ScriptEase ISDK for Java

v 5.01b

Nombas, Inc.

ScriptEase™ ISDK/Java 5.0 Manual, 2nd edition

Table of Contents

Introduction

Thank you for choosing **ScriptEase**!

ScriptEase is a high-performance implementation of JavaScript, highly customizable and portable to a wide variety of operating systems. No matter your scripting needs, ScriptEase has you covered.

In this manual we show you how to add scripting support via **ScriptEase** to your application, use the extensive **ScriptEase API** to control your scripting, and perform a large number of common scripting tasks. This manual does not replace the **ScriptEase** Language Reference. Refer to that guide for information on JavaScript and **ScriptEase** extensions to it.

Let's dive right in by looking at integrating **ScriptEase** into your application.

Integrating the ISDK/Java

This chapter contains the instructions you'll need to integrate the ScriptEase ISDK into your Java application The next chapter builds on this to show you how to use the ScriptEase API to script your application.

Unpacking ScriptEase:ISDK/Java

The distribution you received should first be installed on your system. The distribution consists of a zip file that can be unpacked anywhere on your system. Unpacking the distribution will produce a single directory tree containing the ScriptEase ISDK installation. This tree contains all of the files you will need to integrate the ScriptEase ISDK into your application. It also contains a number of sample ISDK applications. In this manual, we will often refer to filenames and directories. Those should be understood to be relative to the directory you installed ScriptEase into. For instance, if you install ScriptEase in the directory *`c:\se501`* and we refer to the filename *`Samples\Sample\Sample0.java`*, then you should understand that to be referring to the file *`c:\se501\Samples\Sample/Sample0.java`* on your system.

The distribution contains several logical parts. In the `Seisdk` directory can be found the interpreter core jar (`NombasJS.jar`), the ScriptEase standard library jar (`NombasLib.jar`), and the ScriptEase regular expression jar (`SERegexp.jar`) The interpreter jar contains the code to execute scripts along with the ScriptEase API used to integrate the scripting engine into your Java application. The standard library jar contains the code for the standard ECMA function libraries, such as Math or RegExp. The regular expression jar contains classes which are needed by the libraries jar. The NombasLib.jar also contains the SELibraryManager class, which is used to make the standard libraries available to your scripts (The SELibraryManager class is discussed in detail in the Wrapper Functions chapter).

The next major part is the source code for the interpreter and the libraries. The source for the class files contained in the `NombasJS.jar` are located in the `COM\Nombas\jse\Isdk` and `COM\Nombas\jse\utilities` directories, while the source for the `NombasLib.jar` is located in the `COM\Nombas\jse\libraries` directory. If you are using an evaluation version of the ScriptEase ISDK, source code for the interpreter is not included; you use the pre built core jar we provide. (The source for the libraries is provided, however.) The source files found in these directories are not .java files, but instead are .jsrc files. The .jsrc files need to be translated into the corresponding .java files by the included Nombas preprocessor. More information regarding the preprocessor can be found in the following sections.

Sample applications

Before integrating the ScriptEase ISDK into your own application, it is suggested you compile and run some of the sample applications provided with the distribution. They are found in the `Samples` directory. The samples `Sample0` and `Sample1`, are basic samples that are most appropriate to learn from. Each is found in the subdirectory `Samples`.

Once you've build the samples, you may wish to refer to them as working examples while you are adding the ISDK to your own application.

Integration Basics

Integrating the ScriptEase ISDK with your application requires a few basic tasks as outlined below.

Edit your jseopt.jh

The first step is to edit the `jseopt.jh` file found in the `Include` directory of the distribution. This file is self-documenting in its comments. Each preprocessor switch or option has a description of what altering it will do. Change any switch or option to your preference.

Rebuild the ScriptEase interpreter and libraries

In order to incorporate the changes you made to the `jseopt.jh` file, you must first preprocess the source files and then recompile the resulting .java files.

Program your application to invoke ScriptEase

This topic is subject of the rest of the manual. Here we will only mention that all of your source files that are to invoke ScriptEase should import interpreter classes:

```
import COM.Nombas.jse.Isdk.*;
```

Add the ScriptEase classes to your CLASSPATH

The final step for integrating your application with ScriptEase is to add the interpreter and library classes to your CLASSPATH. If you rebuilt the source in a previous step, you would add the directory where those class files were placed to your CLASSPATH. If you did not rebuild the source and plan to use the pre-built jar files which were included in the distribution, you would add those jars to your CLASSPATH. You will also need to include the `SERegexp.jar` file to your CLASSPATH if you plan to use the ECMA RegExp object.

The Nombas Preprocessor

As mentioned above, the source files included in the distribution are not standard .java files. Instead, they are .jsrc files which must be run through the provided preprocessor before being compiled. The preprocessor outputs the corresponding .java files which can then be compiled normally. In addition to the preprocessor commands, the preprocessor also allows source lines in .jsrc files to be appended using the '\' character. For example, the following code:

```
#if DEBUG == 1 && \
   BLAH != 0
```

would be turned into a single line by the preprocessor.

The Preprocessor Commands

The Nombas preprocessor is modeled after the C preprocessor, but only supports a subset of the commands offered by the C preprocessor. Unless stated otherwise, the Nombas preprocessor commands behave exactly like their C counterparts.

#include <filename>|"filename"

Inserts the text of another file into this one. If the file name is enclosed in angle brackets, the preprocessor will only search the current working directory (the directory the preprocessor was invoked from) for the file. If the file name is enclosed in quotation marks, the preprocessor will search the include path for the file. (See below for more information regarding the include path)

The ScriptEase source files include header files which have a .jh extension. These header files are located in the 'Include' directory of the distribution. It is important to place this directory in the preprocessor's include path when processing the ScriptEase source.

#define <name> <body>|<name>(<args>) <body>
#undef <name>

The `#define` command defines a preprocessor macro. When the preprocessor encounters the macro name, it replaces it with the tokens in the specified body. This directive can also be used to defined parameterized macros. The Nombas preprocessor follows the same rules as C regarding the scanning and rescanning of macro expansions. Once a macro is expanded the preprocessor will rescan the resulting tokens for other defined macros. If other macros are found during the rescanning phase, they are expanded and scanned recursively. If a macro is found while scanning its own expansion, directly or as a result of recursion, that macro is not expanded. For instance, given the following macro definitions:

```
#define A B B
#define B C C
#define C A A
```

The code

```
A B C
```

will be expanded into

```
A A A A A A A A B B B B B B B B C C C C C C C C
```

When defining parameterized macros, the '#' character can be used to turn macro parameters into string tokens. For example, the following code results in `Hello World!` being printed to standard output:

```
#define FOO(x) #x
System.out.println(FOO(Hello World!));
```

The `#undef` command is used to undefined a macro.

#if <constant expression>
#elif <constant expression>
#ifdef <name>
#ifndef <name>
#else
#endif

These conditional compilation preprocessor commands are used to include/exclude text from the resulting .java file. The if command includes or excludes text based on a constant expression, which is subject to macro expansion. If the constant expression is true, then the all of the text following the

`#if` up to the next `#endif` is included in the output file. If the expression is false, the text following the #if is ignored. The `#else` and `#elif` commands must follow an #if command and are only evaluated if the corresponding #if expression is false. The `#if` , `#else`, and `#elif` commands work together like the Java `if`, `else`, and `else if` statements.

The `defined` operator is available for use in `#if` and `#elif` statements. This operator results in 1 if the specified name is defined as a preprocessor macro, 0 otherwise.

The `#ifdef` command includes text if the specified macro has been defined and excludes text if the macro has not been defined. The `#ifndef` command is the inverse of the `#ifdef` command.

The `#endif` command is used by all of the other conditional compilation commands to determine where to stop including/excluding text.

#error <message>

Produces an error message with the specified tokens and exits the preprocessor.

Using the Preprocessor

The Preprocessor.jar file is located in the 'Tools' directory of the distribution. A simple example of its usage follows:

```
java -jar Tools\Preprocessor.jar Foo.jsrc
```

This example would preprocess the file Foo.jsrc (assuming it exists) and output Foo.java. The preprocessor accepts the following command line parameters:

```
-I <dir> : adds the <dir> to the Preprocessor's include path.
-O <dir> : places the output files into <dir>
```

To preprocess all of the Nombas source files, run the following commands from the root of the distribution:

```
java -jar Tools\Preprocessor.jar -I Include
COM/Nombas/jse/Isdk/*.jsrc

java -jar Tools\Preprocessor.jar -I Include
COM/Nombas/jse/utilities/*.jsrc

java -jar Tools\Preprocessor.jar -I Include
COM/Nombas/jse/libraries/*.jsrc
```

Since no output directory was specified, the preprocessed .java files are placed in the same directory where the corresponding .jsrc files were found.

Advanced Integration - Debugging

A debug mode build is differentiated by the presence or absence of the `NDEBUG` flag in the `jseopt.jh` file. When this flag is on, asserts are removed and the fastest code is generated while when it is off, asserts are fleshed out to catch errors. ScriptEase follows this convention. When `NDEBUG` is undefined, ScriptEase adds a lot of debugging code.

Using the debug build, many of the common ScriptEase problems, both in integration and scripting, are detected and reported to you. We encourage you to develop your application using the debug-mode of the ScriptEase library before contacting Nombas for technical support. Carefully following this manual chapter and building a debug-mode version will eliminate 90% of the problems commonly encountered and will save you valuable time rather than waiting for technical support to get to your question. If you encounter a true bug in our core, the added information produced in the debug-mode build will allow us to find it and create an errata more quickly.

Before you proceed to the next chapter on scripting your application, please look over the following common problems you will encounter. It is best to keep these in mind first, as many good habits are introduced. If you think of catching bugs as something to be done after your code is written, you will unfortunately spend a lot more time catching them. Prevention is the best cure.

Jsedebug.Log

This is the name of the debug output file. Under DOS, Windows, or OS/2, this file is created in the root of `c:\`. For UNIX and Mac versions, it is put in the current directory. The debug output will be put in this file, appended to whatever the file already contains. When trying to debug your program, delete this file first, run your program, then read the file to see what information it provides. Even if this information is not enough for you to fix the problem, it will be helpful to us, so include it when you contact Nombas for technical support.

Initialization and Contexts

Initialization is necessary before you can actually perform the tasks related to executing scripts. The first task is to initialize the engine itself. Your application only does this once when it starts and terminates the engine once when it exits. Even if you are running multiple threads in your application and running many different scripts, you only initialize the engine once. Here is a code snippet demonstrating initializing and terminating the engine:

```
static public final void main(String[] argv)
{
   SE.seInitialize();

   /* your application, including scripting. */

   SE.seTerminate();
}
```

The two initialization methods (`seInitialize()` and `seTerminate()`) are static members of the `SE` object. The `SE` object contains all the API initialization methods along with all of the constants used by the API.

The other initialization task is to create an `SEContext object`. This is a handle that ties all of your scripting together. Each script you wish to run needs a context. It holds the variables, functions, preprocessor defines, and all the other information a script needs. All of the API calls are methods of the `SEContext` object.

A single context may run more than one script one after the other but not simultaneously. Therefore, if you want to run multiple scripts at once, such as in a multithreaded application, each thread will need its own context. You can create as many contexts as you like. Most applications will create a single context that is used for the life of the application then destroyed.

When a new context is created, none of the standard function libraries will be available for script in it to call. You will need to add the desired function libraries using the `SELibraryManager` object. When function libraries are added to the context, the functions contained in the library are added to the global object. In JavaScript, global variables are just the members of an object, the global object. Any scripts running can see the stock libraries as global variables. This is how a script access stock objects like `eval`, `Math`, `String` and so forth.

To create a context, you must use the SE.seCreateContext ScriptEase method. The only required information for all versions is a context parameters object. The context parameters object implements the `SEContextParams` interface and is passed as the first parameter to `seCreateContext`. The newly created context keeps a reference to the parameter object. The second parameter is a string. If you are using a trial version of the ScriptEase ISDK, you must give your userkey provided to you by Nombas. If you do not do so, the trial version will fail in its construction of a new context and return `null`. If you have a purchased version of ScriptEase, this second parameter is ignored.

The ScriptEase Context Parameter Interfaces

As mentioned in the previous section, when calling seCreateContext you must pass a context parameters object which implements the `SEContextParams` interface. In addition to implementing the SEContextParams interface, the parameters object may implement any of the other ScriptEase Context Parameter Interfaces, depending on the needs of the application. The interfaces are described in the following sections.

interface: SEContextParams

```
 public int seGetOptions();
 public void seSetOptions(int seOptions);
```

The SEContextParams interface is used by the interpreter to access the interpret options you have set for the created context. The interpreter will call the `seGetOptions` method whenever it needs to retrieve your options, and it will call `seSetOptions` when it needs to temporarily turn off certain options for a particular section of code. The following options can be |'ed together in any combination:

SE.DEFAULT

Default behavior

SE.OPT_REQUIREVAR

All variables must be declared using the `var` keyword. If this flag is not used, the normal JavaScript behavior is in effect. When you write to an undeclared variable, the variable is automatically created as a global variable. Reading from an undeclared variable always results in an error.

SE.OPT_DEFAULTLOCAL

Variables used without declaring them with the `var` keyword are declared automatically as global variables as described above under `SE_OPT_REQUIREVAR`. This flag makes them declared as local variables instead. JavaScript standard behavior is to create the variables as global variables.

SE.OPT_WARNBADMATH

If any math operation involves NaN, flag an error. Normally, JavaScript allows NaN to be used in an operation and defines specific results.

SE.OPT_EXTRAPARAMS

Wrapper functions indicate the maximum number of parameters they can take, and extras will cause an error. This flag causes all library functions to take any number of parameters, ignoring excess parameters. It is normally useful to leave out this option, as extra parameters usually signal an incorrect usage of these functions.

SE.OPT_TOBOOLOBJECTS

JavaScript states that any object converted to a Boolean results in `TRUE`. If this flag is on, objects are first converted to a primitive then to a boolean. For instance, without this flag the object `new Boolean(False)` or `new Number(0)` will convert to `TRUE`, but with the flag they become `FALSE`.

SE.OPT_DEBUGGER

A debugger is in use, so ignore `SE.INFREQUENT_CONT` for all `seEval` calls.

interface: SEErrorHandler

```
public void sePrintErrorFunc(SEContext context, String
text);
```

The `sePrintErrorFunc` method is called by the interpreter to print an error to the user. This happens when a script generates an error that is not trapped by a `try/catch` handler. The error needs to be displayed to the user. This is the function that is called by ScriptEase to do so.

interface: SEErrorFunction

```
public void seAtErrorFunc(SEContext context, SEAtErrorInfo
info);
```

The `seAtErrorFunc` method is invoked whenever a script generates an error, at the point of error. This will occur for any error, even if the error is trapped via a `try/catch` handler. Note that some scripts will throw errors as a valid part of their program such as to indicate an error return from a function which will be trapped higher up in the script. This is why normally you do not care about an error until it comes back to you via the `sePrintErrorFunc`, indicating it never is trapped. Getting an immediate notification is primarily useful in implementing a debugger for which the user may want to stop anytime an error is generated even if it will be handled, in order to step through the handling code.

The `seAtErrorFunc` is passed an informational `SEAtErrorInfo` object. Currently, the SEAtErrorInfo has one public method:

```
public boolean getTrapped();
```

The `getTrapped` method returns `true` if the error will be trapped and `false` if the `sePrintErrorFunc` will be called on it.

The actual value of the error is set up in the `SE.RETURN` object, described fully in Chapter V. For an error, the value is an `Error` object as described by the ECMAScript specification. Since working with variables and return values is not described until later chapters, you should revisit this description once you have read those chapters.

interface: SEContinueFunction

```
public void seContinueFunc(SEContext se);
```

The seContinueFunc method is called by the ScriptEase interpreter after every statement while evaluating scripts. It is useful to perform periodic work, such as checking Windows messages in a Windows ScriptEase application. It is also useful in implementing a debugger to regain control after each statement is executed.

When evaluating scripts using `seEval` (described in Chapter VI), you can pass the `SE.INFREQUENT_CONT` flag to have the continue function called much less frequently than once per statement. If all you need to do is an occasional Windows Message processing, calling this function after every statement wastes a lot of processing, which is when this flag is most useful.

You use the standard ScriptEase wrapper return rules to control execution using this method. You can return an error in the normal way, which will abort script execution but can be trapped like any other error. Alternately, you can use set the object/member pair `SE.RETURN,SE.EXIT` to `true` in order to force the program to abort completely. Returning a non error value does nothing, it is ignored. Either you generate an error in order to abort script execution, or you return nothing and the script continues as normal.

interface: SEFileLocation

```
 public String seFileFindFunc(SEContext se,String fileName,
                              boolean findLink);
```

The `seFileFindFunc` is used used by ScriptEase when looking for source files. The filenames being looked for are the filenames passed to the `#include` and `#link` directives. The parameter `findLink` tells you which kind if being looked for: `true` for a `#link` extlib, `false` for a `#include` include file. (Note: the #link directive is not currently implemented for Java, so this parameter should always be `false.`)

If you do not implement this interface, then files are looked for directly, meaning that the filename given must appear exactly as specified in the current directory. By implementing this function, you can handle looking for these files with various extensions in various directories. You are passed `fileName`, the file to be looked for. This is the text that appears in the directive exactly as the user entered it. You return a string containing the translated filename if the file was found. Return `null` if the file could not be found.

interface: SEGetSourceFunction

```
 public boolean seGetSourceFunc(SEContext se,SESourceInfo
                                info,int mode);
```

The `seGetSourceFunc` callback is used to read script files. If you do not provide the callback, files are read using the normal Java File I/O methods. By defining this interface and the `SEFileLocation` interface above, you can completely virtualize your files. Although you can handle the virtualizing of files in this function alone, error reporting is based on the filename returned from `seFindFileFunc` so implementing it is recommended for the user's ease.

The `mode` parameter tells you what the call is intended to do. It can be one of these values:

```
SE.seSourceOpen       open a new file
SE.seSourceGetLine    get the next line from the file
SE.seSourceClose      close the file
```

In each case, the `seGetSourceFunc` method is called with `seSourceOpen`. Returning `false` results in an 'unable to open file' error in the script. Next the method is repeatedly called with `seSourceGetLine` to get the individual lines of the source, until you return `false` to indicate no more lines. Finally, the method is called with `seSourceClose` to close down the file. The `info` parameter points to an object that you use to accomplish these tasks. Each file will be given its own `SESourceInfo` object to work with.

The `SESourceInfo` class defines the following public methods:

```
public String getName();
public void setCode(String code);
public int getLineNumber();
public void setLineNumber(int lineNumber);
public void setUserData(Object userData);
public Object getUserData();
```

The `getName` method is used to access the name of the file, which is the result of your `seFileFindFunc` if you have implemented the `SEFileLocation` interface. The setCode method is where to application puts the successive lines of the file. The `setLineNumber` and `getLineNumber` methods are used by the application to update and retrieve the current line number. Finally, the `setUserData` and `getUserData` methods are used to store and retrieve whatever information the application needs to process the file. A simple implementation would use an `InputStream` for userdata, but a more complex one might need to point to an object keeping necessary data.

interface: SEGetResourceFunction

```
public boolean seGetResourceFunc(SEContext se, int id,
                                 StringBuffer buf);
```

ScriptEase uses a number of text string resources, which it has internal string values for. You can implement this interface to override those values. This is useful for internalization, to translate the text strings into whatever language is appropriate. The `id` parameter indicates which resource ScriptEase is trying to access. You fill in the `buf` with the text you'd like to give the resource.

The list of identifier numbers and the English strings corresponding to them can be found in `Include\rsrccore.jh` and `Include\rsrclib.jh`.

interface: SEPrepareContext

```
public void sePrepareContextFunc(SEContext se);
```

After `seCreateContext` has finished preparing a new context, it invokes the `sePrepareContextFunc`. You can do any final setup on your context here, such as adding your application specific wrapper tables (see Chapter VII). If you do the final preparation here instead of in your code after calling `seCreateContext`, then all calls to `seCreateContext` will do that same preparation. This is useful if you are using utility libraries that create new contexts with `seCreateContext`. It ensures those contexts are properly set up for your application. Nombas has no utility routines that use `seCreateContext`. However, some may be created in the future.

Here is an example of using the ScriptEase Context Parameter Interfaces to create a context:

```
class MyParams implements SEContextParams, SEErrorHandler
{
   private int myOptions;

   MyParams(int options)
   {
      this.myOptions = options;
   }

   public int seGetOptions()
   {
      return myOptions;
   }

   public void seSetOptions(int newOptions)
   {
      this.myOptions = newOptions;
   }

   public void sePrintErrorFunc(SEContext se, String text)
   {
      System.err.println("Error encountered: " + text);
   }
}

public class MyApplication
{
   public static final void main(String[] argv)
   {
      SEContext se;
      MyParams params;

      SE.seInitialize();

      params = new MyParams(SE.DEFAULT);

      se = SE.seCreateContext(params, "");

      /* your application, including scripting using 'se'
       * as the scripting context.
       */

      se.seDestroyContext();

      SE.seTerminate();
   }
}
```

Working with Variables

The ScriptEase engine keeps track of all variables used by the scripts you execute. The ScriptEase API provides functions to examine and modify these variables. The most common place to use these functions is in the body of wrapper functions, which are described in the next chapter. However, that is not the only place you might want to examine variables. For instance, the ScriptEase debugger executes scripts one statement at a time and lets the user examine the variables as it is doing so. The debugger uses the ScriptEase API to do this.

The most important concept to remember is that every variable is a member of some object. There are only a few top-level objects that store all variables and values used by ScriptEase. For instance, if a script says:

```
var a = 4;
```

That global variable `a` is actually a member of an object, the global object, which is one of these top-level objects. All global variables are members of this same object. Similarly, functions can have local variables and parameters, such as in this function, which are also part of an object:

```
function foo(b)
{
   var c = 10;
}
```

This function has two variables, the parameter `b` and the local variable `c`. Both are part of what is called the activation object. Each time a function is called, a new activation object is created for it. There is one global object, but there can be many activation objects. Activation objects are created for a function when it starts executing and destroyed when the function finishes. The ScriptEase API lets you access all activation objects, so you can examine or modify all local variables for functions currently being run.

IDENTIFYING A VARIABLE

The majority of ScriptEase API functions work with variables, retrieving or modifying their values. All the functions share a common way to identify which variable you want to work with. You specify the object and member that the variable resides at as parameters to each function. ScriptEase provides a number of predefined objects that you can use which cover all of the places variables are stored internally. The most common is `SE.GLOBAL`, the global object. Each such object is fully explained below.

Note that `SE.GLOBAL`, `SE.THIS`, and so forth are the names of the object. You pass that exact text to the function to identify that as the object you want to work with.

Later, when we discuss the API functions for examining variables, we will see it is possible for a variable itself to be an object. In that case, `seGetObject` will return an object handle for the object the variable contains. This handle, since it is an object, can be used instead of `SE.GLOBAL` or the other stock objects in further variable identifications. In this way, starting from the top-level objects, you can access any variable on the system.

While it is possible to access any variable in this way, it is not always convenient. For instance, let's say you want to get at the variable `foo[5].goo`. You could do this in steps. You would get `foo` as a member of the global object. After seeing that it is itself an object, you could get the numeric member `5` from it. That gives yet another object from which you could extract the final member `goo`. Not only is that ugly and difficult to understand, but there could be other caveats. A script, when it refers to `foo`, might not be getting a global variable. `foo` could be a local variable, or it could be found because the code is inside a `with` statement. Trying to program all the possibilities would be tedious, lengthy, and error prone.

Fortunately, ScriptEase provides an API call to do this for you. `seVarParse` will take an arbitrary variable name, such as `a` or `foo[5].goo` and tell you what object and member name it is referring to. Once you have the object and member name, you are ready to call any of the variable access functions we will describe below to examine or modify that variable.

LIST OF MEMBER SPECIFIERS

The second half of the variable locator is the object member to access. We will discuss specifying that first, leaving the objects for below, as it reduces the number of forward references.

When you want to access a member of some object, you use one of the following macros to indicate which member you'd like to access.

SE.MEM(String)
SE.UNIMEM(String)

The simplest form, this accesses a named member of the object. In the Java version of the ScriptEase ISDK, `SE.MEM` and `SE.UNIMEM` are identical.

SE.HIDDEN_MEM(String)
SE.HIDDEN_UNIMEM(String)

These are identical to SE.MEM and SE.UNIMEM except that the member accessed is visible only the thei ScriptEase API, and not to the scripts themselves. These hidden members are an excellent way to associate data with an object (such as a Java Object) that you don't want to be seen by the script.

SE.COMPOUND_MEM(String)
SE.COMPOUND_UNIMEM(String)

Similar to SE.MEM and SE.UNIMEM but the member name can represent complex expressions. See seVarParse for limits on these expressions.

SE.NUM(int)

Allows you to access numerically-named members. This is most useful with arrays. Because of the way JavaScript works, member names that are number use the text representation of that number as their name. Thus, `SE_MEM("10")` and `SE_NUM(10)` refer to the same member name. Because of internal optimizations, not only is this naming method more convenient for numeric members, it is faster.

SE.STR(int)

Access a member using an internalized string handle that was received from the ScriptEase API. You can generate such string handles using the `seInternalizeString` API call, and member names passed to your callback functions are in this format.

An internal string handle is faster to use than a string literal member name. When you pass a member name using `SE_MEM`, it has to be converted into an internal string internally before continuing. Therefore, doing it once and referring to members using the resulting handle is faster. In addition, you can compare handles for equality using the == operator, which is much faster than the `String.equals` needed to compare text strings.

SE.INDEX(num)

Internally, all objects members are stored in slots. They are contiguous starting from `0`. `SE_INDEX` lets you access a member by its slot.

The usual use for this method of accessing members is to iterate over all members of an object. The ScriptEase API call `seObjectMemberCount` will tell you how many members an object has, and thus how many slots it is using. Those slots are numbered from `0` to one less than the number of slots.

Note that the slot a particular member uses will changes as members are added to or removed from an object. Do not try to use `SE.INDEX` to access regular members or assume they occupy any particular slot.

SE.STRUCT(SEMemberDesc)

You can use this macro to retrieve the member from an `SEMemberDesc` object and pass it to any of the functions. You store a member in the Object using the `seStoreMember` function. Here is a short code example:

```
SEMemberDesc mem = new SEMemberDesc();
se.seStoreMember(mem,SE.NUM(0));
se.sePutNumber(myObj,SE.STRUCT(mem), 10);
```

Member description structures are useful to pass a member identifier as a parameter to a function.

SE.VALUE

`SE.VALUE` means not to work with any member but rather to work with the object itself. For instance, you can use the object/member pair `SE.GLOBAL,SE.VALUE` to examine or change the global object. For most objects, putting a value to the object itself using `SE.VALUE` will call the operator overload function on that object with the operator `SE.OP_ASSIGN` being overloaded. If the object has no operator overloading, then the operation does nothing and is ignored. Several of the special ScriptEase objects have their own behavior when assigned to the `SE.VALUE` member. For instance, with `SE.GLOBAL`, doing so changes the global object. Read the individual descriptions below of the ScriptEase objects to determine if that object allows a put to itself via `SE.VALUE`, and what that put does.

SE.STOCK(JseStrID)

There are a number of stock member names, which are used via this method. The JseStrID of such a stock string is put as the argument to the method such as

`SE.STOCK(JseStrID.length)` or `SE.STOCK(JseStrID.this)`. For a particular string, such as `length`, `SE.STOCK(JseStrID.length)` is equivalent to `SE.MEM("length")`. The advantage to using the `SE.STOCK` method it that the method is faster. However, only some of the more commonly used member names are available with `SE.STOCK`.

The following variable names have stock IDs and can be used with the `SE.STOCK` method:

`Array`	`Boolean`	`Buffer`	`Date`
`Exception`	`Function`	`Number`	`Object`
`RegEx`	`String`	`__parent__`	`_argc`
`_argv`	`_call`	`_class`	`_construct`
`_prototype`	`_value`	`arguments`	`callee`
`constuctor`	`global`	`length`	`main`
`preferredType`	`prototype`	`this`	`toSource`
`toString`	`valueOf`		

SE.FUNCTION_TEXT

This is a special member that cannot be created, written to, or deleted. Only script functions have this member. For all other objects, `SE.FUNCTION_TEXT` is undefined. The value associated with the member is a text version of the script function's body.

SE.OBJECT_DATA

This is a user-defined member associated with all objects. It is internally a reference to an `Object`, so you'll want to store your data to this member with `sePutPointer` and retrieve it with `seGetPointer`. It allows you to store arbitrary user data with ScriptEase objects in your application. This is commonly used when designing custom object classes specific to your application and is explored more fully in Chapter IX, Objects. All stock ScriptEase objects, described below, do not have an `SE.OBJECT_DATA` member, only user objects have one.

SE.LIBRARY_DATA

All wrapper functions have a piece of user data associated with them, determined by the seAddLibTable call that initialized that function. Use `seGetPointer` to retrieve that data from a wrapper function. Only wrapper functions have library data associated with them.

LIST OF STOCK OBJECTS

ScriptEase scripts contain a large amount of data you might want to access. You access a particular piece of data using one of the following stock object. The name given is the value you pass as the `object` parameter to any of the variable accessing functions described below. After that is a description of the object and what data you will actually be getting or changing when you access and modify its members.

SE.GLOBAL

The members of this object are the global variables of the script. For instance, the global variable `zed` is the member `zed` of this object. You can examine and change global variables by using this stock object. You can also change the global object itself by using using `sePutObject` on `SE.GLOBAL,SE.VALUE`.

Note that all functions remember the global object in effect when they were first created and swap that in when they are executing. This facilitates scoping where multiple scripts are executed using `seEval`, each with a new global object. The various functions remember their global object, so variables created with the `var` keyword in the script the function was evaluated in are accessible whenever the function is executing. For some programs, this behavior is undesirable. For instance, a person might want to create utility functions that always act on the variables in effect and run them with different global objects. There are two ways to change the behavior.

First, by turning off `JSE_MULTIPLE_GLOBAL` in your `jseopt.jh` file, this behavior is turned off completely. Alternately, you can turn the behavior off for any wrapper function by including the `SE.KEEP_GLOBAL` flag in the function flags used to define that wrapper function (see Chapter VII, wrapper functions for more details.)

SE.ARGS

The members of this object are the arguments passed to your wrapper function. If you are outside a wrapper function, this object has no members. You cannot add or delete members from this object, you can only access and update the actual parameters passed to your function.

Since ScriptEase supports named arguments, you can specify a normal member name. However, most wrapper functions have no names for the arguments. Normally, you just use `SE.NUM(x)` to access argument number `x`. For instance, `SE.NUM(0)` is the 0th argument (i.e. the first argument passed to your function) and so forth. For the arguments object, `SE.INDEX` accesses the members just like `SE.NUM` does, they are synonymous.

You cannot use `SE.VALUE` with `SE.ARGS`.

SE_ACTIVATION

An activation object is the object used to store local variables and parameters to a function. Since a wrapper function is written in Java, it has no such variables. However, the calling script function does, and it is often convenient to be able to access them. `SE.ACTIVATION` accesses the calling script function's activation object. If there is no calling script function, `SE.ACTIVATION` accesses the global object, just as `SE.GLOBAL` does. You can get the activation object by using `SE.ACTIVATION,SE.VALUE` but you cannot write to it.

SE.THIS

Whenever a function is called, it has a `this` variable associated with it. For instance, when you call a function such as:

```
foo.func();
```

`foo` is the `this` variable for the function call to `func()`. When you don't specify an explicit `this` variable, such as:

```
func();
```

The `this` variable is implicitly the global object. You can access the `this` variable for your wrapper function, which is always an object, and its members by using the `SE.THIS` stock object. If you use `SE.THIS` outside of a wrapper function, it is always the global object.

You cannot write to `SE.THIS,SE.VALUE`, it is read-only.

SE.SCOPE

Using SE.SCOPE is a way to locate a variable in the same way that the script interpreter would, following the _prototype and __parent__ chain of the active functions. If you know for certain what object a member belongs to (some object or SE.GLOBAL or SE.THIS for example), then using that object directly is more efficient, but SE.SCOPE mimics the flexible scoping rules of the ECMAScript language.

SE.TEMP

Often a ScriptEase API program needs to create variables of its own to store temporary data. `SE.TEMP` refers to an object where such data can be stored. This object lasts for the life of the context, so your data can be long lasting if you desire. You are free to add and remove members from this object as you need. Please see the in depth explanation of this object later in this chapter.

You cannot write to `SE.TEMP,SE.VALUE`.

SE.WRAPPER_TEMP

Like `SE.TEMP`, but the object and all of its members goes away when the current wrapper function finishes. If used outside of a wrapper function, it is identical to `SE.TEMP`. Like `SE.TEMP`, you cannot write to the object itself.

SE.NOWHERE

This is a garbage sink object. Any write to any member of this object is ignored, as are attempts to create new members. Any read of a member returns the undefined value. It is intended for one particular use, namely as a return from functions that return an object/member pair when there is an error. If the programmer doesn't check the error return and just tries to use the returned object/member in this case, a `SE.NOWHERE` member is returned so the access does nothing and doesn't crash.

If you compile the core with the `SE.TRAP_NOWHERE` option on, accesses to this object will trigger assert failures. This is intended to facilitate debugging, so you can find where you have not properly checked your returns from ScriptEase API calls.

SE.DEFINES

The `#define` statement defines a macro, an identifier mapped to be a different value. Note that ScriptEase does not support parameterized macros. This object's members are the defined macros, the value of the members are strings that are what the macro is defined to be. In other words, if you have this statement:

```
#define FOO goo
```

Then one of the members of the `SE.DEFINES` object will be `FOO` and its value will be the string `"goo"`. You can examine and modify the defines, as well as adding new ones by creating new members. Note that these defines will only affect scripts that are executed after you make the change.

You may not write to the `SE.DEFINES` object itself.

SE.RETURN

This is the place in which the return value from your wrapper function is to be stored. This discussion will only be minimal for now, a more complete discussion of this object can be found later in the chapter. `SE.RETURN` mainly works with the `SE.VALUE` member. For the return, the `SE.VALUE` is the value to be returned. That is to say, if you want to return the value 10, you would write:

```
se.sePutNumber(SE.RETURN,SE.VALUE, 10);
```

The object has four secondary members, all of which are booleans. You set one (and only one) of them to `true` to indicate a special return condition. They are:

SE.ERROR

The returned value indicates an error. If this flag is `false`, the value returned in `SE.VALUE` is a normal return like the JavaScript statement `return 10;`. If this flag is `true`, it is the equivelent of `throw 10;`.

SE.EXIT

Exits out of the script with the returned value being the return of the script. This is exactly analogous to a Java program calling `System.exit()`, for instance `System.exit(10);`.

SE.YIELD

Causes the script to drop back to the calling seExec API call. This is useful for fibers, described in Chapter XIII. It will allow the next fiber to take its turn. The return value is still returned as normal once the fiber gets its next turn to run.

SE.SUSPEND

Similar to `SE.YIELD`, in that the fiber drops back to the calling ScriptEase ISDK application. However, the fiber cannot be run until resumed by the application. Any calls to `seExec` will return immediately. The application restores the fiber by putting `false` to that fiber's `SE.RETURN,SE.SUSPEND` member. In addition, the application may also modify the return value after it does so, but before it `seExec`s the fiber. This is useful for implementing wrapper functions that delay the fiber until some needed value is available, then return that value. The application manager can examine the wrapper function's return value to allow the wrapper function to communicate with the manager. Chapter XIII has more details.

In addition to a place to put your return, the `SE.RETURN` object is also where you receive the result from seEval API invocations. The reason for the dual use is simple; in many cases, you want to execute some code using `seEval` then pass along the result. By putting it in this place, you can immediately return from your wrapper function, returning that value.

Note that once `SE.ERROR` is set to true, the return value is locked into place and cannot be changed. The reason is again for convenience. Many times you want to just run your snippet of code and not check for errors. In this way, if an error occurs, it takes precedence. You don't have to check for the error to avoid overwriting it. However, you can reset the `SE.ERROR` boolean back to `false` if you want to erase the error and overwrite it.

SE.AT_EXIT

The `SE.AT_EXIT` object contains a number of members who are themselves functions, wrapper or script. These functions should take no parameters. All of the function members are invoked when the current seEval with the flag `SE.EXIT_LEVEL` completes. The `this` object for these functions will be the `SE.AT_EXIT` object they are a part of.

Members of the `SE.AT_EXIT` object that are not functions are ignored. However, they can be used to store information retrieved by an at-exit function through its `SE.THIS` object.

The `sePutWrapper` API function is useful for creating functions as members of the `SE.AT_EXIT` object to then be called.

You should choose member names for your at-exit functions that are unique and unlikely to be duplicated. Using short, common member names runs the risk of overwriting another at-exit function or its associated data members.

SE.FILENAMES

This object is an array, so it has members `0`, `1`, and so forth. Each member's value is a string, one of the filenames the script is using. These filenames are the source files of the script. Because a script can use the `#include` directive, a script can be made up of several source files.

SE.STACK_INFO(depth)

These objects contain information about all function calls currently being executed. `SE.STACK_INFO(0)` represents info on the wrapper function you are in, `SE.STACK_INFO(1)` is the function that called you, and so forth. The maximum depth you can look back is determined by the compile-time constant `SE.MAX_STACK_INFO_DEPTH` which defaults to 64. Thus, by default, you may use `SE.STACK_INFO(0)` to `SE.STACK_INFO(63).`

Here are the members of each stack info object and what that information is. These members are all read-only:

SE.SI_WRAPPER

A boolean, `true` if the function is a wrapper function, `false` if it is a script function.

SE.SI_FUNCTION

The object for the function.

SE.SI_FUNCNAME

The string name of the function.

SE.SI_TRAPPED

`true` if an error occuring would be trapped if it occured in this function.

SE.SI_GLOBAL

The global object for this function.

SE.SI_THIS

The `this` object for this function.

SE.SI_DATA

The user data associated with this function if it is a wrapper function. This is `null` for script functions. You retrieve this value using `seGetPointer`.

SE.SI_FILENAME

The filename the current line of the script function is in. For wrapper functions, this will be undefined.

SE.SI_LINENUM

The line number of the current line of the script function, or 0 for wrapper functions.

SE.SI_ACTIVATION

The activation object for the script function. For wrapper functions, it will be the same as the global object.

SE.SI_SCOPECHAIN

The current scope chain for the script function. See Chapter VI on script evaluation for more information on scope chains.

SE.SI_DEPTH

The depth of the function. As you go deeper into the stack, the depth gets smaller. The depth from any particular stack info object is the number of stack objects in total. For instance, if a script is in a wrapper function called from the main body of the stack, then the call stack comprises two levels, `SE.STACK_INFO(0)` and `SE.STACK_INFO(1)`. The `SE.SI_DEPTH` is indication of how many function calls are nested beneath this level, including the level itself. In this case, the `SE.SI_DEPTH` of `SE.STACK_INFO(0)` will be 2, indicating 2 items nested. The `SE.SI_DEPTH` of `SE.STACK_INFO(1)` will be 1, as this function has only itself and nothing nested beneath it. The depth can be 0 if `SE.STACK_INFO(0)` is used while no code is executing and thus no function calls are nested at all.

SE.SERVICES

This is a storage space to associate names with arbitrary data. What data you associate is reserved for your application. ScriptEase: Desktop, for instance, stores a number of pointers with names.

You are allowed to retrieve and store values to `SE.SERVICES,SE.VALUE`. This is a single slot that is accessed more quickly than the others. However, there is only one such slot. Therefore, an application writer is given that slot for his application. Any utility function library that need to associate data with a context must use a named member of the `SE.SERVICES` object to store its data.

