ScriptEase[ä]

ISDK for Java

v4.10c

Nombas, Inc.

Nombas, Inc.
64 Salem Street
Medford, MA 02155
USA

http://www.nombas.com/

Contents

APPLICATION PROGRAMMING INTERFACE 35

Introduction: Integrating The ISDK/Java

This chapter describes the methods for integrating the ScriptEase:ISDK/Java into your application. Integration is comprised of three elements:

- the JseContext,
- method wrappers,
- JseVariables.

All methods mentioned here are fully described in the API chapter.

Unpacking and installing ScriptEase:ISDK

The package you received consists of 3 basic parts: the ScriptEase:ISDK Interpreter jar, the ScriptEase Standard Function Library source code, and miscellaneous samples and source files required to support the interpreter engine. Refer to the file README.TXT on the first installation disk for any last minute information and for more details on the installation procedure.

Integration overview

Integrating the ScriptEase interpreter with your application is a matter of including the necessary files with your application and creating the required interfaces. The interpreter is initialized with a call to jseInitializeExternalLink(), a static method of the Jselib object. jseInitializeExternalLink() returns a JseContext object to be used throughout the scripting session; its properties are set according to the parameters passed to jseInitializeExternalLink(). These properties define the scripting session's global variables, functions, security, and operating parameters (such as error handling).

Although usually one JseContext is sufficient, you may use more than one JseContext to create simultaneous and independent scripting sessions, with unique sets of global variables and available functions.

For each method you want to make available, you must write a wrapper method that converts variable types from JavaScript to Java and back. These methods must then be assigned to the JseContext with calls to jseAddLibrary(). Nombas provides many sample methods you can use or adapt to your needs; see the appendix for a full list.

Executing a script from within your application is done with a single call to jseInterpret(). While you are interpreting a script, all of the wrapper methods it contains have been registered with the interpreter and can be called directly from your Java code via jseCallFunction(), an instance method of a JseContext object.

Initializing the ISDK

Your application class must implement COM.Nombas.jse.Isdk.Jse. This imports all of the constants you need to interact with the ISDK. You can also implement some or all of the following interfaces, depending on the needs of your application. Below are descriptions of the functions you will need to write to complete these interfaces.

For example, your application object my be defined as:

```
Public class myApplication implements Com.Nombas.jse.Isdk.Jse,
    JseErrorHandler
{
 . . .
}
```

In the description for the JseErrorHandler interface (found immediately below), there is a sample implementation of the function required. You could add a function similar to it to your class definition above.

interface JseErrorHandler:

jseErrorMessageFunc will be called whenever the interpreter encounters an error condition. As indicated by its name, this function usually prints out an error message, although this doesn't have to be the case.

It requires two parameters, the relevant JseContext object and a string that describes the error. (These parameters will be supplied by the system when it makes the call; what you do with them is up to you.)

Here is an example of a simple jseErrorMessageFunc:

```
public void jseErrorMessageFunc(JseContext jsecontext,
                                 String ErrorString)
{
   System.out.print("Inside jseLibErrorWriteln\n");
   System.out.print("\nJavaISDK Error Handler Says: " +
      ErrorString + "\n");
}
```

interface JseContinueFunction:

jseMayIContinueFunc is called before the interpreter executes a script command. It takes one parameter, the relevant JseContext. If jseMayIContinueFunc returns true, execution will continue as normal; if it returns false, the script will be terminated.

You can use this function:

- to provide a debugging interface (use jseLocateSource() to retrieve the name and line number of the current location in the script being run),
- as a callback monitoring function for your scripts,
- to call multitasking tickler routines, or,
- to check on external status such as the pressing of ctrl-C or break.

Note that including even a simple Continue function will slow execution noticeably, so use it only if you really need to.

interface JseFileLocation:

jseFileFindFunc is called whenever the interpreter needs to open a file, taking a String that is the path to the file to be opened. It determines whether the file is safe to open, returning the full path to the file if it is and null if it is not. If the file is not found, you can specify some alternate action. It requires three parameters: the relevant JseContext, a String containing the file specification of the file the interpreter is trying to open, and a boolean value. This last value is set to *true* if the file is included with a #link command; otherwise, it's set to *false*.

```
public String jseFileFindFunc(JseContext jsecontext,
   String FileSpec,
   boolean FindLink)
```

If the interpreter is unable to find the file or is unable to open it, the function should return *null*.

Note that these strings do not have to be filenames, but may be URLs, decoded strings, or anything else your application uses as a source. See jseToolkitAppIO below for the interface used to read these files.

interface JseToolkitAppIOInterface:

Implement this interface if your scripts will need to open files, whether as parameters to a #include statement or as filenames passed to jseInterpret().

jseGetSource tells the interpreter how to open and read files for internal use. The action performed depends on the third parameter, actionFlag, which will be one of the following values: jseNewOpen, jseGetNext and jseClose.

jseNewOpen opens a file for reading; your function will return *true* if successful and *false* if not.

jseGetNext reads the next line of the file; your function will return *true* if successful and *false* if there are no more lines to read. The data read from the file must be stored in the properties of the JseToolkitAppSource object passed in.

jseClose is sent when the file is ready to be closed; your function will return *true* if the file was successfully closed and *false* if not.

```
public boolean
jseGetSource(JseContext jsecontext,
```

```
                JseToolkitAppSource srcDesc,
                int actionFlag);
```

The interpreter will initialize the JseToolkitAppSource object (the second parameter to jseGetSourceFunc).

The JseToolkitAppSource object has the following methods you may use to access and set its data:

getLineNumber()

Gets the line number of the line currently being read. This is most commonly used when debugging and for creating error messages.

getName()

This method is used when **actionFlag** is jseNewOpen. It returns the filename of the file that is to be opened.

getUserData()

Retrieves the object previously set in a call to setUserData(). This object is not used by the interpreter; it is provided as storage for any external data needed for proper interpretation and execution of the file. The object will be returned.

setCode(String)

This method is used by the toolkit application to set the next line of source code. This is set by the toolkit application when actionFlag is jseGetNext and JseFileLocation.jseFileFindFunc returns true. **string** will be the next line of code in the file being read.

setLineNumber(int)

Sets line number used by API for debugging and error messages. The line number is automatically incremented each time JseFileLocation.jseFileFindFunc is called; this function need only be called when line numbers are skipped to ensure that the line number returned by error messages corresponds to the correct line.

setUserData(Object)

Set a generic object for use by the toolkit application. The ISDK core ignores this value; it is provided as a place to store any data that will be needed for the correct interpretation and execution of the file. To access the object, use the getUserData() method.

The JseExternalLinkParameters object

The properties of this object determine how the interpreter will handle variables and other aspects of the scripting session. It has two properties: jseSecureCode, and an options flag.

jseSecureCode is the path and filename of the script that handles the security for the

session. This is a JavaScript file, so it may be easily edited if you need to do so. The format of this script is described later in this chapter.

The other property of a JseExternalLinkParameters object consists of one or more of the following values or'd together:

jseDefault is a default value (equivalent to 0) you can use if you don't wish to specify any of the optional behaviors. You may use this value anywhere that requires flags to indicate that the default values should be used.

Default_C_Behavior determines how the interpreter handles variables and arrays. If this value is included in the options parameter, variables and arrays will be treated as they are in C: variables are passed by reference rather than by value, and array arithmetic is possible. All functions will behave as if they had been declared with the **cfunction** keyword (see page 109).

Require_Function_Keyword determines whether the **function** (or **cfunction**) keyword is required with all functions. Although this keyword is required in the ECMA JavaScript specification, many implementations of JavaScript do not require it. You can force your users to use the keyword by including this value in the options flag.

Require_Var_Keyword which determines whether your users may use variables that have not been previously declared with the **var** keyword. This is permitted in JavaScript, but all such variables are considered global variables. Include this flag to prohibit this behavior.

Warn_On_Bad_Math determines whether or not illegal mathematical operations (such as dividing by zero) will be flagged as an error or not. In JavaScript it is legal to divide a number by zero; the value *NaN* (Not a Number) will be returned. Including Warn_On_Bad_Math in the options parameter will generate an error condition if you try to make invalid mathematical statements.

Creating a JseContext object

Most Integration:SDK API calls are methods of the JseContext object. An instance of this object is used throughout the scripting session to maintain its state. It is created by passing the JseToolkitApp and JseExternalLinkParameters objects to the Jselib method jseInitializeExternalLink():

```
jsecontext = jselib.jseInitializeExternalLink(
      Object ToolkitAppObject,
      JseExternalLinkParameters LinkParms,
      String globalVarName,
      String AccessKey)
```

The first parameter is the application object (already described), which implements whichever of the interfaces described that you want. The second parameter is an initialized JseExternalLinkParameters object (also already described).

The other two parameters are the name you wish to call the JavaScript global object and the **AccessKey** you received when you registered ScriptEase:ISDK.

Security code

You may provide a security filter to prevent certain functions from executing and to limit scripts to working in certain directories or files. The security filter is itself a script. It will be called before the the user's script is run, and before every non-secure function call. The full path and filename to the file containing the code is passed to jseInitializeExternalLink() when obtaining the JseContext.

The security script contains a SecurityGuard() function, and optionally may contain SecurityInit() or SecurityTerm() functions. SecurityGuard() receives the name of the function being called as its second parameter and tests to see whether the function call is permitted or not. If SecurityGuard() returns *true*, the function is permitted; if it returns *false*, the function may not be used. SecurityInit() is called before the script runs, and SecurityTerm() is called when the script terminates.

Here is an example of a security file. This file is for an application that limits how users may access the OpenDatabase() and CloseDatabase() functions. It also adds some stuff to the PATH variable before running the script, and removes it when it is done.

```
function SecurityInit(SecurityVar)
{
   // initialize SecurityVar
   SecurityVar.TempDir = Clib.getenv("TEMP");
   AddPath(...add a few directories to PATH...);
   return true;
}
function SecurityTerm(SecurityVar)
{
   DeletePath(...remove stuff added to PATH...);
   return true;
}
function SecurityGuard(SecurityVar,testFunction,var1,var2,...)
{
   switch( testFunction )
   {
      case "OpenDatabase":
               //limit how users may use this function
               break;
      case "CloseDatabase":
               //limit how users may use this function
               break;

      default:
         // any other function is allowed
         return true;
   }
}
```

Testing the integration

To be sure your application has integrated the interpreter correctly, add a call to interpret a test script. The script can be simply "a=1", for example:

```
jsecontext.jseInterpret( null, "a=1;", null,
             jseAllNew, JSE_INTERPRET_CALL_MAIN,
             null, null);
```

Add this function after the call to Jselib.jseInitializeExternalLink(). If you can compile, link, and run this test code, then the interpreter has been successfully included in your application and is functioning correctly.

Adding functions to the ScriptEase engine

Once you have initialized the interpreter and created a JseContext, you must register any functions you want to make available to your users. This is a three step process:

- Write wrapper methods for the functions you wish to add to the function library. The wrapper method retrieves the function arguments from the ScriptEase call, translates the data from JavaScript to Java, makes your application call, and then translates any Java values back into JavaScript for return.
- Every function you make available to your users must be entered into a function library table. The function library table is an array of function descriptors containing each function's name as it will be called by your scripts, the corresponding wrapper method, the minimum and maximum number of arguments for the function, and a mask of function attributes. Data properties as well as functions can be added to the table. The next section of this manual describes some Jselib methods that help build a function library table.
- Call jseAddLibrary() (an instance method of the JseContext object) to register the function library table(s) with the ScriptEase interpreter.

Creating a ScriptEase function library table

A ScriptEase function library table is an array of Function Descriptors. It specifies to the interpreter the details of the methods to be added to the ScriptEase library. Each instance of a Function Descriptor assigns the ScriptEase function its name, the Java function, and the minimum and maximum number of arguments the ScriptEase function takes. There is no predefined limit to the number of functions that can be specified in a Function Library Table, nor is there any limit to the number of library tables that may be added to a given JseContext.

The Jselib object has a number of static methods to assist in building the table. Which method you use depends on the type of function being added to the table:

Jselib.JSE_LIBOBJECT - This defines what is being added to the table as an object; functions and variables using the other functions (listed below) will be added as properties and methods of this object, until another call to JSE_LIBOBJECT defines a new current object. If JSE_LIBOBJECT is not called, properties added with these methods will be added to the global object.

Jselib.JSE_LIBMETHOD - To add a method or function to the current object. The method will be added to the last object called with JSE_LIBOBJECT.

Jselib.JSE_PROTOMETH - Adds a method to the current object's prototype.

Jselib.JSE_VARASSIGN - This creates a copy of an already existing variable and assigns it to the current object.

Jselib.JSE_VARNUMBER - Assigns a numerical value as a property of the current object.

Jselib.JSE_VARSTRING - Assigns a string value as a property of the current object.

Jselib.JSE_ATTRIBUTE - Creates an undefined variable with the specified attributes. This method can also be used to change a variable's attributes.

These methods have the following syntax:

JSE_LIBOBJECT(name, methodName, min, max, varAttr, funcAttr)

JSE_LIBMETHOD(name, methodName, min, max, varAttr, funcAttr)

JSE_PROTOMETH(name, methodName, min, max, varAttr, funcAttr)

JSE_VARASSIGN(name, variable, varAttr)

JSE_VARNUMBER(name, var_number, varAttr)

JSE_VARSTRING(name, var_string, varAttr)

JSE_VARATTRIBUTE(name, varAttr)

name is a string representing the name to give to your function in a script. This is the name by which your users will use to refer to it.

methodName is the function called by the interpreter, i.e., the name of the wrapper method that corresponds to the function listed above or an inner class implementations of a JseWrapperFunction.

min is used to specify the minimum number of parameters that can be passed to the function.

max is used to specify the maximum number of parameters that can be passed to the function. If the number of parameters is unknown, set MaxVariableCount to -1. Set the maximum and the minimum to the same value to specify an exact argument count. If a script calls a function whose parameters do not meet the function parameter requirements, the script will be terminated with appropriate error handling.

Note that **min** and **max** represent the number of parameters passed to the JavaScript function, and not the number of parameters for the wrapper method. wrapper methods take only one parameter, the scripting session's JseContext object. The parameters passed to the function must be extracted as described later in this chapter.

varAttr is a bitwise-or of one or more of the following values:

jseDefaultAttr is used as a place holder for this parameter when you don't want to use any of the other options.

jseDontEnum This prevents the property or method from being listed in for...in statements.

jseDontDelete Prevents the delete operator from being used on the variable.

jseReadOnly Makes the property or method read only.

jseImplicitThis Add 'this' to the prototype chain (functions only).

funcAttr is a bitwise-or of one or more of the following:

jseFunc_Default to specify the default behavior.

jseFunc_CBehavior specifies that the function uses C behavior, passing parameters by reference.

jseFunc_Secure indicates that the function is safe to call and does not need to pass through the security filter. If this is not supplied, a security risk is assumed and the function must pass through the security filter before being executed. If there is no security filter, this option does nothing.

variable is a variable that has already been included in the function library table. `JSE_VARASSIGN` adds a copy of this variable to the current object.

var_number is a number variable or value to be assigned to the current object.

var_string is a string variable, enclosed in quotes, or a literal string enclosed in quotes ("\"literal string\"", e.g.), to be added as a property of the current object.

Initializing a ScriptEase function library table

A function library table is initialized and added to a specific context by calling jseAddLibrary(). It is defined as follows:

```
Void jseAddLibrary(
      String objectVariableName,
      JseFunctionDescription functionList[],
      JseLibrary libraryObject)
```

objectVariableName is the name of the object variable to which the library is to be attached. All methods in the library will become methods of this object. If this is *null*, the method will be added to the global object and will be available as a global function...no object scoping is necessary to call such a function.

functionList[] is the array of ScriptEase function descriptors to be added to the library, created with the methods described above.

libraryObject is an instance of the library being added; all functions in function list must be methods of this object. The libraryObject class must implement the JseLibrary interface, which contains these two methods:

```
public JseLibrary
   jseLibraryInitFunction(JseContext jsecontext);
```

This method is used to initialize any type of library-specific structure that the library may need to access. For example, the library may need to keep track of files opened. This method ay return same object as this or create a new one.

```
public void jseLibraryTermFunction(JseContext jsecontext);
```

This function is called when the library is terminated. It can be called multiple times, and every call to the jseLibraryInitFunction will have a matching call to jseLibraryTermFunction. Any initialization performed in the initialization function should be cleaned up here.