SE.SELF

A wrapper function can use this object to refer to itself, the wrapper function that is currently executing. It is used most often to retrieve the library data for the executing wrapper function using the `SE.SELF,SE.LIBRARY_DATA` pair.

EXAMINING VARIABLES

Now that you know how to select the variable you are interested in, let's look at examining its value. JavaScript does not have variables of fixed type. When a script is run, any variable can be assigned a value of any type. Each can be assigned a value of differing types as the script continues. For this reason, you have to look to see what type a variable currently is. You do this using the seGetType API call. This tells you what type an object member is. You can use this to execute different code based on the type of a variable.

Fortunately, you can let ScriptEase worry about converting types and just ask to get a variable's value as a certain type. If the variable is not of the correct type, it is converted. This is useful because many JavaScript functions are designed to work this way; when passing parameters to the standard JavaScript functions, they are converted to the type the function expects automatically. If you allow ScriptEase to do the same when you are accessing variables, you will automatically follow the JavaScript standard that variables are converted to the correct type whenever necessary. The `seGetXXX` functions, where `XXX` is based on the type you'd like to get, convert the ScriptEase variable value to the given type and return it to you, in Java format. Note that the variable is not permanently changed. You can use the `seConvertXXX` API call to do that.

All of the API calls that read a variable's value, either to get its type or get its value, read the value exactly once per API call. This is important to understand the behavior of dynamic objects. If you just use `seGetXXX`, the value is read, converted to the required type, and returned to you. This is the preferred method. However, you may want to read the type then get the value of that particular type, presumably to do different things based on the variable's type. Understand that this involves two calls to API functions that read the value, one to seGetType and one to a `seGetXXX`. This means the value will be read twice. If the object the member is being retrieved from is dynamic, that dynamic `get` will be called twice. It is possible for it to return two different values, defeating the purpose of your code.

In this situation, because `seGetXXX` is safe, your code will not crash just operate unexpectedly. You can ignore such objects and let the object's designer worry about it. Alternately, you can use `seAssign` to grab the value and store it in a temporary location. This will read the value once. Now you can use seGetType and `seGetXXX` on that stored value, knowing it will not change.

MODIFYING VARIABLES

The other thing you do with variables is to change their value. You use `sePutXXX` to put a particular value into variable. Again, `XXX` has various options for the various types of data you can store in a variable. Like a JavaScript assignment, whatever value the variable held before the call is discarded in favor of the new value. If the variable is a member of a dynamic object, a dynamic `put` will be called to store the value.

In the same way as for reading a variable's value, each API call that modifies a variable's value does so once, meaning one dynamic `put` call will be made per API function call. You can use a similar technique to reading, build up the value in a temporary location then put the value once to the real location, so any dynamic `put` is called only once.

USING SE.TEMP AND SE.WRAPPER_TEMP

Using these objects is pretty simple. You create a member, store some value it in it, then delete it when you are done. For `SE.WRAPPER_TEMP`, you often do not delete the members explicitly and instead let them go away automatically when your wrapper function exits. The only problem arises in selecting which member to use. You need to ensure that you do not conflict with some other part of your program that may also be using a temporary member of these objects, or to utility functions potentially written by someone else.

The way to do this is to choose a member name for your temporary variable that is not a simple name like `foo` or `i`. It is suggested that you use a name that incorporates the filename and wrapper function name, since that should be unique for your application. For instance, your member name might be `foo.c:my_wrapper.temp1`. In this way, you can ensure that your program does not mysteriously fail due to conflicting `SE.TEMP` member names.

SE.RETURN EXPLAINED

The SE.RETURN object is potentially the most confusing of the objects. However, it does not have to be. The main object/member pair is `SE.RETURN,SE.VALUE` and is where you put the return value for your wrapper function. For instance, if you want to return the number 10, you would write:

```
se.sePutNumber(SE.RETURN,SE.VALUE, 10);
```

That part is easy. However, the `SE.RETURN` object has four other boolean members: `SE.ERROR`, `SE.EXIT`, `SE_.IELD`, and `SE.SUSPEND`. The last two are used for fibers and are covered in Chapter XIII on fibers. The first two are discussed next.

After you return a value (not before), you can mark that as an error result by setting the `SE.RETURN,SE.ERROR` member to be `true`. Consider the JavaScript statement:

```
   return 10;
```

versus

```
   throw 10;
```

In the first case, the result is 10. In the second case it is also 10, but it is an error result of 10. If you don't understand the `throw` statement, you should consult a JavaScript reference. The `return` statement is identical to the example we gave above. The `throw` statement is done from the ScriptEase API as follows:

```
se.sePutNumber(SE.RETURN,SE.VALUE, 10);
se.sePutBoolean(SE.RETURN,SE.ERROR, true);
```

Throwing arbitrary values in this way is not common and is usually reserved for complex scripts. Most often, you want to throw an exception. Some error happens, such as illegal parameters to your wrapper function, and you want to generate an error. That is a common occurance, and ScriptEase provides the `seThrow` API call to do so. Explicitly setting `SE.RETURN,SE.ERROR` to `true` is very uncommon, and you probably won't ever need to do it.

Similarly, the `SE.EXIT` flag indicates that the script should exit with the given value. Consider the Java statement:

```
System.exit(10);
```

`SE.EXIT` is usually used to abort a script when an error occurs. Most of the time, you will use `seThrow` to generate an error. `seThrow` errors can be trapped using the `try/catch` statement allowing the script to recover from errors. However, if something so drastic has happened that the wrapper function decides the script must abort immediately and should not be trapped, you can duplicate the Java `System.exit()` call using the `SE.EXIT` flag. This code does exactly that:

```
se.sePutNumber(SE.RETURN,SE.VALUE, 10);
se.sePutBoolean(SE.RETURN,SE.EXIT, true);
```

There is one final thing you should know. Normally, you can keep overwriting `SE.RETURN,SE.VALUE`, and the last value returned is the result of the function. However, once any of the four boolean members is turned to `true`, `SE.RETURN,SE.VALUE` becomes read-only. Any error is locked in this way. This means that if you call functions inside your wrapper function that generate an error, that error will also be the result of your own function, and propagated back to the user. This is usual desired behavior. In this way, you often do not need to check the error results of the ScriptEase functions you call, as those errors take precedence over whatever you try to return. This leads to small, easy-to-understand wrapper functions in most cases. If you have a more complex wrapper function that can recover from errors, you can unlock the error result by setting whichever of the four members that is `true` back to `false`.

Script Execution Topics

Before we delve into customizing your scripting environment for your application, it's time to talk about the most common scripting operation: executing a script.

Using seEval

Having created an `SEContext`, you use this context to invoke scripts via the ScriptEase API. The ScriptEase ISDK function to execute script code is `seEval`. This function has a large number of parameters to control its behavior and the behavior of executed code. This chapter is devoted to explaining `seEval`. Let's start with a simple example that uses default values for most of the parameters:

```
se.seEval("var a = 10;",SE.TEXT,
          null,null,SE.DEFAULT,null);
```

All of the `null` values indicate a parameter that we are not interested in providing, using the default value instead. This call as it is written will evaluate the simple script `var a = 10;`. The full prototype of `seEval` is as follows:

```
   boolean
seEval(Object to_interpret,int interp_type,
       String text_args,SEObject stack_args,
       int flags,
       SEEvalParams params);
```

The function returns a boolean indicating whether or not the evaluation succeeded. It would not if the script to evaluate contains an error. In addition to indicating success or failure, the script returns a value using the `return` statement. This value returned from the script or function is stored in the `SE.RETURN` object. This means that if you invoke `seEval` in a wrapper function then immediately return from the wrapper function, the result of the evaluation is passed along as the result of your wrapper function. This is a useful technique which is used, for instance, to implement the ECMAScript `eval` function.

An important concept of an evaluation is that of the global object. All global variables in the script, as well as functions, are put into the global object. When the script completes, all variables and functions are still part of the global object. This means that additional calls to `seEval` will find the variables and functions from past calls. You can specify a particular global object in the `params` parameter to put these variables and functions in that object as is described below.

As was mentioned when describing error returns, once a context has an error as its return, any attempts to change the return value are ignored. Likewise, any calls to `seEval` are ignored for the same reason. It is the most reasonable course of action when some previous API call generated an error. You must first erase the error as was described if you want to use `seEval`.

Let's look at the parameters and explain their use.

TO_INTERPRET, INTERP_TYPE

The first and second parameters are linked together. The second parameter, `interp_type`, indicates what type of object the first parameter,`to_interpret,` is. Here are the possibilities:

```
INTERP_TYPE   TO_INTERPRET
-------------------------------
SE.FILE       String
SE.TEXT       String
SE.PRECOMP    byte[]
SE.FUNC       SEObject
```

`SE.FILE` indicates a filename, which is read, parsed, and interpreted.

`SE.TEXT`, which we've already seen above, indicates the source code as a text string.

`SE.PRECOMP` allows you to execute a precompiled script. You pass as the parameter the script buffer that was given to you by the `sePrecompile` ScriptEase API call.

`SE.FUNC` allows you to execute a function. You pass the function you wish to execute. Remember, in JavaScript, a function is just an object. You can retrieve the function you wish to call via the seGetObjectEx API call.

TEXT_ARGS, STACK_ARGS

The next two parameters are likewise related. You can pass arguments to your script or function via one of them. `stack_args` takes precedence so if you use them both the `text_args` are ignored. In either case, the arguments are extracted and passed to the called script or function. For a function, these are just standard arguments. Script arguments are treated like `argc` and `argv` for the main function in a C program. They are stored for the script in the global variables `_argc` and `_argv`.

For text arguments, specify the arguments in a text string, i.e. `"-v foo"`. This is parsed in exactly the same way as a command line; white space is used to separate the arguments, and each is turned into a string in the `_argv` array. For `stack_args`, you pass in a ScriptEase stack object created via the `seMakeStack` ScriptEase API call. The arguments are defined by setting members of this object numerically using the `SE.INDEX()` member format. In other words `SE.INDEX(0)` is the first argument, `SE.INDEX(1)` is the second, and so forth. This form of parameter passing is more commonly used for functions. Most scripts that handle arguments expect all of their arguments to be text strings. If you pass a script arguments that are not text strings, such as numbers or objects, you will probably confuse it.

When you call a function that passes any parameters by reference, the arguments in the stack object will be updated appropriately, so you can check their final value before destroying the stack object after the function returns.

FLAGS

The `flags` parameter is any of the following values, `|`'ed together:

SE.NO_INHERIT

An eval is normally treated like the script text appeared in the containing script at the point it is executed. The script can see the same variables of its parent, change them, and so forth. If you user this flag, the eval is completely separate. It has no effect on its parent except to return a value.

If you use this flag, the stock libraries previously added to the parent will have new copies initialized for the child. See `SE.NO_LIBRARIES` below.

SE.NO_LIBRARIES

Only used with `SE.NO_INHERIT`, the stock libraries are not made available. This flag is usually not useful, as the script will not be able to call any of your wrapper functions. Still, you may want to just perform a computation that doesn't need to spend the time to reinitialize standard libraries that won't ever be called.

SE.NEW_GLOBALS

If `SE.NO_INHERIT` flag is used, this flag is also automatically used. When the flag is not used, any new variable created is stored in the global object of the parent and is still around after this script finishes executing. If this flag is included, a new global object is created for variables the script uses, and those variables go away when the script completes. Specifying your own global variable in the `params` parameter overrides this flag.

SE.CALL_MAIN

The ScriptEase extension of calling a function `main` after the evaluating the code outside any function will be used. If the flag is not included, a function `main` is not treated as special. This does not apply to calling a function.

SE.FUNCS_ONLY

A script is executed in two parts. First, any function defined in it are extracted and created. Likewise, any variables defined using the `var` keyword are initialized as undefined. This happens at the very start of the script. The second part executes any code in the script. For instance, consider this script:

```
function foo()
{
   return 10;
}

var a = 10;
```

A normal evaluation creates a function `foo` in the global object as well as a variable `a` as the undefined value. Then the code in the script is run, assigning `a` to be 10. If you specify the `SE.FUNCS_ONLY` flag, only the first part is executed. In this case, the function `foo` and the variable `a` are created, but the the script body is not run, so the assignment to `a` is not executed.

Understanding this behavior also is helpful in understanding a subtle ECMAScript rule, that all variables defined with the `var` keyword are extracted and initialized before any code is run to be the undefined value. In this script example, ECMAScript treats the script as if you wrote instead:

```
var a;

function foo()
{
   return 10;
}

a = 10;
```

SE.EXIT_LEVEL

Normally, any new at-exit functions are added to the parent. This means they are not called when the `seEval` is done, but rather when the whole context is cleaned up. If you turn on this flag, at-exit functions created inside the eval are called when the eval is finished.

This may seem like a good idea, but there is an important caveat. At-exit functions normally clean up resources. Since an evaled script can return a value to you, that value may be dependent on those resources. If the at-exit functions have been called, the value is using resources that have been cleaned up. This is why you usually want all at-exit functions held until the context is being destroyed, so you know all such values are no longer used.

SE.NEW_DEFINES

Normally any new defines (i.e. MACROS) are added to a global list, and will remain for any new `seEval` calls. This flag makes a new list only accessible to this evaluation for any new `#defines` defined in the script.

SE.NO_OLD_DEFINES

This flag will automatically turn on `SE_NEW_DEFINES` as well. Defines already created, such as by previous `seEval` calls, will not be applied to this evaluation.

SE.REPORT_ERRORS

In many cases, you just want to interpret a script and continue. The script should print any errors and then you are ready to do something else. That's what this flag means. In other cases, you want whatever the call returns returned to you, even if it was an error. For instance, you may want to pass the result along, error or otherwise. In this case, don't include this flag and the return value of the called function will also be copied to your own return value.

SE.INFREQUENT_CONT

Normally, the `seContinueFunc` function is called after each statement so debuggers can function properly. With this flag, it is called much less frequently. This is useful in Windows in which the continue function must check Window messages so the task doesn't get the 'not responding' problem. However, calling it after each statement wastes a lot of time. This flag causes the continue function to be called far less frequently.

SE.START

The script is initialized but not actually run. You use the seExec API call to execute one block of the script. `seExec` executes the script until the next time a `seContinueFunc` would be needed, so refer to `SE.INFREQUENT_CONT` above. The use of this flag is intented to allow easy cooperative multitasking within your application. You can call `seExec` to execute one small script piece at a time, with whatever other code you desire between calls. You can run several scripts simultaneously, each in their own `SEContext`, by calling `seEval` on each with this flag set, then calling `seExec` on each in turn.

SE.CONSTRUCTOR

This flag is only applicable if calling a function. `seEval` will then call the function as a constructor, i.e. as `new Func()` rather than `Func()`. The `this` you pass is usually `null`, in which case a blank version of the object type is created for the constructor, the default behavior when you do a `new Func()`. You can make the `this` something else in which case the constructor will get it. Watch out, this may confuse constructors. Also, some constructors ignore the provided object and create their own.

SE.NAMED_PARAMS

Passes parameters by name. This can only be used if passing parameters in `stack_args`, and all the object members must have a name. You may use this flag only when calling a function.

SE.INIT_IMPLICIT_THIS

When this flag is specified, initialization code (i.e., global code that is outside of any function) will execute as if it is in a function with the SE.IMPLICIT_THIS flag set. This flag is useful when executing small pieces of code that need function-like scoping behavior, such as events.

SE.INIT_IMPLICIT_PARENTS

When this flag is specified, initialization code (i.e., global code that is outside of any function) will execute as if it is in a function with the SE.IMPLICIT_PARENTS flag set. This flag is useful when executing small pieces of code that need function-like scoping behavior, such as events.

SE.DEFAULT

No special options.

PARAMS

This parameter is a object that contains several access methods. You use the access methods to set the various option variables in the object. You can pass `null` if you do not want to specify any of them. Here are the access methods of the SEEvalParams class:

```
public SEObject getScopeStart();
public void setScopeStart(SEObject scopestart);
public SEObject getScopeEnd();
public void setScopeEnd(SEObject scopeend);
```

The scope chain is how ScriptEase determines what a variable name is referring to. The scope chain is a list of objects. For a typical function, the list contains the activation object in which local variables are stored and the global object in which global variables are stored. The variables themselves are members of the object they are a part of. As a result, for a typical function, the local variables are first search for a variable name followed by the global variables. You can specify your own objects to be added to the list.

The `scopestart` and `scopeend` members of the `params` object are SEObjects created using the `seMakeStack` API call. The members of these objects should be objects to be added to the scope chain. These objects are added at the start or end of the scope chain respectively. Those added at the start are searched first.

Adding objects to the start of the scope chain is analogous to a script execution inside a `with` statement. A `with` statement adds a single object to the start of the scope chain. `setScopeStart` allows you to add a list of objects, but the same principle applies. `setScopeEnd` works similarly but adds objects to be searched after all other places to search for a variable name.

```
public SEObject getGlobal();
public void setGlobal(SEObject global);
```

Indicates a new global variable to evaluate the script using.

```
public SEObject getDefaultThis();
public void setDefaultThis(SEObject default_this);
```

The `default_this` parameter allows you to determine which object will be the `this` variable for the executed script or function. For a script, the `null` value is traditionally used which makes the global variable the default `this` for the evaluated script as well. For a function, if the function is being invoked as a member of some object, that object should be passed as the `default_this` variable instead.

```
public SEObject getSecurityInit();
public void setSecurityInit(SEObject security_init);
public SEObject getSecurityTerm();
public void setSecurityInit(SEObject security_term);
public SEObject getSecurityGuard();
public void setSecurityGuard(SEObject security_guard);
public SEObject getSecurityObject();
public void setSecurityObject(SEObject security_object);
```

These are the standard security functions as described in the ScriptEase language manual chapter on security. These objects work exactly the same in the ScriptEase ISDK as they do for any other ScriptEase security application.

```
public String getFileName();
public void setFileName(String filename);
public int getLineNum();
public void setLineNum(int linenum);
```

These parameters are used when the `SE.TEXT` form of script is executed. They specify the virtual filename and starting line number for the script text. This is helpful in reporting errors that might occur in the script text.

FUNCTION GLOBALS

One ScriptEase feature that you should keep in mind is that all functions remember the global object in effect when they are created and use that as their own global object when called. A script file, especially header files, may be self-contained packages that add functions and variables to the global object in initialization. Those functions cannot work if they are run under a different global object, they need their global object in which their definitions are stored. Therefore, specifying a different global object for a function to run under has no effect, because it is changed back when the function is actually run.
If you'd like to turn off this behavior, you can use define:

```
 #define JSE_MULTIPLE_GLOBAL 0
```

in your jseopt.jh file. Be warned, doing so may make script function libraries written by other people incompatible with your application.

SCOPING

A topic that leads to much confusion is that of scoping, and how to control it. Scoping is the process of resolving a variable name when it is encountered in a script. Normally, local variables are searched for the given variable name, if any, followed by global variables. The JavaScript `with` statement is the most common way to alter scoping. The various scoping rules and issues will now be examined.

SCOPING - GLOBAL CODE

Global code is code outside of any function. Scoping for global code is simple, the global object is searched for variables only. Modifying the scoping of global code is done in the seEval call used to invoke that code using the `ssetScopeStart` and `setScopeEnd` methods of the `SEEvalParams` class as described above. The script user can then modify the scoping by using the `with` statement.

SCOPING - FUNCTIONS

Functions are more complex. The normal behavior for a function is to search its local variables and parameters first. Next, the local variables and parameters of its parent function are searched. This only applies if the function is nested inside a parent function, and it includes all parents if it is nested several levels deep. Finally, the global variables are searched. Again, the user can modify this behavior using the `with` statement.

There are several methods for controlling the scope of functions. If you call the function directly using `seEval`, you can specify additions to the scope chain using the `ssetScopeStart` and `setScopeEnd` methods of the `SEEvalParams` class as described above. This method is rarely used because functions are usually called from within a script.

The second method is to use the `SE.IMPLICIT_THIS` and `SE.IMPLICIT_PARENTS` attributes. A script function can be given these attributes using the `seSetAttribs` API call. Both of these attributes modify the function's scope chain by adding elements to the scope chain after the local variables, but before the global variables. The `SE.IMPLICIT_THIS` flag makes the function add its `this` object to the scope chain. This makes the function behave much like a Java method in that members of the `this` object can be referred to directly without having to qualify them with `this.` as is normal for JavaScript. `SE.IMPLICIT_PARENTS` is similar, except the parents of the `this` variable are added to the scope chain. Parents are linked through the `__parent__` (two underscores on each side) member. `this.__parent__` is the parent of the `this` variable and is added to the scope chain if `SE.IMPLICIT_PARENTS` attribute is set in the called function. Next, the parent of that object is added and so forth for all parents in the chain. This is most useful for implementing browser behavior, notably event handlers. The parents of an event handler, the element it belongs to, the document it is in, and the window it is part of, are all implicitly added in this fashion.

CONTINUE FUNCTION

The continue function is provided to allow an API application to process code while a script is being executed. One use of it is to implement a debugger. A second use is to process Windows messages. This section pulls all the information about the continue function into one place.

The function is provided by the object provided to the `seCreateContext` API call that implements the `SEContinueFunction` interface. During `seEval` script evaluation, the function is called after each statement in the script is processed. However, the `seEval` call can be provided with the `SE.INFREQUENT_CONT` flag to call the function far less often. This flag is useful when code must be processed occassionally, but not nearly as frequently as after each statement. Because debuggers must regain control after each statement, the `SEContextParams` has a flag `SE.OPT_DEBUGGER`. This flag overrides the `SE.INFREQUENT_CONT` flag. This allows the API user to use the `SE.INFREQUENT_CONT` flag whenever it makes sense. A debugger can then optionally be used on the same code correctly, without changing that code.

Finally, when using the `SE.START` flag with `seEval` to execute code one piece at a time, control is automatically returned from the `seExec` API call after each script chunk. The amount of code executed in each call to `seExec` is determined by the presence or absence of the `SE.INFREQUENT_CONT` flag and the `SE.OPT_DEBUGGER` flag. Since control is returned to the caller after each seExec call, the continue function is not called in this case. However, you may wish to call it yourself, depending on your application.

Wrapper functions

Wrapper functions are functions that are written in Java using the ScriptEase API instead of being written in JavaScript. From the script's point of view, they appear just like any other function and can be called identically. Most wrapper functions are initialized by the application before running scripts so as to be available to the script user right from the start. This is done by writing a wrapper function table and adding it to an `SEContext` using the `seAddLibTable` ScriptEase API call. The table is added before the application makes any calls to the `seEval` ScriptEase API call. All of the standard ECMAScript objects, such as `String`, `Math`, or `Number`, are written using wrapper functions and wrapper function tables, so you have a large body of example wrapper functions included with ScriptEase to look at.

Here is a sample wrapper function, to get an idea of how one looks. The rest of this chapter is devoted to demystifying it:

```
public void print()
{
   return new SEWrapper()
   {
      public void wrapperFunction(SEContext se, int argc)
      {
         switch( se.seGetType(SE.ARGS,SE_.NDEX(0)) )
         {
            case SE_TYPE_NUMBER:
            /* for whatever reason, need a specific number
             * format
             */
            System.out.printlm(se.seGetNumber(SE.ARGS,
                                              SE.INDEX(0)));
            break;

            default:
            /* Oh what the heck, just let ScriptEase convert
             * whatever it is to a String.
             */
            System.out.println(se.seGetString(SE.ARGS,
                                              SE.INDEX(0)));
            break;
         }
         /* let's return something because we can */
         se.sePutString(SE.RETURN,SE.VALUE,
                        "Go away, you bother me kid.");
      }
   };
}
```

Before looking into the wrapper function tables, a basic overview of a wrapper function is necessary. The example above is simple but it demonstrates all that a wrapper function does. It gets its arguments, uses them to perform the body of the wrapper function, and returns a result.

THE FUNCTION HEADER

There are two methods used to define wrapper functions in Java, the inner class method and the reflection method. The above example uses the inner class method. The `print` method in the example creates a new inner class that implements the `SEWrapper` interface. The `SEWrapper` interface defines a single public method called `wrapperFunction`, which is where the code for your wrapper function is placed. The `wrapperFunction` method takes two parameters, the scripting context and the number of arguments passed to the wrapper function.

The reflection method is a little more straight-forward. You simply define a public method which has the same signature as the `wraperFunction` method of the `SEWrapper` interface. For example, we could have defined the above print wrapper function like so:

```
public void print(SEContext se, int argc)
{
   …
}
```

Wrapper functions defined using the inner class method as typically faster than those defined using the reflection method, but they also tend to take up more memory. Regardless of how it is defined, a wrapper function returns a `void` result because it uses the ScriptEase API functions to indicate its return value, which will of course be some ScriptEase value. Therefore, the Java return value is not used for a wrapper function.

Wrapper functions are usually called during an `seEval` ScriptEase API function call. `seEval` evaluates a script, and if that script invokes any of your functions that are implemented via a wrapper function, that wrapper function will be called back by ScriptEase. Wrapper functions receives two parameters. The first is the `SEContext` that is doing the callback. A wrapper function can be added to several contexts and it needs to know which one is doing the callback. You should use this provided context in any calls to ScriptEase API functions inside your wrapper function. You can compare the returned reference against any you might have to determine which context is being called back, but it is frowned upon. It is better to store any needed data along with each context using the `SE.SERVICES` object and retrieve it in your wrapper function. The second argument is simply a numeric count of the number of ScriptEase parameters passed to your function. ScriptEase wrapper functions can take varying number of arguments depending on how you define them as we will see later. If your wrapper function takes a fixed number of arguments, you can ignore this parameter.

THE ARGUMENTS

Now that you know how many arguments you have, how do you retrieve them? You use any of the ScriptEase API's retrieval functions such as `seGetNumber`, `seGetString`, and so forth. These functions will automatically convert the value to the correct type if it is not already. You can use `seGetType` to check the type first if you wish to be more stringent. The object/member pair to use for your arguments is `SE.ARGS`, `SE.NUM(x)` where `x` is the argument number. You can also use `SE.INDEX(x)`, as for arguments it is synonymous. Arguments range from `0` to one less than the number of arguments. `SE.ARGS,SE.NUM(0)` is the first argument.

THE RETURN

You return a value by using the standard ScriptEase API calls `sePutNumber`, `sePutString`, and so forth. The object/member pair to put to is `SE.RETURN,SE.VALUE`. See Chapter V for a thorough discussion of using `SE.RETURN`. For the simple case, you just put a value to `SE.RETURN,SE.VALUE`. For instance, your wrapper function could return the number 10 via:

```
se.sePutNumber(SE.RETURN,SE.VALUE, 10);
```

If you return nothing, by default the undefined value is returned. For constructor functions, the pre-constructed object is returned in this case.

WRAPPER TABLES

The basic idea behind a wrapper table is that it is a list of wrapper functions to be made available to your scripts. However, it has additional capabilities as well. You can define entire object classes using the table.

Here is a sample wrapper table that includes many of the options that can be used.

```
SELibraryTable[] my_lib =
{
   SE.NUMLITERAL( "Identification", “1.03”,
                  SE.DONTENUM ),
   SE.STRING( "IdentString", "Version 1.03b",
              SE.DONTENUM ),

   /* Move into "Clib" */
   SE.INOBJECT( "Clib",SE.DONTENUM ),

      /* SE.METHOD is a synonym for SE.FUNCTION */
      SE.FUNCTION( "MyFuncCall", MyFuncCallWrapper(),
                   0, 10, SE.SECURE|SE.BYREF, SE.READONLY ),

      SE.CLASS( "MyDate", MyDateWrapper(), 0,1,
                SE.INSECURE, SE.STOCK_ATTRIBS ),
         /* Stock attributes are DontDelete, ReadOnly,
          * and DontEnum
          */
         SE.PROTO,
            SE.FUNCTION( "valueOf", MyDateValueOfWrapper(),
                         0, 0, SE.SECURE, SE.STOCK_ATTRIBS ),
         SE.END_PROTO,
      SE.END_CLASS,

   SE.END_OBJECT,

   /* And why not some stuff in "SElib", note that 'INOBJECT'
    * is always relative to the global object
    */
   SE.INOBJECT( "SElib", SE.STOCK_ATTRIBS ),
      SE.FUNCTION( "Version", MyVersionWrapper(),
                   0,0, SE.SECURE, SE.STOCK_ATTRIBS ),
   SE.END_OBJECT,

   /* And demonstrate the last two functions. First we make
    * 'MyLib' a copy of 'Clib'. Then we change the attributes
    * on an existing variable.
    */
   SE.COPY( "MyLib", "Clib", SE.READONLY ),
   SE.ATTRIB( "varname", SE.READONLY | SE.DONTDELETE )
}
```

As you can see, the table is a list of elements where each element is one of a number of library table methods or objects such as `SE.FUNCTION` or `SE.CLASS`. We will list each individual method or object and what it does below.

First, some overview. Most methods define something and use a name as the first parameter. For instance, the first element defines a number:

```
SE.NUMLITERAL( "Identification", "1.03",
               SE.DONTENUM ),
```

When the table is added, these definitions in the table are created in the global object. Recall, members of the global object are the global variables of the script. This line therefore declares a new global variable named `Identification`. As the table is parsed, however, certain entries will cause this base object to change from the global object to some other object. This line is such an entry.

```
SE.INOBJECT( "Clib",       SE.DONTENUM ),
```

First, it too has a name `Clib`. It refers to the `Clib` member of the global object. However, the purpose of the line is to make that the new base. Therefore, new names declared after this line are no longer relative to the global object, but rather to the `Clib` member of the global object. A few lines later, the `SE.END_OBJECT` method undoes this, reverting back. This scheme allows a more readable table. It is a simple hierarchical scheme that matches well with the way objects and classes are defined.

There are two conveniences implemented. First, if you specify a name preceeded by `global.`, for instance `global.foo`, the `global.` means that `foo` is relative to the global variable and the base is ignored for this entry only. Second, you can use object notation such as `foo.goo`. For instance, instead of writing:

```
SE.INOBJECT( "Clib", SE.DONTENUM ),
SE.INTEGER(  "foo", 10, SE.DONTENUM ),
```

You could instead write:

```
SE.INTEGER( "Clib.foo", 10, SE.DONTENUM ),
```

When using this notation, the base for further statements is not changed. The `Clib.` part of it applies only to this particular definition. Also, if the object referred to, in this case `Clib`, does not already exist or is not an object, it is converted to an object.

Choose the notation in a particular instance that is clearer. If you are going to define more than one item in an object, it is clearer to move into that object using `SE.INOBJECT` while a single item can be clearer to write out a dot-separated name.

WRAPPER TABLE METHODS AND OBJECT

What follows is a description of each of the methods and objects you can use in a wrapper table and what each does.

SYNTAX:	`SE.NUMLITERAL(name,string,vflags)`
DESCRIPTION:	Create a variable in the current base object with the given name and the given value. The string passed must be parsable as a floating point number. The flags of the variable are set to the `vflags` value. The allowable flags are: `SE.DEFAULT` No special attributes `SE.READONLY` The member is read-only and cannot be modified. `SE.DONTENUM` The member should not be enumerated when a script uses `for..in`. `SE.DONTDELETE` The member cannot be deleted using the JavaScript `delete` operator.

SYNTAX:	`SE.INTEGER(name,number,vflags)`
DESCRIPTION:	Create a variable in the current base object with the given name and the given value. Identical to `SE.NUMLITERAL`, except an integer value is given.

SYNTAX:	`SE.STRING(name,string,vflags)`
DESCRIPTION:	Very similar to `SE.NUMLITERAL`, except the variable is set to a string value.

SYNTAX:	`SE.INOBJECT(name,vflags)`
DESCRIPTION:	The given name is treated as an object, and if the name is not currently an object, it is turned into one. The object has its flags set to the `vflags` value. Finally, that object is the new base for all names until an `SE.END_OBJECT` is found.

SYNTAX:	`SE.END_OBJECT`
DESCRIPTION:	Undoes the `SE.INOBJECT` above so all names are derived from the base before the `SE.INOBJECT` took effect.

SYNTAX:	`SE.FUNCTION(name,func,min_args,max_args,`
	`func_flags,var_flags)`

DESCRIPTION:	Declares a wrapper function. The parameters are the function's name, the function itself (a wrapper function), the minimum and maximum number of arguments, the function flags, and the variable flags.

The overloaded second parameter (the wrapper function) differs depending on how the wrapper function is actually defined. If the wrapper function was defined using the inner class method, then the second parameter will be the class that implements the `SEWrapper` interface. If the wrapper function was defined using the reflection method, the second parameter is a String representing the name of the method.

Using the example print wrapper function from above, this is how we would add the function to the library table:

```
/* Inner class method */
SE.FUNCTION("myPrint", print(), 1, 1, SE.DEFAULT),

/* Reflection method */
SE.FUNCTION("myPrint", "print",1,1,SE.DEFAULT),
```

The maximum number of arguments can be `-1` to specify no limit.

The function flags are one or more from the following:

`SE.DEFAULT` No special flags.

`SE.DYNAUNDEF` The object's dynamic callbacks are only called if the object does not already have the member in its internal storage. See Chapter VIII for a complete description of callbacks and this flag.

`SE.BYREF` Parameters passed to this function are passed by reference, so that any changes to them are reflected in the variables passed as the parameters.

`SE.SECURE` The function is secure. Only mark a wrapper function as secure if it can not perform any dangerous task. When in doubt, do not make it secure. The general rule is that any access to the system, such as reading a file or calling a system function, makes a function insecure.

`SE.KEEP_GLOBAL` Normally when a function is executed, the global object in effect when the function was created is used as the global object when the function is executed. With this flag, the current global object is retained whenever the function is executed.

SYNTAX:
```
SE.METHOD(name,func,min_args,max_args,
          func_flags,var_flags)
```

DESCRIPTION: This is a synonym for `SE.FUNCTION`.

SYNTAX:
```
SE.CLASS(name,func,min_args,max_args,
         func_flags,var_flags)
```

DESCRIPTION: This works similarly to `SE.FUNCTION` in that it adds the given entry as a function. However, as a class, such a function is expected to be used as a constructor. Several additional items are

	therefore created to facilitate this. First, the function is given a `prototype` which has the attributes `SE.STOCK_ATTRIBS`. Second, the prototype is given an `_class` member with a name equal to the name of the class. Finally, the prototype also gets a `constructor` member which points back to the class. All of these items are standard for ECMA classes. After this table entry is finished, the base is moved to the class object so you can add members or use `SE.PROTO` to add prototype members. This works in the same way `SE.INOBJECT` works. Use `SE.END_CLASS` to move back out of the object.

SYNTAX:	`SE_END_CLASS`
DESCRIPTION:	Changes the base to its value before the `SE.CLASS` entry.

SYNTAX:	`SE.PROTO`
DESCRIPTION:	Changes the base to the `prototype` of the current object. This is used to define the methods available to members of the current class. It is identical to: `SE.INOBJECT("prototype")`

SYNTAX:	`SE.END_PROTO`
DESCRIPTION:	Changes the base to its value before the `SE.PROTO` entry.

SYNTAX:	`SE.COPY(name,source(String),var_flags)`
DESCRIPTION:	Acts as an assignment, copying the source value to the given name. It sets the destination flags as well.

SYNTAX:	`SE.ATTRIB(name,var_flags)`
DESCRIPTION:	Sets the variable flags on a given name, changing nothing else about it.

SELIBRARY INTERFACE

A class needs to implement the `SELibrary` interface if any of its methods are to be used as wrapper functions. The object that implements this interface (and thus implements the actual wrapper functions) is passed to the `seAddLibTable` API call along with the corresponding library table. In addition to implementing the wrapper functions, this object could also be used to store any library specific data.

The `SELibrary` interface declares the following two methods:

```
public SELibrary seLibraryInitFunc(SEContext se);
public void seLibraryTermFunc(SEContext se);
```

The `seLibraryInitFunc` method is called whenever the corresponding library table is initialized. This method returns an object which contains any library specific data structure that the library may need access to. For example, the library may need to keep track of files opened. This method may create and return a new object, but you will most likely want to simply return `this`.

Note that the library can be initialized more than once, and you must be prepared to handle that case. The first time the library is initialized is when you call `seAddLibTable`, but the library will be reinitialized in certain circumstances.

The `seLibraryTermFunc` is used is called when the library is terminated. This function should be used to clean up any library specific resources. For each call to the initialization function, there will be one call to the termination function.

THE SELIBRARYMANAGER CLASS

The SELibraryManager class (located in the COM.Nombas.jse.libraries package) is used to make the ScriptEase standard library functions available to your scripts. The SELibraryManager class has one static method, addStandardLibraries, which has the following signature:

public static final void addStandardLibraries(SEContext se, int libs);

The first parameter is the context you want to add the library functions to. The second parameter specifies which libraries to add to the context. This parameter can be any of the following values |'d together:

`SELibraryManager.ECMA_OBJECTS`

Makes the following ECMA objects available to your scripts: Object, Function, Array, String, Boolean, Number, Exception.

`SELibraryManager.ECMA_MISC`

Some miscellaneous ECMA functions: eval, parseInt, parseFloat, escape, unescape, isNaN, isFinite.

`SELibraryManager.ECMA_DATE`

The ECMA Date object.

`SELibraryManager.ECMA_MATH`

The ECMA Math object.

`SELibraryManager.ECMA_REGEXP`

The ECMA RegExp object.

`SELibraryManager.ECMA_ALL`

Makes all of the ECMA objects and functions available to your scripts.

`SELibraryManager.SE_TEST`

The Nombas Test object, used to test and debug scripts.

`SELibraryManager.LANG_MISC`

Miscellaneous language extension functions: defined, undefined, getArrayLength, setArrayLength

`SELibraryManager.LANG_CONVERT`

ECMA conversion functions: ToPrimitive, ToBoolean, ToNumber, ToInteger, ToInt32, ToUint32, ToInt16, ToObject, ToString

For example, if you wanted to make the ECMA Date and Math objects available to your scripts, you would invoke the library manager like this:

```
SELibraryManager.addStandardLibraries(se,
SELibraryManager.ECMA_DATE |
SELibraryManager.ECMA_MATH);
```

The SEToLocaleHandler interface

When using the `SELibraryManager` to add the ECMA_DATE library to your context, you can specify a `SEToLocaleHandler` object for that library. When the `toLocaleString` method of the Date object is invoked from your script, the library will check for the presence of a `SEToLocaleHandler` object. If you have specified one, that object's `toLocale` method will be used to determine the locale string.

The `SEToLocaleHandler` interface defines a single method:

```
public boolean toLocale(SEContext se,
                        StringBuffer buffer,
                        char type,
                        double milli_since_1970);
```

This method is expected to be able to take the milliseconds since Jan 1, 1970 and convert it to locale time. The `type` parameter determines whether the locale time should be determined for just the date ('d'), just the time ('t'), or both ('b'). The converted locale time should then be placed into the provided `buffer`. If this method returns false, then the contents of the buffer will be ignored and the result of the `toLocaleString` function will be "bad date".

Once you have an object that implements the SEToLocaleHanlder interface, you can specify it when you add the Date library using the SELibraryManager:

```
SELibraryManager.addStandardLibraries(se,
SELibraryManager.ECMA_DATE, myToLocaleHandler);
```

or:

```
SELibraryManager.addStandardLibraries(se,
SELibraryManager.ECMA_ALL, myToLocaleHandler);
```

If you specify a `SEToLocaleHandler` object when adding one of the other libraries (like the ECMA Math library), the `SEToLocaleHandler` will be ignored.

Lifetimes

A primary ScriptEase consideration is the lifetime of the return from certain API calls. These calls include `seGetObject`, `seMakeObject` and `seMakeStack`. If you read the API manual section, they will all mention that they follow the standard ScriptEase lifetime model. That is what we describe here.