Writing ScriptEase function wrappers

Writing wrapper methods for new JavaScript functions is a three part task:

- Retrieve the variables passed as parameters to the function. You must first get a handle to the variable (a JseVariable), and then extract its data with one of the jseGetXXX() functions. This translates them from ScriptEase to Java for use in your code;
- Call your internal function or functions, using the Java variables retrieved by the previous step;
- Set the return codes and JseVariable values to be returned to the script.

All wrapper routines have the following syntax:

```
public void functionName(JseContext jsecontext)
```

The parameters passed to the JavaScript function functionName must be extracted as described below. The wrapper method itself takes only one parameter, the relevant JseContext. You can use an inner class method instead of writing a wrapper function.

Retrieving function arguments in a wrapper method

A JseContext object has two methods for retrieving function arguments: jseFuncVar() and jseFuncVarNeed(). The method you use depends on the type of variable, your application environment, and your programming style. Each returns a handle to a variable

you can use to retrieve or store a value passed from a script; the first parameter of each function is the offset (in the parameter list) of the variable desired.

jseFuncVar() returns the variable at the specified offset without checking its type; jseFuncVarNeed() has a second parameter that specifies the type or types of variable that will be accepted, generating an error if an appropriate variable is not found.

If no variable was found at the supplied offset, both functions return *null* and generate an error (the min/max values supplied when the function was added to the library may have already caught this error).

jseFuncVarNeed()

jseFuncVarNeed() will only retrieve variables of the type(s) specified by the second parameter. The script being interpreted will generate an error message and return *null* if the appropriate variable is not found. The variable's type is checked just prior to retrieving its handle, so the handle returned can be relied upon to be a valid JseVariable and of the type expected.

```
JseVariable
JseContext.jseFuncVarNeed(int ParameterOffset,
                          int Varneeded);
```

jseFuncVarNeed() takes two parameters: the offset of the parameter in the parameter list (0 for the first parameter, 1 for the second, etc.); and a value indicating the argument type expected by the script's function. This value may be one or more of the following or'd together, depending on the variable type you expect to receive:

JSE_VN_NUMBER	JSE_VN_BYTE
JSE_VN_BOOLEAN	JSE_VN_BUFFER
JSE_VN_NULL	JSE_VN_OBJECT
JSE_VN_FUNCTION	JSE_VN_ANY
JSE_VN_INT	jselib.JSE_VN_NOT()
JSE_VN_STRING	jselib.JSE_VN_CONVERT(from, to)

If a variable may be of more than one possible type, all possible types should be supplied, joined by a bitwise or (for example, JSE_VN_STRING | JSE_VN_NUMBER).

The last three values on the list are used when the variable type is unknown and in cases where more than one variable type is expected. JSE_VN_ANY will accept a variable of any type. Jselib.VN_NOT() (a static method of the Jselib object) will accept a variable of any type other than those passed to the method. If you are passing more than one value to Jselib.VN_NOT, they should be joined by an or (|). Jselib.VN_CONVERT() (also a static method of the Jselib object) converts a variable to a specific type. It takes two parameters: an or mask of variable types to be accepted, and the type (one of the values listed above) that the variable is to be converted to. Jselib.VN_CONVERT() cannot convert to types of JSE_VN_BYTE, JSE_VN_INT, or JSE_VN_FUNCTION.

jseFuncVarNeed() returns null on failure. If null is returned, your error routine will have been called, and the script being interpreted will abort when it returns from your wrapper method.

jseFuncVar()

If a function expects a variable of unknown type or if the wrapper method does its own type checking and conversion, use jseFuncVar() to obtain the variable's handle. It retrieves a variable handle regardless of its type. jseFuncVar() is prototyped as:

```
JseVariable
JseContext.jseFuncVar(int ParameterOffset);
```

Like jseFuncVarNeed(), jseFuncVar() returns *null* if it is unable to retrieve a variable at the specified offset. If *null* is returned, your error routine will have been called, and the script being interpreted will abort when it returns from the wrapper method.

Getting Data From JseVariables

Once you have a JseVariable handle, the data can be retrieved by calling the appropriate jseGetXXX() function. There is a specific get function for each of the ScriptEase types (jseGetNumber() and jseGetString(), e.g.). The methods jseGetLong() and jseGetByte() may be used to ensure that a number can be converted to an integral or byte value. Like jseFuncVar() and jseFuncVarNeed(), these are all instance methods of JseContext.

For example, if your function had one argument that was a number, you would use jseGetNumber() to get the value of the Scriptease variable.

```
// get the value from a jseLongVar.
int numArgumentVal = jsecontext.jseGetNumber(jseLongVar);
System.out.print(numArgumentVal);
```

If you used jseFuncVar() to get the handle to a function argument, first check the ScriptEase type before accessing its data with jseGetType(). jseGetType() returns one of the following values: jseTypeBoolean, jseTypeNull jseTypeNumber, jseTypeString, jseTypeBuffer, jseTypeObject, or jseTypeUndefined.

Assigning values to JseVariables

Setting the value of a JseVariable is similar to getting the value. Instead of using a jseGetXXX() function, use one of the jsePutXXX() functions. For example, if your function had one argument that was an integer, you would use jsePutLong() to set the value of the ScriptEase variable.

```
longValue = 1000 * 1000;
jsecontext.jsePutLong(jseLongVar, longValue);
```

To ensure that the new value is of the appropriate data type for the JseVariable, use jseConvert() before assigning the new value.

The jsePutXXX() functions will have a permanent effect only if the wrapper method is for a cfunction or an object, as only these variables are passed by reference.

JseVariable Attribute Flags

This is an OR'ed set of flags describing the attributes of the variable. The flags are as follows:

jseDefaultAttr - The default attributes are used (no flags are set)

jseDontDelete - This variable cannot be deleted. If, within a script, the user calls 'delete [variable]', then no action is taken. This does not affect calls to jseDeleteMember().

jseDontEnum - This variable is not enumerated within for . . . in loops. Therefore, if it is a member of an object and the user enumerates the members of the object using a for . . . in loop, this member will be skipped. jseGetNextMember() always returns all members.

jseImplicitParents - This is an attribute that applies only to local functions. It allows the scope chain to be altered based on the __parent__ property of the 'this' variable. If this flag is set, the __parent__ property is present, and a variable is not found in the local variable context (activation object), then the parents of the 'this' variable are searched (as long as there is a __parent__ property) before searching the global object. Here is an example, assuming that jseImplicitParents is set on function foo().

```
var a;
a.value = 4;
var b;
b.__parent__ = a;
b.foo = foo;
b.foo();
function foo()
{
   value = 5;
   // This will actually set a.value to 5
}
```

jseImplicitThis - This attribute applies only to local (script) functions. If this flag is set, then the 'this' variable is inserted into the scope chain before the activation object. This means that if a variable is not found in the local variable context (activation object), the interpreter will then search in the current 'this' variable of the function.

jseReadOnly - This is a read-only variable. Any attempt to write to the variable will fail (nothing will happen).

Returning values from a wrapper method

To return a primitive value from a wrapper method, use the appropriate jseReturnXXX() method. To return a long, use jseReturnLong(), e.g. Both require only one parameter, the value to be returned.

```
// Return a long from a ScriptEase wrapper method.
jsecontext.jseReturnLong(3006);
```

```
// Return a float from a ScriptEase wrapper method.
jsecontext.jseReturnNumber(22.22);
```

Returning an object requires a call to jseReturnVar(). jseReturnVar() can be used to return data of any type, although it is easier to use the typed functions (for example, jseReturnLong(),) if possible. jseReturnVar() puts a generic data type on the jseStack to be returned to the script.

jseReturnVar() takes two arguments:

```
jsecontext.jseReturnVar(JseVariable, jseReturnType);
```

The first argument is the JseVariable to return, and the second argument defines the return action. This argument tells the ScriptEase engine what to do with the variable once the function has returned and the statement that called it has completed. Possible values for this parameter are:

jseRetTempVar - the variable will be destroyed when popped from the stack.Use this option when your wrapper method creates a variable. jseRetTempVar will delete the variable when it is no longer needed, so you do not need to call jseDestroyVariable() on it.

jseRetCopyToTempVar - the variable is copied, and the temporary copy is put on the stack, to be destroyed when popped. The original variable must still be destroyed.

jseRetKeepLVar - the variable will not be put on the stack. It will not be automatically destroyed; you must call jseDestroyVariable() to delete it.

In nearly all cases this should be set to jseRetTempVar.

Using inner class methods

You may use an inner class method instead of a wrapper method if you prefer. Any of the samples in this manual can be easily converted to inner class methods by embedding them in the following code:

```
public JseLibraryFunction MySumFunction()
{
   return new JseLibraryFunction()
   {
      public void libraryFunction(JseContext jseContext)
      {
         /* body of function goes here */
      }
   };
}
```

Unless otherwise stated, inner class and wrapper methods are treated identically throughout the API and this manual.

Passing and returning simple data types

Passing and returning one of the primitive data types (numbers, strings and booleans) involves calling jseFuncVarNeed() to get the appropriately typed jseVariable and then calling jseGetXXX() to extract its value.

The following example is a wrapper for a function that simply adds two integers and returns the result. It would be invoked from the script source like this:

```
sum = MySumFunction(var1, var2);
```

Here is the wrapper method:

```
public void MySumFunction(JseContext jsecontext)
{
   JseVariable jseInt1;
   JseVariable jseInt2;
   int        Cint1;
   int        Cint2;
   int        SumInt;

      jseInt1 = jsecontext.jseFuncVarNeed(0, JSE_VN_NUMBER);
   jseInt2 = jsecontext.jseFuncVarNeed(1, JSE_VN_NUMBER);
   if (jseInt1 == null || jseInt2 == null)
   {
      return;
   }
   Cint1 = jsecontext.jseGetLong(jseInt1);
   Cint2 = jsecontext.jseGetLong(jseInt2);

      SumInt = Cint1 + Cint2;

      jsecontext.jseReturnLong(SumInt);
}
```

If an invalid parameter is passed in, jseFuncVarNeed() returns *null* and calls the user-defined error function. The interpreter doesn't quit until the wrapper method ends, so you must exit the function before it tries to use the bad data. Calls to jseFuncVarNeed() are usually put at the beginning of the function to ensure that all of the data is valid before continuing with the function.

If the call to jseFuncVarNeed() is successful, it returns a variable that holds the value of the corresponding parameter. Call jseGetLong() to extract the value from the variable.

Returning an integer requires just one function call, jseReturnLong(). This function assigns the Java variable's value to the JseContext. The interpreter will internally allocate and free the resources needed to hold the value.

Passing and returning strings and boolean values is essentially the same procedure. Strings and booleans both have their own type parameters to jseFuncVarNeed(): JSE_VN_STRING for strings and JSE_VN_BOOLEAN for boolean values. Use jseGetString() to extract data from strings and jseGetBoolean() to extract data from booleans. Use these functions in place of jseGetLong() in the example. To return strings or booleans use jseReturnVar().

For example, here is a script that passes and returns a string:

```
public void PromptAndGetS(JseContext jsecontext)
{
  java.io.BufferedReader in = new java.io.BufferedReader(
            new java.io.InputStreamReader(system.in));

  String foo = "";
  JseVariable   MyjseBuffer;
  String        szTextBuffer;

  MyjseBuffer = jsecontext.jseFuncVarNeed(0, JSE_VN_STRING);
  szTextBuffer = jsecontext.jseGetString(MyjseBuffer);
  try{
    foo = in.readLine();
  }
  catch (java.io.IOException ioe);
  jsecontext.jsePutStringLength(MyjseBuffer, szTextBuffer,
                                szTextBuffer.length() );
  jsecontext.jseReturnVar(MyjseBuffer, jseRetCopyToTempVar);
  return;
}
```

Passing simple data types by reference

In addition to its return value, your wrapper method can return data directly via its parameters. This is not possible in Java, but is allowed in JavaScript functions declared as cfunctions (or when the jseDefault_C_Behavior flag is set), since cfunctions receive parameters passed by reference and not by value. For example, suppose you had a JavaScript function that modified a number in some way:

```
num = 10;
ModifyNumber(num);
if( num == 0 ) exit(EXIT_ERROR);
```

Here is wrapper method for such a function:

```
void ModifyNumber(jseContext jsecontext)
{
   double   jseL;
   jseL = jsecontext.jseFuncVar(0);

   // If returned null this type can't convert to an integer
   if ( null != jsecontext.jseConvert(jseL, jseTypeNumber) )
   {
      jsecontext.jsePutNumber(jseL, GetANumber());
   }
   return;
}
```

When this function returns, *num* will have been set to the value returned by GetANumber().

In the example above there is no type checking on the JseVariable. Its type is of no importance because we will use jseConvert() to ensure that the variable is of the correct type.

You still need to associate the parameter offset with a JseVariable. Pass the parameter offset to jseFuncVar() to get a JseVariable. Now, convert the JseVariable into one that will hold a long with jseConvert(). jseConvert() requires the JseVariable to convert and the new variable type. In this case, we convert *num* to jseTypeNumber. Finally, the value returned by GetANumber() is passed to jsePutNumber(). Calling jsePutNumber() inserts the Java variable's value into the JseVariable. Upon returning to the script, the script's variable holds the value returned by the function GetANumber().

Working with objects

Passing and returning objects involves an additional step. As with primitive data types, first get a JseVariable for the object with jseFuncVarNeed(). Then get a handle to the property by calling jseMember(). The data may now be extracted from this second JseVariable with a call to one of the jseGetXXX() functions.

jseMember() has three parameters: the name of the object whose members are being accessed, the name of the property being accessed, and its data type.

```
public void jseObjectFunc(JseContext jsecontext)
{
  JseVariable   jseVarObject;
  JseVariable   jseVar;
  int           integer = 0;
  String        string = "";

  jseVarObject = jsecontext.jseFuncVarNeed(0, JSE_VN_OBJECT);
  if (jseVarObject == null) return;
  jseVar = jsecontext.jseMember(jseVarObject, "MyInt",
                                jseTypeNumber );
  if (jseVar != null)
  {
    integer = (int)jsecontext.jseGetLong(jseVar);
  }
  jseVar = jsecontext.jseMember(jseVarObject, "MyString",
                                jseTypeString);
  string = jsecontext.jseGetString(jseVar);
  System.out.print("string = " + string + "integer = " +
                   new Integer(integer).toString() + "\n");
}
```

The example above gets an object with two properties from the interpreter. One of the properties (MyString) is a string, and one of them (MyInt) is a number; they will be stored in the variables `integer` and `string`, respectively, and printed to the screen.

Since jseMember() doesn't create a new JseVariable reference in creating new object properties, you shouldn't try to destroy them when you are through with them. Child variables will be cleaned up by the interpreter engine when the parent object is destroyed.

Functions with a variable number of arguments

Unlike Java, JavaScript functions may accept a variable number of arguments. For example, consider the following function, which takes a string as its first parameter, and has an optional second parameter, a number:

```
ret = OneOrTwoArgs("My Dog Has Fleas");     // returns "M"
ret = OneOrTwoArgs("My Dog Has Fleas", 7);  // returns "H"
```

If the integer parameter is not provided, the function returns the first character of the string. If an integer is provided, the character returned will be at the index position specified by the integer.

Since the first argument is mandatory, there is no need to treat it differently. It may be accessed just as in the previous examples. However, you must determine whether the second parameter exists before you try to extract a value from it. The function jseFuncVarCount() returns the number of parameters passed to the ScriptEase function. If the variable exists, make the usual jseFuncVarNeed() and jseGetLong() calls to check the JseVariable type and extract its data.

```
public void OneOrTwoArgs(jseContext jsecontext)
{
  JseVariable  MyjseString;
  JseVariable  MyjseOptNum;
  String       MyJavastr;
  int          MyJavaoptNumber;
  int          index;

  MyjseString = jsecontext.jseFuncVarNeed(0, JSE_VN_STRING);
  if (MyjseString == null)
  {
    return;
  }
  int str_len;
  MyJavastr = jsecontext.jseGetString(MyjseString);
  MyjseOptNum = 0;
  if (jsecontext.jseFuncVarCount() == 2)
  {
     MyjseOptNum = jsecontext.jseFuncVarNeed(1, JSE_VN_NUMBER);
     MyCoptNumber = jsecontext.jseGetLong( MyjseOptNum);
  }
  if (MyjseOptNum < MyJavastr.length) index = MyjseOptNum;
      else index = 0;
  jsecontext.jseReturnLong(MyJavastr);
  return;
}
```

Accepting a ScriptEase argument of unknown type

If you do not know what type of variable is to be retrieved, you can use the function jseFuncVar() instead of jseFuncVarNeed(). jseFuncVar() will accept a variable of any type. You must then call jseGetType() to determine the variable's type. jseGetType() returns one of the following values, corresponding to the variable's type: jseTypeUndefined, jseTypeObject, jseTypeString, jseTypeBuffer, jseTypeNumber, jseTypeBoolean, jseTypeNull, or jseTypeNumber.