The major point to understand is that the underlying item, we will use an object as an example, always has its lifetime determined by the ScriptEase system. The return value is always a handle to the underlying object, not the underlying object itself. You control the lifetime of the handle only. No ScriptEase API call allows you to destroy an object. You can force a new object to be created via the `seMakeObject` API call but that new object is part of the ScriptEase system and therefore is destroyed when the ScriptEase system sees fit to destroy it.

So what do you get when you call, for instance, `seMakeObject` and retrieve a handle to that object? You get a lock on the object, so that the ScriptEase system will not delete the object until you are done using it. During the time you have the lock, you may use the returned object in any other ScriptEase API call. Once your lock goes away, you are no longer using the object and may not use it in ScriptEase API calls. However, it is quite likely that the object is still in use somewhere on the system which is why the object will not necessarily go away. You should think of getting a lock on an object as a license to use it only. Once you stop using it, it is out of your control and what happens to it is no longer your concern. Don't worry about it, the ScriptEase system will keep the object around as long as it is needed and get rid of it when it is not.

That leads us to the lifetime of your handle. Continuing with the example of an object lock returned via `seMakeObject`, how long can you use that object? The answer is simple; you can use it until the callback you got the lock in returns. Typically, the callback is a wrapper function. The lifetime operates exactly like a Java local variable to your wrapper function. Here is a valid wrapper function:

```
public void foo(SEContext se,int argc)
{
   SEObject myobj = se.seMakeObject();
   se.sePutObject(SE.RETURN,SE.VALUE, myobj);
}
```

The function of this wrapper should be obvious, it creates and returns a new blank object. However, the following wrapper function is invalid:

```
static SEObject myobj = null;

public foo(SEContext se,int argc)
{
   if( myobj==null )
   {
      myobj = se.seMakeObject();
   }

   se.sePutObject(SE.RETURN,SE.VALUE, myobj);
}
```

The intent is clear, to create a new object and return it then keep returning that same object for any further calls to the function. But, as we know, the handle returned by `seMakeObject` is only valid until the end of the wrapper function, so when the wrapper function returns the first time, `myobj` is no longer valid. Trying to use it in later invocations will use it after it has become invalid and obviously not work. We will see how to make this example work as intended shortly.

The careful reader will be wondering what happens if you get a lock when not inside a wrapper function. Locks retrieved in a wrapper function are local variables, locked retrieved not in a wrapper function are global variables. The lock is permanent and lasts for the life of the program. It lasts until the `SEContext` is destroyed and all variables and objects in it are freed.

While you may want to use the lock for your entire program, often you want to manipulate an object for a few lines of code, then you are done with it. Eventually, when everyone else is done using the object, you'd like it to be freed. If you keep the object locked, it can never be freed, and it will continue to use up memory until your program is done. For this reason, you may use the `seFreeXXX` API routines, such as `seFreeObject. seFreeObject` simply tells ScriptEase that you are done using the object at that point and that your lock is to be freed then. Remember, this does not destroy the object. It only tells ScriptEase that you are no longer using the object. As was emphasized before, ScriptEase will actually destroy the object some time in the future when it determines it is safe to do so.

An issue related to object lifetimes was brought up in the second wrapper function described above. How do we keep an object handle past the wrapper function it was created in? The answer is that you use the `seLockXXX` API calls, in this case `seLockObject`. This call indicates that the given object handle is to be valid for the life of the program. Once you pass an `SEObject` to this routine, it will be treated exactly like a handle returned outside of any wrapper function, it lasts until the end of the program unless you explicitly remove it using `seFreeObject`. If the handle already is a global handle, `seLockObject` has no additional effect.

So, the second wrapper function written correctly is:

```
static SEObject myobj = null;

public foo(SEContext se,int argc)
{
   if( myobj==null )
   {
      myobj = se.seMakeObject();
      se.seLockObject(myobj);

      /* make the lock permanent
       * so we can keep using it
       * in every call to this
       * wrapper function.
       */
   }

   se.sePutObject(SE.RETURN,SE.VALUE, myobj);
}
```

Summarizing the basic ScriptEase lifetime rules: Objects returned by the API are handles that allow you to use the given item. The handle, if given to you in a wrapper function, lasts until that wrapper function returns. If given to you outside a wrapper function, the handle is permanent. As long as the handle is valid, you may refer to the item, and the item will not be destroyed. Once you release the handle, you cannot say what will happen to the item, but you shouldn't worry about it as the ScriptEase system will properly take care of it.

You will notice the `seCloneXXX` API calls, such as `seCloneObject`. These calls take a lock, which follows the rules just described, and makes a second lock identical to the first. If the first lock was to be destroyed when the wrapper returns, the second does as well. Once created, you have two independent locks with different references although they both refer to the same ScriptEase object. They both follow the lifetime rules given above independently. For instance, you might clone a regular local lock and pass it to a utility routine. You continue to use the original lock until it goes away at the end of the wrapper function. The utility routine may call `seLockObject` on the cloned lock and use it for a while. That is perfectly valid, both locks are independent.

Objects and Classes

One of the most important tasks for a ScriptEase application writer is to design and implement object classes for the application's scripts to use. Most applications will have underlying data and functions that the script should be able to access in object form. This chapter will start with a discussion of object classes and finish with details on implementing those classes using ScriptEase.

OBJECT CLASSES

An object class starts with a constructor function. A constructor function's job is to initialize a new object of the object class. It can be written in Java using wrapper functions or implemented in JavaScript. When the user wishes to create a new object of your class, he calls your constructor function using the new operator, such as:

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

In this case, the constructor function is `MyClass`. Your constructor function will have a new blank object of its class provided to it as its `this` variable. The function can then add members to the `this` variable as appropriate for the task your object class is designed to perform. Here is a simple circle class constructor written in JavaScript:

```
function circle(radius)
{
   this.radius = radius;
}
```

With this constructor in your script, you can create a new circle object in JavaScript such as `new circle(10)`. Although this example has implemented the circle constructor in JavaScript, you could also implement the circle constructor using the ScriptEase API, as we will demonstrate later in this chapter.

The particular parameters you pass to your constructor as well as how you set up your new object is determined by the object's intended use. The main point to remember is that the `this` variable passed to your constructor is already a blank object of the constructor's object class. All objects of a single class share common members via their `prototype`. This sharing is set up for your premade `this` variable passed to your constructor.

You designate the methods to share by putting them in the `prototype` member of the constructor function. All object's of the constructor's class have access to those members. Here is a simple script that uses a slightly extended version of the `circle` constructor:

```
function circle(radius)
{
   this.radius = radius;
}

function circle.prototype.toString()
{
   return "circle of radius " + this.radius;
}
```

```
var a = new circle(5);
Clib.puts(a.toString());
```

This is a simple program that will print `circle of radius 5`. Although we've implemented this in script form, you can do the same using the ScriptEase API. Here is the version that does so:

```
public void circle(SEContext se,int argc)
{
   assert( argc==1 );
   se.seAssign(SE.THIS,SE.MEM("radius"), SE.ARGS,SE.NUM(0));
}

public void circleToString(SEContext se,int argc)
{
   String str = "circle of radius " +
                (int)se.seGetNumber(SE.THIS,SE.MEM("radius"));

   se.sePutString(SE.RETURN,SE.VALUE, str);
}

SELibraryTable[] circleTable =
{
   SE.CLASS( "circle", "circle", 1, 1, SE.SECURE, SE.DEFAULT),
      SE.PROTO
         SE.METHOD( "toString", "circleToString", 0, 0,
                    SE_SECURE, SE.DEFAULT),
      SE.END_PROTO,
   SE.END_CLASS
}

se.seAddLibTable(circleTable,this);
```

Notice that the circle function in both the JavaScript and ScriptEase API versions returns no value. It instead initializes the provided `this` variable. Constructors can return a value which will override the default preconstructed `this` variable. However, doing so requires you to do all of the initialization for the object you intend to return yourself.

Please refer to a JavaScript language book for more information on objects and object classes.

DYNAMIC OBJECTS

We've seen how to make class objects constructors, and prototype functions. However, it is often desirable to produce objects that are more flexible than a standard object. For instance, you may want to map the object to a real entity in your application and have changes to it immediately reflected. You might want to map an object to your display screen such that when a user writes:

```
displayObj.background = BLUE;
```

Your screen changes to the color blue. You do this using dynamic objects. While dynamic objects are most often used to make flexible class members, any object can be dynamic not just members of a class.

Very often you will want to associate one or more Java Objects directly with your object, so that when your wrapper function retrieve that object that can also retrieve the Java Object. Using seGetPointer() and sePutPointer() along with either SE.HIDDEN_MEM or SE.HIDDEN_UNIMEM or with an seInternalizeStringHidden property, is an excellent way to keep the data on your Java side safe from the script code and always associated with the proper objects.

SciptEase provides a related group of callback interfaces you can implement. The object which implements the interfaces is then associated with your object using the `seSetCallbacks` API function. Normally, you do this in your constructor when initializing an object of your class. These are interfaces for all the object manipulation tasks such as getting a member, putting a value to a member, deleting a member, etc.

Your callbacks will override the normal behavior for the object. To implement the above example, you would override the `put` behavior of the `displayObj` object. Your code would check for your special property `background` and changing the screen color to match the color being put to that member. You can override only some of the behaviors by implementing only the interfaces you are interested in.

Here is a list of the callback interfaces. When implementing any of these interfaces for your own object, remember that `SE.THIS` refers to the object being manipulated.

interface: SEGetCallback

```
public boolean get(SEContext se, int prop, boolean call_hint);
```

The `get` callback is used when a member of the object's value is being accessed. It is also used when trying to determine if an object has a property if you have not implemented `SEHasPropCallback` (see below). Implementing `SEHasPropCallback` is the preferred method.

The `prop` parameter, a parameter to most of these dynamic callback functions, indicates which member of the `SE.THIS` object is to be accessed. Normally, you use `seInternalizeString` at the beginning of your program to internalize your special properties, then you can compare them with the property being accessed using a single integer comparison. The alternative is to turn `prop` into a String using `seGetInternalString` then compare with a `String.equals`, but this is a lot of work and must be done on each get operation.

`call_hint` is a boolean indicating if ScriptEase believes the returned value is going to be used as a function to call. This would be the difference between:

```
a = yourobj.foo; /* call_hint==false */
```

and

```
a = yourobj.foo(); /* call_hint==true */
```

Knowing this information is useful in certain dynamic objects in which a property and a method require different setup routines, such as COM.

Once you've decided what value the dynamic property should have, you return it using the usual `SE.RETURN` object and return `true` from the function. If you've decided the property is not one you are interested in, return `false`. ScriptEase will act just as if the dynamic callback did not exist in this case, looking up the property in its internal storage for the object.

Note that you can access the internal storage of the object within your dynamic callback implementation. You should use the `Direct` versions of the `seGetXXX` and `sePutXXX` API calls in order to bypass your dynamic properties. If you use the non-`Direct` versions, the internal storage will be used for your object, but only for gets. This is because a particular callback for an object is shut off inside that callback, to prevent infinite recursion. However, only that one callback is shut off. If you use the object in a way that uses another callback, ScriptEase will use that callback. On rare occasions, you want that behavior. Most of the time, however, the implementation of a dynamic callback will want to directly access the members of its object. It is usually much clearer and quicker to just use the `Direct` versions of all ScriptEase API calls while implementing a dynamic callback.

interface: SEPutCallback

```
   public boolean put(SEContext se, int prop);
```

This callback is used whenever any of the object's members is being put to. Like the `get` callback, the parameter `prop` indicates the property being updated. The value being put to that property is the first (and only) parameter to the callback, `SE.ARGS,SE.NUM(0)`. Also like `get`, you return `true` if you've handled the put operation and `false` if you still want ScriptEase to update its internal storage for the object in the regular way.

Like the other dynamic callbacks, the `put` callback for your object is turned off while inside of it. However, it is usually better to make this behavior irrelevent by always using the `Direct` versions of the ScriptEase API calls inside a dynamic callback.

interface: SEHasPropCallback

```
   public int hasProp(SEContext se,int prop);
```

This callback is used when searching for a variable. ScriptEase maintains an internal list of objects to search when resolving a variable reference. This list is called the scope chain, and is described fully in the Execution chapter. This list usually consists of the Activation Object, where local variables are stored, followed by the Global Object. It is easy to add objects to the list by using the `with` statement. If your object is on this list, this callback will be used to determine if your object contains any property being searched for.

The return value can one of the following:

```
 SE.HP_HAS
```

The object has the property.

```
 SE.HP_HASNOT
```

The object does not have the property.

```
 SE.HP_CHECK
```

Disregard this callback and check in the normal way. The normal way involves calling your dynamic `get` callback if you have one.

```
SE.HP_DIRECTCHECK
```

Disregard this callback and check for the property in the object's internal storage only, do not call the `get` callback.

interface: SECanPutCallback

```
public boolean canPut(SEContext se,int prop);
```

Before trying to put a value, `canPut` will be called to determine if it is to be allowed. You determine whether or not an property can be updated with this callback. Return `true` to allow the put. It is most useful if you are not implementing a `put` callback, because in that case you can merge the functionality of this callback into the `put` callback by not doing any update.

interface: SEDeletePropCallback

```
public boolean deleteProp(SEContext se,int prop);
```

When a property of an object is to be deleted, this callback will be invoked. As usual, return `false` if you want ScriptEase to delete the property from its internal storage. This routine will also be called when the object itself is to be deleted, a destructor. In this case, the `prop` parameter will be –1.

interface: SEDefaultValueCallback

```
public void defaultValue(SEContext se,int hint);
```

When an object is used in a situation when it has to be converted to a primitive value (i.e. a string or number), this callback is used to do so. The only parameter is `hint`, the type that the system needs the value as. It is permissible to always convert to a single primitive type, which will then itself be converted to the correct value, if you don't want to take the hint into account. Return the value in the `SE.RETURN` object.

interface: SEOperatorOverloadCallback

```
public Boolean operatorOverload(SEContext se,short op);
```

ScriptEase implements operator overloading. Whenever an object is used as the left-hand operand, this callback is invoked. The `op` parameter will be the operator being overloaded, according to this table:

SE.OP_PREINC	++expr
SE.OP_POSTING	expr++
SE.OP_PREDEC	--expr
SE.OP_POSTDEC	expr--
SE.OP_ASSIGN	lhs = expr
SE.OP_NOT	!expr

SE.OP_UNARY_PLUS	+expr
SE.OP_UNARY_MINUS	-expr
SE.OP_BITNOT	~expr
SE.OP_EQUAL	expr==expr
SE.OP_NOTEQUAL	expr!=expr
SE.OP_STRICT_EQUAL	expr===expr
SE.OP_STRING_NOTEQUAL	expr!==expr
SE.OP_LESS	expr<expr
SE.OP_LESS_EQUAL	expr<=expr
SE.OP_GREATER	expr>expr
SE.OP_GREATER_EQUAL	expr>=expr
SE.OP_SUBTRACT	expr-expr
SE.OP_ADD	expr+expr
SE.OP_MULTIPLY	expr*expr
SE.OP_DIVIDE	expr/expr
SE.OP_MOD	expr%expr
SE.OP_SHIFTLEFT	expr<<expr
SE.OP_SHIFTRIGHT	expr>>expr
SE.OP_USHIFTRIGHT	expr>>>expr
SE.OP_OR	expr|expr
SE.OP_XOR	expr^expr
SE.OP_AND	expr&expr

The assign operators, such as `*=`, are performed as two separate operations, as if written `expr = expr * expr` instead of `expr *= expr`.

The right-hand side of the operator is to be found in `SE.ARGS,SE.NUM(0)`. The result of the operation should be returned in the `SE.RETURN` object with a return from the function of `true`. A return of `false` will do the normal operation which will involve converting the object to a primitive type compatible with the other operand and doing the JavaScript operation.

Note that the operator overload will be called with the op `SE.OP_ASSIGN` if the object is assigned to. Normally, this operation is ignored since you cannot assign to an object directly. In a script, you can write:

```
some_obj = 10;
```

but this just discards the object in the given variable and replaces it with 10. If the object has operator overloading, this will call the overload callback instead. If you return `false`, the normal changing of `some_obj's` value takes place. If you return `true`, it does not. Be careful, you can make a variable whose value the user can never change in this way.

inteface: SEGetByIndexCallback

```
   public boolean getByIndex(SEContext se,int index);
```

This callback is used to get an object member's value by index. This will be used solely by the ScriptEase API when a programmer is trying to iterate the members of your dynamic object. There is no hint as there is no way to know how the programmer intends to use the retrieved value. Return `false` if you have no such indexed member.

In order to implement this routine correctly, you need to internally order the members of your dynamic object in a consistent way. A person will be using this routine to iterate all of your members, from 0 on up. You must return each member once only and always in the same index. It is only permissible to reorganize the members if a member is added or removed. Return the member in the `SE.RETURN` object.

interface: SEGetNameByIndexCallback

```
 public int getNameByIndex(SEContext se, int index);
```

A companion routine to `getByIndex`, this is used when a script wants to iterate through your object using the `for..in` statement. You must return the names of your object's members according to their index. Like `getByIndex` above, it is only permissible to reorder your object if a member is added or removed. Return `-1` to indicate an index beyond the number of members in your object. Otherwise return the internalized version of your member's name (see `seInternalizeString`). The internalized string will be freed when you return it just as if you called `seFreeInternalString`. This is useful in the majority of cases in which you create the name to return and no longer need it locked. If you do need to retain a lock on the returned string, use `seCloneInternalString` to make a duplicate to return.

interface: SEMaxIndexCallback

```
   public int getMaxIndex(SEContext se);
```

Return the maximum index of the members of your objects which is equal to the number of members minus one.

For all of the above callbacks, the `SE.DYNA_UNDEF` flag will cause your dynamic property to be called only if the object does not contain the property in its internal storage. This is useful for speed. When your dynamic put callback is invoked on a property, if that property is not special, you can return `false` to put it into the object's internal storage. From then on, that property will be treated normally. The properties you are interested in you do not store in the object, you handle them in your callback. They will continue to be routed through your callbacks each time they are accessed.

FUNCTION REDIRECTION

Normally, an object is either a function or it is not. If it is a function, it can be invoked or used as a constructor such as by `Func();` or `new Func();`. Each of these behaviors can be overridden. Two special members can be added to an object to override, `_call` and `_construct`. They are used in the two instances above, `_call` when invoked as a regular function and `_construct` when used in a constructor with the `new` operator. These special members must themselves be functions that are called in the appropriate case. It is possible to turn a regular non-function object into a callable function by giving it an `_call` member and a constructable object likewise by giving it an `_construct` member.

API Function List

Here are the API functions organized by functionality. For all the API functions, if the call has an output parameter, a one-dimensional array which has its first element filled in by the function, you can always pass `null` if you don't care about that particular output. If you don't pass `null`, you must make sure the array has a length of at least 1, or an exception will be thrown.

Almost of the API functions are instance methods of the SEContext class. The few that aren't (`seInitialize`, `seTerminate`, `seCreateContext`, and `seCreateBlankContext`) are static methods of the SE class.

INITIALIZATION/CONTEXT CREATION

seInitialize

SYNTAX:	`int SE.seInitialize(void);`
WHERE:	None
RETURN:	The ScriptEase engine's version identifier.
DESCRIPTION:	`seInitialize` is used to initialize the engine, once per application. See `seTerminate` for the termination. Call this at the start of your program, before doing any other ScriptEase calls. All ScriptEase applications, even multithreaded ones, should call this routine only once.
SEE:	seTerminate

seTerminate

SYNTAX:	`void SE.seTerminate(void);`
WHERE:	None
RETURN:	None
DESCRIPTION:	Call `seTerminate` once before your application exits to cleanup ScriptEase. You must destroy any `secontext`s created before calling this routine.
SEE:	seInitialize

seCreateContext

SYNTAX:

```
    SEContext
SE.seCreateContext(SEContextParams params,
                         String userkey);
```

WHERE:

`params` is an object which implements the SEContextParams interface.

`userkey` the userkey, if applicable

RETURN:	An `secontext`, suitable to be used in further ScriptEase API calls.
DESCRIPTION:	Create a scripting context. Chapter IV is devoted to initializing and creating new contexts, see it for full details. The userkey is provided to you via email when you download an evaluation version of ScriptEase. Purchased versions ignore this parameter, for which you can pass null.
SEE:	seDestroyContext

seCreateBlankContext

SYNTAX:

```
   SEContext
SE.seCreateBlankContext(
   SEContextParams params,
   String userkey);
```

WHERE:	`params` is an object which implements the SEContextParams interface. `userkey` the userkey, if applicable
RETURN:	An `secontext`, suitable to be used in further ScriptEase API calls.
DESCRIPTION:	This routine creates a scripting context without making a call to the `SEPrepareContextFunc` method of your `SEContextParams` object. The userkey is provided to you via email when you download an evaluation version of ScriptEase. Purchased versions ignore this parameter, for which you can pass null.
SEE:	seCreateContext, seDestroyContext

seCreateFiber

SYNTAX:

```
   SEContext
SEContext.seCreateFiber();
```

WHERE:	None
RETURN:	An `SEContext`, suitable to be used in further ScriptEase API calls.
DESCRIPTION:	Create a fiber context. Chapter XIII is devoted to the topic of multithreading and fiber contexts. See it for full details.
SEE:	seCreateContext, seDestroyContext

seGetContextParams

SYNTAX:

```
   SEContextParams
SEContext.seGetContextParams();
```

WHERE:	`se` the context to get the parameters from.
RETURN:	None
DESCRIPTION:	Get a pointer to the context's parameter structure. The reference to the object which implements the `SEContextParams`

	interface used to create the context is stored with the context. You can get this object and examine or modify it. If you use the same object for multiple contexts, any change will affect all contexts.
SEE:	seCreateContext

seDestroyContext

SYNTAX:	`void SEContext.seDestroyContext();`
WHERE:	None
RETURN:	None
DESCRIPTION:	When you are done with a context, you destroy it to free up all associated resources, such as memory allocated.
SEE:	seCreateContext, seCreateBlankContext, seCreateFiber

seAddLibTable

SYNTAX:

```
    boolean
SEContext.seAddLibTable(
                SELibraryTableEntry[] table,
                SELibrary libdata);
```

WHERE:	`table` the table of functions to add `libdata` the SELibrary object which implements your wrapper functions
RETURN:	A boolean indicating success. Failure can happen on an illegal table or if memory becomes exhausted.
DESCRIPTION:	This routine parses a library table and adds the variables, functions, and classes defined in it to your context. You need to add the tables to your context only once, right after you create it. From then on, all scripts run in the context will have access to the things you've defined in it. You do need to add the table to each context. You should consider consolidating all of your `seAddLibTable` calls into an `sePrepareContextFunc` callback (see Chapter IV) so all contexts created in your application have access to the functions.
SEE:	None

seGarbageCollect

SYNTAX:	`void SEContext.seGarbageCollect(int action);`
WHERE:	`action` the garbage collection action to perform
RETURN:	None
DESCRIPTION:	This routine allows you to manipulate and invoke garbage collection on a given context. `action` can be one of the following:

`SE.GARBAGE_COLLECT`

Perform a garbage collection immediately, even if it has been disabled.

`SE.GARBAGE_OFF`

Disable garbage collection. Instead of collecting to free up unused memory, more memory is always allocated from the system whenever existing storage is exhausted.

`SE.GARBAGE_ON`

Re-enable garbage collection.

Note that each `SE.GARBAGE_OFF` must be paired with one `SE.GARBAGE_ON`. If `SE.GARBAGE_OFF` has been invoked several times, garbage collection will not be restarted until `SE.GARBAGE_ON` has been invoked the same number of times. However, a garbage collection can always be forced using the `SE.GARBAGE_COLLECT` action.

SEE: None

VARIABLE LOCATING

seVarParse

SYNTAX:

```
   boolean
SEContext.seVarParse(SEObject startObject
                     String string,
                     SEObject[] object,
                     int[] member,
                     int flags);
```

WHERE: `startObject` to object to start seaching from

`string` the string name of the variable

`object` an output parameter which has its first element filled in with the object

`member` an output parameter which has its first element filled in with the member

`flags` flags to control variable resolution.

RETURN: A boolean indicating if the variable was found.

DESCRIPTION: Turn a text variable name into the correspond Object,Member pair. Both output-only parameters, `object` and `member`, follow the usual ScriptEase lifetime rules for the returned value types. See `seGetObject` and `seInternalizeString` for more information on the `object` and `member` respectively.

This routine parses a variable name and returns the given variable's location. This allows you to read and update the variable's value. The variable name must be constant. For instance, `foo[5].goo` is acceptable but `foo[goo].goo` is not because in this case `[goo]` would mean to access the variable `goo` as a string and use that member name. Similarly, `goo(5).zoo` is not allowed because it calls a function. The reasoning is that this routine is used to access variables by name, but it should be quick. If you need to use full-fledged expressions, you should use the seEval routine instead although that is much slower.

The primary return is the output parameters which are filled in with an Object/Member pair you use to access or update that variable. You would later pass the `object` and `SE.STR(member)` to any of the other API functions to manipulate that variable. These return values follow standard rules for Lifetimes, and so may need extra code for cleaning up if this is not called in a wrapper function. You may often choose SE.COMPOUND_MEM or SE.COMPOUND_UNIMEM along with standard seGet and sePut calls for simple access to complex representations of a single variable

The flags may be any combination of the follows, `|`'ed togethor:

`SE.DEFAULT`

`SE_GF_COMPOUND_CREATE` - objects will be created if they don't yet exist. see VARIABLE READING for more information on this flag.

SEE: `seGetObject`, `seInternalizeString`, `seInternalizeStringHidden`

seGetName

SYNTAX:

```
   String
SEContext.seGetName(SEObject object,
                    SEMemberDesc member);
```

WHERE: `object` the object the variable is in

`member` the member description (ie. `SE.VALUE`, `SE.NUM(5)`)

RETURN: The text of the variable's name

DESCRIPTION: Given an Object,Member pair, get a name for the variable This function gets the full name of the given variable. It is intended for error reporting. Be warned, this is a very slow function.

SEE: None

seInternalizeString

seInternalizeStringHidden

SYNTAX:

```
   int
SEContext.seInternalizeString(String string)

   Int
SEContext.seInternalizeStringHidden(String
                                    string)
```

WHERE: `string` the text of the member name to internalize

RETURN:	The internalized string handle
DESCRIPTION:	All object member names are internalized by the ScriptEase engine before use. This API call is used to get the internalized version of a particular string. It is useful for commonly-used strings as whenever you use the text of the string, ScriptEase must internalize that string. In addition, ScriptEase internal strings can be directly compared using a single == comparison rather than the much slower `String.equals`. The resulting handle can be used as a member name using the `SE.STR()` member description specifier. In addition, object callbacks return internal handles for the member name being accessed. ScriptEase internal strings are always locked until explicitly freed. Use `seFreeInternalString` to indicate you are finished with a particular internal handle. You can also duplicate an string handle using `seCloneInternalString`. Refer to the standard ScriptEase lifetime model, as internal strings follow that with the exception that no internal strings are freed automatically when a wrapper function exits. The difference between seInternalizeString() and seInternalizeStringHidden() is that seInternalizeStringHidden() will create a property name that is not accesible from scripts (similar to SE.HIDDEN_MEM or SE.HIDDEN_UNIMEM).
SEE:	seCloneInternalString, seFreeInternalString, seGetInternalString

seCloneInternalString

SYNTAX:	`int SEContext.seCloneInternalString(int str);`
WHERE:	`str` the internal string to clone
RETURN:	A duplicate of the internal string
DESCRIPTION:	This is a standard clone function as described in Chapter VIII. The returned string handle acts as an exact duplicate of the original string handle.
SEE:	seInternalizeString, seInternalizeStringHidden, seFreeInternalString, seGetInternalString

seFreeInternalString

SYNTAX:	`void SEContext.seFreeInternalString(int str);`
WHERE:	`str` the internal string to free
RETURN:	None
DESCRIPTION:	This is a standard free function as described in Chapter VIII. Once called, the string handle is freed up and is no longer valid.
SEE:	seInternalizeString, seInternalizeStringHidden,

seCloneInternalString, seGetInternalString

seGetInternalString

SYNTAX:
```
   String
SEContext.seGetInternalString(int str);
```

WHERE: `str` the string handle to get the text of

RETURN: The text of the string

DESCRIPTION: Retrieves the text of a member described by an internal string handle.

SEE: seInternalizeString, seInternalizeStringHidden, seCloneInternalString, seFreeInternalString

VARIABLE READING

seGetBoolEx

seGetNumberEx

seGetPointerEx

seGetObjectEx

seGetStringEx

SYNTAX:
```
   boolean
SEContext.seGetBoolEx(SEObject object,
                      SEMemberDesc member,
                      int fl);

   double
SEContext.seGetNumberEx(SEObject object,
                        SEMemberDesc member,
                        int fl);

   Object
SEContext.seGetPointerEx(SEObject object,
                         SEMemberDesc member,
                         int fl);

   SEObject
SEContext.seGetObjectEx(SEObject object,
                        SEMemberDesc member,
                        int fl);

   String
SEContext.seGetStringEx(SEObject object,
                        SEMemberDesc member,
                        int fl);
```

WHERE: `object` the object half of an Object,Member pair

`member` the member half of an Object,Member pair

`fl` flags determining how the variable is retrieved

RETURN: The Java value for the variable.

DESCRIPTION: These routines are a core element of the ScriptEase API. Given an Object,Member pair, these routines extract the current value as the given type, converting if necessary, and return the result. Note that the underlying variable does not change type, its value is retrieved and converted without changing the source variable. A valid return will always result from these functions. If an internal error occurs, like an illegal conversion, that error will be set up as the result of your function (see `seThrow`), but a valid result is still returned. The intent is that you can write a simple wrapper with no error checking that uses these routines. See the section SE.RETURN EXPLAINED in Chapter V for a discussion of the implications of this behavior. The value returned if an error occurs will always be a stock value. For numbers, it is `SE.NAN` (or `0` for non-floating point numbers). For strings, an empty string, `""`, is returned. For objects, `SE.NOWHERE` is returned. Finally, for booleans `false` is returned.

The flags parameter can be any of the following `|`'d togethor:

`SE.DEFAULT`

`SE.GF_DIRECT`

`SE.DEFAULT` is the default. `SE.GF_DIRECT` means to ignore the object's prototype and dynamic methods when looking for the property. It directly accesses the object's internal structure. It is intended for writing faster dynamic routines. See Chapter IX for more information on using this flag.

In addition, you can specify the flags by using different named functions that have the flags as part of their name. In this case, you do not specify the flags, they are implicit. Taking `seGetNumberEx` as an example:

```
seGetNumber(...) = seGetNumberEx(...,SE_DEFAULT)
```

```
seGetDirectNumber(...) =
seGetNumberEx(...,SE_GF_DIRECT)
```

The return from seGetObjectEx follows the usual ScriptEase lifetime rules described in Chapter VIII.

SEE: None

seFreeObject

SYNTAX:
```
    void
SEContext.seFreeObject(SEObject item);
```

WHERE: `item` the item to free

RETURN: None

DESCRIPTION: You use this function to explicitly free an object returned from seGetObjectEx. Once freed, the object or string is no longer valid. See Chapter VIII for the standard ScriptEase lifetime rules.

SEE: SeGetObjectEx

seCloneObject

SYNTAX:
```
    SEObject
SEContext.seCloneObject(SEObject item);
```

WHERE: `item` the item to clone

RETURN: The cloned object

DESCRIPTION: These calls produce a duplicate of the given `SEObject`. The duplicate and the original handle refer to the same item but are independent. For instance, freeing one of the handles means that handle can no longer be used, but the other handle is still valid until it to is freed.

SEE: SeGetObjectEx

seWeakLockObject

SYNTAX:
```
    void
SEContext.seWeakLockObject(SEObject obj,
                           boolean weak);
```

WHERE: `obj` the object to weak lock

`weak` should the object be weak locked

RETURN: None

DESCRIPTION: Lock an item to produce a weak lock. The object is always locked, so it must eventually be explicitly freed, just as if you use `jseLockObject`. The boolean determines if the lock is weak. `jseLockObject` always produces normal locks. Therefore, this routines is usually used to produce weak locks, so the parameter is `true`. On occasion, you may need to turn off an existing weak lock and restore it to a full lock which is when the parameter may be `false`.

This API function is designed to resolve a common problem. It is typical when mapping a Java object to a JavaScript object for a programmer to want each item to have a reference to the other. This allows both sides to have access to its sibling to perform any needed task. The problem that arises is that the lock on the ScriptEase object by the API keeps the object permanently in memory, even when ScriptEase is no longer using the object. It is a cyclic loop that cannot be detected because the cycle extends outside of the ScriptEase core.

Using seWeakLockObject, the programmer retains a handle to the object but that handle does not lock the object in memory. If ScriptEase is no longer using the object, the presence of this lock does not keep it from being garbage collected. Other than that difference, this function performs exactly like seLockObject.

Be careful with this function. The object can be cleaned up at any time once the script is no longer using it. If you try to use a handle to such an object, you will probably crash the system. You should make sure to add a destructor to the object so that you know when you must stop using the handle, and free the handle using seFreeObject at that time. Any use for this function other than this intended one is likely to crash your application.

SEE: seLockObject, seFreeObject

seLockObject

SYNTAX:

```
   void
SEContext.seLockObject(SEObject obj);
```

WHERE: `obj` the object to lock

RETURN: None

DESCRIPTION: Lock an object. The object will remain locked until it is explicitly freed. `seLockObject` always produces normal locks.

SEE: seFreeObject, `seWeakLockObject`

seGetType

SYNTAX:

```
   int
SEContext.seGetType(SEObject object,
                    SEMemberDesc member);
```

WHERE: `object` the Object half of an Object,Member pair

`member` the Member half of an Object,Member pair

RETURN: The type of the member.

DESCRIPTION: This is a simple function, it returns the current type of the given variable. Members that do not exist are reported as type `SE.TYPE_UNDEFINED`. Members can also exist with type `SE.TYPE_UNDEFINED`. Use the `seExists` api call to differentiate them.

SEE: seExists

seExists

SYNTAX:

```
   boolean
SEContext.seExists(SEObject object,
                   SEMemberDesc member);
```

WHERE: `object` the Object half of an Object,Member pair

`member` the Member half of an Object,Member pair

RETURN: `true` if the member exists, else `false`.

DESCRIPTION: This API function is used in place of `seGetType` when you do not care what the type of the member is, only if it exists or not.

SEE: seGetType

seExistsDirect

SYNTAX:

```
Boolean
SEContext.seExistsDirect(SEObject object,
                         SEMemberDesc memer);
```

WHERE:

`object` the Object half of an Object,Member pair

`member` the Member half of an Object,Member pair

RETURN:

`true` if the member exists, else `false`.

DESCRIPTION:

This API function is used in place of `seGetType` when you do not care what the type of the member is, only if it exists or not. Unlike `seExists`, this function ignores any dynamic properties on the object and only checks if the internal ScriptEase storage for the object contains the given member. Like all the `Direct` functions, it is meant to be used in implementing object dynamic functions so that the dynamic functions themselves can use the internal ScriptEase store without having to take into account the possibility of being called by each other.

SEE:

seGetType, seExists

seGetAttribs

SYNTAX:

```
   int
SEContext.seGetAttribs(SEObject object,
                       SEMemberDesc member);
```

WHERE:

`object` the Object half of an Object,Member pair

`member` the Member half of an Object,Member pair

RETURN:

The attributes of the member

DESCRIPTION:

This function gets a member's attributes. The attributes a member has can be one or more of the following flags, `|`'d together:

`SE.DEFAULT`

No special attributes

`SE.DONTENUM`

The member is not enumerated when the `for..in` statement is used on this object.

`SE.DONTDELETE`

Using the `delete` operator on this member is ignored.

`SE.READONLY`

Attempts to write to the member are ignored.

`SE.IMPLICIT_THIS`

Only objects can have this attribute, a member that is not an object will never have it. When the object is called as a function, the `this` variable is added to the function's scope chain. See Chapter VI, SCOPING for more information.

`SE.IMPLICIT_THIS`

Only objects can have this attribute, a member that is not an object will never have it. When the object is called as a function, the `this` variable's parents are added to the function's scope chain. See Chapter VI, SCOPING for more information.

`SE.DYNA_UNDEF`

Only objects can have this attribute, a member that is not an object will never have it. Dynamic callbacks of the object are only invoked if the object does not have the member being worked on in its internal store. See Chapter IX for more information.

SEE: seSetAttribs

seCompare

SYNTAX:

```
    boolean
SEContext.seCompare(SEObject obj1,
                          SEMemberDesc mem1,
                          SEObject obj2,
                          SEMemberDesc mem2,
                          int[] result);
```

WHERE:

`obj1` the object half of the Object,Member pair for the first operand.

`mem1` the member half of the Object,Member pair for the first operand.

`obj2` the object half of the Object,Member pair for the second operand.

`mem2` the member half of the Object,Member pair for the second operand.

`result` an output parameter which has its first element set to -1, 0, or 1 to indicate that the first member is less than, equal to, or greater than the second member.

RETURN: `true` if the two members are equal

DESCRIPTION: This function uses the ECMAScript rules to determine the relationship between the two variables. Since ECMAScript only defines a less-than relationship, this routine internally has to compare twice to give all the possibilities. For that reason, you can narrow the information you are interested in by specifying one of the following special values as the `result` parameter. In that case, only the boolean return of the function is used as described in each case:

`SE.COMP_EQUAL`

Determine if the two members are equal

`SE.COMP_LESS`

Determine only if the first member is less than the second.

SEE: None

OBJECT ACCESS ROUTINES

seObjectMemberCount

SYNTAX:
```
    int
SEContext.seObjectMemberCount(SEObject object);
```

WHERE: `object` the object to query

RETURN: The number of members the object has.

DESCRIPTION: This call returns the number of members an object has. The usual use is to iterate through all the members, using `SE.INDEX()` from 0 to one less than the result of this call.

SEE: seObjectMemberName

seObjectMemberName

SYNTAX:
```
    String
SEContext.seObjectMemberName(SEObject object,
                             SEMemberDesc mem);
```

WHERE: `object` the object the member is in

`mem` which member to get the name of

RETURN: The member name.

DESCRIPTION: This function is used to get the name of a member. It is most useful when enumerating the members of an object using the `SE.INDEX()` member access macro. If there is no such member then a blank string is returned and `SE.RETURN`,`SE.ERROR` will have been set.

Unlike `seGetName`, only the members name is returned, not a fully-qualified variable name.

SEE: seGetStringEx, seGetName, seObjectMemberCount

seIsFunc

SYNTAX:
```
    boolean
SEContext.seIsFunc(SEObject object,
                   SEMemberDesc member,
                   boolean[] script);
```

WHERE: `object` the object half of an Object,Member pair.

`member` the member half of an Object,Member pair.

`script` an output parameter, `true` if the function is a script function as opposed to a wrapper function.

RETURN: The boolean `true` if the object is a function.

DESCRIPTION: All functions in ScriptEase are objects, but not all objects are functions. The API call lets you determine if an object is in fact a function. If it is, the output boolean `script` will be `true` if the function is a script function, `false` if it is a wrapper function.

SEE: seIsArray

seIsArray

SYNTAX:

```
   boolean
SEContext.seIsArray(SEObject object,
                    SEMemberDesc member,
                    int[] length);
```

WHERE: `object` the object half of an Object,Member pair.

`member` the member half of an Object,Member pair.

`length` an output parameter filled in with one more than the highest numbered element, `0` if no numbered element.

RETURN: The boolean `true` if the object is an ECMA `Array`. Note that objects that are not true `Array`s can still have numbered elements. Thus, the `length` parameter will be filled in for all objects, though it will usually be `0`.

DESCRIPTION: Determine if an object is an ECMA `Array`.

SEE: seIsFunction, seSetArray

seSetArray

SYNTAX:

```
   boolean
SEContext.seSetArray(SEObject object,
                     int length);
```

WHERE: `object` the object to adjust

`length` one more than the new max element

RETURN: This function returns `true` if the object was adjusted, `false` if it was already an `Array` and no elements were eliminated.

DESCRIPTION: This call turns an object into an ECMA `Array` and adjust its elements. Any numbered element greater than or equal to the `length` is deleted. In addition, the object is permanently marked as an ECMA `Array` object. This means that its `length` element corresponds to the numbered elements and adjusting either adjusts the other just as for an `Array`.

If you want to turn an object into an `Array` without altering any elements, use `SE.MAX_INDEX` as the `length` parameters.

SEE: seIsArray

seShareReadObject

SYNTAX:

```
   boolean
SEContext.seShareReadObject(SEObject object);
```

WHERE: `object` the object to make shared read

RETURN: This function returns `true` if the object was successfully shared for reading, `false` if it could not be.

DESCRIPTION: Calling this routine makes the given object, and all its children, read-only and shareable. The object handle no longer follows the usual ScriptEase lifetime rules but rather is valid until the ScriptEase engine is cleaned up using the `seTerminate` call.

All shared objects exist for the life of the program. The `SEObject` handle passed to the routine can be used from then on in any context.

When using this routine, you must remember that any object and all its children are shared. This means the object's base class, which it refers to via its `_prototype` is also shared. If you have a complicated object hierarchy, you may end up sharing a large number of objects. All these objects will persist until the program terminates the ScriptEase engine. Also remember that shared objects and their children become read-only to all contexts, including the one that originally shared them.

Objects can only be made sharable when there is a single context existing. You must mark all objects you wish to share before making additional contexts. This routine will fail once two or more contexts exist.

SEE:

VARIABLE WRITING

sePutBoolEx

sePutNumberEx

sePutPointerEx

sePutObjectEx

sePutStringEx

sePutUndefinedEx

sePutNullEx

SYNTAX:

```
   boolean
SEContext.sePutBoolEx(SEObject obj,
                      SEMemberDesc mem,
                      int fl,
                      boolean val);

   boolean
SEContext.sePutNumberEx(SEObject obj,
                        SEMemberDesc mem,
                        int fl,
                        double val);

   boolean
SEContext.sePutPointerEx(SEObject obj,
                         SEMemberDesc mem,
                         int fl,
                         Object val);

   boolean
```

```
SEContext.sePutObjectEx(SEObject obj,
                        SEMemberDesc mem,
                        int fl,
                        SEObject val);

   boolean
SEContext.sePutStringEx(SEObject obj,
                        SEMemberDesc mem,
                        int fl,
                        String val);

   boolean
SEContext.sePutStringEx(SEObject obj,
                        SEMemberDesc mem,
                        int fl,
                        String val,
                        int len);

   boolean
SEConext.sePutUndefinedEx(SEObject obj,
                          SEMemberDesc mem,
                          int fl);

   boolean
SEContext.sePutNullEx(SEObject obj,
                      SEMemberDesc mem,
                      int fl);
```

WHERE:

`object` the object half of an Object,Member pair

`mem` the member half of an Object,Member pair

`fl` flags determining how the variable is stored

`val` the value to put, type based on which routine you are using

`len` for the overloaded version of sePutString, the length in characters of the String to be put. If `len` is less than the actual length of the String, only `len` characters will be put.

RETURN:

The boolean `true` if the member was created, `false` if it already existed.

DESCRIPTION:

These functions are the inverse of the `seGetXXX` versions, they put a value into the given Object,Member location. Like their get counterparts, these functions have versions that make the flags implicit in their name. However, there is one additional flag, `SE.GF_MUST`. `SE.GF_MUST` means that the value should ignore the `SE.READONLY` attribute. This eases updating internal members in your objects that should be read-only for the script but not for you. It is equivelent to turning off the read-only bit, putting the value, then turning it back on. It is most-often used in combination with `SE.GF_DIRECT`. Here are the name/flag equivelents using sePutNumberEx as an example:

```
sePutNumber(...) =
sePutNumberEx(...,SE.GF_DEFAULT)

sePutDirectNumber(...) =
sePutNumberEx(...,SE.GF_DIRECT)

seMustPutNumber(...) =
sePutNumberEx(...,SE.GF_MUST)

seMustPutDirectNumber(...) =
sePutNumberEx(...,SE.GF_MUST|SE.GF_DIRECT)
```

SEE: None

seDelete

SYNTAX:

```
    boolean
SEContext.seDelete(SEObject obj,
                   SEMemberDesc mem);
```

WHERE: `obj` the object half of the Object,Member pair

`mem` the member half of the Object,Member pair

RETURN: The boolean `true` if the member was deleted or did not exist, `false` if it could not be deleted such as trying to delete a virtual object's member.

DESCRIPTION: This call deletes a member of an object. `seDelete` is not affected by the `SE.DONTDELETE` flag, only the `delete` operator is affected. If you would like to respect the flag, use `seGetAttribs` to check attributes before deleting a member.

SEE: seGetAttribs

seMakeObject

SYNTAX:

```
    SEObject
SEContext.seMakeObject();
```

WHERE: None

RETURN: A handle to the created object.

DESCRIPTION: This call creates a new object. The returned object handle follows the standard object lifetime rules described in chapter VIII. The returned object is blank, meaning it has no members. You'll usually want to store the object using `sePutObject`, either to assign it to a variable or return it as the result of your wrapper function.

SEE: seMakeStack, sePutObject

seMakeStack

SYNTAX:

```
    SEObject
SEContext.seMakeStack();
```

WHERE: None

RETURN: A handle to the created object (stack).

DESCRIPTION: This call creates a new stack. The returned object handle follows the standard object lifetime rules described in chapter VIII.

A stack is an object and can be used wherever an object can be used. However, you probably should not, as stacks are significantly slower to manipulate than objects. Stacks do have the benefit of guaranteeing that members will remain in the order they are created, so that SE.INDEX(0) is always the first member created, SE.INDEX(1) is the second, and so forth. Regular objects do not have this property. Stacks are used when needing to pass a list of items to the API, such as the parameters or the scope chain to `seEval`.

SEE: seMakeObject, seEval

sePutWrapper

SYNTAX:

```
   boolean
SEContext.sePutWrapper(
   SEObject obj,
   SEMemberDesc mem,
   SEFunctionDescription desc,
   Object data);
```

WHERE:

`obj` the object half of the Object,Member pair

`mem` the member half of the Object,Member pair

desc the SEFunctionDescription describing the wrapper function

`data` data associated with the function.

`name` the name of the function for error reporting wrapper_func - the function

`minArgs` the minimum arguments to the function

`maxArgs` the maximum arguments to the function

`funcFlags` the function flags

`varFlags` the variable-type flags

RETURN: A boolean, `true` if the put was successful.

DESCRIPTION: This call turns a variable into a wrapper function. See Chapter VII for complete details about wrapper functions.

The third parameter is an SEFunctionDescription object which describes the wrapper function. An SEFunctionDescription object can be created with the following constructor:

```
public SEFunctionDescription(
   String name,
   SEWrapper/String func,
   int minArgs,
   int maxArgs,
   byte funcFlags,
   short varFlags);
```

This parameters to this constructor correspond to the parameters to the `SE.FUNCTION()` library table entry. Since you specify the exact Object,Member location to put the new wrapper function in, the `name` parameter does not indicate where to put the function. It is used if an error message occurs related to the function.

SEE: None

seSetCallbacks

SYNTAX:

```
   void
SEContext.seSetCallbacks(SEObject obj,
                         SEMemberDesc mem,
                         Object cbs);
```

WHERE:

`obj` the object half of the Object,Member pair

`mem` the member half of the Object,Member pair

`cbs` an object that implements one or more of the callback interfaces

RETURN: None

DESCRIPTION: This routine sets the object callbacks for an object. If the variable is not an object, nothing is done. See Chapter IX for a complete discussion on dynamic objects and object callbacks.

SEE: seEnableDynamicMethod

seEnableDynamicMethod

SYNTAX:

```
   RestoreDynamicMethodState
SEContext.seEnableDynamicMethod(
   SEObject obj, SEMemberDesc mem,
   int whichCallback,
   boolean enable,
   RestoreDynamicMethodState restore_state);
```

WHERE:

`obj` the object half of the Object,Member pair

`mem` the member half of the Object,Member pair

`whichCallback` which method to enable/disable, this may be any of:

```
SE.GET_CALLBACK
SE.PUT_CALLBACK
SE.HASPROP_CALLBACK
SE.CANPUT_CALLBACK
SE.DELETEPROP_CALLBACK
SE.DEFAULTVALUE_CALLBACK
SE.OPERATOROVERLOAD_CALLBACK
SE.GETBYINDEX_CALLBACK
SE.GETNAMEBYINDEX_CALLBACK
SE.GETMAXINDEX_CALLBACK
SE.ALL_CALLBACK /* all of the above */
```

`enable` whether to enable dynamic method

`restore_sate` `null` for first call in pair, for second call set to value returned by the first call

RETURN: In first call in pair, this returns the value to be passed as `retore_state` for second call. For second call in pair the return value has no meaning.

DESCRIPTION: Enable (if enable is `true`) the calling of the dynamic method named methodName, else disable calling of that dynamic method. These methods are disabled during a callback of that method (i.e. `put` is disabled while within `put` to prevent recursion). This is a risky function and not a default part of the API. To enable this API function compile with `JSE_ENABLE_DYNAMETH`.

This function is always used in a pair. For the first call in the pair `restore_state` should be `null`. For the second call, which will undo the first, `restore_state` must be the value returned by the first call of the pair.

SEE: seSetCallbacks

seConvert

SYNTAX:

```
    void
SEContext.seConvert(SEObject obj,
                    SEMemberDesc mem,
                    int type);
```

WHERE: `obj` the object half of an Object,Member pair

`mem` the member half of an Object,Member pair

`type` the type to convert to.

RETURN: None

DESCRIPTION: This routine retrieves the value using a get from the Object,Member pair, converts it, and puts it back to the same location. The possible conversions are:

`SE.TOPRIMITIVE`

Convert to a primitive value. A primitive value is a non-object value.

`SE.TOBOOLEAN`

Converts to a boolean

`SE.TONUMBER`

Converts to a number

`SE.TOINTEGER`

Converts to an integer.

`SE.TOINT32`

Converts to a signed 32-bit integer

`SE.TOUINT32`

Converts to an unsigned 32-bit integer

`SE.TOUINT16`

Converts to an unsigned 16-bit integer

`SE.TOSTRING`

Converts to a string

`SE.TOOBJECT`

Converts to an object.

SEE: None

seSetAttribs

SYNTAX:

```
   void
SEContext.seSetAttribs(SEObject obj,
                       SEMemberDesc mem,
                       int attributes);
```

WHERE: `obj` the object half of an Object,Member pair

`mem` the member half of an Object,Member pair

`attributes` the attributes to set

RETURN: None

DESCRIPTION: This call sets the variable's attributes. The attributes can be any of the following, `|`'d togethor:

`SE.DEFAULT`

The default, no special attributes

`SE.READONLY`

Any attempt to update the member is ignored

`SE.DONTDELETE`

Attempts to delete the member using the ECMAScript `delete` operator are ignored

`SE.DONTENUM`

The member is not included in `for..in` enumerations

The following flags apply only if the variable is an object:

`SE.DYNA_UNDEF`

Dynamic callbacks for the objects are only used if the object does not contain the desired member in the ScriptEase internal storage for the object.

`SE.IMPLICIT_THIS`

When the function is executed, the `this` variable is added to the front of the scope chain so all members of `this` are visible without putting `this.` in front of them. This is exactly as if the entire body of the function was wrapped in the statement `with( this ) ...` This behavior is similar to how Java member functions work.

`SE.IMPLICIT_PARENTS`

Similar to `SE.IMPLICIT_THIS`, the parents of the `this` variable are put into the scope chain. This chain begins with `this.__parent__` and continues as long as each such object itself has a `__parent__` property. Note that there are two underscores on each side of parent above. This is useful for browsers in which event handlers can refer to variables in the enclosing element, document, and window. By chaining these objects together using `__parent__` and adding `SE.IMPLICIT_THIS` and `SE_IMPLICIT_PARENTS` to the event handler function, the desired behavior is achieved.

SEE: seGetAttribs()

seAssign

SYNTAX:

```
   boolean
SEContext.seAssign(SEObject destObj,
                   SEMemberDesc destMem,
                   SEObject srcObj,
                   SEMemberDesc srcMem);
```

WHERE:

`destObj` the desination object half of the Object,Member pair

`destMem` the desination member half of the Object,Member pair

`srcObj` the source object half of the Object,Member pair

`srcMem` the desination member half of the Object,Member pair

RETURN: A boolean, `true` if the destination member was created, `false` if it already existed.

DESCRIPTION: This function does a get on the source Object,Member followed by a put of that value to the destination Object,Member.

SEE: seMustAssign

seMustAssign

SYNTAX:

```
   boolean
SEContext.seMustAssign(SEObject destObj,
                       SEMemberDesc destMem,
                       SEObject srcObj,
                       SEMember srcMem);
```

WHERE:

`destObj` the desination object half of the Object,Member pair

`destMem` the desination member half of the Object,Member pair

`srcObj` the source object half of the Object,Member pair

`srcMem` the desination member half of the Object,Member pair

RETURN: A boolean, `true` if the destination member was created, `false` if it already existed.

DESCRIPTION: This function does a get on the source Object,Member followed by a put of that value to the destination Object,Member. This routine ignores the `SE.READONLY` flag on the destination if it has that flag.

SEE: seAssign

seThrow

SYNTAX:

```
void
SEContext.seThrow(String message);
```

WHERE: `se` the current context

`message` the message of the error message in standard ScriptEase format.

RETURN: None

DESCRIPTION: An error object is constructed and set up in the `SE.RETURN` object. In addition, the error flag is turned on. If the wrapper function calling `seThrow` returns after this call, it's result will be the given error.

The standard error message format for ScriptEase allows information on the type of the error to be included. By default, a stock `Exception` object is constructed with the text passed to this function. The extended form of the string is:

```
!TYPE NUM: MESSAGE
```

For instance, you could pass:

```
!SyntaxError 9999: You made a mistake.
```

The `TYPE` indicates the type of the error. An error object of this type is constructed to contain the error. The error types are:

`SyntaxError`

`ReferenceError`

`ConversionError`

`ArrayLengthError`

`TypeError`

`URIError`

`EvalError`

`RegExpError`

These types are defined by ECMA, including what errors are generated in normal error situations. When writing your own wrapper functions, pick the error type you feel is most appropriate.

The `NUM` indicates a resource number for your error message. Values of 10000 or more are reserved for user-errors, and you should use one. These numbers are used by the `seGetResourceFunc` to internationalize the text associated with various text strings, including error messages.

`MESSAGE` is the text of the error message.

SEE:	None

EXECUTING SCRIPTS

seEval

SYNTAX:

```
    boolean
SEContext.seEval(Object to_interpret,
                  int interp_type,
                  String text_args,
                  SEObject stack_args,
                  int flags,
                  SEEvalParams params);
```

WHERE:

`to_interpret` the script or function to execute

`interp_type` what the `to_interpret` parameter is

`text_args` arguments as a text string

`stack_args` arguments on a stack seobject

`flags` options on how to eval

`params` eval params

RETURN: A boolean, `true` if the evaluation was successful

DESCRIPTION: See "Using seEval" in the chapter "Script Execution Topics" for details on using the `seEval` ScriptEase API call.

SEE: seExec, seEnd

seExec

SYNTAX:

```
    boolean
SEContext.seExec();
```

WHERE: None

RETURN: `true` if there are more statements to execute, `false` when the `seEval` is completed.

DESCRIPTION: This routine executes one script statement from a script started with `seEval` using the `SE.START` option. When the eval is completed, the return value will be stored in the `SE.RETURN` object.

SEE: seEval, seEnd

seEnd

SYNTAX:

```
    void
SEContext.seEnd();
```

WHERE: None

RETURN: None

DESCRIPTION: This call aborts a script started with `seEval` using the `SE.START` option. This will immediately terminate the script which is being executed one statement at a time. There is no

	return value from an aborted script.
SEE:	seEval, seExec

sePrecompile

SYNTAX:

```
    byte[]
SEContext.sePrecompile(String to_interpret,
                       int interp_type,
                       SEEvalParams params);
```

WHERE: `to_interpret` the text of the script or the filename to compile

`interp_type` how to interpret `to_interpret`, either `SE.TEXT` or `SE.FILE`

`params` used to get the virtual file and line number only.

RETURN: The bytecodes

DESCRIPTION: This routine compiles a script into the corresponding bytecodes. The file to precompile is specified exactly like `seEval` and must be either `SE.TEXT` or `SE.FILE`. The given file is precompiled and the resulting bytecodes are returned. Usually, these bytecodes are then written to disk for use later.

The bytecodes can be passed as the item to evaluate in a later `seEval` call with `SE.PRECOMP` as the type. The bytecodes must be freed using `seFreeBytecodes`. Although the script is precompiled, it is not added to the context or run. The context will be unchanged as a result of this call.

You can provide the optional `params` object. Only the `xxxLineNum` and `xxxFileName` methods of the object are used.

SEE: seFreeBytecodes

seFreeBytecodes

SYNTAX:

```
    void
SEContext.seFreeBytecodes(byte[] codes);
```

WHERE: `codes` the bytecodes returned from `sePrecompile`

RETURN: None

DESCRIPTION: Call this routine to free the bytecodes given to you from sePrecompile after you are finished using them, such as after writing them to disk.

SEE: sePrecompile

seIsBreakpoint

SYNTAX:

```
    boolean
SEContext.seIsBreakpoint(String filename,
                         int lineNumber);
```

WHERE: `filename` the file interested in

`lineNumber` the line in that file

RETURN:	A boolean, `true` if the line is a valid breakpoint.
DESCRIPTION:	This function is provided for use by a debugger. It checks to see if it is possible for any function currently loaded to break at the given file and line. This check is intended to be called in response to a user request to set a breakpoint. Be warned that this call is very slow. The filename must match one of the currently loaded filenames (which can be found in the `SE.FILENAMES` object) or it cannot be a breakpoint and this function will return `false`. If you have a `seFindFileFunc` callback in your context (described in Chapter IV), the filenames that are used are the ones returned from this function, the translated filename, not the untranslated ones passed to the callback.
SEE:	None

Core Customization Topics

The ScriptEase core is highly customizable to suit the needs of a variety of scripted applications. This chapter described the customization options available. All of these options are determined by compile-time `#defines`. Once the options are set, the ScriptEase core must be re-preprocessed and recompiled to reflect these options. As a result, only customers who have purchased ScriptEase and thus have the source code to the ScriptEase core can benefit from customizing the core.

The jseopt.jh file found in the distribution contains the same documentation as is here. For each define, it starts either on or off (off being commented out). Read through this chapter and the jseopt.jh file, and change the state of any of the options you like. Below, each option is listed as either on or off, reflecting the default state.

CORE CUSTOMIZATION

The following options modify the internals of the ScriptEase core, and are used mostly to balance performance and memory use.

JSE_MULTIPLE_GLOBAL (on)

By default, ScriptEase remembers the global object in effect when each function is created, and runs each function under its original global object. This flag can be used to turn off this behavior.

JSE_ONE_STRING_TABLE (off, on if SE_SHARED_OBJECTS is defined)

Turning on this define makes a single string table be used, as opposed to each context having its own table. A single string table is compatible with a multithreaded application using more than one context.

JSE_INFREQUENT_COUNT (5000)

ScriptEase normally calls an application's continue function after each script statement is executed. `seEval` provides the option to call less frequently. This define indicates how many statements are to be processed between each call to the continue function.

JSE_GET_RESOURCE (off)

Normally, all resources (text error messages for instance) have their text stored with the application. If this option is on, rather than using the stored text, the application's seGetResourceFunc callback is used to retrieve the text for any resource.

JSE_SHORT_RESOURCE (off)

When this define is turned on, all resource strings retain only their identifier. If you provide an `seGetResourceFunc`, those identifiers will be turned into real strings using it, otherwise error messages and other resources will contain only the error number and no message.

JSE_TRAP_NOWHERE (off)

Normally, when an API function tries to get an object that doesn't exist, or can't be converted to an object, the engine returns `SE_NOWHERE`. This allows your application to use that object without error checking the result. However, you may instead wish to check all your results for errors and ensure that this object is never used. Turning on this define, which defaults to being off, causes an error if `SE_NOWHERE` is ever tried to be used.

JSE_INLINES (off)

In the ScriptEase core, a number of functions are expanded inline to improve speed. These functions, however, take up considerable code space and can actually harm performance in later versions of the JDK (1.3.1 and 1.4).

JSE_PEEPHOLE_OPTIMIZER (on)

The peephole optimizer is run on the bytecodes ScriptEase produces for each function, transforming certain inefficient common sequences into more efficient ones. It speeds up programs and shrinks the resulting bytecodes. The only disadvantage is an increase in compilation time, which may outweigh speed performance if scripts consist of a number of quick and tiny functions.

JSE_CACHE_GLOBAL_VARS (on)

When on, ScriptEase maintains a cache of recently-accessed global variables, speeding access to them in many cases. However, without certain transformations done by the peephole optimizer, the global variable cache can be too aggressive and return the wrong result at times. Therefore, you should only use the global cache if the peephole optimizer is turned on, as is the default.

JSE_GLOBAL_CACHE_SIZE (10)

By default, the number of global variables retained in the cache is 10. Increasing the size of the cache could increase cache hits, but the time to look through them may slow misses. Our internal testing indicates the value of 10 is about optimal. Global caching must be enabled for this define to be useful.

JSE_COMPACT_LIBFUNCS (off)

By default, this is on. ScriptEase stores wrapper functions in a minimal way, expanding them on first use. Since most applications include a large number of wrapper functions, such as the standard ECMA library, and scripts use only a fraction of them, this setting conserves a lot of memory. There doesn't seem to be a good reason to turn it off, but the option exists nonetheless.

JSE_REFCOUNT (off)

JSE_GC (on)

Normally, `JSE_GC` is on, meaning ScriptEase reclaims memory by garbage collecting. If `JSE_GC` is off and `JSE_REFCOUNT` is on, ScriptEase uses a reference counting scheme. This takes more memory and is slower, but it frees objects as soon as they become unused. It cannot detect cyclic loops. If both are on, reference counting is supplemented via garbage collection to find cyclic loops.

`JSE_GC` is noticeably quicker and less memory-hungry than is `JSE_REFCOUNT`. Garbage collection passes are quick as well, on the order of tenths of a milliseconds on a typical machine. Therefore, the main benefit of `JSE_REFCOUNT` is to find objects that have become freed as soon as they have done so. However, in most applications, you are better off leaving `JSE_GC` on and forcing a garbage collection (by calling `seGarbageCollect`) at any critical point that you need to ensure unused objects are freed.

By default, `JSE_GC` is on, and `JSE_REFCOUNT` is off. Note that either one or both of the following may be defined. However, at least one of the two must be defined. If both are turned off, `JSE_GC` will be selected automatically.

SE_OBJ_POOL_SIZE (1024, 128 if JSE_MIN_MEMORY is on)

ScriptEase maintains a pool of objects for its needs so that it doesn't need to allocate and free objects to the system as often. The bigger the pool, the less system allocation is required at the expense of more memory used by the pool. Also, the emptying of the pool triggers garbage collection, so the bigger the pool, the longer ScriptEase can go between garbage collections.

Because emptying a pool triggers garbage collection, reducing the pool sizes below the value for a `JSE_MIN_MEMORY` build will cause ScriptEase's execution speed to slow drastically due to constant collection while freeing up very little memory. It is advised that you treat 128 as the minimum for this define.

SE_MEM_POOL_SIZE (1024, 128 if JSE_MIN_MEMORY is on)

The members of an object are stored in a separate structure that works exactly like `SE_OBJ_POOL_SIZE`. Since each object requires one descriptor for its members, it is usually best to keep these two pool sizes identical.

JSE_PACK_OBJECTS (off)

Objects are packed to use the minimum memory. Turning this on saves significant memory, but on many systems brings an equally significant loss in performance when dealing with objects.

JSE_PER_OBJECT_CACHE (on, off if JSE_MIN_MEMORY is on)

Each object caches the last member in it that was accessed. In many programs, this improves performance. Turning it off reduces memory requirements for objects slightly at the cost of performance.

JSE_PER_OBJECT_MISS_CACHE (on, off if JSE_MIN_MEMORY is on)

Objects store the last member searched for that the object did not have. This speeds up searching searching for global variables in which a chain of objects is searched for a particular member, often missing each time for the first few objects in the chain. By default, non-min memory builds have a per-object miss cache.

SE_APIVARNAME_POOL_SIZE (5)

Varname structures are used by the API for internalized variable names. Like string locking structures, they are needed only when the user has an internal string locked. While many programs will lock a number of such names, they remain locked for the life of the program. Typically, only one or a few names are locked and then freed at a time.

SE_STACK_SIZE (2048, 512 if JSE_MIN_MEMORY is on)

ScriptEase uses an internal stack for resolving function calls and evaluating expressions. Each function call needs a few entries for overhead plus one entry per parameter passed to it and per local variable it allocates. The default size of this stack is 2048 entries, enough to recurse typical functions to a depth in the hundreds. 512 entries is allocated for a min-memory build.

SE_MAX_STACK_INFO_DEPTH (64)

The `SE_STACK_INFO` stock objects allow the API user to examine the call stack of a running script. This macro determines the maximum depth that can be examined. Each depth only requires one pointer (usually four bytes), so increasing the max depth isn't costly. However, it is unlikely any program need more depth.

FEATURE CUSTOMIZATION

The following options all turn on or off certain JavaScript features. These features are part of standard ECMAScript, but you may choose to disable certain features to conserve space for low-memory systems.

JSE_COMPILER (on)

The ScriptEase compiler is necessary to run script code. With the compiler turned off, your application will only be able to run precompiled scripts. The JavaScript `eval` function relies on the compiler being enabled. The compiler is on by default.

JSE_TOOLKIT_APPSOURCE (on)

This define determines whether the seGetSourceFunc is to be used by the compiler when it reads text. By default, it is on. This define is only applicable for the compiler.

JSE_SAVE_FUNCTION_TEXT (on, off if JSE_MIN_MEMORY is on or JSE_COMPILER is off)

ScriptEase needs to retain information about the source of script functions to allow the `toSource` call to turn those functions back into the source they came from. This option tells it to do so.

JSE_PROTOTYPES (on)

Allows the JavaScript prototype-related features. JavaScript is heavily dependent on prototypes, and many of its features are built on top of them. Turning this off is probably a bad idea.

JSE_ALWAYS_IMPLICIT_THIS (off)

With this flag set, the `this` variable is always searched as part of the scope chain. Normally, the user must set the `SE_IMPLICIT_THIS` flag for each function individually. Turning this compile-time option on will make that the default for all functions.

JSE_ALWAYS_IMPLICIT_PARENTS (off)

Similar to `JSE_ALWAYS_IMPLICIT_THIS`, this flag will turn on implicit parents for every function, causing the engine to search the `__parent__` member of the 'this' variable when resolving variable names.

JSE_FUNCTION_ARGUMENTS (on)

When calling a function, some older scripts use the `<function_name>.arguments` syntax rather than just using arguments. This behavior is rare, but by default we support it.

JSE_AUTO_OBJECT (off)

An older ScriptEase behavior, undefined variables are automatically converted to an object when used as an object.

JSE_REGEXP_LITERALS (on)

By default, JavaScript regular expression literals are allowed. They are a standard part of JavaScript. However, if you turn off the regular expression portion of the ECMA library, you should turn this off as well. The regular expression code is large, and is commonly left out of applications running in tight memory.

JSE_FUNCTION_LENGTHS (on)

Each function gets a .length property, the number of parameters it takes. Turning this off will conserve memory slightly. Because it is a JavaScript feature, this option is on by default.

JSE_HTML_COMMENT_STYLE (off)

If this option is turned on, HTML-style comments will be accepted (i.e. `<!-- ... -->`).

SE_ECMA_RETURNS (on)

If no value is explicitly `return`ed from the body of a script, the last expression evaluated is returned implicitly. Normally, each expression evaluated has its result preserved against the possibility it is the last expression evaluated and thus needs to be returned. If this flag is turned off, the last expression is determined by an alternate algorithm. While this algorithm is faster, some cases will confuse the algorithm and cause an improper value to be returned.

SCRIPTEASE FEATURE CUSTOMIZATION

The following options are similar to Feature Customization, but they modify ScriptEase-only features.

JSE_TOKENSRC (on)

Allows ScriptEase to produce tokens. You can use `sePrecompile` only if this define is on. The compiler portion of ScriptEase must also be turned on.

JSE_TOKENDST (on)

Allows ScriptEase to run precompiled scripts. This option is necessary to pass precompiled scripts to `seEval`.

JSE_OBJECTDATA (on)

This option reserves space in each object to store a userdata pointer. The `SE_OBJECT_DATA` member requires this definition. Turning this off will break most Nombas-supplied libraries.

JSE_DYNAMIC_CALLBACKS (on)

By default, objects can be given a callback table to implement their get, put, delete, and other operators using the `seSetCallbacks` API call. Turning off this option removes that capability.

JSE_OPERATOR_OVERLOADING (on)

Operator overloading is one of the object callbacks. It can be individually turned off.

JSE_ENABLE_DYNAMETH (off)

Object callbacks are normally not recursive, they are shut off while active. In other words, if you are implementing the dynamic get for an object and you try to get a member of that object, you won't be stuck in an infinite loop, rather that get will get the property from the internal ScriptEase store. It is possible that a particular application does want the recursive behavior. This option makes available the seEnableDynamicMethod API call, which allows a particular dynamic method to be turned back on.

JSE_GETFILENAMELIST (off)

If your application does not need to use the `SE_FILENAMES` object, you can turn it off and save the code and runtime space associated with storing these filenames.

JSE_BREAKPOINT_TEST (off)

If your application does not need to use the `seIsBreakpoint` call, you can get rid of it to save some space. The ScriptEase debugger requires this call to operate.

JSE_TASK_SCHEDULER (on)

ScriptEase supports the concept of fibers. Turning off this option will turn off the `seCreateFiber` API call as well as the `SE_YIELD` and `SE_SUSPEND` members of the `SE_RETURN` object.

JSE_INCLUDE (on)

Turns on the `#include` directive.

JSE_DEFINE (on)

Turns on the `#define` directive.

JSE_CONDITIONAL_COMPILE (on)

Turns on the `#if`, `#elif`, `#else`, `#ifdef`, and `#ifndef` directives.

JSE_SECUREJSE (on)

By default, the ScriptEase security model is on. If you are not intending to use security, you can turn it off to save space.

JSE_MAIN_ARGC_ARGV (on)

ScriptEase has the legacy option to treat a function named `main` as special, auto-calling it like C. The `SE_CALL_MAIN` option to `seEval` causes this to be done. By default, this is available. If you have no interest in this option, you can get rid of it completely and save some code space.

JSE_TOSOURCE (on)

ScriptEase provides a number of helper functions that ease turning an object class into its appropriate source code, to implement the `toSource` method. Many of the standard ECMA objects we provide rely on this code. Turn it off only if you are not using the ECMA objects, and do not need the helper routines for your own functions.

JSE_NAMED_PARAMS (on)

ScriptEase allows passing parameters by name, such as calling a function like `foo(a:10,b:"blah");`. Turn off this option to remove this capability.

JSE_TIMEZONE_GLOBAL (off)

Some implementations do not have knowledge of the local machines timezone. In this case, you can turn on this define, and fill in this global variable with the information:

```
 extern slong jse_minutes_from_gmtime;
```

JSE_TOLOCALEDATE_FUNCTION (off)

The optional `toLocaleDate` method of the the `Date` object is off by default.

JSE_MILLENIUM (off)

Several JavaScript date functions deal with 2-digit dates, assuming to be from the 20th century. Because such functions are not year-2000 compatible, they are turned off by default.

SE_SHARED_OBJECTS (on)

Activates the API function `seShareReadObject` which allows objects to be shared among threads.

JSE_PASSBYREF (on)

ScriptEase supports the passing of parameters to script functions by reference by using the `&` operator and to wrapper functions by using the `SE.BYREF` flag. This define turns the support on, which is the default.

JSE_FLOATING_POINT (on)

This value defines whether the ScriptEase interpreter and libraries will support floating-point operations. If this is not defined then the engine will only support integers, and any use of a floating-point number or operation will result in an exception (e.g. `pi=3.1415`, `Math.cos()`, `0.0`). An application for a small device may run much smaller and faster when `JSE_FLOATING_POINT` is not defined if floating-point math is not needed.

DEBUGGING CUSTOMIZATION

These options affect debugging and the level of internal checks ScriptEase performs.

NDEBUG

When `NDEBUG` is not defined (i.e. a debug build), ScriptEase does a significant amount of internal checking to check for any bugs, either errors in ScriptEase itself or errors in an application's use of the API. You should do all development without the `NDEBUG` flag, as you can find many bugs this way. When you are ready to release, turn `NDEBUG` back on for the fastest possible code.

JSE_TRACK_OBJECT_USE (off)

This is an internal routine that keeps track of how many times object members are accessed, and whether they are found in the cache or needed to be looked up. It is used primarily to optimize the object cache and detect bottlenecks in member lookup.

JSE_NEVER_FREE (off)

An extreme bug-detection setting used for self-debugging by the core. When on, the garbage collector never frees unused memory, though it does fill the memory with a particular byte value when no longer used. This will cause ScriptEase to use enormous memory, but is useful to isolate internal garbage-collection related bugs.

JSE_DONT_POOL (off)

Another garbage collection setting. ScriptEase normally maintains a pool of objects and reuses them whenever possible. In this mode, all memory is returned to the system when not in use and reallocated when needed. This is a very slow mode, again designed to shake out any internal bugs.

JSE_ALWAYS_COLLECT (off)

The garbage collector normally runs only when ScriptEase detects it is low on memory. If this is on, the garbage collector is run any time it could possibly be run, regardless of memory. This is an important setting to self-diagnose ScriptEase core bugs, but is very slow. 'very slow' means that - VERY slow.

Fibers and Threads

Each `SEContext` can be used by only a single thread at one time. If you want to run multiple scripts simultaneously in a multithreaded application, you need to create one `SEContext` using `seCreateContext` per thread. You can in fact create more than one `SEContext` per thread if you like.

Each context contains a copy of much of the same data. Namely, each context will initialize the standard function libraries into its global object in order to allow its scripts to see them. In addition, each context keeps a pool of various kinds of memory available in order to increase performance. As a result, each context has significant overhead of memory. Fibers exist to help alleviate this problem.

Fibers are like sibling contexts. Each fiber in the same group has access to the same variables, uses the same pools of memory, and so forth. Therefore the overhead described exists only once even when a large number of fibers exist in the same group. However, fibers are not a replacement for separate contexts in multiple threads. All fibers in the same group are considered one context, so they can only be used by a single thread and only one fiber can be active at once. You can use fibers to cooperatively multitask scripts but since only one fiber can be run at once, fibers do not take advantage of multiple processors. If your machine has multiple processors, and you would like to run multiple scripts taking advantage of all the processors, you must use full contexts not fibers.

Fibers are created using the ScriptEase `seCreateFiber` API call. You pass as a parameter an existing context. The new fiber is created as part of the same fiber group the existing context is part of. You create the first context using `seCreateContext` then create any number of fiber siblings using `seCreateFiber`. When you are done, you have a number of related contexts, each which can run its own script. However, all of the contexts share a single global object. You can change the global object in any of the fibers, but the intent of the fibers is to conserve memory so sharing the global object is the norm.

USING SE.START

As was mentioned, each fiber group has the limitation that only one of the fibers can be running at the same time. If you use `seEval` to evaluate a script in one fiber, you must wait for it to complete before evaluating another script in a different fiber. To get around this limitation, `seEval` has the `SE.START` flag option.

`SE.START` initializes an eval and then returns. Successive lines of the script are run using the ScriptEase `seExec` API call. Using this method, when you initialize each fiber you begin the script it is to run using `seEval` and `SE.START`, which then returns to you quickly. You do the same for each fiber. Now you can execute a single line of each script using `seExec`. Typically, you keep evaluating one line on each fiber in a round-robin fashion in this way. As each fiber completes its task, it is removed from the list of fibers to execute in this way. New fibers can be created and added into the list easily.

GLOBAL MANIPULATION

Although the intent of fibers is to conserve memory by sharing overhead, often each fiber should still be independent. For instance, you may not want them to share global variables. This is easy to accomplish. After you create the initial context and set up the libraries in it using `seCreateContext`, you preserve that global object. Then for each fiber (including the original context returned by `seCreateContext`), you give it a new global object with its `_prototype` pointing to the preserved global object. Thus, all new variables created in a fiber will be created in its private global object, yet it still can refer via the global's prototype to the original global object which contains all the standard function libraries.

Here is a short example ScriptEase API application that creates several fibers and runs them all.

```
public class FibersSample implements
             SEContextParams,  SEErrorHandler, SELibrary
{
   static final int MAX_FIBERS = 5;
   private int seOptions;

   /* ----------------------------------------------------------
    * SEContextParams interface
    * ----------------------------------------------------------*/
   public int seGetOptions()
   {
      return this.seOptions;
   }

   public void seSetOptions(int options)
   {
      this.seOptions = options;
   }

   /* ----------------------------------------------------------
    * Error handler
    * ----------------------------------------------------------*/

   public void sePrintErrorFunc(SEContext se, String text)
   {
      System.out.println("Error encountered: " + text);
   }

   /* ----------------------------------------------------------
    * SELibrary interface
    * ----------------------------------------------------------*/
   public SELibrary seLibraryInitFunc(SEContext se)
   {
      return this;
   }

   public void seLibraryTermFunc(SEContext se)
   {
   }

   /* A wrapper function to write out a string. It converts
    * whatever argument it is given to a string then writes it
    * to the terminal using 'printf'. The user would use it like
    * this:
    *
```

```
 *    StringOut("Hello, world!");
 */

/* ---------------------------------------------------------
 * Text output
 * ---------------------------------------------------------*/

public SEWrapper StringOut()
{
   return new SEWrapper()
   {
      public void wrapperFunction(SEContext se, int argc)
      {
         int i;
         String text;


         for( i=0;i<argc;i++ )
         {
            /* Get each successive argument and print them
             */
            text = se.seGetString(SE.ARGS,SE.NUM(i));
            System.out.println(text);
         }
      }
   };
}

SELibraryTableEntry[] SampleFunctionList =
{
   SE.FUNCTION( "StringOut",    StringOut(),      1, -1,
                SE.SECURE, SE.DONTENUM )
};


static void add_fiber(SEContext se,
                      SEContext[] table,
                      int[] number,
                      SEObject glob)
{
   table[(number[0])++] = se;

   /* give the fiber a private global */
   se.sePutObject(SE.GLOBAL,SE.VALUE,se.seMakeObject());

   /* but point back to shared so can see it */
   se.sePutObject(SE.GLOBAL,
                  SE.STOCK(JseStrID._prototype),
                  glob);

   se.seEval("var a = 10;\nStringOut(a);\n",SE.TEXT,
             null,null,SE.START,null);
}

static void remove_fiber(int num,
                         SEContext[] table,
                         int[] number)
{
   /* we are done with the context */
   table[num].seDestroyContext();

   /* remove it from the table */
   while( num<(number[0])-1 )
```

```
      {
         table[num] = table[num+1];
         num++;
      }
      (number[0])--;
   }

   public static final void main(String[] argv)
   {
      SEContext se;
      SEContext[] fibers = new SEContext[MAX_FIBERS];
      int[] fibers_used = {0};
      int fiber_current = 0;
      SEObject shared_global;
      FibersSample sample = new FibersSample();
      sample.seSetOptions(SE.DEFAULT);

      SE.seInitialize();

      /* initialize the main context */
      se = SE.seCreateContext(sample,null);
      if( se==null )
      {
         System.err.println("Invalid user key.");
         System.exit(0);
      }

      shared_global = se.seGetObject(SE.GLOBAL,SE.VALUE);

      /* add libaries so we have the StringOut function */
      se.seAddLibTable(sample.SampleFunctionList,sample);


      /* Add the original context to our fiber list. All contexts
       * including the parent will be treated identically
       */
      add_fiber(se,fibers,fibers_used,shared_global);


      /* Create some more fibers. All are added to one big
       * pool.
       */
      while( fibers_used[0]<MAX_FIBERS )
         add_fiber(fibers[0].seCreateFiber(),fibers,
                   fibers_used,shared_global);


      /* run the fibers until all have exited. For each fiber,
       * execute its next available statement using seExec().
       * Notice that an seEval using SE.START was started in
       * each fiber when it was added above. As each fiber
       * finishes its seEval(), we remove it from the fiber list.
       * We exit when all fibers are done.
       */
      while( fibers_used[0]>0 )
      {
         if( !fibers[fiber_current].seExec() )
         {
            remove_fiber(fiber_current,fibers,fibers_used);
            /* and continue using the fiber that fell into its
             * place
             */
         }
```

```
          else
          {
             fiber_current++;
          }
          if( fiber_current>=fibers_used[0] ) fiber_current = 0;
       }


       /* Done with the sample, shut everything down. */
       SE.seTerminate();
    }
}
```

YIELDING AND SUSPENDING

At times, you may want to control the behavior of your fiber execution more than the simple controls executing a single statement at a time provides. Two methods are provided for you to do so. Both methods are invoked by a wrapper function to affect the execution of the fiber the wrapper functions is within.