The following example demonstrates jseFuncVar().

```
    public void jseExternalLibFunc(JseContext jsecontext)
    {
       JseVariable   jseMysteryVar;
       String        MyString;
       double        MyNumber;
       boolean       MyBoolean;

       jseMysteryVar = jsecontext.jseFuncVar(0);
       switch(jsecontext.jseGetType(jseMysteryVar))
    {
          case jseTypeUndefined:
             /* Can set to a jseType here */
             jsecontext.jseConvert(jseMysteryVar, jseTypeNumber);
             break;
          case jseTypeString:
             MyString = jsecontext.jseGetString(jseMysteryVar);
             break;
          case jseTypeLong:
             MyNumber = jsecontext.jseGetNumber(jseMysteryVar);
             break;
          case jseTypeBoolean:
             MyBoolean = jsecontext.jseGetBoolean(jseMysteryVar)
             break;
          default:
             //ignore boolean, buffer and object values
       }
       return;
}
```

This function executes different code depending on the variable type passed. The correct data extraction function will be called against the ScriptEase variable no matter what the data type is.

Calling interpreted ScriptEase functions

When a script is being interpreted with jseInterpret() or has been loaded in jseInterpret(), its functions are registered with the interpreter. You can then call these functions directly

from Java with jseCallFunction(); this saves the interpreter from having to re-interpret the function. You can use jseGetNextFunction() to list all available local functions in a given JseContext.

There are five steps to calling previously loaded (via jseInterpret()) functions:

- Get a handle to the function variable with jseGetFunction(),
- Create a JseStack to manage the parameters passed to the function,
- Put the parameter variables on the stack with jsePush(),
- Make the function call with jseCallFunction(), and
- Destroy the JseStack with jseDestroyStack().

jseGetFunction() requires two parameters: the name of the function being called, and an error message flag. Set the flag to *true* if you want to use the default error handling system if the function cannot be found, and *false* if you want to use a different error handling system. If *false*, jseGetFunction() returns *null* if the variable is not a function or the processor cannot find the function, and you can take appropriate action. jseGetFunction() returns a handle to the function variable, which will be needed for the call to jseCallFunction().

Creating a jseStack is easily done by calling jseCreateStack(). It returns a handle to the newly created stack, which is used in subsequent calls to jsePush() and jseCallFunction().

Next use jsePush() to put JseVariables on the stack. You must call jsePush() once for each argument you are passing to the function. jsePush() takes three parameters: the handle of the stack you're working on, the variable to be pushed to the stack, and a boolean flag. Set this flag to *true* if you want the JseVariable to be automatically destroyed when the stack is destroyed (with jseDestroyStack(), i.e.). If it is set to *false*, you will have to call jseDestroyVariable() yourself to destroy the variable and free the resources allocated.

Now you are ready to make the function call with jseCallFunction(). jseCallFunction() takes four parameters: a JseVariable for the function being called, the JseStack, an array variable to store whatever value the function returns (the return value will be placed at offset 0 of this array), and a JseVariable to be used as the "this" variable within the function call.

jseCallFunction() returns *true* if the function was successfully executed, otherwise it returns *false*.

Creating and destroying jseVariables

ScriptEase variables are created with one of the following jseCreateXXX() methods, each of which creates and returns a variable of the specified type. All variables created with these methods must be explicitly destroyed with jseDestroyVariable() when you are done using it.

```
jseCreateVariable(int VType);
jseCreateSiblingVariable(jseVariable OlderSiblingVar,
                         int ElementOffsetFromOlderSibling);
jseCreateLongVariable(int value);
jseCreateConvertedVariable(JseVariable VariableToConvert,
                           int ConversionType)
```

There are two ways to destroy a ScriptEase variable:

- jseDestroyVariable() will destroy any variable created with the above calls.
- If RetAction is jseRetTempVar, jseReturnVar() will destroy the JseVariable after it is used. Do not destroy the variable explicitly if it is used as a return variable in this manner.

Interpreting a script with jseInterpret()

A script is executed with the JseContext method jseInterpret(). jseInterpret() takes seven parameters that indicate which script is to be run and how it will inherit variables from the context it is called from. Although there is a large number of possible combinations of these parameters, the three most commonly used situations are described at the end of this section.

```
boolean
jsecontext.jseInterpret(
      String sourceFile,
      String sourceText,
      byte[] pretokenizedBuffer,
      int jseNewContextSettings,
      int howToInterpret,
      JseContext localVariableContext,
      JseVariable[] returnVar);
```

jseInterpret() returns a boolean value to indicate the success or failure of the script, returning *true* if the script executed completely, and *false* if it did not. This value is in no way related to the value returned by the script, although if jseInterpret() returned *false*, no value will be returned from the function, since it failed to interpret.

sourceFile is a string containing the filename and path to a ScriptEase script file or null if you are interpreting ScriptEase source from memory. Any parameters that need to be passed to this file for execution should be passed in the following parameter (**sourceText**).

sourceText is either a string containing a block of ScriptEase code to be interpreted or the optional arguments to pass to the script (if interpreting code from a file).

howToInterpret specifies the interpreter's mode:

> **JSE_INTERPRET_NO_INHERIT** indicates that the new JseContext should not inherit global variable and functions from its parent JseContext.

JSE_INTERPRET_CALL_MAIN instructs the interpreter to run the main() function following any global code.

JSE_INTERPRET_LOAD - Interpret and execute the script within the current JseContext so that the interpreted functions and variables are available to subsequent calls to jseInterpret(). (This option is maintained for backwards compatibility with earlier versions of the ISDK. We recommend that you use JSE_INTERPRET_DEFAULT instead.

JSE_INTERPRET_DEFAULT - Variables and methods will be inherited from the calling context, and main will not be called.

The **jseNewContextSettings** parameter determines which of the jseContext elements will be created anew in the child JseContext. If a new context element is not specified, it will be inherited from the parent JseContext. Use one or more of the following flags or'ed together:

jseNewNone - Do not create any new elements.

jseNewDefines - Create new defines.

jseNewLibrary - Create new function libraries.

jseNewGlobalVarGroup - Create a new global variable group.

jseNewFunctions - Create new scripted functions.

jseNewAtExit - Create new atexit functions.

jseNewSecurity - Reinitialize the security script (i.e., call SecureInit() before running the script; see page 9).

jseAllNew - Create new elements for all categories above, except for functions, which will be inherited from the parent JseContext.

LocalVariableContext tells a new level of jseInterpret() that local variables of that JseContext should be treated as global variables of the new interpretation. For example,

```
foo()
{
   a = 4;
   interpret("a++");
}

is only identical (in result) to,
foo()
{
   a = 4;
   a++;
}
```

- if the new level of interpret knows to make local variables of foo() be global variables of the new interpret.

ReturnVar - If a script includes a return or exit statement and return value, it will be stored in ReturnVar[0] when jseInterpret() returns. If there is no specified return variable, an undefined value will be returned.

If JseInterpret() returns *true* then you are responsible for destroying this returned variable with jseDestroyVariable(). Pass in *null* if you do not need to receive the returned value.

jseInterpret() - flags for the most common situations

These are the three most commons situations encountered when executing a script:

- You want to execute the code as if it were the only thing running; all variables created will be destroyed.
- You want your code to be able to use all variables that are currently available for the jseContext, and all variables created by the script will remain after the script terminates.
- You want your code to be able to use all variables that are currently available for the jseContext, but you don't want the variables created by the script to remain after the script terminates.

The flags to use for the jseNewContextSettings and howToInterpret for these three situations are as follows:

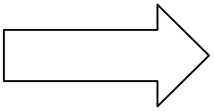
1. jseNewContextSettings = jseAllNew & ~jseNewSecurity

   howToInterpret = JSE_INTERPRET_CALL_MAIN |

   JSE_INTERPRET_NO_INHERIT
2. jseNewContextSettings = jseNewNone

   howToInterpret = JSE_INTERPRET_CALL_MAIN
3. jseNewContextSettings = jseNewFunctions

   howToInterpret = JSE_INTERPRET_CALL_MAIN



Application Programming Interface

The following methods are listed for reference in alphabetical order.

A Note on Terminology

`jseContext` refers to the JseContext class. Most of the API functions are members of this class, so when a function template reads:

```
jseVariable jseContext.jseActivationObject()
```

The "`jseContext.`" indicates that jseActivationObject() is a member of the JseContext class.

`jsecontext` (lower-cased) refers to a specific instance of this class. It is used in code samples and would be replaced in actual code by another JseContext instance you are actually working with.

If the code fragment was:

```
a = jsecontext.jseActivationObject(),
```

and your JseContext instance was, for example, "foo", you might have,

```
jseContext foo = …;
A = foo.jseActivationObject()
```

JseActivationObject

DESCRIPTION	Get the local variable object for the function currently being executed.
SYNTAX	`JseVariable JseContext.jseActivationObject();`
RETURN	This method returns the current activation object, whose members are the local variables, of the last local (script) function. Thus the local variable "a" of the last script function would be a member of this object.
SEE ALSO	jseGlobalObject

jseAddLibrary

DESCRIPTION Add an external method library to a given JseContext.

SYNTAX

```
void JseContext.jseAddLibrary(string
          objectVariableName,
          JseFunctionDescription[] FunctionList,
          JseLibrary LibraryObject);
```

PARAMETERS objectVariableName - The name of the object the properties will be associated with. If *null* is supplied in this field, then the global object will be used.

LibraryObject - the object which implements JseLibrary, associated with this library.

functionList - An array of function descriptions to add to the context.

COMMENTS Use this function to add library functions to the jseContext. A static table of JseFunctionDescription structures is defined, and this table is passed as the second parameter to the function.

Here is an example of usage:

```
JseFunctionDescription[] myFuncs={
JseLib.JSE_LIBMETHOD(“foo,” “fooFunc”)
0,0, JseDefaultAttr, JseDefaultAttr)
};
jsecontext.jseAddLibrary(“MyFuncs”,myFuncs,
MyLibraryObject);
```

RETURN None.

SEE ALSO jseFunctionDescription, jseLibraryData

jseAppExternalLinkRequest

DESCRIPTION Create a new JseContext using the jseAppLinkFunc provided in the jseExternalLinkParameters structure.

SYNTAX

```
JseContext.
jseAppExternalLinkRequest(boolean Initialize)
```

PARAMETERS **Initialize** - The second parameter passed to the AppLink function.

COMMENTS If the applicable object implements the JseAppExternalLinkRequest interface, it is used to create a new JseContext. This function should first be called with *true* as the second parameter to initialize a new context, and then be called a second time with *false* in order to clean up the returned context.

EXAMPLE

```
newcontext = jsecontext.JseAppExternalLinkRequest(
   true);
if( newcontext == null )
   PrintError( "Initialization failed");
/... Use the new context here ... */
newcontext.jseAppExternalLinkRequest(
     false);
```

RETURNS *null* on failure, otherwise a valid JseContext.

NOTE This function is not used frequently. You should use jseInitializeExternalLink() instead. It is provided for compatibility with the C version of the ScriptEase API.

SEE ALSO jseInitializeExternalLink, jseTerminateExternalLink

jseAssign

DESCRIPTION Copy the current value of one variable to another variable.

SYNTAX

```
boolean
JseContext.jseAssign(JseVariable destVar,
                     JseVariable srcVar );
```

PARAMETERS **destVar** - The ScriptEase variable to set

srcVar - The ScriptEase variable to assign from.

This function assigns the value of the ScriptEase data defined by destVar to be equivalent to the value of the ScriptEase data defined by srcVar. This function provides the same functionality as the '=' operator.

RETURN Return boolean *true* for success, else it returns *false* if the assignment was unsuccessful.

SEE ALSO jseGetType, jseConvert, jseCreateConvertedVariable

jseBreakpointTest

DESCRIPTION Test to see if the current line is a valid breakpoint.

SYNTAX

```
boolean
JseContext.jseBreakpointTest(String fileName,
                             int lineNumber);
```

PARAMETERS **fileName** - Name of the file to be tested.

lineNumber - The line number to check for breakpoint

COMMENTS Check if currently-running script has a breakpoint in this **fileName** at this **lineNumber**. This function is provided to facilitate debugging.

RETURN Return *true* if on a valid breakpoint, else return *false*.

SEE ALSO jseLocateSource

jseCallAtExit

DESCRIPTION Add a function to be called when exiting a jseContext

SYNTAX

```
void
JseContext.jseCallAtExit(String exitFunction,
                         object exitObject);
```

PARAMETERS **exitFunction** - The function to call at exit.

exitOject - the object of which this function is a method.

COMMENTS The object must have a method of the given name of the format "void function(JseContext);". This method will be called when the top-level interpreted JseContext is destroyed. Any number of functions may be registered with jseCallAtExit(); they will be called in the reverse order in which they're added. At-exit functions are called regardless of the reason for the exit. If an error condition exists, the error flag will be turned off while calling these functions. These functions will be called before any libraries added with jseAddLibrary are terminated.

Note: this function is very "un-Java-like." It has this format for compatibility with the ScriptEase ISDK/C API.

RETURN None.

SEE ALSO jseAtExitFunc

jseCallFunction

DESCRIPTION Call a ScriptEase function.

SYNTAX

```
boolean
JseContext.jseCallFunction(JseVariable jsefunc,
      JseStack jsestack,
      JseVariable[]  returnVar,
      JseVariable thisVar);
```

PARAMETERS **jsefunc** - The function to be called.

jsestack - The parameters to pass to the specified function.

returnVar - Variable in which to place the return variable. The value will be stored at **returnVar**[0]. Use *null* if the return variable will be ignored by the script. If jseCallFunction() fails, no value will be returned.

thisVar - The variable to be used as the 'this' var; use *null* for the global object.

COMMENTS This function is used to make a call to a ScriptEase function from within your application.

RETURN **Returns** *true* if the call was successful, *false* otherwise. The context error flag will have been cleared when this function returns, therefore you should use this return value to determine if the function failed.

Note: the returned variable must **not** be destroyed with jseDestroyVariable().

SEE ALSO jseCurrentFunctionName, jseGetFunction, jseCreateStack, jseDestroyStack, jsePush

jseCompare

DESCRIPTION Compare two script variables for greater-than, less-than or equal comparison.

SYNTAX

```
boolean
JseContext.jseCompare(JseVariable variable1,
      JseVariable variable2,
      int[] compareResult);
```

PARAMETERS **variable1** - The first variable to compare.

variable2 - The second variable to compare.

compareResult - Integer holder to store the result of this function. On return, the first element of this array (i.e. compareResult [0]) will be set to:

< 0 if variable 1 is less than variable 2
0 if variable 1 is equal to variable 2
> 0 if variable 1 is greater than variable 2.

COMMENTS This routine compares two JseVariables. In its most basic form, it simply compares if two variables are equal, in that the data they contain are equivalent, or that they point to the same object. In addition, one of the following predefined values can be passed as compareResult to use the standard ECMAScript comparison routines:

JSE_COMPEQUAL - Compare using ECMAScript equality rules.

JSE_COMPLESS - Compare using ECMAScript less-than rules (different from equality rules).

Typically, to do ECMAScript comparisons, the user should never call this function directly. Use the functions jseCompareLess() and jseCompareEquality(), which map to the equivalent flags above.

RETURN In a standard comparison, *true* if this call succeeds, *false* if it fails (comparing incomparable types). When using JSE_COMPEQUAL, returns a boolean value as to whether the two variables are equal. When using JSE_COMPLESS, returns a boolean value as to whether the first variable is less than the second variable.

SEE ALSO jseEvaluateBoolean, jseAssign, jseCompareLess, jseCompareEquality

jseCompareEquality

DESCRIPTION Compare two script variables for equality using ECMAScript rules.

SYNTAX

```
boolean
JseContext.jseCompareEquality(
      JseVariable variable1,
      JseVariable variable2);
```

PARAMETERS **variable1** - The first variable to compare.

variable2 - The second variable to compare.

COMMENTS This function is equivalent to calling jseCompare with the result value JSE_COMPEQUAL.

If one variable is a string and the other a number, the string will be converted to a number before comparing. Boolean values will be converted to numbers before being compared.

RETURN *True* if the variables are equal to each other, *false* if they are not.

SEE ALSO jseEvaluateBoolean, jseAssign, jseCompare, jseCompareLess

jseCompareLess

DESCRIPTION See if one variable's value is less than another's, using ECMAScript rules.