When a wrapper function is ready to return, it sets up its value in the `SE.RETURN` object. Two members of the object, `SE.YIELD` and `SE.SUSPEND`, can likewise be set. Both are boolean members and are set `true` to invoke their relevent behavior.

First is `SE.YIELD`. By yielding, the fiber ensures that the current `seExec` statement is immediately ended. Recall that the `SE.INFREQUENT_CONT` option to `seEval` means that several statements are executed for each call to `seExec`. If a wrapper function yields, the `seExec` returns immediately. The return value for the wrapper function is still treated normally. The next time the fiber is executed using `seExec`, execution resumes with the code that called the wrapper function getting that value as the return.

The second option is `SE.SUSPEND`. Suspending functions is similar to yielding in that the calling `seExec` finishes immediately. However, the fiber is put into a suspended state. This means that further calls to `seExec` will immediately return without executing any code of the fiber. It is the job of your application to determine when the fiber is ready to be restored and remove its suspended state. This is done by assigning `false` to the fiber's `SE.RETURN,SE.SUSPEND` member. After the suspend is removed, the application can also change the return value to be returned by the wrapper function before again executing any code. It does this by assigning the new value to the `SE.RETURN,SE.VALUE` member as normal. If it does not, the value returned by the wrapper function is used. This is useful if the value to be returned is unknown when the wrapper function suspended, perhaps that is why it needed to be suspended. Remember, though, that the `SE.RETURN,SE.VALUE` member is read-only as long as any of the boolean members is `true` including the `SE.SUSPEND` member. You must turn the suspension off before you are allowed to write a new return value. Of course, you must do it also before you call `seExec` on the fiber after it is unsuspended.

OTHER CONSIDERATIONS

It is important to understand that values returned from the ScriptEase API that follow the usual ScriptEase lifetime rules, such as `SEObject`s, are tied to the context they were created in. Any ScriptEase call that is passed that parameter must be passed the same context used to initially get that item. You cannot use an `SEObject` created in one context with another, for instance.

Fibers are the exception. Fibers are designed to allow several contexts to share the same variables. All fibers in the same fiber group can share these items and use them in any context of the same fiber group. Items created in this way still must not be used with a context that is not part of the fiber group, however.

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 others.

There are two formats for comments: single-line comments (end of line comments) and multi-line comments (block comments). Single-line comments may contain any character except a line terminator character ("\n").

- Single-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.
- Multi-line 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. Multi-line comments may not be nested within other multi-line comments, but single-line comments can exist within multi-line comments.

The following code fragments are examples of valid comments:

```
// this is a single-line comment

/* this is a multi-line 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 or tokens have special meaning for the interpreter and should not or may not be used as identifiers, neither as variable nor as function names. See identifiers to avoid for a list of words to shun as identifiers.

abstract	as	break	boolean	byte	case
catch	char	class	const	continue	debugger
default	delete	do	double	else	enum
export	extends	false	final	finally	float
for	function	goto	if	implements	import
in	instanceof	int	interface	is	long
namespace	native	new	null	package	private
protected	public	return	short	static	super
switch	synchronized	this	throw	throws	transient
true	try	typeof	use	var	void
volatile	while	with			

Identifiers to avoid

Safe programming suggests that the following words or tokens should not be used as identifiers, neither as variable nor as function names. See prohibited identifiers for a list of words to that simply should not be used as identifiers.

arguments	Array	Boolean	Date
decodeURI	decodeURIComponent	encodeURI	
Error	escape	eval	EvalError
Function	Infinity	isFinite	isNaN
Math	NaN	Number	Object
parseFloat	parseInt	RangeError	ReferenceError
RegExp	String	SyntaxError	TypeError
undefined	unescape	URIError	

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 floats

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

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

Scientific floats

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

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

Boolean type

Booleans may have only one of two possible values: `false` or `true`. Since 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

The value `null` is a special data type that indicates that a variable is empty, a condition that is different from being `undefined`. A `null` variable holds no value, though it might have previously. The `null` type is represented literally by the identifier, `null`. Since 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.

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

NaN

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

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

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

Number constants

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

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

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, global.parseInt() and global.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, Math.abs() is a method of the Math object. It may be accessed as follows:

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

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

The `Math.abs()` method is a static method, that is, it is used directly with the Math object instead of an instance of the object. Many methods are instance methods, that is, they are used with instances of an object instead of the object itself. The String substring() method is an instance method of the String object. An instance method is not used with an object itself but only with instances of an object. The `String substring()` method is never used with the String object as `String.substring()`. The following fragment declares and initializes a string variable, which is an instance of the String object, and then uses the `String substring()` method with this instance by using the object operator.

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

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

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

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.

Concatenation operator

The plus + may also be used to concatenate strings. The following expression:

```
"one" + "--" + "two"
```

results in the following string:

```
"one--two"
```

delete operator

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

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

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

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

There are several ways to eliminate the data in a property of an object or in an element of an array. The delete operator is the most complete way. Three other techniques use `undefine()`, `undefine`d, and `void`, as illustrated next:

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

a[1] = undefined;
o.five = undefined;

a[1] = void a[1];
o.five = void o.five;
```

These three techniques may be used with any variable, whereas the delete operator may be used only with properties of objects and elements of arrays.

Generally, delete is the best to use with properties of objects and elements of arrays, thought in specific situations the other techniques might be preferable.

See global.undefine() and undefined for more information.

in operator

The `in` operator determines if a property exists in an object. The following script fragment illustrates for the discussion in this section:

```
var isProp;
var obj = {one:111, two:222, Three:333};
var test = 'one';

isProp = test in obj;

if (isProp)
   Screen.writeln(isProp);
Screen.writeln('two' in obj);
Screen.writeln('three' in obj);

/********************************
Display is:
   true
   true
   false
********************************/
```

The script above defines an object, `obj`, with three properties: "one", "two", and "Three" (note the capitalization). The `in` operator is used three times to see if the following strings are properties in `obj`: "one", "two", "three" (note the capitalization). The first two uses of `in` result in `true` and the third in `false`. Look at the expression "`test in obj`". The expression to the left, in this case `test`, of the `in` operator must be a string or be able to convert to a string (since properties of objects are represented as strings). The expression to the right must be an object or array.

ScriptEase JavaScript has a global.defined() function which is useful. The `in` operator may be used in a similar way. In the following fragment, both `in` and `defined()` result in `true`, and the display is:

```
true
true
```

The fragment is:

```
var test = 'TEST';

Screen.writeln('test' in global);
Screen.writeln(defined(test));
```

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);
```

The display is:

```
true
```

The second line could also be written as:

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

The `instanceof` operator does not work with the class of an object, rather it determines if a variable was constructed from an object. In the example above, the variable `s` was defined as an instance of `String` so it is an instance of the String object and is in the class of String. That is, both of the following lines display true:

```
Screen.writeln(s instanceof(String));
Screen.writeln(s._class == "String");
```

The display is:

```
true
true
```

The following code defines a new object and defines the variable `ms` as an instance of `MyString`, a user defined object. In this case, the variable `ms` is an instance of `MyString` but is in the class of `Object`.

```
var ms = new MyString("abcde");
Screen.writeln(ms instanceof(MyString));
Screen.writeln(ms._class == "Object");
ms.show();

function MyString(string)
{

   this.data = string;
   return this;
} // MyString

function MyString.prototype.show()
{
   Screen.writeln(this.data);
} // MyString.prototype.show
```

The display is:

```
true
true
abcde
```

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, `switch`, while, and do ... while. The `break` statement terminates the innermost loop of `for`, `while`, or `do` statements. The program resumes execution on the next line following the loop. The following code fragment does nothing but illustrate the `break` statement.

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

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

continue

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

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 Clib.strcmp() type of 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 may have code in it to detect unusual results and to **throw an exception**. The function is called from inside a **`try`** statement block which will **try 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 thrown exceptions have been caught and handled.

`catch` will receive an error object that can be printed directly as a string, and which will contain these properties

- `name` - Name of the exception class, e.g. "ConversionError"
- `message` - text of error, e.g. "1607: Variable "b" is undefined."
- `fileName` - Name of the source file where error occurred, if available, e.g. "c:\foo\myscript.jsa"
- `lineNum` - Line number if file where error occurred, if available, e.g. "173"
- `functionName` - Name of executing function where error occurred, if available, e.g. "foobar"

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);
       Screen.write("Error occurred at line " + err.lineNum );
       if ( err.fileName )
          Screen.write(" of file " + err.fileName );
       if ( err.functionName )
          Screen.writeln(" in function " + err.functionName );
       Screen.writeln("");
    }
    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 directive in the preprocessor.

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:

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

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

Simulated named parameters

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

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

The following line uses traditional ordered parameters:

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

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

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

The function above could be rewritten as:

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

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

Though JavaScript allows many variations in how objects may be used, this straightforward example illustrates the essence of simulating named parameters in JavaScript. See the section "Named parameters in JavaScript" in the *ScriptEase Tutorial* for a detailed discussion about simulating named parameters in JavaScript.

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"`.

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.

Terminology for objects

The terminology used to describe the methods and properties of objects is not consistent in the programming community. The following list shows three common naming schemes.

- Object members
  Object methods
  Object properties

- Object properties
  Object methods
  Object attributes

- Object properties
  Object methods
  Object properties

In the first scheme uses "members" as the term to encompass "methods" and "properties". The second scheme uses "properties" as the term to encompass "methods" and "attributes". The third scheme uses "properties" as the term to encompass "methods" and "properties". The order in which the schemes are presented is in order from least ambiguous to most ambiguous. Unfortunately this order does not conform to age of use of the schemes nor to the popularity of the schemes. As a result of the lack of consensus in the programming community, all of these naming schemes are represented in ScriptEase documentation, though the first one is preferred because it is not ambiguous. The following paragraphs explain these schemes in more detail.

The first scheme uses the term "member" for the both functions and data. A "method" is a function attached to an object and a "property" is a datum attached to an object. This scheme has the advantages of being clear and of using common terminology. The disadvantage is that the use of "members" to refer to "methods" and "properties" is not the most common. This scheme is preferred in ScriptEase documentation for a couple of reasons. First, objects are thought of as collections of routines and data, which is an intuitive and useful metaphor for describing objects. The term "member" fits nicely with the metaphor and is distinct from the terms for items in the collection. Second, the use of the terms "method" and "property" for the routines and data attached to or collected in an object is the most common usage.

The second scheme uses the term "property" for the both functions and data. A "method" is a function attached to an object and an "attribute" is a datum attached to an object. This scheme has the advantages of being clear and of using common terminology. The disadvantage is that the use of "attributes" for the data of objects is not the most common.

The third scheme uses the term "property" for the both functions and data. A "method" is a function attached to an object and a "property" is a datum attached to an object. This scheme is inherently confusing because it uses the same term "property" for two different concepts about an object.

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.

SElib.getObjectProperties() is similar to the ECMAScript for/in loop. The important difference is that a for/in loop does not enumerate properties that have DONT_ENUM as part of their attributes (global.setAttributes()), whereas SElib.getObjectProperties() includes them in the array that it returns. See Object propertyIsEnumerable().

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 goto and labels to jump into or out of the middle of a with statement block.

_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.

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 CString object. 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 CString. Thus, references to ScriptEase as the C portion of ScriptEase usually involve Clib, SElib, or CString.

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 CString.

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 CString. 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.

Automatic and JavaScript Arrays

C style automatic arrays have just been discussed. Perhaps some simple and direct comparisons of the two different kinds of arrays would be helpful.

The following lines of code create an automatic array of 3 elements:

```
var a;
a[0] = 0;
a[1] = "one";
a[2] = 2;
```

The following line of code creates an automatic array consisting of objects that have information about files in the root directory of drive C. See SElib.directory(). Several functions from the C library objects, Clib and SElib, return automatic arrays.

```
var a = SElib.directory("c:\\*.*");
```

The following two lines of code both produce identical JavaScript arrays of 3 elements each.

```
var a = new Array(0, "one", 2);
var a = [0, "one", 2];
```

The following lines of code also produce a JavaScript array which is identical to the two immediately preceding arrays:

```
var a = new Array();
a[0] = 0;
a[1] = "one";
a[2] = 2;
```

The elements of automatic and JavaScript arrays are accessed in the same way using indices, for example:

```
a[3] = "three";
Screen.writeln(a[3]);
```

These lines behave the same for both automatic and JavaScript arrays. But what are some of the differences? The following fragment:

```
var aa;
aa[0] = 0;
aa[1] = "one";

var ja = [0, "one"];

Screen.writeln(typeof(aa));
Screen.writeln(typeof(ja));

Screen.writeln(aa._class);
Screen.writeln(ja._class);
```

results in the following display:

```
object
object
Object
Array
```

which shows that both automatic and JavaScript arrays are of type `object`. But automatic arrays belong to the Object class and JavaScript arrays belong to the Array class. See array.jsh - arrays and objects for more information on the differences. The Array class inherits the properties of the Object class but the Object class does not have the properties of the Array class. What does that mean?

Instances of both automatic arrays and JavaScript arrays may use the properties and methods of the Object class, but only JavaScript arrays may use the properties of the Array class. For example and using the two arrays defined immediately above, the `length` property of the Array class may only be used with the JavaScript array `ja`, that is, `var len = ja.length` is valid but `var len = aa.length` is an error. To get the length of `aa`, an automatic array, the function global.getArrayLength() must be used with it. As just explained, the JavaScript array `ja` may also be used with the function. That is, both of the following are valid: `getArrrayLength(aa)` and `getArrrayLength(ja).`

Having both types of arrays is a result of providing the C standard library in the Clib and SElib objects. If you want to simplify matters use the Object convert() method to convert arrays from one class to the other. In general, if you convert automatic arrays to JavaScript arrays and work only with JavaScript arrays, your scripting will be simpler and more powerful. But, remember, if you are only going to do simple things with arrays, then working with either class of array is quick and simple.

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 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.

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

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

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

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

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

Passing variables by reference

By default, lvalues in ScriptEase are passed to functions by value (that is, the function cannot alter the lvalue) . But if a variable is declared in a function with the "&" symbol then it is passed by reference. I a function alters a pass-by-reference (i.e. &argument) 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);
   ...
}
```

```
function 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);
```

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]`.

Because it is common in C to use address arithmetic on string, ScriptEase providces the CString object, which provides most of the array and address functionaliity of a C string pointer. 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

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

Remember that in functions, 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;
}
```

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 ");

function 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

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

ScriptEase

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

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

Another step in converting C to 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. 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.

Security

As a scripting language, ScriptEase provides the power to completely control a computer system. But there are times when this power can be dangerous. Many applications, such as those using distributed scripting, might need to run scripts that you do not want to have access to all of the power of ScriptEase. You do not want these scripts to delete files on your machine, read and transmit important data to a remote machine, execute arbitrary system programs, or any other such activities. ScriptEase security allows you to limit scripts so they cannot do these things.

ScriptEase security works by dividing functions on the system into **secure functions,** those which can perform no dangerous actions, and **insecure functions,** those which can perform dangerous activities. When you execute a script, you can attach a security manager to it. This manager will determine which insecure functions can be called.

If the script tries to call an insecure function which the manager does not allow, it will not call the function but will generate a security error. By using ScriptEase security, you can run scripts you trust and give them full access to dangerous functions, such as `Clib.system()` and `Clib.remove()`, while denying access to these same functions to other scripts you do not trust.

Writing a Security Manager

Whenever you wish to interpret a script, via the API using `jseInterpret()` or in a script using `SElib.interpret()`, you can attach a security manager to the child script that you are running. As long as that child script calls other functions only within that script, it is allowed to do so. If it tries to call an insecure function, your security gets called. Obviously, insecure wrapper functions are always checked.

In the case of a script using `SElib.interpret()` to interpret a child script, that child may be able to try to call functions in the parent. Since the security you added only applies to the child script, the functions in your original script are also considered insecure to the child. The child must get permission to call them exactly like it would need to get permission to call an insecure wrapper function directly.

You can think of your security manager as a big wall with a heavily guarded door. As long as the script stays on its side of the wall, it is fine. The parent script and all wrapper functions are on the other side of the wall. If the child script wants to get access to them, it must convince the guards to let it through.

Let's look at the pieces that make up these security guards.

jseSecurityInit

This function is the main security function. It is run before the script that it is protecting is run, and it sets up the security the child is going to be run under. It specifies which functions the child will be allowed to call. By default, the child will not be allowed to call any insecure functions. In this function, you explicitly specify which insecure functions the child will be allowed to call. You do this by calling the `setSecurity()` method, which is a method of all ScriptEase functions.

In case that is confusing, a quick example of a `jseSecurityInit` function should clear it up:

```
function jseSecurityInit(security_var)
{
   Clib.remove.setSecurity(jseSecureAllow);
}
```

This particular security initialization function is written in ScriptEase script. However, you can also implement all of these functions using the ScriptEase API and wrapper functions. We will implement the examples as scripts for clarity. The first thing you notice about the function is that it takes a parameter, we have named it `security_var`. We did not use it in this example. This parameter is the "security variable" described below."

The body of the function usually lists which functions are to be allowed. Notice that we call the `setSecurity()` method of the particular function we want to allow. This method takes one parameter, the security state of the function. `jseSecureAllow` specifies that this function is allowed to be called.

There are two other values we could have used instead. The value `jseSecureReject` causes calls to the function to fail. This is the default for all functions, so it is usually redundant to specify it. However, if `setSecurity()` is called more than once for the same function, the last call takes precedence. You can use this value to undo allowing access to a particular function.

The final value is `jseSecureGuard`, which says that any time this function is called, we must first call the `jseSecurityGuard` function to determine if the call will be allowed. This function is described below.

Note: The `setSecurity()` method can only be called in a security initialization function. Trying to call it at other times generates errors.

jseSecurityTerm

Whenever you have an initialization function, you have a corresponding termination function. Like `jseSecurityInit`, this function gets a single parameter, the security variable (described below.) This function is rarely needed, and you can simply not specify it most of the time. It is included so that you can clean up the security variable before exiting. You do not need to *`unset`* the `setSecurity()` calls done, as the engine knows that they go away when they are no longer used. The security termination function looks like this:

```
function jseSecurityTerm(security_var)
{
   /* do any necessary cleanup */
}
```

This function is not usually called until the end of the program (not just the end of the script.) Why is this? For ISDK developers, if you have read the advanced concepts chapter, you know that all of the functions in a `jseInterpret()` stick around in the global object, even after the `jseInterpret()` call itself is finished. This is why you can *load* functions using `jseInterpret()` and later call them. Whatever security they had when they were created is not forgotten.

All functions remember the security in effect when they were created, and that applies if they are again called later. So, the security termination function is not actually called until all of the functions have gone away, which happens at the end of the program when the ScriptEase engine cleans up everything.

jseSecurityGuard

Usually it is enough to specify which functions you want to allow to be called in the `jseSecurityInit` function and leave it at that. There can be cases in which you want to allow a function to be called with certain parameters but reject it with others. For instance, you may want to limit creating sockets to certain ports or limit opening files to certain filenames. You specify `jseSecureGuard` for the `setSecurity()` options for these functions, and before they can be called, your `jseSecurityGuard` function will first be called to validate this call.

Here is an example:

```
function jseSecurityGuard(security_var, func, filename)
{
   if( func==Clib.fopen )
   {
      /* get the full path so the user can't trick us with
      * something like: 'c:\\temp\\..\\windows\\win.ini'
      */
      var actualname = SElib.fullpath(filename);

      /* We only want to allow files in this directory
      *to be opened.
      */
      return Clib.strnicmp("c:\\temp\\",actualname,8)==0;
   }
   else
   {
      return false;
   }
}
```

This function, like the other two, gets the security variable as its first parameter. Again, we will describe that shortly. The second parameter is the actual function being called. In this example, we compare it to `Clib.fopen()` so that we can validate a call to `Clib.fopen()`. The security guard function must return `true` to allow the call or `false` to disallow it. In this case, we return `false` if it is not `Clib.fopen()`. Presumably, we only label `Clib.fopen()` as `jseSecureGuard`, so only `Clib.fopen()` will be using this guard function.

We include the else clause because it is always a good idea to cover all bases. If it is something we do not expect, we just say no. This is good programming practice in general. If the parameters are not what you expect, even if you think it is impossible for them not to be, still do something sensible even if that turns out not to be the case.

Notice that this function has a third parameter, `filename`. All of the parameters that are being passed to the called function are also passed to the security guard function after the two parameters it always gets. The first parameter to the called function is the third to security guard, the second we receive as our fourth, and so on. This allows us to examine the parameters that the function will get when deciding if we want to allow the call. In fact, there would be little point in not examining the parameters. If we are always going to reject or accept a particular call regardless of the parameters, we can instead just set that up in the `jseSecurityInit` function.

Perceptive readers will note that `Clib.fopen()` actually takes two parameters, but we have only named one of them. In JavaScript, you can pass extra parameters to script functions, more than are named in the parameter list. These parameters are still there and can be accessed using the `arguments` object. In this case, `filename` is the same as `arguments[2]`, and we could have referred to it that way. The file mode parameter to `Clib.fopen()` will also be passed to us. We can refer to it as `arguments[3]`, or we can name it in the parameter list if we need to check it as well.

This example checks the name and only allows file access in the *C:\temp\* directory. We could limit it in any way we choose, this is just one possibility.

securityVariable

We mentioned above that each function gets a security variable passed to it. Each security manager has a single variable associated with it. You can specify this when you specify your security functions (see below for specifying security). Alternately, if you do not, a blank ScriptEase object is created (as if calling `new Object()`) and used. This variable cannot be accessed by the script being run, but it is passed to each security function whenever it is called. This allows you to store data needed to implement your security and keep it safe from the script being run.

Specifying Security

The ScriptEase API call `jseInterpret()` has among its settings `jseNewSecurity`. If you turn this on, then the script being run will have security applied to it. If you leave it off, no security applies and all functions can be called. The four security items we just finished discussing correspond to the four fields of the `jseExternalLinkParameters` structure of the same name. Before you interpret the script, you use `jseGetExternalLinkParameters()` to get the parameters structure, fill in these fields, then call `jseInterpret()` with the `jseNewSecurity` flag turned on. You must fill in the `jseSecurityInit` function. If you do not, the `jseNewSecurity` flag will be ignored.

Since the parameters are jseVariables, you set them to any function you like. You can use `jseCreateWrapperFunction()` to create a wrapper function to do the security tasks. In the example above, we used script examples. ScriptEase Desktop implements security this way. The three functions are put in a script. You tell ScriptEase Desktop the name of the script using the command line parameter `/secure=<security script name>`. ScriptEase Desktop interprets that script first, picks out the security functions, and uses them when it interprets the script you are really interested in. The functions in the security script must be given the names we described above.

When you interpret a script from within a script, using `SElib.interpret()`, you can also specify the security for that child script. See the manual description of SElib.interpret() for details on how you do this.

Wrapper Functions And Security

Wrapper functions are insecure because they are labeled that way. When you write your own wrapper functions and add them using `jseAddLibrary()`, you get to label them as either secure or insecure. Remember, if there is any possible way the function could be misused, make it insecure. If you are in doubt about whether a particular function should be labeled secure or insecure, choose insecure.

When you are writing a wrapper function, it is possible for it to use `jseCallFunction()` or `jseInterpret()` to execute more code. These calls are affected by security. This allows security to propagate. For instance, the ECMAScript function `eval()` executes a text string as script code exactly like the text string appeared directly in the script. In this case, the wrapper acts just as a pass through, and the code it executes should follow all of the standard security rules. In fact, the ECMAScript `eval()` function itself is secure; whatever text it executes has the same security as what was already executing. ScriptEase uses this model when you use these two API calls. As a result, the following behavior applies:

When calling a function using `jseCallFunction()`, the call is treated as if the wrapper function's caller was making the call. This means that the calling script function will need to get approval to call the new function. Typically, a wrapper function that just turns around and uses `jseCallFunction()` is itself secure.

`jseInterpret()` has different behavior depending on the wrapper function itself. If the wrapper function is insecure, then the script run with `jseInterpret()` starts with no security. If the wrapper function is secure, then `jseInterpret()` starts with the same security as the calling function.

So, for instance, ECMAScript `eval()` is secure as we already mentioned. Thus, when it runs a new script, that script has the existing security restrictions still on it. If the function was labeled insecure, then it has already passed a security check to be able to call it, and it can continue to do dangerous things, so any scripts it interprets are likewise at this high level of security. `jseInterpret()` allows security to be added using the `jseNewSecurity` flag. This is on top of whatever security it already has as specified above.

Sample Script

Here is a sample ScriptEase Desktop security script. If you use it, then the desktop scripts will not be allowed to use any insecure functions except a few file-related ones. In addition, `Clib.fopen()` will only be allowed to open files in the *C:\temp\* directory.

```
function jseSecurityInit(security_var)
{
   /* allow basic file manipulations, but nothing fancy, and
   * make sure to examine all open calls very carefully.
   */
   Clib.fopen.setSecurity(jseSecureGuard);
   Clib.fclose.setSecurity(jseSecureAllow);
   Clib.fprintf.setSecurity(jseSecureAllow);
   Clib.fread.setSecurity(jseSecureAllow);
   Clib.fwrite.setSecurity(jseSecureAllow);
}

function jseSecurityGuard(security_var, func, filename)
{
   /* we only guard the fopen call, so this should be it */
   Clib.assert( security_var==Clib.fopen );

   /* get the full path so the user can't trick us with something
   * like: 'c:\\temp\\..\\windows\\win.ini'
   */
   var actualname = SElib.fullpath(filename);

   /* We only want to allow files in this directory to be opened.
   */
   return Clib.strnicmp("c:\\temp\\",actualname,8)==0;
}
```

Internal Objects

See:

- Global object
- Array object
- Blob Object
- Boolean Object
- Buffer Object
- Clib Object
- Date Object
- Dos Object
- Function Object
- Math Object
- Number Object
- Object Object
- RegExp Object
- SElib Object
- String Object
- Unix Object

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._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: main() function, global._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: main() function, global._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(), in operator, undefined

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(str)`

WHERE: str - with special characters that need to be handled specially, that is, escaped.

RETURN: string - with special characters escaped or fixed so that the string may be used in special ways, such as being a URL.

DESCRIPTION: The `escape()` method receives a string and escapes the special characters so that the string may be used with a URL. This escaping conversion may be called encoding. All uppercase and lowercase letters, numbers, and the special symbols, @ * + - . /, remain in the string. All other characters are replaced by their respective unicode sequence, a hexadecimal escape sequence. This method is the reverse of global.unescape().

SEE: global.unescape(), String escape()

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, `global.getArrayLength()` and global.setArrayLength(), when working with array lengths. The `getArrayLength()` function returns the length of a dynamic array, which is one more than the highest index of an array, if the first element of the array is at index 0, which is most common. If the parameter minIndex is passed, then it is used to set to the minimum index, which will be zero or less. You can use this function to get the length of an array that was not created with the `Array()` constructor function. This function and its counterpart, `setArrayLength()`, are intended for use with dynamically created arrays, that is, arrays not created with the `Array()` constructor function. Use the

	Array length property to get the length of arrays created with the constructor function and not `getArrayLength()`.
SEE:	global.setArrayLength(), Array length

EXAMPLE:

```
   // automatic object array
var arr;
arr[0] = "zero";
arr[1] = 1;
arr[2] = 2;
Screen.writeln(getArrayLength(arr));   // 3

   // JavaScript Array object
var arr = ["zero", 1, 2]
Screen.writeln(arr.length);            // 3
```

global.getAttributes()

SYNTAX: `getAttributes(variable)`

WHERE: variable - a variable identifier, name.

RETURN: number - representing the attributes set for a variable. If no attributes are set, the return is 0. See global.setAttributes() for a list of predefined constants for the attributes that a variable may have.

DESCRIPTION: Gets and returns the variable attributes for the parameter variable. Variable attributes may be set using the function `setAttributes()`. See global.setAttributes() for more information and descriptions of the attributes of variables that can be set.

SEE: global.setAttributes()

global.parseFloat()

SYNTAX: `parseFloat(str)`

WHERE: str - to be converted to a decimal float.

RETURN: number - the float to which the string converts, else `NaN`.

DESCRIPTION: This method is similar to global.parseInt() except that it reads decimal numbers with fractional parts. In other words, the first period, ".", in the parameter string is considered to be a decimal point, and any following digits are the fractional part of the number. The method `parseFloat()` does not take a second parameter.

SEE: global.parseInt()

EXAMPLE:

```
var i = parseInt("9.3");
```

global.parseInt()

SYNTAX: `parseInt(str[, radix])`

WHERE: str - to be converted to an integer.

radix - the number base to use, default is 10.

RETURN: number - the integer to which string converts, else `NaN`.

DESCRIPTION: This method converts an alphanumeric string to an integer number. The first parameter, str, is the string to be converted, and the second parameter, radix, is an optional number indicating which base to use for the number. If the radix parameter is not supplied, the method defaults to base 10, which is decimal. If the first digit of string is a zero, radix defaults to base 8, which is octal. If the first digit is zero followed by an "x", that is, "0x", radix defaults to base 16, which is hexadecimal.

White space characters at the beginning of the string are ignored. The first non-white space character must be either a digit or a minus sign (-). All numeric characters following the string will be read, up to the first non-numeric character, and the result will be converted into a number, expressed in the base specified by the radix variable. All characters including and following the first non-numeric character are ignored. If the string is unable to be converted to a number, the special value `NaN` is returned.

SEE: global.parseFloat()

EXAMPLE:

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

global.setArrayLength()

SYNTAX: `setArrayLength(array[, minIndex[, length]])`

WHERE: array - may be an array, buffer, or string. Though there are multiple ways to set length on these data types, `setArrayLength()` may be used on all three.

minIndex - the minimum index to use. Default is 0.

length - the length of the array to set.

RETURN: void.

DESCRIPTION: This function sets the first index and length of an array. Any elements outside the bounds set by MinIndex and length are lost, that is, become `undefined`. If only two arguments are passed to `setArrayLength()`, the second argument is length and the minimum index of the newly sized array is 0. If three arguments are passed to `setArrayLength()`, the second argument, which must be 0 or less, is the minimum index of the newly sized array, and the third argument is the length.

SEE: global.getArrayLength(), Array length, Blob.size()

EXAMPLE:

```
#include <string.jsh>

var arr = [4,5,6,7];
Screen.writeln(getArrayLength(arr));
setArrayLength(arr, 5);
// arr is now [4,5,6,7,,];

/********************************
```

```
   The examples below illustrate using
   setArrayLength() with:
      arrays
      strings
      buffers

   When appropriate alternatives exist for setting
   length, they are shown as comments.

   These examples are not 100% exhaustive, but show
   most ways to use setArrayLength().
*******************************/

// Two ways to create an array
// with 5 undefined elements
var a1 = new Array();
setArrayLength(a1, 5);
//a1.length = 5     // Does the same

var a2 = [];
setArrayLength(a2, 5);
//a2.length = 5     // Does the same

// Two ways to create a string
// of five "\0" characters
var s1 = "\0".repeat(5);

var s2 = "";
setArrayLength(s2, 5);

// Three ways to create a buffer
// of five "\0" characters
var b1 = new Buffer(s1);

var b2 = new Buffer(5);

var b3 = new Buffer(5);
setArrayLength(b3, 5);
//Blob.size(b3, 5);    // Does the same
//b3.length = 5;       // Does the same
```

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: global.getAttributes()

EXAMPLE:

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

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

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

global.ToBoolean()

SYNTAX: `ToBoolean(value)`

WHERE: value - to be cast as a boolean.

RETURN: boolean - conversion of value.

DESCRIPTION: The following list indicates how different data types are converted by this function.

- `Boolean`

same as value

- `Buffer`
  same as for String
- `null`
  `false`
- `Number`
  `false`, if value is 0, +0, -0 or `NaN`, else `true`
- `Object`
  `true`
- `String`
  `false` if empty string, "", else `true`
- `undefined`
  `false`

global.ToBuffer()

SYNTAX:	`ToBuffer(value)`
WHERE:	value - to be cast as a buffer.
RETURN:	buffer - conversion of value.
DESCRIPTION:	This function converts value to a buffer in a manner similar to global.ToString() except that the resulting array of characters is a sequence of ASCII bytes and not a unicode string.
SEE:	global.ToBytes()

global.ToBytes()

SYNTAX:	`ToBytes(value)`
WHERE:	value - to be cast as a buffer.
RETURN:	buffer - conversion of value.
DESCRIPTION:	This function converts value to a buffer and differs from global.ToBuffer() in that the conversion is actually a raw transfer of data to a buffer. The raw transfer does not convert unicode values to corresponding ASCII values. For example, the unicode string `"Hit"` is stored in a buffer as `"\0H\0\i\0t"`, that is, as the hexadecimal sequence: 00 48 00 69 00 74.
SEE:	global.ToBuffer()

global.ToInt32()

SYNTAX:	`ToInt32(value)`
WHERE:	value - to be cast as a signed 32-bit integer.
RETURN:	number - conversion of value.
DESCRIPTION:	This function is the same as global.ToInteger() except that if the return is an integer, it is in the range of -2^{31} through $2^{31}-1$.
SEE:	global.ToInteger(), global.ToNumber()

global.ToInteger()

SYNTAX: `ToInteger(value)`

WHERE: value - to be cast as an integer.

RETURN: number - conversion of value.

DESCRIPTION: This function converts value to an integer type. First, call global.ToNumber(). If result is `NaN`, return +0. If result is +0, -0, +Infinity or -Infinity, return result. Else return floor(abs(result)) with the appropriate sign. For example, the value -4.8 is converted to -4.

SEE: global.ToInt32(), global.ToNumber()

global.ToNumber()

SYNTAX: `ToNumber(value)`

WHERE: value - to be cast as a number.

RETURN: number - conversion of value.

DESCRIPTION: The following table lists how different data types are converted by this function.

- `Boolean`
  +0, if value is `false`, else 1
- `Buffer`
  same as for String
- `null`
  +0
- `Number`
  same as value
- `Object`
  first, call ToPrimitive(), then call ToNumber() and return result
- `String`
  number, if successful, else `NaN`
- `undefined`
  `NaN`

SEE: global.ToInteger(), global.ToInt32()

global.ToObject()

SYNTAX: `ToObject(value)`

WHERE: value - to be cast as an object.

RETURN: object - conversion of value.

DESCRIPTION: The following table lists how different data types are converted by this function.

- `Boolean`
  new Boolean object with value
- `null`

generate runtime error

- `Number`
  new Number object with value
- `Object`
  same as parameter
- `String`
  new String object with value
- `undefined`
  generate runtime error

SEE: global.ToPrimitive()

global.ToPrimitive

SYNTAX: `ToPrimitive(value)`

WHERE: value - to be cast as a primitive.

RETURN: value - conversion of value to one of the primitive data types.

DESCRIPTION: This function does conversions only for parameters of type Object. An internal default value of the Object is returned.

SEE: global.ToObject()

global.ToSource()

SYNTAX: `ToSource(value)`

WHERE: value - a variable or value to convert to a source string that will reproduce `value` when the string is evaluated or interpreted.

RETURN: string - a string representation of `value`, which can be evaluated or interpreted.

DESCRIPTION: A variable or value may be represented by a string comprised of JavaScript statements which, when evaluated or interpreted, reproduce the variable or value. The source string may be evaluated by global.eval() or by SElib.interpret(). It is sometimes convenient or powerful to use source strings, for example, in the Data object the DSP object.

Though the source string may be read by humans, it is daunting. Remember, `ToSource()` is designed for interpretation by the ScriptEase interpreters, not by users.

The example below compares source strings created by the `global.ToSource()` function and the Object toSource() method. In these examples, the source strings are identical, which is not guaranteed always to be so. But, no matter which one is used, the source strings can be evaluated or interpreted.

SEE: Object toSource(), global.eval(), SElib.interpret()

EXAMPLE:

```
// An Array
var a = [1, '2', 3];

Screen.writeln(ToSource(a));
Screen.writeln();
Screen.writeln(a.toSource());
Screen.writeln();
/*******************************
Displays:

((new Function("var tmp1 = [1,\"2\",3]; tmp1[\"0\"] =
1;
tmp1[\"1\"] = \"2\"; tmp1[\"2\"] = 3; return
tmp1;"))())

((new Function("var tmp1 = [1,\"2\",3]; tmp1[\"0\"] =
1;
tmp1[\"1\"] = \"2\"; tmp1[\"2\"] = 3; return
tmp1;"))())
*******************************/

// An Object
var o = {one:1, two:'2', three:3};

Screen.writeln(ToSource(o));
Screen.writeln();
Screen.writeln(o.toSource());
Screen.writeln();
/*******************************
Displays:

((new Function("var tmp1 = new Object();
tmp1[\"three\"] = 3;
tmp1[\"one\"] = 1; tmp1[\"two\"] = \"2\"; return
tmp1;"))())

((new Function("var tmp1 = new Object();
tmp1[\"three\"] = 3;
tmp1[\"one\"] = 1; tmp1[\"two\"] = \"2\"; return
tmp1;"))())
*******************************/
```

global.ToString()

SYNTAX: `ToString(value)`

WHERE: value - to be cast as a string.

RETURN: string - conversion of value.

DESCRIPTION: The following table lists how different data types are converted by is this function.

- `Boolean`
  "false", if value is `false`, else "true"
- `null`
  "null"
- `Number`
  if value is `NaN`, return "NaN". If +0 or -0, return "0". If Infinity, return "Infinity". If a number, return a string representing the number. If a number is negative, return "-" concatenated with the string representation of the number.
- `Object`
  first, call ToPrimitive(), then call ToString() and return result

- `String`
  same as value
- `undefined`
  "undefined"

SEE: global.ToPrimitive(), global.ToNumber()

global.ToUint16()

SYNTAX: `ToUint16(value)`

WHERE: value - to be cast as a 16 bit unsigned integer.

RETURN: number - conversion of value.

DESCRIPTION: This function is the same as global.ToInteger() except that if the return is an integer, it is in the range of 0 through $2^{16} - 1$.

SEE: global.ToUint32(), global.ToInteger()

global.ToUint32()

SYNTAX: `ToUint32(value)`

WHERE: value - to be cast as a 32 bit unsigned integer.

RETURN: number - conversion of value.

DESCRIPTION: This function is the same as global.ToInteger() except that if the return is an integer, it is in the range of 232 - 1.

SEE: global.ToInt32(), global.ToInteger()

global.unescape()

SYNTAX: `unescape(str)`

WHERE: str - holding escape characters.

RETURN: string - with escape characters replaced by appropriate characters.

DESCRIPTION: This method is the reverse of the global.escape() method and removes escape sequences from a string and replaces them with the relevant characters. That is, an encoded string is decoded.

SEE: global.escape(), String unescape()

EXAMPLE:
```
unescape("Hello%20there%21");
// Returns "Hello there!"
```

global.undefine()

SYNTAX: `undefine(value)`

WHERE: value - value, variable, or property to be `undefined`.

RETURN: void.

DESCRIPTION: This function undefines a variable, Object property, or value. If a value was previously defined so that its use with the function global.defined() returns `true`, then after using `undefine()`

with the value, `defined()` returns `false`. Undefining a value is different than setting a value to `null`.

The delete operator may be used only with properties of objects and elements of arrays and is more complete than `undefine()`. Two other techniques, using undefined and `void`, are equivalent to `undefine()`. The following three techniques for undefining `test` are equivalent:

```
 var test = 111;

 undefine(test);
 test = undefined;
 test = void test;
```

SEE: global.defined(), delete operator, undefined

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);
```

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 to ease the programming task by providing another easy to use tool for scripters. The current section is about Array objects.

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

Array elements can be of any data type. The elements in an array do not all need to be of the same type, and there is no limit to the number of elements an array may have.

The following statements demonstrate assigning values to arrays.

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

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

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

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

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

```
for(var x = 1; x < 32; x++)
   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.

SEE: array.jsh - arrays and objects

Creating arrays

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

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

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

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

In this case, an array with length 31 is created.

Finally, you can pass a list of elements to the `Array()` function, which creates an array containing all of the parameters passed. For example:

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

creates an array with a length of 6. c[0] is set to 5, c[1] is set to 4, and so on up to c[5], which is set to the string "blast off". Note that the first element of the array is array[0], not array[1].

Arrays may also be created dynamically. By referring to a variable with an index in brackets, a variable is created as or converted to an array. The array that is created is an automatic or dynamic array which is different than an instance of an Array object created as described in this section. Automatic arrays, created as described in this paragraph, are unable to use the methods and properties described below, so it is recommended that you use, in most circumstances, the `new Array()` constructor function to create arrays.

Initializers for arrays and objects

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

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

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

Arrays may be initialized using syntax similar to the following:

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

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

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

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

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

The following code fragment shows the differences.

```
var o = {a:1, b:2, c:3};
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(), global.getArrayLength(), global.setArrayLength()

EXAMPLE:

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

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

var bee = new Array();
```

```
bee[0] = 88;
bee[3] = 99;

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

Array object instance methods

Array()

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

WHERE: length - If this is a number, then it is the length of the array to be created. Otherwise, it is the element of a single-element array to be created.

elementN - list of elements to be in the new Array object being created.

RETURN: object - an Array object of the length specified or an Array object with the elements specified.

DESCRIPTION: The array returned from this function is an empty array whose length is equal to the `length` parameter. If `length` is not a number, then the length of the new array is set to 1, and the first element is set to the `length` parameter. Note that this can also be called as a function, without the new operator.

The alternate form of the Array constructor initializes the elements of the new array with the arguments passed to the function. The arguments are inserted in order into the array, starting with element 0. The length of the new array is set to the total number of arguments. If no arguments are supplied, then an empty array of length 0 is created.

SEE: Automatic array allocation

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

Array concat()

SYNTAX:
```
array.concat([element1, ...])
```

WHERE: elementN - list of elements to be concatenated to this Array object.

RETURN: object - a new array consisting of the elements of the current object, with any additional arguments appended.

DESCRIPTION: The return array is first constructed to consist of the elements of the current object. If the current object is not an Array object, then the object is converted to a string and inserted as the first element of the newly created array. This method then cycles through all of the arguments, and if they are arrays then the elements of the array are appended to the end of the return array, including empty elements. If an argument is not an array, then it is first converted to a string and appended as the last element of the array. The length of the newly created array is adjusted to reflect the new length. Note that the original object remains unaltered.

SEE: String concat()

EXAMPLE:

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

Array join()

SYNTAX: `array.join([separator])`

WHERE: separator - a value to be converted to a string and used to separate the list of array elements. The default is an empty string.

RETURN: string - string consisting of the elements, delimited by separator, of an array.

DESCRIPTION: The elements of the current object, from 0 to the length of the object, are sequentially converted to strings and appended to the return string. In between each element, the separator is added. If `separator` is not supplied, then the single-character string "," is used. The string conversion is the standard conversion, except the `undefined` and `null` elements are converted to the empty string "".

The Array `join()` method creates a string of all of array elements. The `join()` method has an optional parameter, a string which represents the character or characters that will separate the array elements. By default, the array elements will be separated by a comma. For example:

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

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

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

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

SEE: Array toString()

EXAMPLE:

```
// The following code:

var array = new Array( "one", 2, 3, undefined );
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 Array 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 Array 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 relative 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.

SEE: Buffer object, Win32 structure definitions

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 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, (such as SElib.dynamicLink(), Clib.fread(), and Clib.fwrite(), which might 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`.

See Win32 structure definitions.

blobDescriptor example

The Win32 API function *GetOpenFileName* is being used for this example. The syntax, in the Win32 API documentation, for this function is:

```
BOOL GetOpenFileName(LPOPENFILENAME lpofn);
```

The ScriptEase syntax for calling the Win32 API is:

```
SElib.dynamicLink(library, procedure, convention
                  [, [desc,] param …])
```

The first three parameters in the ScriptEase syntax are standard for all calls to the Win32 API and are not discussed here. See SElib.dynamicLink() - for Win32 for a complete discussion. In the current section, we are only dealing a structure parameter since the lpofn parameter in the *GetOpenFileName* API function is a pointer to a structure. Other parameters, not discussed here, such as, integers and double words, are handled in a straightforward way.

An actual call to the *GetOpenFileName* function might look like the following:

```
var rtn;
rtn = SElib.dynamicLink("COMDLG32", "GetOpenFileNameA", STDCALL,
                        ofnDesc, ofn);
```

We are concerned with the parameters: `ofnDesc` and `ofn`. The original function only required one parameter, *lpofn*, but we are passing two parameters. (Remember that the first three parameters: `library`, `procedure`, and `convention`, are parameters for `SElib.dynamicLink` and that the parameters to API functions begin after the first three.) Why two parameters? Because `ofn` is a structure and ScriptEase requires a description of the structure. Hence, `ofnDesc` is a blobDescriptor object and `ofn` is a structure, and, in ScriptEase, a structure is considered a binary large object.

Lets look at the *OpenFileName* structure that is defined in the Win32 API and required by the *GetOpenFileName* function. The structure is defined as:

```
typedef struct tagOFN {// ofn
     DWORD         lStructSize;
     HWND          hwndOwner;
     HINSTANCE     hInstance;
     LPCTSTR       lpstrFilter;
     LPTSTR        lpstrCustomFilter;
     DWORD         nMaxCustFilter;
     DWORD         nFilterIndex;
     LPTSTR        lpstrFile;
     DWORD         nMaxFile;
     LPTSTR        lpstrFileTitle;
     DWORD         nMaxFileTitle;
     LPCTSTR       lpstrInitialDir;
     LPCTSTR       lpstrTitle;
     DWORD         Flags;
     WORD          nFileOffset;
     WORD          nFileExtension;
     LPCTSTR       lpstrDefExt;
     DWORD         lCustData;
     LPOFNHOOKPROC lpfnHook;
     LPCTSTR        lpTemplateName;
 } OPENFILENAME;
```

In ScriptEase, the `blobDescriptor` for the *`OpenFileName`* structure above could look like the following:

```
var ofnDesc = new blobDescriptor();

ofnDesc.lStructSize       = UWORD32;
ofnDesc.hwndOwner         = UWORD32;
ofnDesc.hInstance         = UWORD32;
ofnDesc.lpstrFilter       = UWORD32;
ofnDesc.lpstrCustomFilter = UWORD32;
ofnDesc.nMaxCustFilter    = UWORD32;
ofnDesc.nFilterIndex      = UWORD32;
ofnDesc.lpstrFile         = UWORD32;
ofnDesc.nMaxFile          = UWORD32;
ofnDesc.lpstrFileTitle    = UWORD32;
ofnDesc.nMaxFileTitle     = UWORD32;
ofnDesc.lpstrInitialDir   = UWORD32;
ofnDesc.lpstrTitle        = UWORD32;
ofnDesc.Flags             = UWORD32;
ofnDesc.nFileOffset       = UWORD16;
ofnDesc.nFileExtension    = UWORD16;
ofnDesc.lpstrDefExt       = UWORD32;
ofnDesc.lCustData         = UWORD32;
ofnDesc.lpfnHook          = UWORD32;
ofnDesc.lpTemplateName    = UWORD32;
```

As you can see, the ScriptEase `blobDescriptor` functions like a structure definition in another language and, specifically, like *`struct`* in C. The *`OpenFileName`* shown above is used with *`typedef`* for a *`struct`*, which might be a more useful comparison than just a structure definition. In any case, the similarity between structures and blobDescriptors is evident. Each property of the `blobDescriptor` object describes or determines how much memory is used by an element of a structure. For example, the first element of the Win32 API *`OpenFileName`* structure is *`lStructSize`* of type *`DWORD`*. In ScriptEase, the corresponding first property in `ofnDesc` is `lStructSize` and is defined as `UWORD32`. Both *`DWORD`* in the Win32 API and `UWORD32` in ScriptEase designate 32 bits of memory to hold data. Thus, the memory requirements, for a structure, in the Win32 API and in ScriptEase are coordinated.

Notice that the original structure element name is *`lStructSize`* and the object property name `lStructSize` are the same. They did not need to be. The property names in a `blobDescriptor` object can be any names of your choosing. It is the size designations, such as, `UWORD32`, that are important. This `blobDescriptor` is the parameter `desc` in the syntax statement for `SElib.dynamicLink()`.

Now we need to define the parameter `param` that is described. (Remember, `desc` is required only if the following `param` is a structure.) In our current example, `ofn` is the structure that is passed as `param` following the `ofnDesc` which is passed as `desc`. How might `ofn` be built since ScriptEase no longer has structure data types? Objects may be used as structures with object properties being equivalent to structure elements. So the following lines of code could be used:

```
#include "comdlg32.jsh"
```

```
#define MAXFILESIZE 65

var ofn = new Object();
   // Size of the ofn structure
ofn.lStructSize = Blob.size(ofnDesc);

   // Handle of owner, a ScriptEase screen in this example
ofn.hwndOwner = Screen.handle();

   // Set a buffer to pass and receive a filespec
var fileSpec;
fileSpec = new Buffer(MAXFILESIZE);
fileSpec.putString(`c:\bat\*.bat`);
fileSpec = fileSpec.toString();
   // Actually pass a pointer to this buffer
ofn.lpstrFile = SElib.pointer(fileSpec);
   // Set the maxsize for a filespec to pass and received
ofn.nMaxFile = MAXFILESIZE - 1;

   // Do the API call and get the function return
var rtn;
rtn =  SElib.dynamicLink("COMDLG32", "GetOpenFileNameA", STDCALL,
                          ofnDesc, ofn);
```

This code fragment would create a common open file dialog in a directory `c:\bat` and would show files with extensions of `bat`. The last statement is the `SElib.dynamicLink()` call. The object/structure `ofn` is passed, corresponding to the *`lpofn`* parameter in the original Win32 API syntax. The `ofnDesc blobDescriptor` is passed to describe ofn to ScriptEase so that ScriptEase may communicate properly with the Win32 API.

Notice two things about the `ofn` object/structure.

- The property names match the properties in the blobDescriptor `ofnDesc` that describes the `ofn` object/structure.
- Not all of the properties of the ofn object/structure needed to be initialized to values. We created a simple open dialog that did not need any data except the properties/elements that we defined. Often, it is not necessary to define data elements that are passed to an API function, if the data is not used. Be careful though. If you are not sure about whether or not to initialize all elements, it is a safe practice to initialize them to default values specified by API documentation.

Another thing of interest in this code fragment is how it handles string data. The `lpstrFile` property/element is used to pass a string to and receive a string from the *`GetOpenFileNameA`* API function. The method shown here is one way, among other techniques to handle string data. The API *`OpenFileName`* structure requires a point to a string buffer, not the string itself. Therefore, this fragment creates a buffer `filespec` of the proper size. It then puts the string with a file specification into the buffer and then converts the buffer to a string. ScriptEase strings may contain "`\0`" characters. The Buffer toString() method creates a string of the same length as the buffer and includes all of the "`\0`" characters after the string `` `c:\bat\*.bat` ``. Then the element `lpstrFile` is assigned a pointer to the string `filespec`, which started its existence as a Buffer object. The file name selected in the open dialog will be returned in the `filespec` string/buffer. We have been discussing the following lines:

```
var fileSpec;
fileSpec = new Buffer(MAXFILESIZE);
fileSpec.putString(`c:\bat\*.bat`);
fileSpec = fileSpec.toString();
   // Actually pass a pointer to this buffer
ofn.lpstrFile = SElib.pointer(fileSpec);
```

We could have accomplished the task of passing and receiving string data with the following lines (which are similar to the ones above):

```
var fileSpec;
fileSpec = new Buffer(MAXFILESIZE);
fileSpec.putString(`c:\bat\*.bat`);
   // Actually pass a pointer to this buffer
ofn.lpstrFile = SElib.pointer(fileSpec.data);
```

The main difference is that the string data is in a buffer when passed and returned. To work with the returned string data, the buffer must be converted to a string if you want to use string methods and functions with it.

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()` method returns a string corresponding to the value of a Boolean object or primitive data type.

SEE: Boolean toString(), boolean type

EXAMPLE:

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

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, from a string, another Buffer object, or from most any data type or object (see global.ToBuffer()).

(See the helper file buffer.jsh for enhancements to the Buffer object.)

ScriptEase 5.00 introduced an important change in buffers. Prior to version 5.00 ScriptEase JavaScript had a buffer data type, as well as, a Buffer object. Beginning with ScriptEase 5.00, buffer data types no longer exist, only the Buffer object. The scripts distributed with ScriptEase Desktop have been updated to reflect the changes, but users might have some personal scripts that need changing. So, some key differences in working with buffers as objects only, without a unique data type, are discussed now.

First, many (probably most) script statements, using buffers, do need to be changed. All of the Buffer and Blob methods still work as they did before. The primary difference is in the use of the Buffer() function without the `new` constructor, the use of the `data` property in a Buffer object, and the use of the `length` and `size` properties.

Previously, the `Buffer()` function returned a buffer data type and the `new Buffer()` constructor returned a Buffer object. The `data` property of the Buffer object could be used to access the actual buffer data in the object. The `length` property could be used with a buffer data type to get the length of a buffer. Now both `Buffer()` and `new Buffer()` return a Buffer object, and the data property no longer exists. Plus, only the `size` property may be used to get the size or length of a buffer. The following fragments illustrate these differences.

Prior to ScriptEase 5.00

```
var b1 = Buffer("abc");      // b1 is buffer data type
var b2 = new Buffer("abc"); // b2 is Buffer object
Screen.writeln(b1);          // abc
Screen.writeln(b1.data);     // abc
Screen.writeln(b1.length);   // 3
Screen.writeln(b2.length);   // 3
Screen.writeln(b2);          // abc
Screen.writeln(b2.data);     // abc
Screen.writeln(b1.size);     // 3
Screen.writeln(b2.size);     // 3
```

Starting with ScriptEase 5.00

```
var b1 = Buffer("abc");      // b1 is Buffer object
var b2 = new Buffer("abc"); // b2 is Buffer object
Screen.writeln(b1);          // abc
Screen.writeln(b1.data);     // undefined
Screen.writeln(b1.length);   // undefined
Screen.writeln(b2.length);   // undefined
Screen.writeln(b2);          // abc
Screen.writeln(b2.data);     // undefined
Screen.writeln(b1.size);     // 3
Screen.writeln(b2.size);     // 3
```

Most ScriptEase functions and methods are now smart enough to handle Buffer objects as they once did buffer data types, but small changes might be needed in some places. For example, previously, some SElib.dynamicLink() calls required a buffer data type. But now a Buffer object may be used. So if you have code (prior to ScriptEase 5.00) like:

```
var buf = Buffer(256); // creates a buffer data type
SElib.dynamicLink("user32", "GetClassNameA", STDCALL,
                  this.handle, buf, buf.length);
this.className = buf.getString();
return this.className;
```

With code starting after ScriptEase 5.00, the fragment above should work with the one correction, **`length`** to **`size`**, as shown in bold.

```
var buf = Buffer(256); // creates a Buffer object
SElib.dynamicLink("user32", "GetClassNameA", STDCALL,
                  this.handle, buf, buf.size);
this.className = buf.getString();
return this.className;
```

The example above is taken from the `Window.prototype.getClassName()` instance method definition in *winobj.jsh*. To illustrate changes due to the use of the data property, we take our example from the `Window.prototype.getClientRect()` instance method defined in *winobj.jsh*. Prior to ScriptEase 5.00 it was defined as:

```
var rtn;
var Rect = new Buffer(4+4+4+4);  // hold 4 integers of 4 bytes

if (rtn = SElib.dynamicLink("user32", "GetClientRect", STDCALL,
                            this.handle, Rect.data))
{
   this.client.left   = Rect.getValue(4);
   this.client.top    = Rect.getValue(4);
   this.client.right  = Rect.getValue(4);
   this.client.bottom = Rect.getValue(4);
}
```

In ScriptEase 5.00 it needs to be corrected in one place: the removal of the data property.

```
return rtn != NULL;

var rtn;
var Rect = new Buffer(4+4+4+4);  // hold 4 integers of 4 bytes

if (rtn = SElib.dynamicLink("user32", "GetClientRect", STDCALL,
                            this.handle, Rect))
{
   this.client.left   = Rect.getValue(4);
   this.client.top    = Rect.getValue(4);
   this.client.right  = Rect.getValue(4);
   this.client.bottom = Rect.getValue(4);
}

return rtn != NULL;
```

SEE: Blob 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 Buffer getValue() and Buffer putValue(). This value is set when a buffer is created, but may be changed at any time. This property defaults to the state of the underlying OS and processor.

SEE: Buffer unicode, Buffer()

EXAMPLE:

```
/********************************
The default behavior for a Windows 2000
using an i386 type of processor results
in the following buffer or 6 bytes:
   65 00 66 00 67 00
********************************/
var i;
var b = new Buffer("ABC", true);

/********************************
With bigEndian set, as in the following,
the buffer is:
   00 65 00 66 00 67
********************************/
var i;
var b = new Buffer("ABC", true, 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, then the buffer is extended to accommodate the new position, and filled with `NULL`, "\0", 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, Buffer size

EXAMPLE:

```
var b = new Buffer("@ABCDE");
// now b.cursor == 0
b.cursor = 3;
Screen.writeln(b.getValue()); // 67 - ASCII for "C"
```

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 Buffer size to a value smaller than the current position of the Buffer cursor, then the cursor is moved to the end of the new buffer.

SEE: Buffer cursor

EXAMPLE:
```
var n = buffer.size;
```

Buffer unicode

SYNTAX:
```
buffer.unicode
```

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

When the Buffer toString() method is used with a Unicode buffer, an ASCII based string, of type string, is returned.

The size of an Unicode buffer is usually twice the size of an ASCII based, or byte based, buffer. For example,

```
var b1 = new Buffer("abc");
var b2 = new Buffer("abc", true);
```

Result in the following:

```
b1.size == 3
b2.size == 6;
```

SEE: Buffer bigEndian, Buffer()

EXAMPLE:
```
var b1 = new Buffer("abc", true);
// b1.unicode == true;
```

Buffer[] Array

SYNTAX:
```
buffer[offset]
```

RETURN: number - the value in a buffer at the index or `offset` specified. If a value is being assigned to this position, the value assigned is returned.

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

Every time an index value is used, the Buffer cursor property for an object is set to the next index, as with Buffer putValue() and Buffer getValue().

SEE: Buffer getValue(), Buffer putValue()

EXAMPLE:
```
var b = new Buffer("ABC");
// b.cursor == 0
Screen.writeln(b[1]); // 66 - ASCII code for "B"
```

```
// now b.cursor == 2
b[0] = 68; // "DBC"
// now b.cursor == 1
```

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 Buffer unicode property of this buffer.

bigEndian - numeric description of the initial state of the Buffer bigEndian property of this buffer.

RETURN: object - the new Buffer object created.

DESCRIPTION: To create a Buffer object, use syntax as shown by the following:

```
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 Buffer cursor location and the Buffer size.

All of the Buffer construction calls above may be done without the `new` constructor starting with ScriptEase 5.00.

SEE: Blob object

Buffer getString()

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

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

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

The Buffer cursor property is updated to the position after the string returned.

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 compare()

SYNTAX: `buffer.compare(buffer2)`

WHERE: buffer2 - buffer to compare against.

RETURN: negative if buffer < buffer2,
zero if buffer == buffer2
positive if buffer > buffer2

DESCRIPTION: This function is identical to calling `Buffer.compare(buffer,buffer2).`

SEE: Buffer(), Buffer.compare (), Buffer.equal (), Buffer equal()

EXAMPLE:

```
// The following code:
var bufa = Buffer("ab");
var bufb = new Buffer("ab\0");
var cmp = bufa.compare(bufb)
// will set the variable cmp to be < 0
```

Buffer equal()

SYNTAX:	`buffer.equal(buffer2)`
WHERE:	buffer2 - buffer to compare against.
RETURN:	true if two buffers are equal, else false
DESCRIPTION:	This function is identical to calling `Buffer.equal(buffer,buffer2).`
SEE:	Buffer(), Buffer equal(), Buffer.compare (), Buffer compare()

EXAMPLE:

```
// The following code:
var bufa = Buffer("ab");
var bufb = new Buffer("ab\0");
var cmp = bufa.equal(bufb)
// will set the variable cmp to be false
```

Buffer getValue()

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

EXAMPLE:

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

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

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

Buffer putString()

SYNTAX:	`buffer.putString(string)`
WHERE:	string - Any string.

RETURN: void.

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

EXAMPLE:

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

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

Buffer putValue()

SYNTAX: `buffer.putValue(value[, valueSize[, valueType]])`

WHERE: value - the value, a number, to be put into the buffer at the position indicated by the `cursor` property.

valueSize - a positive number describing the number of bytes to be used and defaults to 1. The following are acceptable values: 1,2,3,4,8, and 10

valueType - One of the following types: "`signed`", "`unsigned`", or "`float`". The default type is: "`signed`."

RETURN: void

The `value` is put into buffer at the current `cursor` position, and the cursor value is automatically incremented by the size of the value to reflect this addition.

DESCRIPTION: This method puts the specified `value` into a buffer. The value must be a number. The parameter `valueSize` or both `valueSize` and `valueType` may be passed as additional parameters. The parameter `valueSize` is a positive number describing the number of bytes to be used and defaults to 1. Acceptable values for `valueSize` are 1, 2, 3, 4, 8, and 10, providing that it does not conflict with the optional `valueType` flag. (See listing below.)

The parameter `valueType` must be one of the following: "`signed`", "`unsigned`", or "`float`". It defaults to "`signed`." The `valueType` parameter describes the type of data to be read. Combined with `valueSize`, any type of data can be put. The following list describes the acceptable combinations of `valueSize` and `valueType`:

valueSize	valueType
1	signed, unsigned
2	signed, unsigned
3	signed, unsigned
4	signed, unsigned, float
8	float

```
              10          float   (Not supported on every system)
```

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

SEE: Buffer getValue(), Buffer[] Array

EXAMPLE:

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

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

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

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

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

Buffer subBuffer()

SYNTAX: `buffer.subBuffer(begin, end)`

WHERE: begin - start of offset

end - end of offset (up to but not including this point)

RETURN: object - another Buffer object consisting of the data between the positions specified by the parameters: `begin` and `end`.

DESCRIPTION: If the parameter `begin` 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. 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 position 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"
```

Buffer object static methods

Buffer.compare ()

SYNTAX: `Buffer.compare(buffer1,buffer2)`

WHERE: buffer1,buffer2 - Two buffer objects to be compared.

RETURN: negative if buffer1 < buffer2,
zero if buffer1 == buffer2
positive if buffer1 > buffer2

DESCRIPTION: There is no insensitive compare, Buffer.comparei(), since buffers are not strings.To compare buffers as strings, use the toString() or valueOf() methods and make string comparisons. By using these methods, ASCII and Unicode buffers, holding string data, may be compared.

SEE: Buffer(), Buffer compare(), Buffer.equal (), Buffer equal()

EXAMPLE:

```
// The following code:
var bufa = Buffer("ab");
var bufb = new Buffer("ab\0");
var cmp = Buffer.compare(bufa,bufb)
// will set the variable cmp to be < 0
```

Buffer.equal ()

SYNTAX: `Buffer.equal(buffer1,buffer2)`

WHERE: buffer1,buffer2 - Two buffer objects to be compared.

RETURN: true if two buffers are equal, else false

DESCRIPTION: There is no insensitive compare, Buffer equali(), since buffers are not strings. To compare buffers as strings, use the toString()

or valueOf() methods and make string comparisons. By using these methods, ASCII and Unicode buffers, holding string data, may be compared.

SEE: Buffer(), Buffer equal(), Buffer.compare (), Buffer compare()

EXAMPLE:

```
// The following code:
var bufa = Buffer("ab");
var bufb = new Buffer("ab\0");
var cmp = Buffer.equal(bufa,bufb)
// will set the variable cmp to be false
```

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 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 one 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 `Clib.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 Clib.kbhit().

SEE: Clib.getch()

Clib.gets()

SYNTAX: `Clib.gets(str)`

RETURN: str - buffer to hold the same string that is returned.

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.fputc()

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.puts()

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 `Clib.printf()` except that it takes a variable argument list using valist.

See Clib.printf() and Clib.va_start() for more information. The method Clib.vprintf() returns the number of characters written on success, else a negative number on error.

The example function acts just like a `Clib.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 `Clib.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 `Clib.scanf()` except that it takes a variable argument list. See Clib.scanf() and Clib.va_start() for more information.

The method Clib.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 `Clib.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()

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 Clib.time() function.

Clib.gmtime()

SYNTAX:	`Clib.gmtime(timeInt)`
WHERE:	timeInt - an integer time value.
RETURN:	object - a time object reflecting the value timeInt (as returned by the `Clib.time()`.
DESCRIPTION:	Takes the integer timeInt (as returned by the Clib.time() function) and converts it to a Time object representing the current date and time expressed as Greenwich mean time. See Clib.localtime() for a description of the returned object.
SEE:	Clib.mktime(), Date Object, Date toGMTString()

Clib.localtime()

SYNTAX: `Clib.localtime(timeInt)`

WHERE: timeInt - an integer time value.

RETURN: object - a time object reflecting the value timeInt (as returned by the Clib.time() function).

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.

SEE: Clib.mktime(), Date Object, Date toDateString(), Date toLocaleDateString()

EXAMPLE:

```
// Show today's date
// Return day of the year in USA format
ShowToday()
{
      // get current time structure
   var tm = Clib.localtime(Clib.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 Clib.localtime()) to the time format returned by Clib.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 `Clib.localtime()` converts from a time integer to a Time object, `Clib.mktime()` converts from a Time object to a time integer.

Clib.strftime()

SYNTAX: `Clib.strftime(str, formatString, timeObj)`

WHERE: str - 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 str. The parameter formatString describes what the string will look like, and timeObj is a time object as returned by Clib.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. `Clib.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(function)`

WHERE: function - a function to be called when a script is exited. Use the actual function name or ID and not a string.

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.

SEE: Clib.exit()

EXAMPLE:

```
Screen.writeln("Starting the script");

Clib.atexit(Finished);
/*
   Not:
   Clib.atexit("Finished");
*/

function Finished()
{
   Screen.writeln("Exiting the script");
} // Finished
```

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 `Clib.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 Clib.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 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 information about the error that is more specific. ScriptEase

	implements errno as a macro to the internal function `_errno()`. This property can be accessed with Clib.perror() or Clib.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 Clib.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 be opened.

RETURN: number - a file pointer to the file opened, `null` in case of failure.

DESCRIPTION: This method opens the file specified by filename for file operations specified by mode, returning a file pointer to the file opened. `null` is returned in case of failure.

The parameter filename is a string. It may be any valid file name, excluding wildcard characters.

The parameter mode is a string composed of one or more of the following characters. For example, "`r`" or "`rt`"

- `r`
  open file for reading; file must already exist
- `w`
  open file for writing; create if doesn't exist; if file exists then truncate to zero length
- `a`
  open file for append; create if doesn't exist; set for writing at end- of- file
- `b`
  binary mode; if b is not specified then open file in text mode (end- of- line translation)
- `t`

text mode

- +
  open for update (reading and writing)

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

SEE: Clib.fclose(), 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 Clib.fopen(). This method returns an integer which is non-zero if the file cursor is at the end of the file, and 0 if it is NOT at the end of the file.

SEE: Clib.fopen()

Clib.fflush()

SYNTAX: `Clib.fflush(filePointer)`

WHERE: filePointer - pointer to file to use.

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

Clib.fgetc()

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

SEE:	Clib.fgetc()

Clib.fprintf()

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

filePointer - pointer to file to use.

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

DESCRIPTION: This method closes the file associated with filePointer, ignoring any close errors, opens filename according to mode, as with Clib.fopen(), and reassociates filePointer with the new file specification. 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 Clib.fgetpos() on the same open file. Returns zero for success, otherwise returns non- zero and stores an error value in Clib.errno.
SEE:	Clib.fseek()

Clib.ftell()

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

Clib.fwrite()

SYNTAX:	`Clib.fwrite(srcVar, varDescription, filePointer)`
WHERE:	srcVar - variable to hold data to write to file. varDescription - description of the data to write, that is, how and how much. filePointer - pointer to file to use.
RETURN:	number - elements written on success, else 0 if a write error

occurs.

DESCRIPTION: This method writes the data in srcVar to the file indicated by filePointer and returns the number of elements written. 0 will be returned if a write error occurs. Use Clib.ferror() to get more information about the error. varDescription is a variable that describes the how and how much data is to be read. If srcVar is a buffer, it will be the length of the buffer. If srcVar is an object, varDescription must be an object descriptor. If srcVar is to hold a single datum then varDescription must be one of the values listed in the description for Clib.fread().

The ScriptEase version of `Clib.fwrite()` differs from the standard C version in that the standard C library is set up for writing arrays of numeric values or structures from consecutive bytes in memory. This is not necessarily the case in JavaScript.

SEE: Clib.fread()

EXAMPLE:

```
// To write the 16_bit integer "i",
// the 32_bit float "f", and
// then 10_byte buffer "buf" into open file "fp",
// use 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()

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

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 the 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 it 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 it 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: string - 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 it 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 Clib.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 Clib.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 an 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 String length is that `length` counts `null` characters as part of a string, whereas `Clib.strlen()` considers them 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 the terminating `null` byte is reached. 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 the terminating `null` byte is reached. 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 the terminating `null` byte is reached. 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:

function main()
{
   var Phrase = CString("To be or not to be? 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!
//  be or not to be? beep!
//  be? 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.strupr(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: Clib.toascii(), Clib.tolower(), Clib.toupper(), String toLowerCase(), String toLowerCase(), String invertCase()

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: Clib.toascii(), Clib.tolower(), Clib.toupper(), String toLowerCase(), String toLowerCase(), String invertCase()

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:	Clib.toascii(), Clib.tolower(), Clib.toupper(), String toLowerCase(), String toLowerCase(), String invertCase()

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:	Clib.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.ldexp().
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(mantissa, exp)`
WHERE:	mantissa - 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 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 property arguments[] 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 - an optional initial parameter for the variable parameter list.

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()

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()

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. See Date() for a summary.

```
var aDate = new Date(milliseconds);
var bDate = new Date(datestring);
var cDate = new Date(year, month, day);
var dDate = new Date(year, month, day, hour, minute, second,
millisecond);
```

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.

- **hour**
  An hour is specified as a number from 0 to 23. Midnight is 0, and 11 p.m. is 23.
- **minute**
  A minute is specified as a number from 0 to 59. The first minute of an hour is 0, and the last is 59.
- **second**
  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 Date getDate() method, the call would be: `aDate.getDate()`. Static methods have "`Date.`" at their beginnings since these methods are called with literal calls, such as `Date.parse()`. These methods are part of the Date object itself instead of instances of the Date object.

Date object instance methods

Date()

SYNTAX:

```
new Date()
new Date(milliseconds)
new Date(string)
new Date(year, month[, day[, hour[,
          minute[, second[, millisecond]]]]])
```

WHERE:

milliseconds - number of milliseconds since midnight January 1, 1970 GMT, as returned by Date.parse().

string - a string with date information. The string should be in the following format: `Friday, October 31, 1998 15:30:00 GMT`, or a substring of this format. The string accepted by `Date()` is the same as for Date.parse().

year - four digit year, see Date setYear(). If `year` is passed alone, it is recognized as `milliseconds`.

month - number, 0 - 11, month of year, see Date setMonth().

day - number, 1 - 31, day of month, see Date setDate().

hour - number, 0 - 24, hour of day, see Date setHours().

minute - number, 0 - 59, minute of hour, see Date setMinutes().

second - number, 0 - 59, second of minute, see Date setSeconds().

millisecond - number, 0 - 999, millisecond of second, see Date setMilliseconds().

RETURN: object - a Date object set according to the arguments passed. If no arguments are passed, then the current date and time are set.

DESCRIPTION: ScriptEase JavaScript has a rich set of methods for working with dates and times. The JavaScript Date object is a variable type that is different from the Clib date and time methods. The Date.fromSystem() and Date toSystem() methods allow conversion from and to the C style methods. See the Date Object for a complete description of the Date() function.

If the `new` operator is used, for example, `new Date(1999, 2)`, then a Date object is created using any parameters passed to the `Date()` constructor. However, if the `new` operator is not used, then all parameters are ignored and `Date()` returns a string representation of the current date and time, for example, "Wed Sep 4 11:54:16 2002".

SEE: Date Object, Date toSystem(), Date.fromSystem(), Date object instance methods, Date object static methods, Clib.time(), Clib.gmtime(), Clib.localtime(), Clib.mktime()

EXAMPLE:

```
var d = new Date()    // date in a Date object
// d == Mon Aug 20 16:29:53 2001 // ie, current
// typeof(d) == object
// d._class == Date
var d = Date()        // date as a String
// d == Mon Aug 20 16:29:53 2001 // ie, current
// typeof(d) == string
// d._class == String
var d = new Date().getDay()
// d == 1 // which is Monday
```

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 milliseconds from midnight on January 1, 1970,

GMT, to the date and time specified by a Date object.

Date getTimezoneOffset()

SYNTAX: `date.getTimezoneOffset()`
RETURN: number - minutes.

DESCRIPTION: This method returns the difference, in minutes, between Greenwich Mean Time (GMT) and local time.

Date getUTCDate()

SYNTAX: `date.getUTCDate()`
RETURN: number - a day of a month.

DESCRIPTION: This method returns the UTC day of the month, as a number from 1 to 31, of a Date object. The first day of a month is 1, and the last is 28, 29, 30, or 31.

Date getUTCDay()

SYNTAX: `date.getUTCDay()`
RETURN: number - a day in a week.

DESCRIPTION: This method returns the day of the week, as a number from 0 to 6, of a Date object. Sunday is 0, and Saturday is 6.

Date getUTCFullYear()

SYNTAX: `date.getUTCFullYear()`
RETURN: number - four digit year.

DESCRIPTION: This method returns the UTC year, as a number with four digits, of a Date object.

Date getUTCHours()

SYNTAX: `date.getUTCHours()`
RETURN: number - an hour in a day.

DESCRIPTION: This method returns the UTC hour, as a number from 0 to 23, of a Date object. Midnight is 0, and 11 p.m. is 23.

Date getUTCMilliseconds()

SYNTAX: `date.getUTCMilliseconds()`
RETURN: number - a millisecond in a second.

DESCRIPTION: This method returns the UTC millisecond, as a number from 0 to 999, of a Date object. The first millisecond in a second is 0, and the last is 999.

Date getUTCMinutes()

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

Date getUTCMonth()

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

Date getUTCSeconds()

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

Date getYear()

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

Date setDate()

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

Date setFullYear()

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

The parameter day is the same as for Date setDate().

Date setHours()

SYNTAX:
```
Date.setHours(hour[, minute[, second[,
                    millisecond]]])
```

WHERE: hour - an hour in a day.

minute - a minute in an hour.

second - a second in a minute.

millisecond - a millisecond in a second.

RETURN: number - time in milliseconds as set.

DESCRIPTION: This method sets the hour, as a number from 0 to 23, of a Date object to the parameter hours. Midnight is 0, and 11 p.m. is 23.

The parameter minute is the same as for Date setMinutes().

The parameter second is the same as for Date setSeconds().

The parameter milliseconds is the same as for Date setMilliseconds().

Date setMilliseconds()

SYNTAX:
```
date.setMilliseconds(millisecond)
```

WHERE: millisecond - a millisecond in a minute.

RETURN: number - time in milliseconds as set.

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

Date setMinutes()

SYNTAX:
```
date.setMinutes(minute[, second[, millisecond]])
```

WHERE: minute - a minute in an hour.

second - a second in a minute.

millisecond - a millisecond in a second.

RETURN: number - time in milliseconds.

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

The parameter second is the same as for Date setSeconds().

The parameter milliseconds is the same as for Date setMilliseconds().

Date setMonth()

SYNTAX:
```
Date.setMonth(month[, day])
```

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

Date setSeconds()

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

Date setTime()

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

Date setUTCDate()

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

Date setUTCFullYear()

SYNTAX:	`date.setUTCFullYear(year[, month[, date]])`
WHERE:	year - a four digit year. month - a month in a year. day - a day in a month.

RETURN:	number - time in milliseconds as set.
DESCRIPTION:	This method sets the UTC year of a Date object to the parameter year. The parameter year is expressed with four digits. The parameter month is the same as for Date setUTCMonth(). The parameter day is the same as for Date setUTCDate().

Date setUTCHours()

SYNTAX:
```
Date.setUTCHours(hour[, minute[, second[,
                          millisecond]]])
```

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

Date setUTCMilliseconds()

SYNTAX:
```
date.setUTCMilliseconds(millisecond)
```

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

Date setUTCMinutes()

SYNTAX:
```
date.setUTCMinutes(minute[, second[,
                             millisecond]])
```

WHERE:	minute - a minute in an hour. second - a second in a minute. millisecond - a millisecond in a second.
RETURN:	number - time in milliseconds.
DESCRIPTION:	This method sets the UTC minute, as a number from 0 to 59, of a Date object to the parameter minute. The first minute of an hour is 0, and the last is 59.

The parameter second is the same as for Date setUTCSeconds().

The parameter milliseconds is the same as for Date setUTCMilliseconds().

Date setUTCMonth()

SYNTAX: `Date.setUTCMonth(month[, day])`

WHERE: month - a month in a year.

day - a day in a month.

RETURN: number - time in milliseconds.

DESCRIPTION: This method sets the UTC month, as a number from 0 to 11, of a Date object to the parameter month. January is 0, and December is 11.

The parameter day is the same as for Date setUTCDate().