SYNTAX

```
boolean
JseContext.jseCompareLess(JseVariable variable1,
                          JseVariable variable2);
```

PARAMETERS **variable1** - The first variable to compare.

variable2 - The second variable to compare.

COMMENTS This function converts variables to primitive values before they are compared. If the two variables are both strings, they will be compared as strings; otherwise they will be converted to numbers and compared.

RETURN *True* if variable1 is less than variable2, *false* if variable1 is greater than or equal to variable2.

SEE ALSO jseEvaluateBoolean, jseAssign, jseCompareEquality, jseCompare

jseConvert

DESCRIPTION Convert a variable to a new jseDataType.

SYNTAX

```
void
JseContext.jseConvert(JseVariable variable,
                      int dType );
```

PARAMETERS **variable** - The ScriptEase variable to convert.

dType - The data type the variable is being converted to.

COMMENTS This function changes a variable from one type to another. This function does not preserve the current contents of the variable, but instead is much like destroying the previous variable and creating a new variable with this type. If the variable is already of the specified type, no conversion is performed and no data is lost.

RETURN None.

SEE ALSO jseDataType, jseGetType, jseAssign

jseCopyBuffer

DESCRIPTION Copy a section of a buffer from a JseVariable to a local buffer.

SYNTAX

```
int JseContext.jseCopyBuffer(JseVariable variable,
                             byte[] buffer,
                             int start,
                             int length);
```

PARAMETERS **variable** - The buffer variable which contains the data to be copied.

buffer - The local buffer that will be filled with the copied data.

start - The offset within the variable where the copying will start from.

length - The length of data to be copied from the buffer variable.

RETURN None.

SEE ALSO jseCopyString, jseGetBuffer

jseCopyString

DESCRIPTION Copy string data from a jseVariable to a user allocated buffer.

SYNTAX
```
int JseContext.jseCopyString(JseVariable variable,
                             byte[] string,
                             int start,
                             int length);
```

PARAMETERS **variable** - The variable containing the string to be copied.

buffer - The buffer which will be filled with the string data

start - The offset in the variable of the first character to be copied.

length - The length of the string to be copied from the variable.

RETURN None.

SEE ALSO jseCopyBuffer, jseGetString

jseCreateCodeTokenBuffer

DESCRIPTION Compile a block of ScriptEase code into executable tokens

SYNTAX
```
byte[] JseContext.jseCreateCodeTokenBuffer(
   String source
   Boolean sourceIsFileName);
```

PARAMETERS **source** - ScriptEase source code to tokenize.

sourceIsFileName - *true* if Source is a filename, else *false* if Source is a block of code.

COMMENTS This call will compile the code in the source parameter into a binary sequence of tokens which can later be executed with jseInterpret or jseInterpInit by passing the returned buffer as the tokenized code parameter

RETURN The return value is a byte array of compiled tokens.

SEE ALSO jseInterpret, jseInterpInit

jseCreateConvertedVariable

DESCRIPTION Create a new variable from another variable and convert its data.

SYNTAX

```
JseVariable
JseContext.jseCreateConvertedVariable(
      JseVariable variableToConvert
      int targetType);
```

PARAMETERS **variableToConvert** - variable to be used as a model for the new variable.

targetType - type of variable to convert to.

jseToBoolean - Convert to a boolean value. The contents of the variable depends on the original variable.

- jseTypeUndefined
- jseTypeNull
- jseTypeBoolean
- jseTypeBuffer
- jseTypeNumber
- jseTypeString
- jseTypeObject

jseToBuffer - Convert to a buffer type. This conversion is done in the same manner as jseToString, but it is converted to an ASCII sequence of bytes, rather than a Unicode string.

jseToBytes - Convert to a buffer type, but instead of converting each unicode value to a corresponding ASCII value, a raw transfer of data is done. That is, the Unicode string "Hi" would be converted to the buffer '\0H\0i' or 'H\0i\0', depending on the endianness of the system, and a floating point value would give the actual bytes that share it rather than a test representation of the value.

jseToInt32 - Convert to a 32-bit integer. This is done by converting like jseToInteger does except the range of valid values is 0 to 0xffffffff.

jseToInteger - Convert to an integral type. The value is first converted with jseToNumber. If the result is NaN, then return +0. If the result is +0, -0, +inf, or -inf, return 0. Otherwise, return sign(result) * floor(abs(result)). In other words, the value -4.8 would be converted to -4, shortened to fit. Only values in the range -0x80000000 to 0x7fffffff can be stored.

jseToNumber: Convert to a jseTypeNumber variable based on its type as follows:

- jseTypeUndefine: NaN
- jseType`null`: +0
- jseTypeBoolean: The result is 1 if the argument is *True*. The result is+0 if the argument is *False*.
- jseTypeBuffer: Same as jseTypeString
- jseTypeNumber: Same as original
- jseTypeString: The string is interpreted as a number, using a complicated set of rules, which are intended to convert human-readable number strings such as "45" and "-45.34" to numbers. If there is an error converting the string to a number, then the result is NaN. More information on these rules can be found in the ECMAScript Language Specifications in section 9.3.
- jseTypeObject - Convert input using jseToPrimitive, then convert result with jseToNumber, and return the result.

jseToObject - Convert to an Object. If the original type is jseType *null* or jseTypeUndefined, then a runtime error is generated. No conversion is done if the original type is an object. Otherwise, the value is converted to the corresponding object wrapper type (i.e. for jseTypeString, new String() will be called with the value as the parameter).

jseToPrimitive - No conversion is done if the variable is any type but jseTypeObject. Otherwise, the internal **defaultValue()** function of the object is called and that value returned.

jseToString - Convert to a string based on this table:

jseTypeUndefined - "undefined"jseTypeNull - "`null`"

jseTypeBoolean - If the argument is *True*, then the result is "True", if the argument is `false`, then the result is "False".

jseTypeString - No conversion done

jseTypeObject - Convert with jseToPrimitive on the object then convert the result with jseToString

jseToUint16 - Convert to an unsigned 16-bit integer. Convert with **jseToInteger**, and then preserve the least significant 16 bits as an unsigned value.

jseToUint32 - Convert to an unsigned 32-bit integer. Convert with **jseToInt32**, and then convert to be unsigned.

COMMENTS This function will convert the **variableToConvert** into a variable of the new **targetType** using the standard ECMAScript conversion rules. See the description of jseConversionTarget for a description of these rules. This differs from jseConvert in that it uses ECMAScript conversion, rather than simply erasing the data and creating a blank type.

RETURN If successful, a new JseVariable of the type specified by **targetType** and containing the same value as variableToConvert (converted to the specified type). Returns *null* if the interpreter is unable to convert the variable to a requested type.

SEE ALSO jseConvert, jseCreateVariable, jseCreateSiblingVariable, jseDestroyVariable, jseConversionTarget

jseCreateFunctionTextVariable

DESCRIPTION Return the source text of a function.

SYNTAX

```
JseVariable
JseContext.jseCreateFunctionTextVariable(
      JseVariable functionVariable);
```

PARAMETERS **functionVariable** - Variable to get the source from.

COMMENTS This function takes a variable and returns the source text of the function. This is equivalent to calling toString() on the function. You must destroy the variable using jseDestroyVariable when you are done with it.

RETURN Returns a string containing the source text of functionVariable.

SEE ALSO jseCreateVariable, jseCreateSiblingVariable, jseDestroyVariable

jseCreateLongVariable

DESCRIPTION Shortcut to create a ScriptEase variable of an integer value.

SYNTAX
```
JseVariable
JseContext.jseCreateLongVariable(int value);
```

PARAMETERS **value** - Value to initialize this ScriptEase variable to.

COMMENTS This function creates a ScriptEase variable of type jseTypeNumber and puts the specified value in the variable. This is equivalent to creating a variable of type jseTypeNumber and then calling jsePutLong() to put a value into it.

RETURN If successful, a pointer to the JseVariable created. If there is not enough system memory to create the variable (extremely unlikely), *null* will be returned.

This variable must be destroyed with jseDestroyVariable() when you are done with it.

SEE ALSO jseCreateVariable, jseCreateSiblingVariable, jseCreateConvertedVariable, jseDestroyVariable, jsePutLong

jseCreateSiblingVariable

DESCRIPTION Create a ScriptEase Sibling Variable.

SYNTAX

```
JseVariable
JseContext.jseCreateSiblingVariable(
      JseVariable olderSiblingVar,
      int elementOffsetFromOlderSibling );
```

PARAMETERS **olderSiblingVar** - The variable that you are basing the new sibling variable on.

elementOffsetFromOlderSibling - The index into the array you are creating this sibling variable from (if a string or buffer).

COMMENTS This routine creates a sibling ScriptEase JseVariable. A sibling variable is a variable that references an already existing ScriptEase Variable. Changes to sibling variables affect each other. The offset parameter is used in conjunction with buffer and string variables, as it specifies an offset into the data at which to begin the sibling variable. The original variable and the sibling variable still reference the same variable, but calling jseGetString on the new variable will start at the new offset into the original.

EXAMPLE

```
JseVariable original =
jsecontext.jseCreateVariable (jseTypeString);
jsecontext jsePutString(original,"one two");
jseVariable news =
jsecontext.jseCreateSiblingVariable
(original,4);
String data = jsecontext.jseGetString (news);
/* data now points to "two", and any changes to
 * original or new will affect the other
 */
```

RETURN If successful, a pointer to the sibling jseVariable created. If there is not enough system memory to create the variable (extremely unlikely), *null* will be returned. This variable must be destroyed with jseDestroyVariable() when you are done with it.

SEE ALSO jseCreateVariable, jseCreateConvertedVariable, jseCreateLongVariable, jseDestroyVariable

jseCreateStack

DESCRIPTION Create a jseStack.

SYNTAX
```
JseStack
JseContext.jseCreateStack();
```

COMMENTS This function creates a JseStack which is used for pushing parameters and calling functions from within the ISDK.

RETURN Returns a pointer to the new JseStack. *null* will be returned if there is insufficient memory to create the stack. You must destroy the stack using jseDestroyStack() when you are done with it.

SEE ALSO jseCallFunction, jseDestroyStack, jsePush

jseCreateVariable

DESCRIPTION Create a jseVariable of a given type.

SYNTAX
```
JseVariable
JseContext.jseCreateVariable(int VType);
```

PARAMETERS **VType** - The type of ScriptEase variable to create.

RETURN If successful, a pointer to the JseVariable is created. If there is not enough system memory to create the variable (extremely unlikely), *null* will be returned. You must destroy this variable with jseDestroyVariable() when you are done with it.

SEE ALSO jseCreateSiblingVariable, jseCreateConvertedVariable, jseCreateLongVariable, jseDestroyVariable

jseCreateWrapperFunction

DESCRIPTION Create a JseVariable object that is a callable function.

SYNTAX
```
JseVariable
JseContext.jseCreateWrapperFunction(
      JseFunctionDescrioption desc,
      Object caller);
```

PARAMETERS **desc** - A function description created with "new JseFunction Description (...)"

Object - a JseLibrary of which these functions are a part.

RETURN If successful, this returns the JseVariable created. If there is not enough system memory to create the variable (extremely unlikely), *null* will be returned. This variable must be destroyed by calling jseDestroyVariable() when you are done with it. The variable is a function object which will call your wrapper function.

jseCurrentContext

DESCRIPTION Return the current JseContext based on any level of previous context.

SYNTAX
```
JseContext
JseContext.jseCurrentContext();
```

COMMENTS This function returns the most current JseContext. This may be used in interrupt-type situations to located the JseContext of the current depth of scripted function calls.

RETURN The current context for the current thread of execution.

SEE ALSO jsePreviousContext

jseCurrentFunctionName

DESCRIPTION Get the currently executing ScriptEase function.

SYNTAX
```
String
JseContext.jseCurrentFunctionName();
```

COMMENTS Returns the name of the current function.

RETURN Name of the function currently executing. Returns *null* if a function is not currently executing.

SEE ALSO jseGetFunction

jseDeleteMember

DESCRIPTION Delete a property of an object.

SYNTAX
```
void
JseContext.jseDeleteMember(JseVariable objectVar,
                           String name );
```

PARAMETERS **objectVar** - ScriptEase variable pointer.

name - The name of the object property to delete.

COMMENTS This function deletes a property of an object. This function ignores the jseDontDelete attribute (which is only used for the 'delete' operator within scripts).

RETURN None.

SEE ALSO jseGetMember, jseGetNextMember

jseDestroyStack

DESCRIPTION Destroy a JseStack.

SYNTAX
```
void
JsecContext.jseDestroyStack(JseStack stack);
```

PARAMETERS **stack** - The stack to destroy.

COMMENTS This function destroys the specified stack.

RETURN None.

SEE ALSO jseCallFunction, jseCreateStack, jsePush

jseDestroyVariable

DESCRIPTION Destroy a ScriptEase variable.

SYNTAX

```
void
JseContext.jseDestroyVariable(
      JseVariable variable );
```

PARAMETERS **variable** - the ScriptEase variable to destroy.

COMMENTS Use this routine to free up the system resources allocated to a ScriptEase variable when it is no longer needed. Variables created with one of the jseCreateXXX() functions must be destroyed; passing a variable to jseReturnVar() with the flag “jseReturnTemp” is another way to release a variable you own.

This is probably the most easily misunderstood concept in the API. “Destroying a variable” does **not** mean destroying the contents of the variable. Instead, it means to destroy your handle or lock on the variable.

If this is the last such lock, then the contents are destroyed. When you get a JseVariable handle, sometimes it is a lock that you must destroy, but sometimes you must not destroy it. The description of the API function will specify which case it is, but the general rule is that if the API function has the word 'create' in it, you are getting a lock you must destroy.

The API jseReturnVar() in several modes accepts a lock that it will destroy when it is done; by passing the variable to it, you are transferring your lock. If you have a variable that you aren't supposed to destroy and pass it to this function, you will have a problem. Either use jseRetKeepLVar to tell jseReturnVar() not to destroy the variable or create a lock using jseCreateSiblingVariable() which you can then pass to it.

RETURN None.

SEE ALSO jseCreateVariable, jseCreateSiblingVariable, jseCreateConvertedVariable, jseCreateLongVariable,

jseReturnVar

jseEvaluateBoolean

DESCRIPTION Determine if a ScriptEase Variable is *true* or *false*.

SYNTAX
```
boolean
JseContext.jseEvaluateBoolean(
     JseVariable variable);
```

PARAMETER **variable** - The ScriptEase variable to test.

COMMENTS Test to see if a ScriptEase variable evaluates to *true* or *false*. Pass a variable of type jseTypeBoolean.

RETURN The boolean value of variable.

jseFindVariable

DESCRIPTION Search for a variable with a given name.

SYNTAX
```
JseVariable
JseContext.jseFindVariable(
   Stringname, int flags);
```

PARAMETERS **name** - The name of the variable sought.
flags - Either 0 or "jseCreateVar" to create a variable you must destroy.

COMMENTS This variable searches the current scope chain for a variable with the given name. Usually, you want to search the scope chain as it was for the function that called you, since someone will likely write something like:

```
function myfunc()
{
   var a;
   wrapper("a");
}
```

The 'a' refers to the 'a' from the point of view of the calling function, not your wrapper function (which does not have the locals of the calling function as part of its scope chain.) In most cases, thus, the correct way to call this function is to use `JseContext.jsePreviousContext()` as the context you pass to this function.

RETURN Returns the variable if it is found, *null* if no such variable can be found. Do not destroy the variable unless you use the flag "JseCreateVar".

SEE ALSO jseGetVariableName

jseFuncVar

DESCRIPTION Get a ScriptEase function wrapper argument.

SYNTAX
```
JseVariable
JseContext.jseFuncVar(int ParameterOffset);
```

PARAMETERS **ParameterOffset** - The offset of the argument you are trying to access starting at 0. Variables are passed from left to right.

COMMENTS This function gets a parameter passed to a wrapper function. It returns a JseVariable, but does no type checking.

RETURN Returns a JseVariable if a valid index is given. Otherwise returns *null*. If index is invalid then the error handling routines will have been called.

SEE ALSO jseFuncVarCount, jseGetFunction, jseFuncVarNeed

jseFuncVarCount

DESCRIPTION Get the number of parameters passed to a wrapper function.

SYNTAX
```
int
JseContext.jseFuncVarCount();
```

COMMENTS This function determines how many arguments were passed to a ScriptEase function.

RETURN The number of arguments passed to this wrapper function.