Date setUTCSeconds()

SYNTAX: `date.setUTCSeconds(second[, millisecond])`

WHERE: second - a second in a minute.

millisecond - a millisecond in a second.

RETURN: number - time in milliseconds.

DESCRIPTION: This method sets the UTC second, as a number from 0 to 59, of a Date object to the parameter second. The first second of a minute is 0, and the last is 59.

The parameter milliseconds is the same as for Date setUTCMilliseconds().

Date setYear()

SYNTAX: `date.setYear(year)`

WHERE: year - four digit year, unless in the 1900s in which case it may be a two digit year.

RETURN: number - time in milliseconds as set.

DESCRIPTION: This method sets the year of a Date object to the parameter year. The parameter year may be expressed with two digits for a year in the twentieth century, the 1900s. Four digits are necessary for any other century.

Date toDateString()

SYNTAX: `date.toDateString()`

RETURN: string - representation of the date portion of the current object.

DESCRIPTION: Returns the Date portion of the current date as a string. This string is formatted to read "Month Day, Year", for example, "May 1, 2000". This method uses the local time, not UTC time.

SEE: Date toString(), Date toTimeString(), Date toLocaleDateString()

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

Date toGMTString()

SYNTAX: `date.toGMTString()`

RETURN: string - string representation of the GMT date and time.

DESCRIPTION: This method converts a Date object to a string, based on Greenwich Mean Time.

EXAMPLE:
```
var d = new Date();
Screen.writeln(d.toGMTString());

// The fragment above would produce something like:
// Mon May 1 15:48:38 2000 GMT
```

Date toLocaleDateString()

SYNTAX: `date.toLocaleDateString()`

RETURN: string - locale-sensitive string representation of the date portion of the current date.

DESCRIPTION: This function behaves in exactly the same manner as Date toDateString(). This function is designed to take in the current locale when formatting the string. Locale reflects the time zone of a user.

SEE: Date toString(), Date toLocaleTimeString(), Date toLocaleString()

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

Date toLocaleString()

SYNTAX: `date.toLocaleString()`

RETURN: string - locale-sensitive string representation of the current date.

DESCRIPTION: This function behaves in exactly the same manner as Date toString(). This function is designed to take in the current locale when formatting the string, though this functionality is currently unimplemented. Locale reflects the time zone of a user.

SEE: Date toString(), Date toLocaleTimeString(), Date toLocaleDateString()

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

Date toLocaleTimeString()

SYNTAX: `date.toLocaleTimeString()`

RETURN: string - locale-sensitive string representation of the time portion of the current date.

DESCRIPTION: This function behaves in exactly the same manner as Date toTimeString(). This function is designed to take in the current

	locale when formatting the string. Locale reflects the time zone of a user.

Date toString()

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

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

Date toSystem()

SYNTAX:	`date.toSystem()`
RETURN:	number - the Date object date and time value converted to the system date and time.
DESCRIPTION:	This method converts a Date object to a system time format which is the same as that returned 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 `Clib.time()`, 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 should be in the following format: `Friday, October 31, 1998 15:30:00`, or a substring of this format. The full format is returned by the Date toGMTString() method, by email and by Internet applications. The day of the week, time zone, time specification or seconds field may be omitted.

SEE: Date object, Date setTime(), Date toGMTString(), Date.UTC

EXAMPLE:

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

```
//is equivalent to:
var theDate = new Date(datestring);
// The following are valid, but not exhaustive
var ms;
ms = Date.parse(new Date().toGMTString());
ms = Date.parse("Mon Aug 20 14:41:01 2001 GMT");
ms = Date.parse("Mon Aug 20 14:41:01 2001");
ms = Date.parse("Mon Aug 20 14:41:01 2001");
ms = Date.parse("August 20 09:35:50 2001");
ms = Date.parse("Aug 20 09:35:50 2001");
ms = Date.parse("August 20, 2001");
ms = Date.parse("August 20 2001");
ms = Date.parse("Aug 20, 2001");
ms = Date.parse("Aug 20 2001");
```

Date.UTC()

SYNTAX:

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

WHERE:

year - A year, represented in four or two-digit format after 1900. NOTE: For year 2000 compliance, this year MUST be represented in four-digit format

month - A number between 0 (January) and 11 (December) representing the month

day - A number between 1 and 31 representing the day of the month. Note that `Month` uses 1 as its lowest value whereas many other arguments use 0

hours - A number between 0 (midnight) and 23 (11 PM) representing the hours

minutes - A number between 0 (one minute) and 59 (59 minutes) representing the minutes. This is an optional argument which may be omitted if `Seconds` and `Minutes` are omitted as well.

seconds - A number between 0 and 59 representing the seconds. This parameter is optional.

milliseconds - A number between 0 and 999 which represents the milliseconds. This is an optional parameter.

RETURN: number - milliseconds from midnight, January 1, 1970, to the date and time specified.

DESCRIPTION: The method interprets its parameters as a date. The parameters are interpreted as referring to Greenwich Mean Time (GMT).

SEE: Date object, Date.parse(), Date setTime()

EXAMPLE:

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

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 SElib.peek() and SElib.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 - an object/structure with properties/elements that correspond to the registers of an 8086 processor. The registers will be set to these values when the method is called.

regOut - an object/structure with properties/elements that will be set to the corresponding registers of the processor when the function is exited.

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 and 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()

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 are required. Note that the similar method Function call() can receive the same arguments as a list. Compare the following ways of passing arguments:

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

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

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

SEE: Function(), Function call()

EXAMPLE:

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

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

Function call()

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

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

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

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

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

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

SEE: Function(), Function.apply()

EXAMPLE:

```
// The following code:

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

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

Function toString()

SYNTAX: `function.toString()`

RETURN: string - a representation of the function.

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

EXAMPLE:

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

// This fragment will print the following
// to the screen:

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

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.

x - y coordinate of the point.

RETURN: number - an implementation-dependent approximation to the arc tangent of the quotient, `y/x`, of the arguments `y` and `x`, where the signs of the arguments are used to determine the quadrant of the result.

DESCRIPTION: It is intentional and traditional for the two-argument arc tangent function that the argument named `y` be first and the argument named `x` be second. The return value is expressed in radians and ranges from `-pi` to `+pi`.

EXAMPLE:

```
//The arctangent of the quotient y/x
//is returned in the
//following function:
function compute_arctangent_of_quotient(x, y)
{
   return Math.arctangent2(x, y)
}
```

Math.ceil()

SYNTAX: `Math.ceil(x)`

WHERE: x - any number or numeric expression.

RETURN: number - the smallest number that is not less than the argument and is equal to a mathematical integer.

DESCRIPTION: If the argument is already an integer, the result is the argument itself. Returns `NaN` if `x` cannot be converted to a number.

EXAMPLE:

```
//The smallest number that is
//not less than the argument and is
//equal to a mathematical integer is returned
//in the following function:
function compute_small_arg_eq_to_int(x)
{
   return Math.ceil(x)
}
```

Math.cos()

SYNTAX: `Math.cos()`

WHERE: x - an angle, measured in radians.

RETURN: number - an implementation-dependent approximation of the cosine of the argument

DESCRIPTION: The argument is expressed in radians. Returns `NaN` if `x` cannot be converted to a number. In order to convert degrees to radians you must multiply by `2pi/360`.

EXAMPLE:

```
//The cosine of x is returned
//in the following function:
function compute_cos(x)
{
   return Math.cos(x)
}
```

Math.exp()

SYNTAX: `Math.exp(x)`

WHERE: x - either a number or a numeric expression to be used as an exponent

RETURN: number - an implementation-dependent approximation of the exponential function of the argument.

DESCRIPTION: For example returns e raised to the power of the `x`, where e is the base of the natural logarithms. Returns `NaN` if `x` cannot be converted to a number.

EXAMPLE:
```
//The exponent of x is returned
//in the following function:
function compute_exp(x)
{
   return Math.exp(x)
}
```

Math.floor()

SYNTAX: `Math.floor(x)`

WHERE: x - a number.

RETURN: number - the greatest number value that is not greater than the argument and is equal to a mathematical integer.

DESCRIPTION: If the argument is already an integer, the return value is the argument itself.

EXAMPLE:
```
//The floor of x is returned
//in the following function:
function compute_floor(x)
{
   return Math.floor(x)
}
//If 6.78 is passed to compute_floor,
//7 will be returned. If 89.1
//is passed, 90 will be returned.
```

Math.log()

SYNTAX: `Math.log(x)`

WHERE: x - a number.greater than zero.

RETURN: number - an implementation-dependent approximation of the natural logarithm of `x`.

DESCRIPTION: If a negative number is passed to `Math.log()`, the return is `NaN`

EXAMPLE:
```
//The natural log of x is returned
//in the following function:
function compute_log(x)
{
   return Math.log(x)
}
//If the argument is less than 0 or NaN,
//the result is NaN
//If the argument is +0 or -0,
//the result is -infinity
//If the argument is 1, the result is +0
//If the argument is +infinity,
//the result is +infinity
```

Math.max()

SYNTAX: `Math.max(x, y)`

WHERE: x - a number.

y - a number.

RETURN: number - the larger of `x` and `y`.

DESCRIPTION: Returns `NaN` if either argument cannot be converted to a number.

EXAMPLE:
```
//The larger of x and y is returned
//in the following function:
function compute_max(x, y)
{
   return Math.max(x, y)
}
//If x = a and y = 4 the return is NaN
//If x > y the return is x
//If y > x the return is y
```

Math.min()

SYNTAX: `Math.min(x, y)`

WHERE: x - a number.

y - a number.

RETURN: number - the smaller of `x` and `y`. Returns `NaN` if either argument cannot be converted to a number.

DESCRIPTION: Returns `NaN` if either argument cannot be converted to a number.

EXAMPLE:
```
//The smaller of x and y is returned
//in the following function:
function compute_min(x, y)
{
   return Math.min(x, y)
}
//If x = a and y = 4 the return is NaN
//If x > y the return is y
//If y > x the return is x
```

Math.pow()

SYNTAX: `Math.pow(x, y)`

WHERE: x - The number which will be raised to the power of `Y`

y - The number which `X` will be raised to

RETURN: number - the value of `x` to the power of `y`.

DESCRIPTION: If the result of `Math.pow()` is an imaginary or complex number, `NaN` will be returned. Please note that if `Math.pow()` unexpectedly returns infinity, it may be because the floating-point value has experienced overflow.

EXAMPLE:
```
//x to the power of y is returned
//in the following function:
function compute_x_to_power_of_y(x, y)
{
   return Math.pow(x, y)
}
//If the result of Math.pow is
//an imaginary or complex number,
//the return is NaN
```

```
//If y is NaN, the result is NaN
//If y is +0 or -0, the result is 1,
//even if x is NaN
//If x = 2 and y = 3 the return value is 8
```

Math.random()

SYNTAX: `Math.random()`

RETURN: number - a number which is positive and pseudo-random and which is greater than or equal to 0 but less than 1.

DESCRIPTION: Calling this method numerous times will result in an established pattern (the sequence of numbers will be the same each time). This method takes no arguments. Seeding is not yet possible.

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
```

Number Object

`platform: All OS, All version of SE`

Number object instance methods

Number toExponential()

SYNTAX:	`number.toExponential(fractionDigits)`
WHERE:	fractionDigits - the digits after the significand's decimal point.
RETURN:	string - A string representation of this number in exponential notation.
DESCRIPTION:	This method returns a string containing the number represented in exponential notation with one digit before the significand's decimal point and fractionDigits digits after the significand's decimal point.

Number toFixed()

SYNTAX:	`number.toFixed(fractionDigits)`
WHERE:	fractionDigits - the digits after the decimal point.
RETURN:	string - A string representation of this number in fixed-point notation.
DESCRIPTION:	This method returns a string containing the number represented in fixed-point notation with fractionDigits digits after the decimal point.

Number toLocaleString()

SYNTAX:	`number.toLocaleString()`
RETURN:	string - a string representation of this number.
DESCRIPTION:	This method behaves like Number toString() and converts a number to a string in a manner specific to the current locale. Such things as placement of decimals and comma separators are affected.
SEE:	Number toString()

EXAMPLE:	`var n = 8.9;`
	`var s = n.toLocaleString();`

Number toPrecision()

SYNTAX:	`number.toPrecision(precision)`
WHERE:	precision - significant digits in fixed notation, or digits after the significand's decimal point in exponential notation.
RETURN:	string - A string representation of this number in either exponential notation or in fixed notation.
DESCRIPTION:	This method returns a string containing the number represented in either in exponential notation with one digit before the

	significand's decimal point and precision-1 digits after the significand's decimal point or in fixed notation with precision significant digits.

Number toString()

SYNTAX:	`number.toString([radix])`
WHERE:	radix - an optional radix, base number, determining the string representation of this number.
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. If the `radix` is specified, the string will have digits representing that number base.
SEE:	Number toLocaleString()

EXAMPLE:

```
var n = 8.9;
var s = n.toString(); // "8.9"

var a = 16;
Screen.writeln(a.toString());   //"16"
Screen.writeln(a.toString(10)); //"16"
Screen.writeln(a.toString(16)); //"10"
Screen.writeln(a.toString(15)); //"11"
Screen.writeln(a.toString(8));  //"20"
Screen.writeln(a.toString(2));  //"10000"
```

Object Object

```
platform: All OS, All version of SE
```

Object object instance methods

Object()

SYNTAX: `new Object([value])`

WHERE: value - a value or variable, usually a primitive, to convert to an object.

RETURN: object - a new top level object.

DESCRIPTION: Create a new top level object. I a `value` is passed, convert the `value` to an object, else create a new object. The examples below illustrate several ways to use `Object()` and how to accomplish similar things using different strategies.

If `Object()` is invoked as a function instead of as a constructor (that is, without `new`), it performs a type conversion on `value`. That is, it returns `value` as data type `Object`.

SEE: Internal objects

EXAMPLE:

```
/***************************************
First we create var s as a string data type.
Then we will convert s to an object data type,
namely, to a String object.
***************************************/
// Create s as data type string
var s = 'my string';
// Display the data type 'string'
Screen.writeln(typeof s);
// Display 'my string'
Screen.writeln(s);

// Convert s to data type object (String object)
var o = new Object(s);
// Display the data type 'object'
Screen.writeln(typeof o);
// Display 'my string'
Screen.writeln(o);

/***************************************
Next we create var so as a String object --
in one statement.
***************************************/
// Create so as a String object in one statement
var so = new String('my string');
// Display the data type 'object'
Screen.writeln(typeof so);
// Display 'my string'
Screen.writeln(so);

/***************************************
Next we create var o as an Object object data type.
We add the property o.mystring.
Note that the property may be accessed as:
   o.mystring
   or
   o['mystring']
***************************************/
// Create a new top level object
var o = new Object();
// Add the property mystring
o.mystring = 'my string';
// Display the data type 'object'
Screen.writeln(typeof o);
// Display 'my string'
Screen.writeln(o.mystring);
// Display 'my string'
Screen.writeln(o['mystring']);
```

Object hasOwnProperty()

SYNTAX: `object.hasOwnProperty(propertyName)`

WHERE: property - a string with the 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 determines if the object has a property with the name propertyName. To return `true`, the property must be an instance property created for this instance of an object and may not be an inheritable or prototype property. 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.

EXAMPLE:

```
function Atest()
{
   this.name = "";
} // Test

Atest.prototype.city = "Fort Worth";

var t = new Atest();

Screen.writeln(t.city);                     // Fort
Worth

Screen.writeln(t.hasOwnProperty("name"));  // true
Screen.writeln(t.hasOwnProperty("city"));  // false
```

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.

EXAMPLE:

```
function Atest()
{
   this.name = "";
} // Test

Atest.prototype.city = "Fort Worth";

var t = new Atest();

// Fort Worth
Screen.writeln(t.city);

// true
Screen.writeln(t.propertyIsEnumerable("name"));
// true
Screen.writeln(t.propertyIsEnumerable("city"));
```

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 as if `toString()` had been called on the original object.

SEE: Object toString()

Object toSource()

SYNTAX: `object.toSource()`

RETURN: string - a string representation of this object, which can be evaluated or interpreted.

DESCRIPTION: An object may be represented by a string comprised of JavaScript statements which, when evaluated or interpreted, reproduce the object. The source string may be evaluated by global.eval() or by SElib.interpret(). It is sometimes convenient or powerful to use source strings, for example, in the Data object the DSP object.

Though the source string may be read by humans, it is daunting.

Remember, `toSource()` is designed for interpretation by the ScriptEase interpreters, not by users.

The example below compares source strings created by the Object `toSource()` method and the global.ToSource() function. In these examples, the source strings are identical, which is not guaranteed always to be so. But, no matter which one is used, the source strings can be evaluated or interpreted.

SEE: Global.ToSource(), global.eval(), SElib.interpret()

EXAMPLE:

```
// An Array
var a = [1, '2', 3];

Screen.writeln(a.toSource());
Screen.writeln();
Screen.writeln(ToSource(a));
Screen.writeln();
/********************************
Displays:

((new Function("var tmp1 = [1,\"2\",3]; tmp1[\"0\"] =
1;
tmp1[\"1\"] = \"2\"; tmp1[\"2\"] = 3; return
tmp1;"))())

((new Function("var tmp1 = [1,\"2\",3]; tmp1[\"0\"] =
1;
tmp1[\"1\"] = \"2\"; tmp1[\"2\"] = 3; return
tmp1;"))())
********************************/

// An Object
var o = {one:1, two:'2', three:3};

Screen.writeln(o.toSource());
Screen.writeln();
Screen.writeln(ToSource(o));
Screen.writeln();
/********************************
Displays:

((new Function("var tmp1 = new Object();
tmp1[\"three\"] = 3;
tmp1[\"one\"] = 1; tmp1[\"two\"] = \"2\"; return
tmp1;"))())

((new Function("var tmp1 = new Object();
tmp1[\"three\"] = 3;
tmp1[\"one\"] = 1; tmp1[\"two\"] = \"2\"; return
tmp1;"))())
********************************/
```

Object toString()

SYNTAX: `object.toString()`

RETURN: string - a string representation of this object.

DESCRIPTION: When this method is called, the internal class property, `_class`, is retrieved from the current object. A string is then constructed whose contents are "[object classname]", where classname is the

	value of the property from the current object. Note that this function is rarely called directly, rather it is called implicitly through such functions as global.ToString().
SEE:	Object toLocaleString()

Object valueOf()

SYNTAX:	`object.valueOf()`
RETURN:	object - the value of this object
DESCRIPTION:	Generally, this method returns the object itself. However, if object is a wrapper for a host object, the host object may be returned. Such wrappers for host objects may be created with the Object constructor.
SEE:	Object toSource(), Object()

RegExp Object

Regular expressions do not seem very regular to average people. Regular expressions are used to search text and strings, searches that are very powerful if a person makes the effort to learn how to use them. Simple searches may be done like the following:

```
var str = "one two three";
str.indexOf("two");   // == 4
```

The String indexOf() method searches `str` for "two" and returns the beginning position of "two", which is 4. What if you wanted to find "t" and "o" with or without any characters in between, an "o" only at the beginning of a string, or an "e" only at the end of a string? Before answering, lets consider wildcards.

Most computer users are familiar with wildcards in searching, especially since they may be used in finding files. For example, the DOS command:

```
dir t*o.bat
```

will list all files that begin with "t" and end "o" in the filename and that have an extension of "bat". JavaScript does not use wildcards to extend search capability. Instead, ECMAScript, the standard for JavaScript, has implemented regular expression searches that do everything that wildcards do and much, much more. Regular expressions follow the PERL standard, though the syntax has been made easier to read. Anyone who can use regular expressions in PERL already knows how to use JavaScript regular expressions. For advanced information on regular expressions, there are many books in the PERL community, in addition to JavaScript books, that explain regular expressions.

Now lets answer the question about how to find the three cases mentioned above.

```
var str = "one two three";
var pat = /t.*o/;
str.search(pat);   // == 4
```

This fragment illustrates one way to use regular expressions to find "t" followed by "o" with any number of characters between them. Two things are different. One the variable `pat` which is assigned `/t.*o/`. The slashes indicate the beginning and end of a regular expression pattern, similar to how quotation marks indicate a string. The String search() method is a method of the String object that uses a regular expression pattern to search a string, similar to the String `indexOf()` method. In fact, they both return 4, the start position of "two", in these examples.

The String object has three methods for searching using regular expression patterns. The three methods are:

```
String match()
String replace()
String search()
```

The methods in the RegExp object, for using regular expressions, are explained below in this section. Before we move on to the cases of an "o" at the start or an "e" at the end of a string, consider the current example a little further. What do the slashes "`/ . . . /`" do? First, they define a regular expression pattern. Second, they create a RegExp object. In our example, the quotes cause `str` to be a String object, and the slashes cause `pat` to be a RegExp object. Thus, `pat` may be used with RegExp methods and with the three String methods that use regular expression patterns.

```
var str = "one two three";
var pat = /t.*o/;
pat.test(str);   // == true
```

By using a method, such as `test()`, of the RegExp object, the string to be searched becomes the argument rather than the pattern to search for, as with the string methods. The RegExp test() method simply returns `true` or `false` indicating whether the pattern is found in the string.

```
var str = "one two three";
var pat = /t.*o/;
str.match(pat);   // == an Array with pertinent info
pat.exec(str);    // == an Array with pertinent info
```

The String match() and RegExp exec() methods return very similar, often the same, results in an Array. The return may vary depending on exactly which attributes, discussed later, are set for a regular expression.

To find an "o" only at the start of a string, use something like:

```
var str = "one two three";
var pat = /^o/;
str.search(pat);   // == 0
```

The caret "^" has a special meaning, namely, the start of a string or line. It anchors the characters that follow to the start of a string or line and is one of the special anchor characters.

To find an "e" only at the end of a string, use something like:

```
var str = "one two three";
var pat = /e$/;
str.search(pat);   // == 12
```

The dollar sign "$" has a special meaning, namely, the end of a string or line. It anchors the characters that follow to the end of a string or line and is one of the special anchor characters.

Note that there is a very important distinction between searching for pattern matches using the String methods and using the RegExp methods. The RegExp methods execute much faster, but the String methods are often quicker to program. So, if you need to do intensive searching in which a single regular expression pattern is used many times in a loop, use the RegExp methods. If you just need to do a few searches, use the String methods. Every time a RegExp object is constructed using `new`, the pattern is compiled into a form that can be executed very quickly. Every time a new pattern is compiled using the RegExp compile() method, a pattern executes much faster. Other than the difference in speed and script writing time, the choice of which methods to use depends on personal preferences and the particular tasks at hand.

In general, the RegExp object allows the use of regular expression patterns in searches of strings or text. The syntax follows the ECMAScript standard, which may be thought of as a large and powerful subset of PERL regular expressions.

Regular expression syntax

The general form for defining a regular expression pattern is:

```
/characters/attributes
```

Assume that we are searching the string "THEY, the three men, left". The following are valid regular expression patterns followed by a description of what they find:

```
/the three/      // "the three"
/THE THREE/ig    // "the three"
/th/             // "th" in "the"
/th/igm          // "th" in "THEY", "the", and "three"
```

The slashes delimit the characters that define a regular expression. Everything between the **slashes** is part of a **regular expression**, just as everything between quotation marks is part of a string. Three letters may occur after the second slash that are not part of the regular expression. Instead, they define **attributes** of the regular expression. Any one, two, or three of the letters may be used, that is, any one or more of the attributes may be defined. Thus, a regular expression has three elements: literals, characters, and attributes.

Regular expression literals

Regular expression literals delimit a regular expression pattern. The literals are a slash "/" at the beginning of some characters and a slash "/" at the end of the characters. These regular expression literals operate in the same way as quotation marks do for string literals. The following two lines of code accomplish the same thing, namely, they define and create an instance of a RegExp object:

```
var re = /^THEY/;
var re = new RegExp("^THEY");
```

and so do the following two lines:

```
var re = /^THEY/i;
var re = new RegExp("^THEY", "i");
```

Regular expression characters

Each character or special character in a regular expression represents one character. Though some special characters, such as, the range of lowercase characters represented by `[a-z]`, may have multiple matches, only one at a time is matched. Thus, `[a-z]` will only find one of these 26 characters at one position in a string being searched. Just as strings have special characters, namely, escape sequences, regular expression patterns have various kinds of special characters and metacharacters that are explained below.

Regular expression attributes

The following table lists allowable attribute characters and their effects on a regular expression. No other characters are allowed as attributes.

Character	Attribute meaning
`g`	Do a global match. Allow the finding of all matches in a string using the RegExp and String methods and properties that allow global operations. The instance property `global` is set to `true`. Example: `/pattern/g`

`i`	Do case insensitive matches. The instance property `ignoreCase` is set to `true`. Example: `/pattern/i`
`m`	Work with multiple lines in a string. When working with multiple lines the "`^`" and "`$`" anchor characters will match the start and end of a string and the start and end of lines within the string. The newline character "`\n`" in a string indicates the end of a line and hence lines in a string. The instance property `multiline` is set to `true`. Example: `/pattern/m`

Attributes are the characters allowed after the end slash "/" in a regular expression pattern. The following regular expressions illustrate the use of attributes.

```
 var pat = /^The/i;    // any form of "the" at start of a string
 var pat = /the/g;     // all occurrences of "the" may be found
 var pat = /test$/m;   // first "test" at the end of any line
 var pat = /test$/igm; // all forms of "test" at end of all lines
    // The following four examples do the same as the first four
 var pat = new RegExp("^The",  "i");
 var pat = new RegExp("the",   "g");
 var pat = new RegExp("test$", "m");
 var pat = new RegExp("test$", "igm");
```

Regular expression special characters

Regular expressions have many special characters, which are also known as metacharacters, with special meanings in a regular expression pattern. Some are simple escape sequences, such as, a newline "`\n`", with the same meaning as the same escape sequence in strings. But, regular expressions have many more special characters that add much power to working with strings and text, much more power than is initially recognized by people being introduced to regular expressions. For anyone who works with strings and text, the effort to become proficient with regular expression parsing is more than worthwhile.

Regular expression summary

Search pattern

`?`	zero or one of previous, `{0,1}`	`be?t`
`*`	zero or more of previous, maximal, `{0,}`	`b.*t`
`*?`	zero or more of previous, minimal, `{0,}?`	`b.*?t`
`+`	one or more of previous, maximal, `{1,}`	`b.+t`
`+?`	one or more of previous, minimal, `{1,}?`	`b.+?t`
`{n}`	n times of previous	`be{n}t`
`{n,}`	n or more times of previous, maximal	`b.{n,}t`
`{n,}?`	n or more times of previous, minimal	`b.{n,}?t`
`{n,m}`	n to m times of previous	`be{1,2}t`
`.`	any character	`b.t`
`[]`	any one character in a class	`[a-m]`
`[^]`	any one not in a character class	`[^a-m]`
`[\b]`	one backspace character	`my[\b]word`

`\d`	any one digit, `[0-9]`	`file\d`
`\D`	any one not digit, `[^0-9]`	`file\D`
`\s`	any one white space character, `[ \t\n\r\f\v]`	`my\sword`
`\S`	any one not white space character, `[^ \t\n\r\f\v]`	`my \sord`
`\w`	any one word character, `[a-zA-Z0-9_]`	`my big\w`
`\W`	any one not word character, `[^a-zA-Z0-9_]`	`my\Wbig`
`^`	anchor to start of string	`^string`
`$`	anchor to end of string	`string$`
`\b`	anchor to word boundary	`\bbig`
`\B`	anchor to not word boundary	`\Bbig`
`|`	or	`(bat)|(bet)`
`\n`	group n	`(bat)a\1`
`()`	group	`my(.?)fil`
`(?:)`	group without capture	`my(?:.?)fil`
`(?=)`	group without capture with positive look ahead	`my(?=.?)fil`
`(?!)`	group without capture with negative look ahead	`my(?!.?)fil`
`\f`	form feed character	`string\f`
`\n`	newline	`string\n`
`\r`	carriage return character	`string\r`
`\t`	horizontal tab character	`one\tfour`
`\v`	vertical tab character	`one\vtwo`
`\/`	`/` character	`\/fil`
`\\`	`\` character	`\\fil`
`\.`	`.` character	`fil\.bak`
`\*`	`*` character	`one\*two`
`\+`	`+` character	`\+fil`
`\?`	`?` character	`when\?`
`\\|`	`|` character	`one\\|two`
`\(`	`(` character	`\(fil\)`
`\)`	`)` character	`\(fil\)`
`\[`	`[` character	`\[fil\]`
`\]`	`]` character	`\[fil\]`
`\{`	`{` character	`\{fil\}`
`\}`	`}` character	`\{fil\}`
`\C`	a character itself. Seldom used.	`b\at`
`\cC`	a control character	`one\cIfour`
`\x##`	character by hexadecimal code	`\x41`
`\###`	character by octal code	`\101`

Replace pattern

`$n`	group n in search pattern, `$1, $2, . . . $9`	`big$1`
`$+`	last group in search pattern	`big$+`
`` $` ``	text before matched pattern	`` big$` ``
`$'`	text after matched pattern	`big$'`
`$&`	text of matched pattern	`big$&`
`\$`	`$` character	`big\$`

Regular expression repetition characters

Notice that the character "`?`" pulls double duty. When used as the only repetition specifier, "`?`" means to match zero or more occurrences of the previous character. For example, `/a?/` matches one or more "a" characters in sequence. When used as the second character of a repetition specifier, as in "`*?`", "`+?`", and "`{n,}?`", a question mark "`?`" indicates a minimal match. What is meant by a minimal match?

Well obviously, it is the counterpart to a maximal match, which is the default for JavaScript and PERL regular expressions. A maximal match will include the maximum number of characters in a text that will qualify to match a regular expression pattern. For example, in the string `"one two three"`, the pattern `/o.*e/` will match the text `"one two three"`. Why? The pattern says to match text that begins with the character "`o`" followed by zero or more of any characters up to the character "`e`". Since the default is a maximal match, the whole string is matched since it begins with "`o`" and ends with "`e`". Often, this maximal match behavior is not what is expected or desired.

Now consider a similar match using the minimal character. The string is still `"one two three"`, but the pattern becomes `/o.*?e/`. Notice that the only difference is the addition of a question mark "`?`" as the second repetition character after the "`*`". The text matched this time is `"one"`, which is the minimal number of characters that match the conditions of the regular expression pattern.

So, it might be a good habit to begin reading regular expression patterns with a maximal and minimal vocabulary. As an example, lets spell out how we could read the two patterns in the current example.

- "`o.*e`" - match text that begin with "`o`" and has the maximum number of characters possible until the last "`e`" is encountered.
- "`o.*?e`" - match text that begins with "`o`" and has the minimum number of characters possible until the first "`e`" is encountered.

Sometimes a maximal match is called a greedy match and a minimal match is called a non-greedy match.

Repetition	**How many characters matched**
`?`	Match zero or one occurrence of the previous character or sub pattern. Same as `{0,1}`
`*`	Match zero or more occurrences of the previous character or sub pattern. A maximal match, that is, match as many characters as will fulfill the regular expression. Same as `{0,}`
`*?`	Match zero or more occurrences of the previous character or sub pattern. A minimal match, that is, match as few characters as will fulfill the regular expression. Same as `{0,}?`
`+`	Match one or more occurrences of the previous character or sub pattern. A maximal match, that is, match as many characters as will fulfill the regular expression. Same as `{1,}`
`+?`	Match one or more occurrences of the previous character or sub pattern. A minimal match, that is, match as few characters as will

	fulfill the regular expression. Same as `{1,}?`
`{n}`	Match `n` occurrences of the previous character or sub pattern.
`{n,}`	Match `n` or more occurrences of the previous character or sub pattern. A maximal match, that is, match as many characters as will fulfill the regular expression.
`{n,}?`	Match `n` or more occurrences of the previous character or sub pattern. A minimal match, that is, match as few characters as will fulfill the regular expression.
`{n, m}`	Match the previous character or sub pattern at least `n` times but not more than `m` times.

Regular expression character classes

Class	**Character matched**
`.`	Any character except newline, `[^\n]`
`[...]`	Any one of the characters between the brackets
`[^...]`	Any one character not one of the characters between the brackets
`[\b]`	A backspace character (special syntax because of the `\b` boundary)
`\d`	Any digit, `[0-9]`
`\D`	Any character not a digit, `[^0-9]`
`\s`	Any white space character, `[ \t\n\r\f\v]`
`\S`	Any non-white space character, `[^ \t\n\r\f\v]`
`\w`	Any word character, `[a-zA-Z0-9_]`
`\W`	Any non-word character, `[^a-zA-Z0-9_]`

Regular expression anchor characters

Anchor characters indicate that the following or preceding characters must be next to a special position in a string. The characters next to anchor characters are included in a match, not the anchor characters themselves. For example, in the string "The big cat and the small cat", the regular expression `/cat$/` will match the "cat" at the end of the string, and the match will include only the three characters "cat". The "`$`" is an anchor character indicating the end of a string (or line if a multiline search is being done).

The following table lists the anchor characters, metacharacters, and their meanings.

Character	**Anchor meaning**
`^`	The beginning of a string (or line if doing a multiline search). (See `\A` below.)
	Example: `/^The/`
`$`	The end of a string (or line if doing a multiline search). (See `\Z` below.)

	Example: `/cat$/`
`\A`	Matches the beginning of a string only. (See `$` above.)
`\b`	A word boundary. Match any character that is not considered to be a valid character for a word in programming. The character class "\W", not a word character, is similar. There are two differences. One, "\b" also matches a backspace. Two, "\W" is included in a match, since it is regular expression character, but "\b" is not included in a match. Example: `/\bthe\b/`
`\B`	Not a word boundary. The character class "\w" is similar. The most notable difference is that "\w" is included in a match, and "\B" is not. Example: `/l\B/`
`\Z`	Matches the end of a string only. (See ^ above.)

Regular expression reference characters

Character	**Meaning**
`|`	Or. Match the character or sub pattern on the left **or** the character or sub pattern on the right.
`\n`	Reference to group. Match the same characters, not the regular expression itself, matched by group `n`. Groups are sub patterns that are contained in parentheses. Groups may be nested. Groups are numbered according to the order in which the left parenthesis of a group appears in a regular expression.
`(...)`	Group with capture. Characters inside of parentheses are handled as a single unit or sub pattern in specified ways, such as with the first two explanations, `|` and `\n`, in this table. The characters that are actually matched are captured and may be used later in an expression (as with `\n`) or in a replacement expression (as with `$n`). For example, if the string "one two three two one" and the pattern `/(o.e).+(w.+?e)/` are used, then the back references `$1` or `\1` use the text "one".
`(?:...)`	Group without capture. Matches the same text as `(...)`, but the text matched is not captured or saved and is not available for later use using `\n` or `$n`. The overhead of not capturing matched text becomes important in faster execution time for searches involving loops and many iterations. Also, some expressions and replacements can be easier to read and use with fewer numbered back references with which to keep up. For example, if the string "one two three two one" and the pattern `/(?:o.e).+(w.+?e)/` are used, then the back references `$1` or `\1` use the text "wo thre".
`(?=...)`	Positive look ahead group without capture. The position of the match is at the beginning of the text that matches the sub pattern. For example, `/ScriptEase (?=Desktop|ISDK)/` matches "ScriptEase " in "ScriptEase Desktop" or "ScriptEase ISDK", but

	not "ScriptEase " in "ScriptEase Web Server". When a search continues, it begins after "ScriptEase ", not after "Desktop" or "ISDK". That is, the search continues after the last text matched, not after the text that matches the look ahead sub pattern.
`(?!...)`	Negative look ahead group without capture. The position of the match is at the beginning of the text not matching the sub pattern. For example, `/ScriptEase (?!Desktop|ISDK)/` matches "ScriptEase " in "ScriptEase Web Server", but not "ScriptEase " in "ScriptEase Desktop" or "ScriptEase ISDK". When a search continues, it begins after "ScriptEase ", not after "Desktop" or "ISDK". That is, the search continues after the last text matched, not after the text that matches the look ahead sub pattern.

Regular expression escape sequences

Sequence	**Character represented**
`\f`	Form feed, `\cL`, `\x0C`, `\014`
`\n`	Line feed, newline, `\cJ`, `\x0A`, `\012`
`\r`	Carriage return, `\cM`, `\x0D`, `\015`
`\t`	Horizontal tab, `\cI`, `\x09`, `\011`
`\v`	Vertical tab, `\cK`, `\x0B`, `\013`
`\/`	The character: /
`\\`	The character: \
`\.`	The character: .
`\*`	The character: *
`\+`	The character: +
`\?`	The character: ?
`\\|`	The character: |
`\(`	The character: (
`\)`	The character:)
`\[`	The character: [
`\]`	The character:]
`\{`	The character: {
`\}`	The character: }
`\C`	A character itself, if not one of the above. Seldom, if ever, used.
`\cC`	A control character. For example, `\cL` is a form feed (`^L` or `Ctrl-L`), same as `\f`.
`\x##`	A character represented by its code in hexadecimal. For example, `\x0A` is a newline, same as `\n`, and `\x41` is "`A`".
`\###`	A character represented by its code in octal. For example, `\012` is a

newline, same as `\n`, and `\101` is "`A`".

Regular expression replacement characters

All of the special characters that have been discussed so far pertain to regular expression patterns, that is, to finding and matching strings and patterns in a target string. If all you want to do is find text, then you do not need to know about regular expression replacement characters. However, most people not only want to do powerful searches, but they also want to make powerful replacements of found text. This section describes special characters that are used in replacement strings and that are related to special characters used in search patterns.

Expression	Meaning
`$1, $2 ... $9`	The text that is matched by sub patterns inside of parentheses. For example, `$1` substitutes the text matched in the first parenthesized group in a regular expression pattern. See the groups, `(...)`, `(?:...)`, `(?=...)`, and `(?!...)`, under regular expression reference characters.
`$+`	The text matched by the last group, that is, parenthesized sub pattern.
`` $` ``	The text before, to the left of, the text matched by a pattern.
`$'`	The text after, to the right of, the text matched by a pattern
`$&`	The text matched by a pattern
`\$`	A literal dollar sign, `$`.

Regular expression precedence

The patterns, characters, and metacharacters of regular expressions comprise a sub language for working with strings. Some of the metacharacters can be understood as operators, and, like operators in all programming languages, there is an order of precedence. The following tables list regular expression operators in the order of their precedence.

Operator	Descriptions
`\`	Escape
`()`, `(?:)`, `(?=)`, `(?!)`, `[]`	Groups and sets
`*`, `+`, `?`, `{n}`, `{n,}`, `{n,m}`	Repetition
`^`, `$`, `\metacharacter`	Anchors and metacharacters
`|`	Alternation

RegExp object instance properties

RegExp global

SYNTAX:	`regexp.global`
DESCRIPTION:	A read-only property of an instance of a RegExp object. It is `true` if "g" is an attribute in the regular expression pattern being

used.

Read-only property. Use RegExp compile() to change.

SEE: Regular expression attributes

EXAMPLE:
```
var pat = /^Begin/g;
//or
var pat = new RegExp("^Begin", "g");
```

RegExp ignoreCase

SYNTAX: `regexp.ignoreCase`

DESCRIPTION: A read-only property of an instance of a RegExp object. It is `true` if "i" is an attribute in the regular expression pattern being used.

Read-only property. Use RegExp compile() to change.

SEE: Regular expression attributes

EXAMPLE:
```
var pat = /^Begin/i;
//or
var pat = new RegExp("^Begin", "i");
```

RegExp lastIndex

SYNTAX: `regexp.lastIndex`

DESCRIPTION: The character position after the last pattern match and which is the basis for subsequent matches when finding multiple matches in a string. That is, in the next search, `lastIndex` is the starting position. This property is used only in global mode after being set by using the "g" attribute when defining or compiling a search pattern. RegExp exec() and RegExp test() use and set the `lastIndex` property. If a match is not found by one of them, then `lastIndex` is set to 0. Since the property is read/write, you may set the property at any time to any position.