SEE ALSO jseFuncVar, jseGetFunction, jseFuncVarNeed

jseFuncVarNeed

DESCRIPTION Get a ScriptEase function wrapper argument and validate its type.

SYNTAX
```
JseVariable
JseContext.jseFuncVarNeed(int parameterOffset,
                          int jseVarNeeded );
```

PARAMETERS **parameterOffset** - The offset of the argument you are trying to access, starting at 0.

jseVarNeeded - The type of the argument you are trying to access. It can be one or more of the following values:

(If you are supplying two possible types, they should be OR'ed together.)

JSE_VN_UNDEFINED gets an undefined variable.

JSE_VN_NUMBER gets a number.

JSE_VN_NULL gets a *null* variable.

JSE_VN_STRING gets a string or byte array.

JSE_VN_BOOLEAN gets a boolean variable. JSE_VN_INT get a number that can be represented as a long with no loss of precision.

JSE_VN_FUNCTION gets a function object.

JSE_VN_BYTE gets a number and cast it as a byte.

JSE_VN_BUFFER gets a buffer.

JSE_VN_OBJECT gets an object.

JSE_VN_ANY accepts any variable type.

Jselib.JSE_VN_NOT() accepts any variable not passed as a parameter. For example:

Jselib.VN_NOT(JSE_VN_NUMBER | JSE_VN_STRING) will accept any variable that is not a number or a string.

Jselib.JSE_VN_CONVERT(from, to) This macro converts variables of the type indicated by the first parameter to the type indicated by the second parameter. For example:

Jselib.VN_CONVERT(JSE_VN_ANY, JSE_VN_STRING) will convert any type of variable received to a string. You cannot convert from JSE_VN_INT, JSE_VN_BYTE, or JSE_VN_FUNCTION.

Jselib.JSE_VN_COPYCONVERT This option indicates that if a variable must be converted (with JSE_VN_CONVERT() or with the jseOptLenientConversion option), a copy of the variable will be made and converted, so that the original variable retains its type and value. You may also use the macro JSE_FUNC_VAR_NEED(). If the index or type are invalid this macro will not return and the scripting session will be terminated.

JSE_VN_CREATE - create variable for explicit jseDestroyVariable.

JSE_VN_READ - variable is for reading only.

JSE_VN_WRITE - variable is for writing only.

COMMENTS This function is used to access function arguments to a ScriptEase wrapper function. It returns a JseVariable, and does type checking and possible conversion.

RETURN Returns a JseVariable if a valid index is given and the type specified is found. Otherwise returns *null*. If index is invalid or the type is incorrect, an error message will have been called, and you should return from the function.

SEE ALSO JseFuncVar, jseGetFunction, jseFuncVarCount

jseGetArrayLength

DESCRIPTION Get the span of elements in a ScriptEase variable object, string or buffer.

SYNTAX

```
int
JseContext.jseGetArrayLength(
      JseVariable variable,
      int[] MinIndex );
```

PARAMETERS **variable** - array variable for which to check the span.

MinIndex - When the function returns this will be set to the index value of the first element in the ScriptEase array. This value will not be greater than zero.

In evaluation objects, this function will only consider elements with numeric indices. For example with this code the length of "foo" is 4:

```
var foo= new object();
foo[3] = "hello"
foo.blah = "goodbye
```

COMMENTS This routine determines the size (length) of a ScriptEase object, string or buffer.

RETURN If **variable** is an object, returns the highest valid index of the object +1; only properties with numeric names (i.e., array elements,) will be considered. This function may be used to get the length of dynamic arrays (i.e., arrays not created with the Array() constructor function). If **variable** is a string or a buffer, it returns the length of the string or buffer.

SEE ALSO jseCreateVariable, jseCreateSiblingVariable, jseCreateLongVariable, jseDestroyVariable, jseSetArrayLength

jseGetAttributes

DESCRIPTION Get a variable's attributes.

SYNTAX
```
int
JseContext.jseGetAttributes(
      JseVariable variable );
```

PARAMETERS **variable** - The ScriptEase variable to read.

COMMENTS This function is used to access the data associated with a JseTypeByte variable.

RETURN The attributes assigned to variable.

SEE ALSO jseSetAttributes

jseGetBoolean

DESCRIPTION Get boolean from a JseVariable.

SYNTAX
```
boolean
JseContext.jseGetBoolean(jseVariable variable );
```

PARAMETERS **variable** - The ScriptEase variable to read.

COMMENTS This function retrieves the data associated with a jseTypeBoolean variable.

jseGetBuffer

DESCRIPTION Get buffer data from a JseVariable.

SYNTAX
```
byte[]
JseContext.jseGetBuffer(JseVariable variable);
```

PARAMETERS **variable** - The jseVariable for the buffer being accessed.

COMMENTS Get buffer data from a JseVariable. Buffer data can have binary and ‘/0' characters in'/0' the block and although it will always be ‘/0' terminated. The final ‘/0' is not considered part of the data and is not part of the length. The returned data can not be modified.

RETURN The buffer data.

SEE ALSO jseGetWritableBuffer, jseGetString

jseGetByte

DESCRIPTION Get the-byte value of a numeric variable.

SYNTAX
```
byte
JseContext.jseGetByte(JseVariable variable);
```

PARAMETERS **variable** - The ScriptEase variable to read.

COMMENTS This function gets the data associated with a jseTypeByte variable.

RETURN The value contained in the numeric variable as a byte.

SEE ALSO jsePutByte

jseGetCurrentThisVariable

DESCRIPTION Get the current “this” variable.

SYNTAX
```
JseVariable
JseContext.jseGetCurrentThisVariable();
```

COMMENTS This function is used to get the current “this” variable.

RETURN Returns the current “this” variable.

SEE ALSO jseGlobalObject

jseGetExternalLinkParameters

DESCRIPTION Get to the external link parameters structure.

SYNTAX
```
JseExternalLinkParameters
JseContext.jseGetExternalLinkParameters();
```

COMMENTS Use this function to get the external link parameter structure. Use the structure to temporarily change the options. You must remember to restore their original value.

RETURN JseExternalLinkParameters structure.

SEE ALSO jseInitializeExternalLink, jseTerminateExternalLink

jseGetFileNameList

DESCRIPTION This function returns a list of all files opened by the script.

SYNTAX
```
String[]
JseContext.jseGetFileNameList();
```

RETURN An array of strings representing containing the names of source files needed to run the script, *null* will be returned if there are no such files.

jseGetFunction

DESCRIPTION Get a ScriptEase function variable.

SYNTAX
```
JseVariable
JseContext.jseGetFunction(JseVariable object,
     String functionName,
     boolean errorIfNotFound);
```

PARAMETERS **object** - The object the function will be associated with. Use *null* to associate the function with the global object.

functionName - A string containing the name of the ScriptEase function you are searching for.

errorIfNotFound - If this flag is set to *true*, an error message will be displayed if the requested function can not be found.

COMMENTS This function gets a given ScriptEase Library function, or any other function in the script being executed.

RETURN A JseVariable for the requested function. This function will cause a temporary variable to be freed when the current context is ended, such as when returning from a wrapper function. To avoid the temporary variable (e.g. not calling from a wrapper or calling frequently) use jseMemberEx(...jseCreateVariable) and test that the variable is a function with jseIsFunction()

This function will return *null* if the requested function wasn't found.

SEE ALSO jseCallFunction, jseCurrentFunctionName

jseGetIndexMember

DESCRIPTION Get a JseVariable for a numerically indexed property.

SYNTAX

```
JseVariable
JseContext.jseGetIndexMember(
      JseVariable objectVariable,
      int index);
```

PARAMETERS **objectVariable** - The jseVariable from which to get a property.

index - The index of the desired property.

COMMENTS This routine gets a JseVariable for an object property. This function is intended for use with the numbered properties of array objects. To get a property that is named with a string, use jseGetMember().

RETURN JseVariable is returned or *null* if the index is invalid.

SEE ALSO jseGetNextMember, jseIndexMember, jseMember, jseDeleteMember, jseGetIndexMemberEx

jseGetIndexMemberEx

DESCRIPTION Get a JseVariable of a numerically indexed object property.

SYNTAX

```
JseVariable
JseContext.jseGetIndexMemberEx(
      JseVariable objectVariable,
      int index
      int flags);
```

PARAMETERS **objectVariable** - The JseVariable of the object from which to get a property.

index - The index of the desired property.

flags - see jseMemberEx in this chapter for flags.

COMMENTS This routine gets a JseVariable for an object property. This function is intended for use with the numbered properties of objects. To get a property that is named with a string, use jseGetMemberEx().

RETURN A JseVariable is returned or *null* if the index is invalid.

SEE ALSO jseGetNextMember, jseIndexMember, jseMember, jseIndexMemberEx, jseMemberEx, jseGetIndexMember, jseGetMember, jseGetIndexMemberEx, jseGetMemberEx, jseDeleteMember

jseGetJavaObject

DESCRIPTION Get the Java 'Object' variable associated with a JseVariable.

SYNTAX
```
Object JseContext.jseGetJavaObject(
     JseVariable var);
```

COMMENTS It is often useful to be able to associate an arbitrary Java item with a JseVariable and later retrieve it. This is anologous to storing a C pointer by casting it to an int. This function retrieves an Object previously stored via jseSetJavaObject().

Remember, any Java item object of any type can be cast to an (Object) and stored via this function. Cast it back to its original type when you retrieve it with this function. You can only store a Java Object in a JseVariable that is itself of type jseTypeObject.

RETURN The Java Object stored with jseSetJavaObject.

SEE ALSO jseSetJavaObject.

jseGetToolkitApp

DESCRIPTION Returns the toolkit application object for this JseContext.

SYNTAX
```
Object JseContext.jseGetToolKitApp()
```

RETURN Return the object passed to the jseIntializeExternalLink call which created this context.

SEE ALSO

jseGetLong

DESCRIPTION Get the long value of a numeric variable.

SYNTAX
```
int JseContext.jseGetLong(JseVariable variable );
```

PARAMETERS **variable** - The ScriptEase variable to read.

COMMENTS Use this function to access the data of a jseTypeNumber variable, cast to an integer value.

RETURN The value contained in the numeric variable as an integer.

SEE ALSO jsePutLong

jseGetMember

DESCRIPTION Get a JseVariable for the property of a ScriptEase object.

SYNTAX

```
JseVariable JseContext.jseGetMember(
      jseVariable objectVariable,
      String Name );
```

PARAMETERS **objectVariable** - The jseVariable to the object from which to get a property. Use *null* to indicate the global variable. The prototype will be searched.

Name - The name of the object property.

COMMENTS This routine gets a JseVariable for an object property.

RETURN A JseVariable to the requested object property, or *null* if the object does not exist.

SEE ALSO jseGetNextMember, jseMember, jseDeleteMember

jseGetMemberEx

DESCRIPTION Get a JseVariable for a ScriptEase object property.

SYNTAX

```
JseVariable
   JseContxt.jseGetMemberEx(
      JseVariable objectVariable,
      String[] name,
      int flags );
```

PARAMETERS **objectVariable -** The JseVariable of the object from which to get a property. Use *null* to indicate the global variable.

Name - The name of the object property.

flags - see jseMemberEx in this chapter.

COMMENTS This routine gets a JseVariable of an object property.

RETURN A JseVariable for the requested object property, or *null* on failure.

SEE ALSO jseMemberEx, jseGetNextMember, jseMember, jseMemberEx, jseIndexMember, jseIndexMemberEx, jseGetIndexMember, jseGetIndexMemberEx, jseDeleteMember

jseGetNextMember

DESCRIPTION Get the next property of an object.

SYNTAX

```
JseVariable JseContext.jseGetNextMember(
      JseVariable objectVar,
      JseVariable prevMemberVariable,
      String[] name );
```

PARAMETERS **objectVar** - The JseVariable for the object from which to retrieve properties. Use *null* to indicate the global variable.

prevMemberVariable - The previous object property. If this is set to *null*, the first member will be returned.

name - On return, the name of the object property that was returned at name[0].

COMMENTS This function allows you to get all the properties of a ScriptEase object variable by stepping through them one at a time. It isn't necessary to know the names of the properties. In the first call, *null* is provided as the previous property; the first property of the object will be returned. This function will return all properties, even those which have the JseDontEnum attribute set. You should check each variable's properties if you want to ignore such numbers.

RETURN A JseVariable for the next object property. This value should be used on subsequent calls to retrieve the next properties. When *null* is returned, there are no more object properties.

SEE ALSO jseGetMember, jseMember, jseDeleteMember

jseGetString

DESCRIPTION Get string data from a ScriptEase variable.

SYNTAX String JseContext.jseGetString (JseVariable variable);

PARAMETERS **variable** - The ScriptEase variable to read.

COMMENTS Get string data from a variable. The returned data must not be modified.

RETURN The data will be '\0'-terminated, but this terminating '\0' character is not considered part of the variable and not considered when determining the variable length. Note also that ECMAScript strings may contain embedded '\0''s.

SEE ALSO jseGetBuffer, jseGetWritableString, jseGetWritableBuffer, jseCopyString, jseCopyBuffer, jsePutString, jsePutBuffer

jseGetType

DESCRIPTION Get the type of a JseVariable.

SYNTAX
```
int JseContext.jseGetType(
      JseVariable variable );
```

PARAMETERS **variable** - The JseVariable whose type is being checked.

COMMENTS This function is used to determine the specified JseVariable's type.

RETURN The type of the variable passed as the argument. Valid types are jseTypeUndefined, jseTypenull, jseTypeNumber, jseTypeString, jseTypeBuffer, jseTypeObject, and jseTypeBoolean.

SEE ALSO jseConvert, jseAssign

jseGetVariableName

DESCRIPTION Get the name of a script variable corresponding to the given jseVariable.

SYNTAX
```
String JseContext.jseGetVariableName(
      jseVariable variableToFind);
```

PARAMETERS **variable** - The variable you want to get.

COMMENTS This function gets the name of the variable corresponding to variableToFind. For example, if there is an error in executing the script and you wish to inform the user that a variable is of the wrong type, you can use this function to get the name of the variable as it is referred to in the script.

RETURN *true* if successful, *false* if the variable was not found

SEE ALSO jseGetType, jseGetFunctionName

jseGetWriteableBuffer

DESCRIPTION Get buffer data from a JseVariable.

SYNTAX
```
byte[] JseContext.
   jseGetWriteableBuffer(JseVariable variable);
```

PARAMETERS **variable** - The jseVariable handle to the buffer being accessed.

COMMENTS Get buffer data from a variable. Buffer data can have binary and '\0' characters in the block.

RETURN The buffer data.

SEE ALSO jseGetBuffer, jseGetString

jseGetWriteableString

DESCRIPTION Get string data from a ScriptEase variable.

SYNTAX
```
String JseContext.
jseGetWriteableString(JseVariable variable)
```

PARAMETERS **variable** - The jseVariable handle to the string variable being accessed.

COMMENTS Get string data from a ScriptEase variable. Since Java strings are immutable, this funciton is identical to jseGetString(). It is provided for compatibility with the C API.

RETURN The string data contained in variable.

SEE ALSO jseGetString, jseGetNumber

jseGlobalObject

DESCRIPTION Get the current global object.

SYNTAX
```
JseVariable
JseContext.jseGlobalObject();
```

COMMENTS This function is used to get the current global object.

RETURN Returns a pointer to the current global object.

SEE ALSO jseGetCurrentThisVariable

jseIndexMember

DESCRIPTION Retrieve a numerically indexed variable from an object; create it if it does not exist.

SYNTAX

```
JseVariable.JseContext.jseIndexMember(
     JseVariable objectVar,
     int index,
     int jseDataType);
```

PARAMETERS **objectVar** - The object to query.

index - The index of the variable to retrieve.

jseDataType - The type of the desired variable.

COMMENTS This function is intended to get the numbered properties of objects. To get named properties, use jseMember().

RETURN The desired variable. If it does not exist it will be created.

SEE ALSO jseGetIndexMember, jseIndexMemberEx, jseGetIndexMemberEx

jseIndexMemberEx

DESCRIPTION Retrieve a variable from a numerically-indexed object; create it if it does not exist.

SYNTAX

```
JseVariable.JseContext
jseIndexMemberEx(jseVariable objectVar,
                 int index,
                 int type
                 int flags);
```

PARAMETERS **objectVar** - The object to query.

index - The index of the variable to retrieve.

type - The type of the desired variable.

flags - see jseMemberEx in this chapter.

COMMENTS This function is intended to get the numbered properties of objects. To get named properties, use jseMemberEx().

RETURN The desired variable. If it does not exist it will be created.

SEE ALSO jseIndexMember, jseGetIndexMember, jseGetIndexMemberEx

jseInitializeEngine

DESCRIPTION This call initializes the ScriptEase Interpreter Engine.

SYNTAX
```
int
jseInitializeEngine();
```

COMMENTS Call this before any other call in the toolkit to initialize the processor.

RETURN Returns the ID of the engine for version number verification.

SEE ALSO jseTerminateEngine

jseInitializeExternalLink

DESCRIPTION Routine to initialize a ScriptEase context.

SYNTAX
```
JseContext JseLib;
jseInitializeExternalLink(object ToolkitGetObject
     JseExternLinkParameters linkParms,
     String globalVarName,
     String accessKey);
```

PARAMETERS **ToolkitGetObject** - The object that is contructing the context should be the one that implements any of the Jse instances.

linkParms - this structure (jseExternalLinkParameters) contains the user defined properties of the ISDK. They are described in full below.

globalVarName - this parameter, a string, is the name you wish to give the global object.

accessKey - this is the key (supplied by Nombas) needed to activate your copy of ScriptEase:Integration SDK. Java does not require a key, but if you're using the JNI version, the key must be valid.

The jseExternalLinkParameters structure has this prototype:
```
String jsesecurecode;
int options;
```

jseSecureCode - Either a full file name and path or a block of JavaScript code that performs the security checking. Set this parameter to *null* if no security checking is needed.

Options - this is an or mask of the following flags. They define how the interpreter treats variables.

jseDefault - Use this flag to use the system defaults.

jseOptRequireVarKeyword - Use this flag if you want to force your users to use the 'var' keyword when creating variables.

jseOptRequireFunctionKeyword - Use this flag if you want to force your users to use the 'function' keyword when creating functions.

jseOptDefaultLocalVars - Use this flag if you want variables declared in a local environment to be local, regardless of whether the var keyword is used or not. (In JavaScript, variables declared without the var keyword would normally be global). If there is a like-named global variable, instead of creating a local variable the global variable would be used.

jseOptDefaultCBehavior - If this flag is defined, functions will be treated as if they were created with the 'cfunction' keyword, regardless of what keyword they were defined with.

jseOptWarnBadMath - If this flag is set, the interpreter will notify you when you make illegal mathematical calculations (such as dividing by zero). In JavaScript, dividing by zero normally returns the value *NaN* and does not generate an error.

jseOptLenientConversion - this option causes variables to automatically be converted to the required type if possible, instead of generating an error. With this option set the macro JSE_VN_CONVERT() will always behave as if the first parameter passed were JSE_VN_ALL. The jsePutxxx() functions will convert the variable to the required type. If you are retrieving data from a variable, if the variable is not of the correct type a copy of the variable will made, converted to the correct type, and returned.

jseOptIgnoreExtraParameters - If this option is set, the interpreter will ignore any parameters greater than the maximum allowed for the function (specified in the Function Descriptor table added to the context with jseAddLibrary().

RETURN returns a JseContext initialized with the values provided.

SEE ALSO jseGetExternalLinkParameters

jseInterpret

DESCRIPTION Interpret a ScriptEase script

SYNTAX

```
boolean JseContext.jseInterpret(
      String sourceFile,
      String sourceText,
      byte[] pretokenizedBuffer,
      int jseNewContextSettings,
      int howToInterpret,
      JseContext localVariableContext,
      JseVariable[]  returnVar);
```

PARAMETERS **sourceFile** - This argument is a string of the filename and path to a JavaScript file or *null* if you are interpreting JavaScript source from memory.

sourceText - This argument is either the text of the script to interpret, or, if interpreting code from a file, the optional arguments to pass to the script. If you do not need to use this parameter it should be set to *null*.

pretokenizedBuffer - If you are interpreting code that has been pretokenized with the jseCreateCodeTokenBuffer(), the code should go here. Otherwise, set this parameter to *null*.

jseNewContextSettings - These flags specify which elements of the jseContext about be created will be created new. Otherwise the elements will be inherited from the current jseContext. Use one or more of the following flags or'ed together:

jseNewNone - Do not create any new elements.

jseNewFunctions - Create new functions.

jseNewSecurity - Reinitialize security before interpreting the script.

jseAllNew - Create new elements for all categories (functions will be inherited from the parent JseContext).

howToInterpret - A flag to specify the method of interpretation. Use one or more of the following, joined by a bitwise or (|):

JSE_INTERPRET_NO_INHERIT - This flag prevents global variables from being passed to the new jseContext.

JSE_INTERPRET_CALL_MAIN - Call main() after running initialization code.

localVariableContext - This parameter is a JseContext or *null*. If you are calling jseInterpret from within a wrapper function, pass jseContext.jsePreviousContext; otherwise pass *null*.

returnCode - If the function executes successfully (i.e., returns *true)*, on return this will contain the value returned by the JavaScript

being executed. This variable must later be destroyed with jseDestroyVariable(). If you don't need to use this value, pass in *null*. The return value will be cleaned up automatically.

COMMENTS This call is the heart of the ScriptEase engine. After your ScriptEase toolkit environment is set up, call this routine to interpret scripts.

RETURN *true* if the script was successfully executed, *false* if not.

---

jseInterpExec

DESCRIPTION Interpret a ScriptEase script

SYNTAX

```
JseContext
JseContext.jseInterpretExec();
```

COMMENTS See jseInterpInit() for a description on using this function.

RETURN The context to pass to the next call to this function. *null* indicates the script is done executing.

jseInterpInit

DESCRIPTION Interpret a ScriptEase script

SYNTAX

```
boolean JseContext.
jseInterpret(String sourceFile,
      string sourceText,
      byte() pretokenizedBuffer,
      int jseNewContextSettings,
      int howToInterpret,
      JseContext localVariableContext,
      JseVariable[]  returnVar);
```

COMMENTS jseInterpInit(), jseInterpExec() and jseInterpTerm() provide an alternative to jseInterpret() for interpreting scripts. The two systems work in slightly different ways. jseInterpret() will call the MayIContinueFunc defined in the JseContext before each script line is executed.

With jseInterpInit() et al. you have more control over how the script executes. jseInterpInit() initializes the script for interpretation. It takes the same parameters as jseInterpret(). jseInterpInit() returns a new JseContext for the script, which is then passed to jseInterpExec(). The script is executed through repeated calls to jseInterpExec() taking this JseContext as its only parameter and returning an updated JseContext that must be passed again to jseInterpExec to execute successive lines. If there are no more lines to execute, jseInterpExec() returns *null*. The MayIContinueFunc will not be called.

When jseInterpExec() returns *null*, the script has completed, and you should call jseInterpTerm() to clean up the interpret. jseInterpTerm() takes one parameter, the original JseContext passed to jseInterpInit(), and not one of jseContexts returned from jseInterpInit() or jseInterpExec(). See jseInterpret for a description of parameters.

RETURN The context to use with jseInterpExec() the first time or null if some error prevented the interpreting from being initialized..

jseInterpTerm

DESCRIPTION

Terminate a ScriptEase script interpretation session.

SYNTAX

```
JseVariable JseContext.
jseInterpTerm();
```

COMMENTS See jseInterpInit() for a description of using this function.

RETURN The variable returned as the result of the script. You must destroy it when you are done with it. *null* is returned if there was an error interpretting the script.

SEE ALSO jseInterpInit, jseInterpExec.

jseIsFunction

DESCRIPTION Test whether a variable is a script or wrapper function registered with the supplied JseContext.

SYNTAX

```
boolean JseContext.jseIsFunction(
     JseVariable functionVariable );
```

PARAMETERS **functionVariable** - The variable being tested.

COMMENTS This function tests whether **functionVariable** is a registered function or not. If **functionVariable** was retrieved from a call to jseGetFunction(), this test is not necessary.

RETURN *true* if functionVariable is a registered function; *false* if it is not.

SEE ALSO jseCreateWrapperFunction, jseIsLibraryFunction

jseIsLibraryFunction

DESCRIPTION Test whether a variable is a wrapper function registered with the supplied JseContext.

SYNTAX
```
boolean JseContext.
jseIsLibraryFunction(JseVariable
                        functionVariable );
```

PARAMETERS **functionVariable** - The variable being tested.

COMMENTS This function tests whether **functionVariable** is a function added with jseAddLibrary or not.

RETURN Returns *true* if the function was added with jseAddLibrary(); otherwise returns *false*.

SEE ALSO jseCreateWrapperFunction, jseIsFunction

jseLibErrorPrintf

DESCRIPTION Prints a string describing the error encountered and flags the interpreter to quit execution.

SYNTAX
```
void JseContext.jseLibErrorPrintf(string text);
```

PARAMETERS **text** - the text of the error message.If an error condition has already been flagged, then this function performs no action.

COMMENTS The function sets the error flag for the JseContext and prints the string. The string lets you supply information about why the error occurred.

RETURN None.

SEE ALSO jseLibSetErrorFlag, jseLibSetExitFlag

jseLibSetErrorFlag

DESCRIPTION Mark the context as having encountered an error.

SYNTAX
```
void JseContext.jseLibSetErrorFlag();
```

COMMENTS Use this function sets the error flag to indicate that an error condition exists. The script will be terminated and any necessary cleanup performed when the current wrapper function is exited.

SEE ALSO jseLibErrorPrintf, jseLibSetExitFlag

jseLibSetExitFlag

DESCRIPTION Set the ScriptEase Lib exit flag.

SYNTAX
```
void JseContext.jseLibSetExitFlag(
     jseVariable exitVariable);
```

PARAMETERS **exitVariable** - The value to be returned by the script. This is the variable returned from jseInterpret.

COMMENTS Sets exit flag for this JseContext and saves exit variable. The script will clean-up and exit on return from this wrapper function.

SEE ALSO jseLibErrorPrintf, jseLibSetErrorFlag

jseLocateSource

DESCRIPTION Get the file information for the currently running script.

SYNTAX
```
String JseContext.jseLocateSource(
   int[] lineNumber );
```

PARAMETERS **lineNumber** -Holder for the current source file number.

COMMENTS Returns the name of the source file for the code currently being executed, and sets lineNumber[0] to the line number currently being executed or parsed. If there is no current file (as when interpreting a string) *null* will be returned.

RETURN A string containing the name of the source file for the currently executing code.

SEE ALSO jseBreakpointTest

jseMember

DESCRIPTION Get or create a JseVariable for a ScriptEase object property.

SYNTAX

```
JseVariable JseContext.jseMember(
      JseVariable objectVar,
      String name,
      int jseDataType)
```

PARAMETERS **objectVar** - The object to get a property from.

name - The name of the object property.

jseDataType - This argument specifies the type of object property variable that will be created if the variable does not already exist.

Note: this function has been deprecated in version 4.03. Internally it calls jseMemberEx() with the flags parameter set to jseDefault.

COMMENTS This routine gets a ScriptEase variable reference for an object's property. Once the JseVariable reference is obtained, use the data access functions to get the data. If the variable does not exist, it will be created when it is read from or written to.

RETURN A JseVariable pointer to the requested object property, or null on failure. If the property does not exist it will be created. Failure means the interpreter ran out of memory.

SEE ALSO jseMemberEx, jseGetMember, jseGetMemberEx, jseIndexMember, jseIndexMemberEx, jseGetNextMember, jseDeleteMember, jseGetIndexMember, jseGetIndexMemberEx

jseMemberEx

DESCRIPTION Get a ScriptEase variable reference to a ScriptEase structure element.

SYNTAX

```
JseVariable
   JseContext.jseMemberEx(JseVariable objectVar,
                          String Name,
                          int Dtype
                          int flags)
```

PARAMETERS **objectVar** - The object to get a property from.

name - The name of the object property.

DType - This argument specifies the type of object property variable that will be created.

flags - this should be set to one (some or all?) of the following (values OR'ed together):

jseCreateVar - If this flag is set, then the variable returned must be explicitly destroyed with jseDestroyVariable(). If this flag is not specified then the variable is tracked internally, and any variable returned from these functions is added to a list of variables to be destroyed when the current context is finished. This can cause problems with long-running persistent contexts because many temporary variables can be added without ever being deleted.

jseDontCreateMember - This only applies to the member functions. If the member does not exist and it is set, *null* is returned instead of creating the member. Therefore,

```
jseGetMember(jsecontext,var,name)
```

is the same as,

```
jseMemberEx(jsecontext,var,name,type,
            jseDontCreateMember).
```

jseDontSearchPrototype - This flag applies only to member functions. The default is to search for any prototype of the object not found in itself. This flag needs to be set to prevent prototype searching.

jseLockRead - This flag allows finer control over what the returned variable looks like. By default, a reference is returned. If this flag is set, the variable is retrieved once when the function is called and should be only used for reading from that point on. This flag and jseLockWrite are mutually exclusive.

jseLockWrite - Similar to jseLockRead, but the variable is locked for writing instead of reading.

COMMENTS This routine gets a ScriptEase variable reference for an object's property. Once the JseVariable reference is attained, use the data access functions to get the data. If the variable does not exist, it will be created.

RETURN A JseVariable for the requested object property, or *null* on failure. If the property does not exist it will be created. Failure means the interpreter ran out of memory.

SEE ALSO jseGetMember, jseGetNextMember, jseDeleteMember

jseMemberWrapperFunction

DESCRIPTION Attach a new object method to a wrapper function.

SYNTAX

```
JseVariable
JseContext.jseMemberWrapperFunction(
     JseVariable objectVar
     String functionName
     String);
 or
JseLibraryFunction function,
     int minVariableCount,
     int maxVariableCount,
     int varAttributes,
     int funcAttributes,
     Object libObj);
```

PARAMETERS **objectVar** - The object that the function is a method of. Use *null* to make it a global function

functionName - is the name of your function in a script. It should be a string such as "GetString". Your users will refer to the function by this name.

function - is the name of the Java method (or an instance of an inner class that implements JseLibraryFunction) corresponding to the function above.

COMMENTS This routine creates a function variable as an object method. It must eventually be destroyed with jseDestroyVariable().

RETURN If successful, this returns the JseVariable created. If there is not enough system memory to create the variable (extremely unlikely), *null* will be returned.

jsePreDefineNumber

DESCRIPTION Define a string alias for a ScriptEase number value.

SYNTAX

```
void
JseContext.jsePreDefineNumber(
     String findString,
     double replaceL );
```

PARAMETERS **findString** - String to match in source.

replaceL - Number to substitute for findString when parsing source.

You can use this function to override the #define statements in a script.

COMMENTS Use this routine to define a float for use by interpreted scripts. When parsing the ScriptEase source, any instance of findString (case sensitive) that might otherwise refer to a variable is replaced with the value for replaceL.

This use:

```
 jsecontext.jsePreDefineNumber("PI",3.1415927);
```

is similar to the script having this statement:

```
 #define PI 3.1415927
```

RETURN None.

SEE ALSO jsePreDefineLong, jsePreDefineString

jsePreDefineLong

DESCRIPTION Define a string alias for a long-integer value.

SYNTAX

```
void
JseContext.jsePreDefineLong(
   String FindString,   int ReplaceL );
```

PARAMETERS **FindString** - string to match in ScriptEase source.

ReplaceL - Integer to substitute for FindString when parsing ScriptEase source.

You can use this function to override the #define statements in a script.

COMMENTS Use this routine to define a long for use by interpreted scripts. When parsing the ScriptEase source, any instance of FindString (case sensitive) that might otherwise refer to a variable is replaced with the integer value for ReplaceL.

This use:

```
Jsecontext.jsePreDefineLong("MILLION",1000000);
```

is similar to the ScriptEase source having a statement such as:

```
#define MILLION 1000000
```

RETURN None.

SEE ALSO jsePreDefineNumber, jsePreDefineString

jsePreDefineString

DESCRIPTION Define a JavaScript string value.

SYNTAX

```
void
JseContext.jsePreDefineString(
   String findString,
   String replaceString );
```

PARAMETERS **FindString** - string to match in source.

ReplaceString - String to substitute for findString when parsing source. The replace string may be any sequence. You can use this function in your application to override the #define statements in any script it runs.

#define is used for text-replacement only, i.e. before the script is interpreted, all instances of findString are replaced with "replaceString," and the resulting text is interpreted as ScriptEase code.

COMMENTS Use this routine to define a string for use by interpreted scripts. When parsing the source, any instance of findString (case sensitive) that might otherwise refer to a variable is replaced with replaceString. This use:

```
jseContext.jsePreDefineString("VERSION_STR",
     "Version 1.2.4 Beta");
```

is similar to the source having a statement such as:

```
#define VERSION_STR "Version 1.2.4 Beta"
```

RETURN None.