Read/write property.

SEE: RegExp exec(), String match()

EXAMPLE:
```
var str = "one tao three tio one";
var pat = /t.o/g;
pat.exec(str);
   // pat.lastIndex == 7
```

RegExp multiline

SYNTAX: `regexp.multiline`

DESCRIPTION: A read-only property of an instance of a RegExp object. It is `true` if "m" is an attribute in the regular expression pattern being used. There is no static (or global) RegExp `multiline` property in ScriptEase JavaScript since the presence of one is based on old technology and is confusing now that an instance property exists.

This property determines whether a pattern search is done in a multiline mode. When a pattern is defined, the `multiline` attribute may be set, for example, `/^t/m`. A pattern definition such as this one, sets the instance property `regexp.multiline` to `true`.

Read-only property. Use RegExp compile() to change.

SEE: Regular expression attributes

EXAMPLE:

```
// In all these examples, pat.multiline is set
// to true. If there were no "m" in the attributes,
// then pat.multiline would be set to false.
var pat = /^Begin/m;
//or
var pat = new RegExp("^Begin", "igm");
//or
var pat = /^Begin/m;
//or
var pat = new RegExp("^Begin", "igm");
```

RegExp source

SYNTAX: `regexp.source`

DESCRIPTION: The regular expression pattern being used to find matches in a string, not including the attributes.

Read-only property. Use RegExp compile() to change.

SEE: Regular expression syntax

EXAMPLE:

```
var str = "one tao three tio one";
var pat = /t.o/g;
pat.exec(str);
   // pat.source == "t.o"
```

RegExp returned array properties

Some methods, String match() and RegExp exec() return arrays in which various elements and properties are set that provide more information about the last regular expression search. The properties that might be set are described in this section, not the contents of the array elements.

index (RegExp)

SYNTAX: `returnedArray.index`

DESCRIPTION: When String match() is called and the "`g`" is not used in the regular expression, String `match()` returns an array with two extra properties, `index` and `input`. The property `index` has the start position of the match in the target string.

SEE: input (RegExp), RegExp exec(), String match()

EXAMPLE:

```
var str = "one tao three tio one";
var pat = /(t.o)\s(t.r)/g;
var rtn = pat.exec(str);
   // rtn[0] == "tao thr"
   // rtn[1] == "tao"
   // rtn[2] == "thr"
   // rtn.index == 4
   // rtn.input == "one tao three tio one"
```

input (RegExp)

SYNTAX: `returnedArray.input`

DESCRIPTION: When String match()is called and the "`g`" is not used in the regular expression, String `match()` returns an array with two extra properties, `index` and `input`. The property `input` has a copy of the target string.

SEE: index (RegExp), RegExp exec(), String match()

EXAMPLE:

```
var str = "one two three two one";
var pat = /(t.o)\s(t.r)/g;
var rtn = pat.exec(str);
   // rtn[0] == "two thr"
   // rtn[1] == "two"
   // rtn[2] == "thr"
   // rtn.index == 4
   // rtn.input == "one two three two one"
```

RegExp object instance methods

RegExp()

SYNTAX: `new RegExp([pattern[, attributes]])`

WHERE: pattern - a string containing a regular expression pattern to use with this RegExp object.

attributes - a string with the attributes for this RegExp object.

RETURN: object - a new regular expression object, or `null` on error.

DESCRIPTION: Creates a new regular expression object using the search pattern and options if they are specified.

If the attributes string is passed, it must contain one or more of the following characters or be an empty string `""`:

`i` - sets the ignoreCase property to `true`
`g` - sets the global property to `true`
`m` - set the multiline property to `true`

SEE: Regular expression syntax, String match(), String replace(), String search()

EXAMPLE:

```
// no options
var regobj = new RegExp( "r*t", "" );
// ignore case
var regobj = new RegExp( "r*t", "i" );
// global search
var regobj = new RegExp( "r*t", "g" );
// set both to be true
var regobj = new RegExp( "r*t", "ig" );
```

RegExp compile()

SYNTAX: `regexp.compile(pattern[, attributes])`

WHERE: pattern - a string with a new regular expression pattern to use with this RegExp object.

attributes - a string with the new attributes for this RegExp object.

RETURN: void.

DESCRIPTION: This method changes the pattern and attributes to use with the current instance of a RegExp object. An instance of a RegExp object may be used repeatedly by changing it with this method.

If the attributes string is supplied, it must contain one or more of the following characters or be an empty string `""`:

`i` - sets the ignoreCase property to `true`
`g` - sets the global property to `true`
`m` - set the multiline property to `true`

SEE: RegExp(), Regular expression syntax

EXAMPLE:

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

RegExp exec()

SYNTAX: `regexp.exec([str])`

WHERE: str - a string on which to perform a regular expression match. Default is `RegExp.input`.

RETURN: array - an array with various elements and properties set depending on the attributes of a regular expression. Returns `null` if no match is found.

DESCRIPTION: This method, of all the RegExp and String methods, is both the most powerful and most complex. For many, probably most, searches, other methods are quicker and easier to use. A string, the target, to be searched is passed to `exec()` as a parameter. If no string is passed, then RegExp.input, which is a read/write property, is used as the target string.

When executed without the global attribute, "g", being set, if a match is found, element 0 of the returned array is the text matched, element 1 is the text matched by the first sub pattern in parentheses, element 2 the text matched by the second sub pattern in parentheses, and so forth. These elements and their numbers correspond to groups in regular expression patterns and replacement expressions. The `length` property indicates how many text matches are in the returned array. In addition, the returned array has the `index` and `input` properties. The `index` property has the start position of the first text matched, and the `input` property has the target string that was searched. These two properties are the same as those that are part of the returned array from String match() when used without its global attribute being set.

When executed with the global attribute being set, the same results as above are returned, but the behavior is more complex which allows further operations. This method `exec()` begins searching at the position, in the target string, specified by `this.lastIndex`. After a match, `this.lastIndex` is set to the position after the last character in the text matched. Thus, you can easily loop through a string and find all matches of a pattern in it. The property `this.lastIndex` is read/write and may be set at anytime. When no more matches are found, `this.lastIndex` is reset to 0.

Since RegExp exec() always includes all information about a match in its returned array, it is the best, perhaps only, way to get all information about all matches in a string.

As with String match(), if any matches are made, appropriate RegExp object static properties, such as RegExp.leftContext, RegExp.rightContext, RegExp.$n, and so forth are set, providing more information about the matches.

SEE: String match(), RegExp object static properties

EXAMPLE:

```
var str = "one two three tio one";
var pat = new RegExp("t.o", "g");

while ((rtn = pat.exec(str)) != null)
   Screen.writeln("Text = " + rtn[0] +
                  " Pos = " + rtn.index +
                  " End = " + pat.lastIndex);
// Display is:
//   Text = two Pos = 4 End = 7
//   Text = tio Pos = 14 End = 17
```

RegExp test()

SYNTAX: `regexp.test([str])`

WHERE: str - a string on which to perform a regular expression match. Default is `RegExp.input`.

RETURN: boolean - `true` if there is a match, else `false`.

DESCRIPTION: Tests a string to see if there is a match for a regular expression pattern.

This method is equivalent to `regexp.exec(string)!=null`.

If there is a match, appropriate RegExp object static properties, such as RegExp.leftContext, RegExp.rightContext, RegExp.$n, and so forth, are set, providing more information about the matches.

Though it is unusual, `test()` may be used in a special way when the global attribute, "g", is set for a regular expression pattern. Like with RegExp exec(), when a match is found, the `lastIndex` property is set to the character position after the text match. Thus, `test()` may be used repeatedly on a string, though there are few reasons to do so. One reason would be if you only wanted to know if a string had more than one match.

SEE: RegExp exec(), String match(), String search()

EXAMPLE:

```
var rtn;
var str = "one two three tio one";
var pat = /t.o/;
   // rtn == true
rtn = pat.test(str);
```

RegExp object static properties

RegExp.$n

SYNTAX: `RegExp.$n`

DESCRIPTION: The text matched by the nth group, that is, the nth sub pattern in parenthesis. The numbering corresponds to `\n`, back references in patterns, and `$n`, substitutions in replacement patterns.

Read-only property.

SEE: Regular expression reference characters, regular expression replacement characters

EXAMPLE:

```
var str = "one two three two one";
var pat = /(t.o)\s/
str.match(pat)
   // RegExp.$1 == "two"
```

RegExp.input

SYNTAX: `RegExp.input`

DESCRIPTION: If no string is passed to RegExp exec() or to RegExp test(), then `RegExp.input` is used as the target string. To be used as the target string, it must be assigned a value. `RegExp.input` is equivalent to `RegExp.$_`, for compatibility with PERL.

Read/write property.

SEE: RegExp exec(), RegExp test()

EXAMPLE:

```
var pat = /(t.o/;
RegExp.input = "one two three two one";
pat.exec();
   // "two" is matched
```

RegExp.lastMatch

SYNTAX: `RegExp.lastMatch`

DESCRIPTION: This property has the text matched by the last pattern search. It is the same text as in element 0 of the array returned by some methods. `RegExp.lastMatch` is equivalent to `RegExp["$&"]`, for compatibility with PERL.

Read-only property.

SEE: RegExp exec(), String match(), RegExp returned array properties

EXAMPLE:

```
var str = "one two three two one";
var pat = /(t.o)/
pat.exec(str);
   // RegExp.lastMatch == "two"
```

RegExp.lastParen

SYNTAX: `RegExp.lastParen`

DESCRIPTION: This property has the text matched by the last group, parenthesized sub pattern, in the last pattern search. `RegExp.lastParen` is equivalent to `RegExp["$+"]`, for compatibility with PERL.

Read-only property.

SEE: RegExp.$n

EXAMPLE:

```
var str = "one two three two one";
var pat = /(t.o)+\s(t.r)/
pat.exec(str);
   // RegExp.lastParen == "thr"
```

RegExp.leftContext

SYNTAX: `RegExp.leftContext`

DESCRIPTION: This property has the text before, that is, to the left of, the text matched by the last pattern search. `RegExp.leftContext` is equivalent to `RegExp["$`"]`, for compatibility with PERL.

Read-only property.

SEE: RegExp.lastMatch, RegExp.rightContext

EXAMPLE:

```
var str = "one two three two one";
var pat = /(t.o)/
pat.exec(str);
   // RegExp.leftContext == "one "
```

RegExp.rightContext

SYNTAX: `RegExp.rightContext`

DESCRIPTION: This property has the text after, that is, to the right of, the text matched by the last pattern search. `RegExp.leftContext` is equivalent to `RegExp["$'"]`, for compatibility with PERL.

Read-only property.

SEE: RegExp.lastMatch, RegExp.leftContext

EXAMPLE:

```
var str = "one two three two one";
var pat = /(t.o)/
pat.exec(str);
   // RegExp.leftContext == " three two one"
```

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.bound()

SYNTAX:	`SElib.bound()`
RETURN:	boolean - true if the currently running script is bound using the /bind command line option, else false.
DESCRIPTION:	ScriptEase scripts may be compiled to standalone executable, exe, files using the bind command line option. Sometimes it is

important to know if a script is being interpreted or being run as a standalone executable. Binding a script is a step more than compiling a script to be interpreted by a ScriptEase interpreter. (See SElib.compileScript())

SEE: SElib.compileScript(), SElib.version(), Using Library Files

EXAMPLE:

```
if (SElib.bound())
{
   Screen.writeln('Running a bound script');
   // Do this
}
else
{
   Screen.writeln('Running an unbound script');
   // Do that
}
```

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(), SElib.bound(), *sebind.jse*, *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: array - 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 filespec path.
.attrib  File flags, as defined in IncAttr, number.
.size    Size of file, number in bytes, number.
.access  Date and time of last file access, number.
.write   Date and time of last write, number.
.create  Date and time of file creation, number.
```

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`.

`SElib.getObjectProperties()` is similar to the ECMAScript for/in loop. The important difference is that a for/in loop does not enumerate properties that have `DONT_ENUM` as part of their attributes (global.setAttributes()), whereas `SElib.getObjectProperties()` includes them in the array that it returns.

SEE: for/in, Object propertyIsEnumerable()

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 SEdesk 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 global.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 that 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()

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 in which to put 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()

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 method 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(), Clib.isalnum(), *autoload.jse*

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 be 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(), File splitName(), 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.version()

SYNTAX: `SElib.version()`

RETURN: object - an object with properties that provide information about the version and operating system of the currently executing ScriptEase interpreter. The object returned as the following properties:

```
.os - string, identifying operating system
.se.engineVersion - number, major minor version
.se.versionString - string, sub minor version
.buildTime - string, date/time of build
.bindable - boolean, can bind to executable
.security - boolean, uses security features
```

DESCRIPTION: This method provides a variety of useful information about the version of the currently executing ScriptEase interpreter.

The `.os` string will be something like:

```
 DOS
 DOS32
 MAC
 NWNLM
 OS2
 WINDOWS
 WIN32.NTCON
 WIN32.NTWIN
 WIN32.95CON
 WIN32.95WIN
 UNIX
```

SEE: getSEver(), getSEversion(), predefined constants and values `VERSION_MAJOR`, `VERSION_MINOR`, `VERSION_STRING`

EXAMPLE:

```
var ver;
ver = SElib.version();
Screen.writeln(ver.os);
Screen.writeln(ver.se.engineVersion);
Screen.writeln(ver.se.versionString);
Screen.writeln(ver.buildTime);
// Displays something like:
//  WIN32.95CON
//  440
//  B
//  Apr 26 2001 16:06:49
```

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. If 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*

SElib.dynamicLink()

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.

See Win32 structure definitions.

SElib.dynamicLink() - for Win32

SYNTAX:
```
SElib.dynamicLink(library, procedure,
                      convention[, [desc,] param …])
```

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.
```

desc - a blobDescriptor that describes the following param if param is a structure. (See blobDescriptor example.) A blobDescriptor is only used in front of params that are structures and is required for such params. A Blob (Binary Large Object) and a Buffer are very similar in ScriptEase. The Blob is the type that was used, in the early days of ScriptEase, to work with data in sections of memory. The Buffer is the newer type. Structure types may be created in Blobs or Buffers and blobDescriptors may be used to describe the data in either type. So, in ScriptEase, you will sometimes see blobDescriptor before a param of type Blob or a param of type buffer. In either case, the blobDescriptor is describing how data is stored in the param, even if the data is a string.

param - a variable for a section of memory that holds data in the form of a structure of elements or a buffer a string.

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: Blob object, blobDescriptor example, Win32 structure definitions, Clib.fread()

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[,
                    [desc,] param …])
```

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.
```

desc - a blobDescriptor that describes the following param if param is a structure. (See blobDescriptor example.) A blobDescriptor is only used in front of params that are structures and is required for such params. A Blob (Binary Large Object) and a Buffer are very similar in ScriptEase. The Blob is the type that was used, in the early days of ScriptEase, to work with data in sections of memory. The Buffer is the newer type. Structure types may be created in Blobs or Buffers and blobDescriptors may be used to describe the data in either type. So, in ScriptEase, you will sometimes see blobDescriptor before a param of type Blob or a param of type buffer. In either case, the blobDescriptor is describing how data is stored in the param, even if the data is a string.

param - a variable for a section of memory that holds data in the form of a structure of elements or a buffer a string.

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, blobDescriptor example, Win32 structure definitions, Clib.fread()

SElib.dynamicLink() - for OS/2

SYNTAX:
```
SElib.dynamicLink(library, procedure, bitSize,
                  convention[, [desc,] param …])
```

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.
```

desc - a blobDescriptor that describes the following param if param is a structure. (See blobDescriptor example.) A blobDescriptor is only used in front of params that are structures and is required for such params. A Blob (Binary Large Object) and a Buffer are very similar in ScriptEase. The Blob is the type that was used, in the early days of ScriptEase, to work with data in sections of memory. The Buffer is the newer type. Structure types may be created in Blobs or Buffers and blobDescriptors may be used to describe the data in either type. So, in ScriptEase, you will sometimes see blobDescriptor before a param of type Blob or a param of type buffer. In either case, the blobDescriptor is describing how data is stored in the param, even if the data is a string.

param - a variable for a section of memory that holds data in the form of a structure of elements or a buffer a string.

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, blobDescriptor example, Clib.fread()

String Object

The String object is a data type and is a hybrid that shares characteristics of primitive data types and of composite data types. The String is presented in this section under two main headings in which the first describes its characteristics as a primitive data type and the second describes its characteristics as an object.

String as data type

A string is an ordered series of characters. The most common use for strings is to represent text. To indicate that text is a string, it is enclosed in quotation marks. For example, the first statement below puts the string `"hello"` into the variable `hello`. The second sets the variable `word` to have the same value as a previous variable `hello`:

```
var hello = "hello";
var word = hello;
```

Escape sequences for characters

Some characters, such as a quotation mark, have special meaning to the interpreter and must be indicated with special character combinations when used in strings. This allows the interpreter to distinguish between a quotation mark that is part of a string and a quotation mark that indicates the end of the string. The table below lists the characters indicated by escape sequences:

`\a`	Audible bell	
`\b`	Backspace	
`\f`	Formfeed	
`\n`	Newline**Error! Reference source not found.**	
`\r`	Carriage return	
`\t`	Horizontal Tab	
`\v`	Vertical tab	
`\'`	Single quote	
`\"`	Double quote	
`\\`	Backslash character	
`\0`	Null character	(e.g., "`\0`"is the `null` character)
`\###`	Octal number (0-7)	(e.g., "`033`"is the escape character)
`\x##`	Hex number (0-F)	(e.g., "`x1B`"is the escape character)
`\u####`	Unicode number (0-F)	(e.g., "`u001B`"is escape character)

Note that these escape sequences cannot be used within strings enclosed by back quotes, which are explained below.

Single quote

You can declare a string with single quotes instead of double quotes. There is no difference between the two in JavaScript, except that double quote strings are used less commonly by many scripters.

Back quote

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.

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 properties
```

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.

SEE: RegExp()

EXAMPLE:
```
var s = new String(123);
```

String charAt()

SYNTAX: `string.charAt(position)`

WHERE: position - offset within a string.

RETURN: string - character at position

DESCRIPTION: This method gets the character at the specified position. If no character exists at location `position`, or if `position` is less than 0, then `NaN` is returned.

SEE: String charCodeAt()

EXAMPLE:
```
// To get the first character in a string,
// use as follows:

var string = "a string";
string.charAt(0);

// To get the last character in a string, use:
string.charAt(string.length - 1);
```

String charCodeAt()

SYNTAX: `string.charCodeAt(index)`

WHERE: position - index of the character the encoding of which is to be returned.

RETURN: number - representing the unicode value of the character at position index of a string. Returns `NaN` if there is no character at the position.

SEE: String charAt(), String.fromCharCode()

DESCRIPTION: This method gets the nth character code from a string.

String concat()

SYNTAX: `string.concat([string1, ...])`

WHERE: stringN - A list of strings to append to the end of the current object.

RETURN: This method returns a string value (not a string object) consisting of the current object and any subsequent arguments appended to it.

DESCRIPTION: This method creates a new string whose contents are equal to the current object. Each argument is then converted to a string using global.ToString() and appended to the newly created string. This value is then returned. Note that the original object remains unaltered. The '+' operator performs the same function.

SEE: Array concat()

EXAMPLE:

```
// The following line:

var proverb = "A rolling stone " + "gathers no moss."

// creates the variable proverb and
// assigns it the string
// "A rolling stone gathers no moss."
// If you try to concatenate a string with a number,
// the number is converted to a string.

 var newstring = 4 + "get it";

// This bit of code creates newstring as a string
// variable and assigns it the string
// "4get it".

// The use of the + operator is the standard way of
// creating long strings in JavaScript.
// In 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. Default is 0.

RETURN: number - index of the first appearance of a substring in a string, else -1, if `substring` not found.

DESCRIPTION: String `indexOf()` searches the string for the string specified in `substring`. The search begins at `offset` if `offset` is specified; otherwise the search begins at the beginning of the string. If `substring` is found, String `indexOf()` returns the position of its first occurrence. Character positions within the string are numbered in increments of one beginning with zero.

SEE: String charAt(), String lastIndexOf(), String substring()

EXAMPLE:

```
var string = "what a string";
string.indexOf("a")

// returns the position, which is 2 in this example,
// of the first "a" appearing in the string.
// The method indexOf()may take an optional second
// parameter which is an integer indicating the index
// into a string where the method starts searching
// the string. For example:

var magicWord = "abracadabra";
var secondA = magicWord.indexOf("a", 1);

// returns 3, index of the first "a" to be found in
// the string when starting from the second letter of
// the string.
// Since the index of the first character is 0, the
// index of second character is 1.
```

String lastIndexOf()

SYNTAX: `string.lastIndexOf(substring[, offset])`

WHERE: substring - The substring that is to be searched for within string

offset - An optional integer argument which specifies the position within string at which the search is to start. Default is 0.

RETURN: number - index of the last appearance of a substring in a string, else -1, if `substring` not found.

SEE: String indexOf()

DESCRIPTION: This method is similar to String indexOf(), except that it finds the last occurrence of a character in a string instead of the first.

String localeCompare()

SYNTAX: `string.localeCompare(compareStr)`

WHERE: compareStr - a string with which to compare an instance string.

RETURN: number - indicating the relationship of two strings.

- `< 0` if string is less than compareStr
- `= 0` if string is the same as compareStr
- `> 0` if string is greater than compareStr

DESCRIPTION: This method returns a number that represents the result of a locale-sensitive string comparison of this object with that object. The result is intended to order strings in the sort order specified by the system default locale, and will be negative, zero, or positive, depending on whether string comes before compareStr in the sort order, the strings are equal, or string comes after compareStr.

SEE: Clib.strcmpi(), Clib.stricmp()

EXAMPLE:

String match()

SYNTAX: `string.match(pattern)`

WHERE: pattern - a regular expression pattern to find or match in string. May be a regular expression or a value, such as, a string, that may be converted into a regular expression using the RegExp() constructor. For example, both of the following are equivalent:

```
var rtn = "one two three".match(/two/);
var rtn = "one two three".match("two");
```

RETURN: array - an array with various elements and properties set depending on the attributes of a regular expression. Returns `null` if no match is found.

DESCRIPTION: This method behaves differently depending on whether pattern has the "g" attribute, that is, on whether the match is global.

If the match is not global, string is searched for the first match to pattern. A `null` is returned if no match is found. If a match is found, the return is an array with information about the match. Element 0 has the text matched. Elements 1 and following have the text matched by sub patterns in parentheses. The element numbers correspond to group numbers in regular expression reference characters and regular expression replacement characters. The array has two extra properties: index and `input`. The property `index` has the position of the first character of the text matched, and `input` has the target string.

If the match is global, string is searched for all matches to pattern. A `null` is returned if no match is found. If one or more matches are found, the return is an array in which each element has the text matched for each find. There are no `index` and `input` properties. The `length` property of the array indicates how many matches there were in the target string.

If any matches are made, appropriate RegExp object static properties, such as RegExp.leftContext, RegExp.rightContext, RegExp.$n, and so forth are set, providing more information about the matches.

SEE: RegExp exec(), String replace(), String search(), Regular expression replacement characters, RegExp object static properties

EXAMPLE:

```
   // not global
var pat = /(t(.)o)/;
var str = "one two three tio one";
   // rtn == "two"
   // rtn[0] == "two"
   // rtn[1] == "two"
   // rtn[2] == "w"
   // rtn.index == 4
   // rtn.input == "one two three two one"
rtn = str.match(pat);

   // global
var pat = /(t(.)o)/g;
var str = "one two three tio one";
   // rtn[0] == "two"
   // rtn[1] == "tio"
   // rtn.length == 2
rtn = str.match(pat);
```

String replace()

SYNTAX: `string.replace(pattern, replexp)`

WHERE: pattern - a regular expression pattern to find or match in string.

replexp - a replacement expression which may be a string, a string with regular expression elements, or a function.

RETURN: string - the original string with replacements in it made according to pattern and replexp.

DESCRIPTION: This string is searched using the regular expression pattern defined by pattern. If a match is found, it is replaced by the substring defined by replexp. The parameter replexp may be a:

- a simple string
- a string with special regular expression replacement elements in it
- a function that returns a value that may be converted into a string

If any replacements are done, appropriate RegExp object static properties, such as RegExp.leftContext, RegExp.rightContext, RegExp.$n, and so forth are set, providing more information about the replacements.

The special characters that may be in a replacement expression are (see regular expression replacement characters):

- `$1, $2 ... $9`
  The text that is matched by regular expression patterns inside of parentheses. For example, $1 will put the text matched in the first parenthesized group in a regular expression pattern. See (...) under regular expression reference characters.
- `$+`
  The text that is matched by the last regular expression pattern inside of the last parentheses, that is, the last group.
- `$&`
  The text that is matched by a regular expression pattern.
- `` $` ``
  The text to the left of the text matched by a regular expression pattern.
- `$'`
  The text to the right of the text matched by a regular expression pattern.
- `\$`
  The dollar sign character.

SEE: String match(), String search(), Regular expression replacement characters, RegExp object static properties

EXAMPLE:

```
var rtn;
var str = "one two three two one";
var pat = /(two)/g;

   // rtn == "one zzz three zzz one"
rtn = str.replace(pat, "zzz");
   // rtn == "one twozzz three twozzz one";
rtn = str.replace(pat, "$1zzz");
   // rtn == "one 5 three 5 one"
rtn = str.replace(pat, five());
   // rtn == "one twotwo three twotwo one";
rtn = str.replace(pat, "$&$&);

function five()
{
   return 5;
}
```

String search()

SYNTAX: `string.search(pattern)`

WHERE: pattern - a regular expression pattern to find or match in string.

RETURN: number - the starting position of the first matched portion or substring of the target string. Returns -1 if there is no match.

DESCRIPTION: This method returns a number indicating the offset within the string where the pattern matched or -1 if there was no match. The return is the same character position as returned by the simple search using String indexOf(). Both `search()` and `indexOf()` return the same character position of a match or find. The difference is that `indexOf()` is simple and `search()` is powerful.

The `search()` method ignores a "g" attribute if it is part of the regular expression pattern to be matched or found. That is, `search()` cannot be used for global searches in a string.

After a search is done, the appropriate RegExp object static properties are set.

SEE: String match(), String replace(), RegExp exec(), Regular expression syntax, RegExp Object, RegExp object static properties

EXAMPLE:

```
var str = "one two three four five";
var pat = /th/;
str.search(pat);      // == 8, start of th in three
str.search(/t/);      // == 4, start of t in two
str.search(/Four/i); // == 14, start of four
```

String slice()

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

WHERE: start - index from which to start.

end - index at which to end.

RETURN: string - a substring (not a String object) consisting of the characters.

SEE: String substring()

DESCRIPTION: This method is very similar to String substring(), in that it returns a substring from one index to another. The only difference is that if either `start` or `end` is negative, then it is treated as `length + start` or `length + end`. If either exceeds the bounds of the string, then either 0 or the length of the string is used instead.

String split()

SYNTAX: `string.split([delimiterString])`

WHERE: delimiterString - character, string or regular expression where the string is split. If `substring` is not specified, an array will be returned with the name of the `string` specified. Essentially this will mean that the string is split character by character.

RETURN: object - if no delimiters are specified, returns an array with one element which is the original string.

DESCRIPTION: This method splits a string into an array of strings based on the delimiters in the parameter delimiterString. The parameter delimiterString is optional and if supplied, determines where the string is split.

SEE: Array join()

EXAMPLE:

```
/*
For example, to create an array of all
of the words in a sentence, use code similar
to the following fragment:
*/

var sentence = "I am not a crook";
var wordArray = sentence.split(' ');
```

String substr()

SYNTAX: `string.substr(start, length)`

WHERE: start - integer specifying the position within the string to begin the desired substring. If start is positive, the position is relative to the beginning of the string. If start is negative, the position is relative to the end of the string.

length - the length, in characters, of the substring to extract.

RETURN: string - a substring starting at position start and including the next number of characters specified by length.

DESCRIPTION: This method gets a section of a string. The start parameter is the first character in the new string. The length parameter determines how many characters to include in the new substring.

This method, `substr()` differs from String substring() in two basic ways. One, in substring() the start position cannot be negative, that is, it must be 0 or greater. Two, the second parameter in `substring()` indicates a position to go to, not the length of the new substring.

SEE: String substring()

EXAMPLE:

```
var str = ("0123456789");
str.substr(0, 5)      // == "01234"
str.substr(2, 5)      // == "23456"
str.substr(-4, 2)     // == "56"
```

String substring()

SYNTAX: `string.substring(start, end)`

WHERE: start - integer specifying the position within the string to begin the desired substring.

end - integer specifying the position within the string to end the desired substring.

RETURN: string - a substring starting at position start and going to but not including position end.

DESCRIPTION: This method retrieves a section of a string. The start parameter is the index or position of the first character to include. The end parameter marks the end of the string. The end position is the index or position after the last character to be included. The length of the substring retrieved is defined by end minus start. Another way to think about the start and end positions is that end equals start plus the length of the substring desired.

SEE: String charAt(), String indexOf(), String lastIndexOf(), String slice(), String substr()

EXAMPLE:

```
// For example, to get the first nine characters
// in string, use a Start position
// of 0 and add 9 to it, that is,
// "0 + 9", to get the End position
// which is 9. The following fragment illustrates.

var str = "0123456789";
str.substring(0, 5)    // == "01234"
str.substring(2, 5)    // == "234"
str.substring(0, 10)   // == "0123456789"
```

String toLocaleLowerCase()

SYNTAX: `string.toLocaleLowerCase()`

RETURN: string - a copy of a string with each character converted to lower case.

DESCRIPTION: This method behaves exactly the same as String toLowerCase(). It is designed to convert the string to lower case in a locale sensitive manner, though this functionality is currently unavailable. Once it is implemented, this function may behave differently for some locales (such as Turkish), though for the majority it will be identical to `toLowerCase()`.

SEE:	String toLowerCase(), String toLocaleUpperCase()

String toLocaleUpperCase()

SYNTAX:	`string.toLocaleUpperCase()`
RETURN:	string - a copy of a string with each character converted to upper case.
DESCRIPTION:	This method behaves exactly the same as String toUpperCase(). It is designed to convert the string to upper case in a locale sensitive manner, though this functionality is currently unavailable. Once it is implemented, this function may behave differently for some locales (such as Turkish), though for the majority it will be identical to `toUpperCase()`.
SEE:	String toUpperCase(), String toLocaleLowerCase()

String toLowerCase()

SYNTAX:	`string.toLowerCase()`
RETURN:	string - copy of a string with all of the letters changed to lower case.
DESCRIPTION:	This method changes the case of a string.
SEE:	String toUpperCase(), String toLocaleLowerCase()

EXAMPLE:

```
var string = new String("Hello, World!");
string.toLowerCase()

// This will return the string "hello, world!".
```

String toUpperCase()

SYNTAX:	`string.toUpperCase()`
RETURN:	string - a copy of a string with all of the letters changed to upper case.
DESCRIPTION:	This method changes the case of a string.
SEE:	String toLowerCase(), String toLocaleUpperCase()

EXAMPLE:

```
var string = new String("Hello, World!");
string.toUpperCase()

// This will return the string
//  "HELLO, WORLD!".
```

String object static methods

String.fromCharCode()

SYNTAX:	`String.fromCharCode(chrCode[, ...])`
WHERE:	chrCode - character code, or list of codes, to be converted.
RETURN:	string - string created from the character codes that are passed to it as parameters.
DESCRIPTION:	The identifier String is used with this static method, instead of a variable name as with instance methods. The arguments passed

to this method are assumed to be unicode characters.

SEE: String(), String charCodeAt()

EXAMPLE:

```
// The following code:
var string = String.fromCharCode(0x0041,0x0042)
// will set the variable string to be "AB".
```

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.

SEE: Unix.setuid()

Unix.setsid()

SYNTAX: `Unix.setsid()`

RETURN: number - 0 for success, -1 for error.

DESCRIPTION: Creates a new session with no terminal, most 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.

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 `Unix.wait()` will suspend execution until a child process terminates, then return the id of the particular child that exited. The status parameter is 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.
```

Appendices

See:

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

Clib qsort()	Sort an array.
global.getArrayLength()	Determines size of an array.
global.setArrayLength()	Sets the size of an array.

For Array objects

Array concat()	Concatenate to array.
Array join()	Creates a string from array elements.
Array pop()	Get last element of array.
Array push()	Add element to end of array.
Array reverse()	Reverses the order of elements of an array.
Array shift()	Get first element of array.
Array slice()	Get a subset of an array.
Array sort()	Sorts array elements.
Array splice()	Insert elements into array.
Array unshift()	Add elements to start of array.

Array properties

Array length	Returns the length of array.

array.jsh - arrays and objects
item.jsh - delimited strings/arrays

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 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.

dlgobj.jsh - Dialog object
getit.jsh - getItem and getLine
inout.jsh - routines for input/output
inputbox.jsh - input box
key.jsh - keys and keyboard
msgbox.jsh - message boxes

Routines for conversion/casting

global.escape()	Escapes special characters in a string.
global.parseFloat()	Converts a string to a Float.
global.parseInt()	Converts a string to an Integer.
global.unescape()	Removes escape sequences in a string.
global.ToBoolean()	Converts a value to a Boolean.
global.ToBuffer()	Converts a value to a buffer.
global.ToBytes()	Converts a value to a buffer, raw transfer.
global.ToInt32()	Converts a value to a large Integer.

global.ToInteger()	Converts a value to an Integer.
global.ToNumber()	Converts a value to a Number.
global.ToObject()	Converts a value to an Object.
global.ToPrimitive()	Converts a value to a Primitive.
global.ToString()	Converts a value to a String.
global.ToUint16()	Converts a value to an unsigned Integer.
global.ToUint32()	Converts a value to an unsigned large Integer.

array.jsh - arrays and objects

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.
blobDescriptor object	Describe data in a Blob.
global.defined()	Tests if variable has been defined.
global.getAttributes()	Gets attributes of a variable.
global.isFinite()	Determines if a value is finite.
global.isNaN()	Determines if a value is Not a Number.
global.setAttributes()	Sets attributes of a variable.
global.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.

profobj.jsh - Profile object for ini files
regobj.jsh - Registry object

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.strftime()	Formatted write of date/time to a string.
Clib.time()	Gets current time.
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 toDateString()	Returns the date portion of current date as string.
Date toGMTString()	Converts a Date object to a string.
Date toLocaleDateString()	Same as date.toDateString using local time.
Date toLocaleString()	Returns a string for local date and time.
Date toLocaleTimeString()	Same as date.toTimeString using local time.
Date toSystem()	Converts a Date object to a system time.
Date toTimeString()	Returns the time portion of current date as string.
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.

datetime.jsh - date and time

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.

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 specification.

Routines for DOS

Dos.address()	Set a memory address.
Dos.asm()	Execute machine code in a memory location.
Dos.inport()	Get byte from a hardware port.
Dos.inportw()	Get word from a hardware port.
Dos.interrupt()	Execute 8086 interrupt.
Dos.offset()	Get offset of memory address.
Dos.outport()	Write byte to hardware port.
Dos.outportw()	Write word to hardware port
Dos.segment()	Get segment of memory address.

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.baseWindowFunction()	Call base procedure for a window.
SElib.breakWindow()	Release control of a window.
SElib.compileScript()	Compiles script into executable code.
SElib.doWindows()	Start ScriptEase window manager.
SElib.dynamicLink()	Make a call to the API.
SElib.inSecurity()	Calls security manager initialization routine.
SElib.instance()	Get instance handle of currently executing script.
SElib.interpret()	Interprets ScriptEase code or source file.
SElib.interpretInNewThread()	Creates a new thread within a current process.
SElib.makeWindow()	Create window to be managed.
SElib.messageFilter()	Restrict messages to a window.
SElib.multiTask()	Toggle multitasking on and off.
SElib.ShellFilterCharacter()	Add character filter to ScriptEase shell.
SElib.ShellFilterCommand()	Add command filter to ScriptEase shell.
SElib.spawn()	Runs an external executable.
SElib.subclassWindow()	Hooks a windowFunction in message loop.
SElib.suspend()	Suspends program execution for a while.
SElib.windowList()	Get handles of child windows.

exec.jsh - execute programs
getopt.jsh - get options
keypush.jsh - keyboard simulation
menuctrl.jsh - control menus
message.jsh - for windows
mouseclk.jsh - mouse simulation
optparms.jsh - get parameters

winexec.jsh - execute programs

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.fread()	Reads data from a file.
Clib.freopen()	Assigns new file spec to a file handle.
Clib.fscanf()	Formatted input from a file stream.
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.

copyfile.jsh - copying files
fileobj.jsh - File objects

Routines for general use

profobj.jsh - Profile object for ini files
regobj.jsh - Registry object
seutil.jsh - ScriptEase header

Routines for math

Math methods

Clib.abs()	Returns the absolute value of an integer.
Clib.acos()	Calculates the arc cosine.
Clib.asin()	Calculates the arc sine.
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.floor()	Rounds down.
Clib.fmod()	Modulus, calculate remainder.
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.NEGATIVE_INFINITY	Number below MIN_VALUE
Number.POSITIVE_INFINITY	Number above MAX_VALUE

Routines for memory manipulation

Clib.bsearch()	Binary search in memory/array/buffer.
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 objects and functions

SElib.getObjectProperties()	Get names of properties of an object.
Object hasOwnProperty()	Determine if an object has a property.
Object isPrototypeOf()	Determine if a property is part of prototype.
Object propertyIsEnumerable()	Is the attribute of a property `DONT_ENUM`.
Object toLocaleString()	Object to string using local settings.

array.jsh - arrays and objects

Function apply()	Apply arguments array to a function.
Function call()	Call function with argument list.

Routines for regular expressions

RegExp compile()	Sets a regular expression for the object.
RegExp exec()	Performs regular expression search.
RegExp test()	Tests a regular expression search.
String match()	Regular expression match
String replace()	Regular expression search/replace
String search()	Regular expression search

Routines for strings/byte arrays

Methods for strings

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 character to byte array.
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.strstri()	Case insensitive version of Clib.strstr.
Clib.strtod()	Converts a string to a floating-point value.
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 concat()	Concatenate a string.
String indexOf()	Returns index of first substring in a string.
String lastIndexOf()	Returns index of last substring in a string.
String localeCompare()	Compare string using local settings.
String slice()	Get a substring from a string.
String split()	Splits a string into an array of strings.

String substring()	Retrieves a section of a string.
String toLocaleLowerCase()	Returns lowercase string using local settings.
String toLocaleUpperCase()	Returns uppercase string using local settings.
String toLowerCase()	Converts a string to lowercase.
String toUpperCase()	Converts a string to uppercase.

String.fromCharCode()	Creates a string from character codes.

item.jsh - delimited strings/arrays

string.jsh - more for strings

String properties

String length	Holds the length of a string in characters.

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.

Routines for UNIX

Unix.fork()	Create duplicate processes.
Unix.kill()	Wrapper for UNIX kill command.
Unix.setgid()	Change group id.
Unix.setsid()	Create a new session.
Unix.setuid()	Change user id.
Unix.wait()	Suspend execution until child process stops.
Unix.waitpid()	Suspend execution with additional controls.

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 sample

String length

SYNTAX: string.length

DESCRIPTION:

SEE:

EXAMPLE:

String instance methods sample

String indexOf

SYNTAX: string.indexOf(substring[, offset])

WHERE:

RETURN:

DESCRIPTION:

SEE:

EXAMPLE:

String static properties sample

String.illus

SYNTAX: String.illus

DESCRIPTION:

SEE:

EXAMPLE:

String static methods sample

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();
```

Index