SEE ALSO jsePreDefineNumber, jsePreDefineLong

jsePush

DESCRIPTION Push a JseVariable onto a JseStack.

SYNTAX

```
void
JseContext.JsePush(JseStack jsestack,
     JseVariable var,
     boolean destroyWhenFinished);
```

PARAMETERS **stack** - The stack to receive the variable.

var - The JseVariable to push onto the stack.

destroyWhenFinished - A boolean flag, specifying whether or not the JseVariable on the stack should be destroyed when the stack is destroyed. You only set this to *true* if you are responsible for destroying a variable and wish to get rid of this responsibility. For instance, if you used jseCreateVariable() to construct a variable to pass as a parameter. By telling this routine to destroy it when done, you no longer have to worry about destroying it yourself.

COMMENTS This function pushes a JseVariable onto the JseStack.

RETURN None.

SEE ALSO jseCreateStack, jseDestroyStack

jsePreviousContext

DESCRIPTION Retrieve the previous context.

SYNTAX

```
JseContext
Jsecontext.jsePreviousContext();
```

COMMENTS Given the current context, jsePreviousContext will find the previous one. The previous context will be the one that represents the script function that called the current function.

RETURN The previous ScriptEase context, or *null* if there wasn't one.

jsePutBoolean

DESCRIPTION Put boolean data into a JseVariable.

SYNTAX
```
void
JseContext.jsePutBoolean(JseVariable variable,
     boolean value);
```

PARAMETERS **variable** - The ScriptEase variable to write.

value - Value to set the variable to; use *true* or *false*

COMMENTS This function is used to write data to a jseTypeBoolean variable.

RETURN None.

SEE ALSO jseGetBoolean

jsePutBuffer

DESCRIPTION Put buffer data into a JseVariable.

SYNTAX void

JseContext.jsePutBuffer(JseVariable variable, byte[] data);

PARAMETERS **variable** - The ScriptEase variable to write data to.

data - Pointer to buffer data.

COMMENTS This function writes data to a jseTypeBuffer variable.

RETURN None.

SEE ALSO jseGetBuffer, jseGetWritableBuffer

jsePutByte

DESCRIPTION Write data to a variable as a byte.

SYNTAX
```
void
JseContext.jsePutByte(JseVariable variable,
     byte byteValue);
```

PARAMETERS **variable** - The ScriptEase variable to write.

byteValue - Value to which the variable is to be set.

COMMENTS This function is used to write data to a variable of jseTypeNumber.

RETURN None.

SEE ALSO jseGetNumber, jsePutNumber, jsePutLong

jsePutNumber

DESCRIPTION Write numeric data to a JseVariable.

SYNTAX
```
void
JseContext.jsePutNumber(JseVariable variable,
                        double number);
```

PARAMETERS **variable** - The ScriptEase variable to write to.

number - Value to which the variable is to be set.

COMMENTS This function is used to write data to a jseTypeNumber variable.

RETURN None.

SEE ALSO jseGetNumber, jsePutLong, jsePutByte

jsePutLong

DESCRIPTION Write integer data to a JseVariable.

SYNTAX
```
void
JseContext.jsePutLong(jseVariable variable,
     int longvalue);
```

PARAMETERS **variable** - The ScriptEase variable to write.

longvalue - Value to which the variable is set.

COMMENTS This function is used to write data to a jseTypeNumber variable.

RETURN None.

SEE ALSO jseGetLong

jsePutString

DESCRIPTION Write string to a JseVariable.

SYNTAX

```
void
JseContext.jsePutString(JseVariable variable,
      String data);
```

PARAMETERS **variable** - The ScriptEase variable to write.

data - Value to set the variable to.

COMMENTS This function writes string data to a jseTypeString variable. The length of the string will be assumed to be the extra string's length. If you wish to explicitly pass a string length, use jsePutStringLength().

RETURN None.

SEE ALSO jsePutStringLength, jseGetString, jseGetWritableString

jsePutStringLength

DESCRIPTION Write string to a ScriptEase variable.

SYNTAX

```
void
JseContext.jsePutStringLength(
      JseVariable variable,
      String data,
      size);
```

PARAMETERS variable - The ScriptEase variable to write.

data - Value to set the variable to.

size - The length of the string in data.

COMMENTS This function writes string data to a jseTypeString variable.

RETURN None.

SEE ALSO jsePutString, jseGetString, jseGetWritableString

jseQuitFlagged

DESCRIPTION Check if current context has been flagged to terminate execution.

SYNTAX

```
int
JseContext.jseQuitFlagged();
```

COMMENTS Returns 0 if a call has not been made on this context to Exit (jseLibSetExitFlag()), or to report an error via any of the error reporting functions (jseLibSetErrorFlag() or jseLibErrorPrintf()). It is not necessary to call these functions after the jseXXX library functions, which include error status (if applicable) in their return codes.

This function can be valuable during debugging (e.g., in assert() statements) to ensure that the JseContext is valid. If you add functions that may set the error or exit flags and that don't indicate this information in their return codes, or if you are not checking return codes in some sections, then jseQuitFlagged() may be used.

Another use for this function is the case where your context may be handled in a callback, so you can save the context in a global and check later if there was a problem.

If your script should exit due to an exit flag or due to an error, then this function will return one of the following non-0 (non-*false*) values:

```
JSE_CONTEXT_ERROR       // ERROR flag set
JSE_CONTEXT_EXIT        // EXIT flag set
```

RETURN (0) if this context is not flagged for exit due a call to jseLibSetExitFlag() or to an error call, else return reason for exit, indicated by one of the values described above.

SEE ALSO jseLibSetErrorFlag, jseLibSetExitFlag, jseLibErrorPrintf

jseReturnNumber

DESCRIPTION Return a number from a ScriptEase wrapper function.

SYNTAX

```
void
JseContext.jseReturnNumber(
   double number);
```

PARAMETERS **number** - The numeric value to return.

COMMENTS This function is used to return a numeric value from a ScriptEase wrapper function. If you call any of the jseReturnXXX() functions again, the last call takes precedence. It creates a variable of type jseTypeNumber, assigns the number to it, and makes that the return from the wrapper function. It is not like 'exit()' in that your wrapper function continues executing. Typically, a call to this function is the last thing your wrapper function does before returning.

RETURN None.

SEE ALSO jseReturnLong, jseReturnVar

jseReturnLong

DESCRIPTION Return an integer from a ScriptEase wrapper function.

SYNTAX

```
void
JseContext.jseReturnLong(
   int longValue);
```

PARAMETERS **longValue** - The value to return.

COMMENTS This function is used to return a long value from a ScriptEase wrapper function. If you call any of the jseReturnXXX() functions again, the last call takes precedence. It creates a variable of type jseTypeNumber, assigns the longValue to it, and makes that the return from the wrapper function. It is not like 'exit()' in that your wrapper function continues executing. Typically, a call to this function is the last thing your wrapper function does before returning.

RETURN None.

SEE ALSO jseGetLong

jseReturnVar

DESCRIPTION Returns a jseVariable from a wrapper function.

SYNTAX

```
Void
JseContext.jseReturnVar(JseVariable variable,
     int jseReturnAction);
```

PARAMETERS **variable** - The variable to be returned from this function.

retAction - Specifies how the variable to be returned shall be treated once you are done using it. The return action can be one of the following values:

jseRetTempVar - This is variable you own and are expected to delete. By passing it along using this flag, you no longer have to delete it. You have passed ownership to the system and it will delete it when it is finished with it.

jseRetCopyToTempVar - Create a new variable, copy to that variable (with jseAssign()), and then return that new variable to be destroyed when it is popped. Don't return this variable; return the copy. If you own this variable and are expected to delete it, you still must do so.

jseRetKeepLVar - This is similar to jseRetCopyToTempVar in that you still own the variable and must delete if appropriate. It differs in that no copy is made. If you change the variable (such as with jseConvert()), the change will be reflected in the value returned from the function.

COMMENTS This function is used to generate a return value from a ScriptEase wrapper function. It will return the specified ScriptEase variable. If you call any of the jseReturnXXX() functions more than once, the last call takes precedence.

RETURN None.

SEE ALSO jseReturnNumber, jseReturnLong

jseSetAttributes

DESCRIPTION Set the attributes of a JseVariable.

SYNTAX

```
void
JseContext.jseSetAttributes(
      JseVariable variable,
      int jseAttributes);
```

PARAMETERS **variable** - The variable to have its attributes updated.

attr - The attributes to be applied to variable.

The return action can be any of the following values OR'ed together:

jseDefaultAttr - Standard ECMAScript behavior.

jseDontEnum - Ignore thise member during for...in enumerations

jseDontDelete - Cannot be deleted by the delete operator

jseReadOnly - Makes the variable read only.

jseImplicitThis - Puts the 'this' variable in the scoping chain. This only applies to calling this member if it is in fact a function.

RETURN jseGetAttribute

jseSetArrayLength

DESCRIPTION Set the length of a string, buffer or numerically-indexed object.

SYNTAX

```
void
JseContext.jseSetArrayLength(
     JseVariable variable,
     int MinIndex,
     int length);
```

PARAMETERS **variable** - the ScriptEase variable for which the length will be set.

MinIndex - the index value to use for the first element of the array. Must be less than or equal to zero.

length - length of the string or buffer, or one greater than the maximum numerically indexed property or an object. Must be greater than or equal to zero.

COMMENTS This routine sets the length of a ScriptEase string, buffer, or numerically-indexed object. This function will create new array entries if they are needed, and destroy those that are no longer needed, i.e., that are outside of the bounds of the new array.

RETURN None.

SEE ALSO jseGetArrayLength

jseSetJavaObject

DESCRIPTION Set the Java 'Object' variable associated with a jseVariable.

SYNTAX

```
void
JseContext.jseGetJavaObject(JseVariable var,Object
javaObj);
```

COMMENTS This often useful to be able to associate an arbitrary Java item with a JseVariable and later retrieve it. This is analogous to storing a C pointer by casting it to an int. This function associates an Object with a JseVariable. If an object was already associated, it is replaced. Remember, any Java item object of any type can be cast to an (Object) and stored via this function. You can only store a Java Object in a JseVariable that itself of type jseTypeObject.

RETURN None.

SEE ALSO jseGetJavaObject.

jseTellSecurity

DESCRIPTION Call the security routine defined for the jseContext.

SYNTAX
```
boolean
JseContext.jseTellSecurity(JseVariable infoVar);
```

PARAMETERS **infoVar** - The variable to be passed to the security filter. Your application and its security filter may use it however you choose.

COMMENTS This function will call the security manager's initialization routine (i.e. the jseSecurityInit() function); it 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 the script.

Typically, you will use this when executing a particularly insecure piece of code (such as a script received over the network) to downgrade the security level, restoring it when the script completes. The only parameter is any JseVariable.

RETURN returns *true* if there is a security filter, and *false* if there is not.

jseTerminateEngine

DESCRIPTION A call to terminate the ScriptEase Interpreter Engine.

SYNTAX
```
void
jseLib.jseTerminateEngine();
```

COMMENTS Call this function after all JseContext links have been terminated. This function cleans up all the resources allocated and initialized by jseInitializeEngine().

RETURN None.

SEE ALSO jseInitializeEngine

jseTerminateExternalLink

DESCRIPTION Terminate a link to a given jseContext.

SYNTAX
```
void
JseContext.jseTerminateExternalLink();
```

COMMENTS This routine is used to terminate the given JseContext. After this call, any references to the supplied context are invalid and will cause an error to occur.

RETURN None.

SEE ALSO jseInitializeExternalLink, jseGetExternalLinkparameters

jseVarNeed

DESCRIPTION Check the type of a given ScriptEase argument variable.

SYNTAX
```
boolean
      JseContext.jseVarNeed(
      JseVariable variable,
      int jseVarNeeded);
```

PARAMETERS **variable** - The variable being checked for type.

need - The type of the argument you are trying to verify. It can be one of the values specified in jseFuncVarNeed.

COMMENTS This function verifies that a function argument to a ScriptEase wrapper function is of a given type.

RETURN *True* if the variable specified is of the type specified or can be convented according to the flags described in jseFuncVarNeed. *null* otherwise and a error message will have been called.

SEE ALSO jseGetVar, jseFuncVar, jseFuncVarNeed

ScriptEase JavaScript Language

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[1] 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")
```

[1] "Code fragment" and "fragment" are used interchangeably. They both refer to lines of script or code that perform some scripting or programming task. The lines of code may or may not be complete scripts or programs.

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. 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

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.

Whitespace characters

Whitespace characters, space, tab, carriage-return and new-line, govern the spacing and placement of text. Whitespace makes code more readable for humans, but is ignored by the interpreter2.

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 whitespace characters between identifiers as simply whitespace, 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
```

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

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

Comments

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

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

---

2 The phrase, "the interpreter," is used synonymously with, "the ScriptEase interpreter." ScriptEase, like JavaScript and many other popular languages, is an interpreted language.

The following code fragments are examples of valid comments:

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

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

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

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

Expressions, statements, and blocks

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

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

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

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

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

Statements within blocks are often indented for easier reading.

Identifiers

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

Identifiers may use only ASCII letters, upper or lower case, digits, the underscore, "_", and the dollar sign, "$". That is, they may use only characters from the following sets of characters.

```
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
"abcdefghijklmnopqrstuvwxyz"
"0123456789"
"_$"
```

Identifiers may **not** use letters of the following characters.

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

Identifiers must begin with a letter, underscore, or dollar sign, but may have digits anywhere else.

Identifiers may not have whitespace in them since whitespace separates identifiers for the interpreter.

Identifiers may be as long a programmer needs.

The following identifiers, variables and functions, are valid:

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

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

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

Prohibited identifiers

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

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

Variables

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

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

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

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

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

Variable scope

Variables in ScriptEase may be either global or local. Global variables may be accessed and modified from anywhere in a script. Local variables may only be accessed from the functions in which they are created. There are no absolute rules for preferring or using global or local variables. Each type has merit. 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 DefaultLocalVariables 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".

Functions

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 that 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

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

Decimal

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

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

Scientific

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

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

Boolean

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

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.

Two commonly used composite data types are: Object and Array.

Object

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

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 (typeof test == "undefined")
        Screen.writeln("test is undefined")
```

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

NULL

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

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

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

```
var test = null;
```

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

NaN

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

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

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

Number constants

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

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

Automatic type conversion

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

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

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

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

Properties and methods of basic data types

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

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

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

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

The following two methods are common to all variables.

.toString()

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

.valueOf()

This method returns the value of a variable.

Operators

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) (the 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 multiplication and division is calculated for an expression before addition and subtraction 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  3.333, (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      i >>>= 2
        with zeros
&       bitwise and                i = i & 1
&=      assignment bitwise and     i &= 1;
|       bitwise or                 i = i | 1
|=      assignment bitwise or      i |= 1;
^       bitwise xor, exclusive     i = i ^ 1
        or
^=      assignment bitwise xor,    i ^= 1
        exclusive or
~       Bitwise not, complement    i = ~i;
```

Logical operators and conditional expressions

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

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 following, 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, otherwise *false*. Do not confuse the equality operator, ==, with the assignment operator, =.
`!=`	inequality	*true* if the values are not equal, else *false*.
`<`	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.

typeof operator

The typeof operator provides a way to determine and to test the data type of a variable and may use either of the following notations, with or without parentheses.

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

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

Flow decisions statements

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

if

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

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

else

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

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

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

The following fragment illustrates using *else* with *if*.

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

while

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

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

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

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

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

do {...} while

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

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

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

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

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

for

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

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

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

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

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

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

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

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

break

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

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

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

continue

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

switch, case, and default

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

```
switch( switch_variable )
{
case value1:
       statement1
       break;
case value2:
       statement2
       break;
       .
       .
       .
default:
       default_statement
}
```

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

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

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

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

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 else 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 *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));
```

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

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

ScriptEase allows you to have two functions with the same name. The interpreter uses the function nearest the end of the script, that is, the last function to load is the one 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 variables 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.

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.

Function properties -- 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.

The 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".

The cfunction keyword

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

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

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

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

Array arithmetic

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

For example, the statement:

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

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

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

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

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

Arrays

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

Array elements can be of any data type. The elements in an array do not 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[fool = “creeping things”
array[goo + 1] = “etc.”
```

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

Since arrays use a number to identify the data they contain, they provide an easy way to work with sequential data. For example, suppose you wanted to keep track of how many jelly beans you ate each day, so you can graph your jelly bean consumption at the end of the month.

Arrays provide an ideal solution for storing such data.

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

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

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

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

Creating arrays

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

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

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

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

In this case, an array with length 31 is created.

Finally, you can pass a number 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 *c*[0], not *c*[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. Arrays created in this manner are unable to use the methods and properties described below, so it is recommended that you use the Array() constructor function to create arrays.

Methods and properties of arrays

When an array is created with the Array() constructor function, a number of methods and properties become available to it.

Properties of arrays

.length

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.

For example, suppose we had two arrays "ant" and "bee", with the following elements:

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

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

By changing the value of the length property, you can remove array elements. For example, if you change ant.length to 2, ant 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.

Methods of arrays

.join()

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

.sort([compareFunction])

The .sort() method sorts members of an array and puts them in alphabetic order. If no compare function is supplied, then elements are converted to strings to do the conversion, which may cause some confusion. For example, the following code:

```
var a = new Array(32, 5, 6, 3)
a.sort();
var string = a.join();
```

creates a string "3, 32, 5, 6".

This behavior is often not what you want in a sort function. Fortunately, the .sort() method allows you to specify a different way to sort the array elements. The name of the function you want use to compare values is passed as the only parameter to sort().

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

- If compareFunction(a, b) is less than zero, sort b to a lower index than a.
- If compareFunction(a, b) returns zero, leave a and b unchanged to each other.

C 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
}
```

.reverse()

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

The following code:

```
ar 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"
```

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

```
var Rectangle;

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

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

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

Predefining objects with constructor functions

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

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

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

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

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

Constructor functions create objects belonging to the same class. Every object created by a constructor function is called an instance of that class. The examples above create 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.

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 = new Rectangle(3,4);
var sally = new 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.

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.

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

This fragment creates a value, which in this case is a function, for joe.area that 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 (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 DONT_ENUM attribute are not accessible to a *for . . . in* statement.

with

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

The following fragment illustrates using the Clib object.

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

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

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

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

Dynamic objects

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

._get(property, ExpectCall)

Whenever the value of a property is accessed, the ._get() method is called. By defining a new ._get() method for an object, you modify the way it accesses property values.

The 4.20 _get function now receives a second parameter. This parameter is called "ExpectCall" and is *true* if the parameter is being retrieved to make a function call, and *false* for other situations.
For example, in this case:

```
obj.foo;
```

The second parameter will be false. But in this case

```
obj.foo();
```

the second parameter will be true.

The example following modifies the Rectangle object created earlier with a new ._get() method. Whenever you access the value of one of the object's properties, it will inform you if the Rectangle is a square. After the object is initialized, the main() function creates an instance of the object with the *width* and *height* properties both set to 3. When the value of the Rectangle.area() method is retrieved, used in a Clib.printf() statement, the dynamic ._get() function is called, which displays, "The rectangle is a square," since *width* and *height* are equal.

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

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

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

Rectangle.prototype.area = rectangle_area;

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

._put(property, value)

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

._canPut(property)

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

._hasProperty(property)

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

._delete(property)

This method is called whenever a property is deleted with the delete operator. The property will be "_delete" when the object itself is being deleted.

._defaultValue(hint)

This method returns the primitive value of a variable.

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

._construct(. . .)

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

._call(. . .)

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

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

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

var Annoying = new Object;

Annoying.get = myget;
```

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

._operator(op,operand)

Operator Overloading

ScriptEase allows you to overload the standard arithmetic operators when used with your objects. Consider this example:

```
    var a = obj + 10;
```

If 'obj' is one of your own objects, you may have some special meaning you'd like the addition operator to have when applied to it. Operator overloading allows this to be done.

Whenever an object is the first operand to an arithmetic operation, it has the opportunity to redefine what that operation means.
All of the arithmetic operators can be overloaded, such as +, -, /, >>, and so forth.
In addition, the unary operators (i.e. those that have only one operand, the object) can also be overloaded. These are the operators ~, !, ++, --, + , and -.
Finally, the assignment operator (=) can also be overloaded.

Please note that the compound assignment operators (i.e., *=, +=, etc) are treated exactly like you wrote out the statement. In other words,

```
a += b;
```

is treated just like:

```
a = a + b;
```

Overloading the operators will work that way. In this case, if 'a' is an object with overloaded operators, that statement will involve two operators, a '+' and an '='.

To overload operators on a particular object, you simply give the object the method '_operator'. This works like all of the other dynamic object methods. For instance, you can put the '_operator' method in a prototype so that all objects of that class inherit the operator overloading. Here is an example:

```
function overload(op,operand)
{
   Clib.printf("overloading occurring on operator
               '%s'\n",op);
   return DYN_DEFAULT;
}

var myObject = new Object();
myObject._operator = overload;

myObject = 10;
```

The operator overloading function is passed two parameters. The first parameter is the operator itself, in the form of a string. It will be "+" or "-" or "++", etc. The second parameter is the second operand to the operator.
If the object operation being overloaded is 'obj + 4', for instance, then the first parameter is "+" and the second parameter is the number 4. The unary operators (such as '-obj') do not have a second operand, so the second parameter is undefined. You can use this to distinguish the operators + and - which can be used either way, i.e. the difference between 'obj + 4' and '+obj'.

Whatever value the operator function returns is taken to be the value of the expression. If the operator function returns DYN_DEFAULT or OPERATOR_DEFAULT_BEHAVIOR, then the normal operation is done.
In many cases, you will not want to override all of the operators that could be applied to your object, so you will return this value if the operator is not one you are interested in. In

the example above, we print out a message when the object is used in an operation, but we don't change what the operation does. We always return DYN_DEFAULT and thus do the normal ECMAScript operation.

The global object and its properties

Global variables are members of the global object. To access global properties, you do not need to use an object name. For example, to access the isNaN() method, which tests to see whether a value is equal to the special value NaN you can call either of the following.

```
isNaN(value);
```

or

```
global.isNaN(value);
```

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.

Properties of the global object

._argc

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.

._argv

This property is an array of strings. Each string is a parameter passed to the script's 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.

Methods of the global object

.eval(expression)

This method 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.

.parseInt(string [, radix])

This method converts an alphanumeric string to an integer number. The first parameter, string, is the string to be converted, and the second parameter, radix, is an optional number indicating which base to use for the number. If the radix parameter is not supplied, the method defaults to base 10 which is decimal. If the first digit of string is a zero, radix defaults to base 8 which is octal. If the first digit is zero followed by an "x", that is, "0x", radix defaults to base 16 which is hexadecimal.

Whitespace characters at the beginning of the string are ignored. The first non-whitespace 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 will be returned.

.parseFloat(string)

This method is similar to parseInt() except that it reads decimal numbers with fractional parts. In other words, the first period, ".", in the parameter string is considered to be a decimal point, and any following digits are the fractional part of the number. The method .parseFloat() does not take a second parameter.

.escape(string)

The .escape() method receives a string and escapes the special characters so that the string may be used with a URL. All uppercase and lowercase letters, numbers, and the special symbols, @ * + - . /, remain in the string. All other characters are replaced by their respective Unicode sequence.

.unescape(string)

This method is the reverse of the .escape() method and removes escape sequences from a string and replaces them with the relevant characters.

.isNaN(number)

This method returns *true* if the parameter, *number*, evaluates to NaN, Not a Number. Otherwise it returns *false*.

.isFinite(number)

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*.

Exception Handling via Scripts

First for script code, exceptions are trapped with try:

```
try
{
    do something;
}
catch( e )
{
    Clib.printf("Something bad happened:
%s\n",e.toString();
}
```

A catch clause 'eats' the error, so the rest of the script continues. If you 'throw' something, that something is passed up the chain as an error. You can throw the error object you caught in a catch statement to make the error be 'unhandled'.

For instance:

```
try
{
    do something;
}
catch( e )
{
    Clib.printf("Something bad happened:
%s\n",e.toString();
    throw e;
}
```

In this case, if there is an error, it will be printed out, but then the program will still stop with that error.

You can raise arbitrary errors as you like in a program, i.e.:

```
throw new TypeError("You are not my type!");
```

A try block can also have a finally clause, e.g.:

```
try
{
    do something;
}
finally
{
    Clib.printf("Always happens.\n");
}
```

The finally clause ALWAYS is executed right before the block is left, even if left by a goto, return, error, or whatever. If the finally block does a control transfer (i.e. it does a

goto, throw, or return), that takes precedence, else whatever transfer was pending actually does happen.

So if you do:

```
try
{
    return 10;
}
finally
{
    Clib.printf("BYE!\n");
}
```

This will print BYE! then return 10 from the function. If you do:

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

no_way:  ...
```

In this case, the goto takes precedence over the return, so the return is ignored and execution continue with the '...' code.

Preprocessing

This section describes directives that affect the processing of a ScriptEase script prior to finally compiling, tokenizing, and executing the script.

Preprocessor Directives

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

#define

The #define directive is used to replace a token or almost any identifier with other characters. The #define directive is executed while the script is being read into the interpreter, before the script itself is executed. The #define directive causes one string to

be replaced by another in the script that goes to the interpreter. All substitutions are made before the code is interpreted. A #define directive has the following structure.

```
#define token replacement
```

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

```
#define NumberOfCountriesInSouthAmerica 13
```

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

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

```
#define ROW_COUNT 25
```

to

```
#define ROW_COUNT 50
```

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

#include

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

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

Any one of these lines make all of the functions in the library file gdi.jsh available to the script that has the line. The quote characters, ' or ", may be used in place of the angled brackets < and >.

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

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

The ScriptEase interpreter will not include a file more than once, so if a file has already been included, a second or subsequent #include directive has no effect. ScriptEase ships with a large number of libraries of pre-written functions that you can use. Library files are plain text files, as are all ScriptEase scripts, and have the extension .jsh as a default.

Since these libraries are external to ScriptEase, they are less static than the standard function libraries, and can be easily expanded or modified as the need arises. The most recent versions of .jsh libraries are listed on the Nombas downloads page at the following

web site:

www.nombas.com

#if, #ifdef, #elif, #else, #endif

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

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

```
var fullPathOfFile = Clib.rsprintf("%s\\%s\\%s\\%s",
```

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

```
                    rootdirectory, subdirectory1,
                        subdirectory2, filename);
```

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

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

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

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

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

The *#ifdef* directive is another limited form of #if that is equivalent to "*#if!defined(var)*".

#link

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

```
#link "sesock"
```

lets the interpreter use the functions for TCI/IP socket communication. #link takes no parameters other than the name of the library being linked.

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

Nombas supplies many #link libraries, such as:

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

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

www.nombas.com

Integrating the ScriptEase Debugger

> ***Please Note:***
>
> *The current version of the ScriptEase:ISDK/Java does not have the debugger implemented. Please check our web site, Nombas.com, for updated information for when this feature is implemented and your ISDK software can be upgraded. This chapter is included in the manual for when that upgrade is available.*

Using the ScriptEase:ISDK, you can debug your applications using Nombas's debugger. The debugger itself is a Windows application, so you'll need a windows machine to do your debugging on. However, your application can be running on any machine that can communicate with your debugging machine. Nombas provides support for debugging via TCP-IP, but you can extend the debugger to use other communication protocols.

For end-user information on using the debugger, please see the chapter, "Using the ScriptEase Debugger."

Using a Nombas protocol model

Nombas has provided two models of debugging to cover many situations. First, on windows systems, you can communicate via shared memory. In this case, the debugger and the application must both be running on the same Windows machine. Either the application can start the debugger, or the debugger can start the application (depending on how you set it up.)

If you are debugging using the TCP-IP model, you need to run the ScriptEase IDE Network Extender (called the proxy) on the debugging machine before running your application. The application will communicate with the proxy in place of the debugger. The proxy will make sure the debugger starts up and receives the information it needs to debug your application.

Defining your own protocol model

To define a new protocol model for communication between the debugger and your application requires you to provide a number of routines linked with your application. These are documented at the top of the file 'srcdbg\debugme.h'. You can examine the file 'srcdbg\debugme.c' to see how these routines are implemented in the Nombas-provided models. If you are defining your own model, you will need to also add that model to the proxy.

Code changes to your application

You must do six things to make sure your application is debuggable. They are described in order:

Set Options

```
#define JSE_DEBUGGABLE 1
```

If you are using TCP-IP, set these flags:

```
#define JSE_DEBUG_TCPIP
#define JSE_DEBUG_MASTER
#define JSE_DEBUG_RUN
#define JSE_DEBUG_FILES
#define JSE_DEBUG_REMOTE
#define JSE_DEBUG_PASSWORD
```

(Note: `JSE_DEBUG_PASSWORD` only activates the password code. You must still actually setup a password if you are going to use it.)

Otherwise, if using shared memory set these flags:

```
#define JSE_DEBUG_MEMORY
#define JSE_DEBUG_RUN
```

This setup for shared memory assumes you want the debugger to start the application. If you'd like it to be the other way around (i.e. the application starts the debugger), add:

```
#define JSE_DEBUG_MASTER
```

Add files to your project

Next add the file 'srcdbg\debugme.c' to your application. Make sure the 'srcdbg' directory is in your include path if it isn't already.

Update your ToolkitAppData structure and jseopt.h

If you don't currently allocate one, you must do so. You can get a definition for one by include 'seclib\seseclib.h'. Alternately, if you already are using your own such structure, add the following to it:

```
#if defined(JSE_DEBUGGABLE)
        struct debugMe * debugme;
#endif
```

You need some includes added at the end of your jseopt.h file:

```
#if defined(JSE_DEBUGGABLE)
          #if defined(__JSE_OS2TEXT__) ||
              defined(__JSE_OS2PM__)
              #include <sys\socket.h>
              #include <netinet\in.h>
              #include <netdb.h>
              #include <utils.h>
              #include <nerrno.h>
              #include <sys\ioctl.h>
          #endif
          #include "dbgshare.h"
          #include "proxy.h"
          #include "debugme.h"
#endif
```

Initialize debugging

After you have initialized your external link and added any libraries, but before you start interpretting you must initialize the connection to the debugger. This is done with the following code:

```
#if defined(JSE_DEBUGGABLE)
          debugmeInit(jsecontext,<command line>,<instance>);
#endif
```

The 'command line' is only needed for shared memory debugging if the debugger is going to be starting up your application. It should be the entire command line, which is easily constructed by concatenating the entries of the argv[] array separated by spaces. The routine will extract the debugging command information from the command line. When it returns, you must reparse the command line into individual arguments (which is easily accomplished using strtok().) For the TCP-IP model, the command line parameter is ignored, so you can safely pass NULL.

The 'instance' parameter is the Windows HINSTANCE value for your program. It is only needed if debugging on a Windows platform using a windowed application (as opposed to a console application.) On other platforms, debugmeInit() does not take a third parameter.

Finally, for the TCP-IP version, you must specify what machine will be the debugging

machine. You do this by setting the environment variable 'REMOTE_ADDR' to the machine host name of the debugging machine. You can set this either before launching your program or within your program before calling debugmeInit(). The machine in question needs to have the proxy running as described above.

Call the debugger hook

Finally, you must call the debugger in your MayIContinue function. Here is an example. If you already do some code in your function, do this in addition.

```
            jsebool JSE_CFUNC FAR_CALL
  ContinueFunction(jseContext jsecontext)
  {
            struct ToolkitAppData * SeData =
                ToolkitAppDataFromContext(jsecontext);

            #if defined(JSE_DEBUGGABLE)
                if ( NULL != SeData->debugme )
                {
                    debugmeDebug(SeData->debugme,jsecontext);
                    if ( jseQuitFlagged(jsecontext) )
                        return False;
                }
            #endif

            jsecontext = jsecontext;      /* to prevent warning
                about unused
                                          /* variable */
            return True;
  }
```

Terminate debugging

This code shows you how to terminate debugging. It assumes 'AppData' is a pointer to your application data structure.

```
#   if defined(JSE_DEBUGGABLE)
            {
                struct debugMe *debugme = AppData->debugme;

                if ( NULL != debugme )
                {
                    debugmeHasTerminated(debugme);

                    while ( debugme )
                    {
                         debugmeDebug(debugme,jsecontext);
                         debugme = AppData->debugme;
                       }
                }
                debugmeTerm(jsecontext);
            }
#   endif
```

You must terminate debugging before you destroy the context. You usually terminate debugging right before you exit. This means all scripts you interpret will be debugged in a single session. However, you can terminate then restart debugging if you want each jseInterpret() to be in its own debug session.

Notes

Once you have made these changes, your application can be debugged. You can make all of these changes and still not debug your application if you skip Initialization of debugging. So, if you want, you can only initialize the connection to the debugger if your user selects a special 'debug application' menu item or such.

You can currently only debug scripts that have a filename (i.e. if you tell jseInterpret() to interpret the contents of a file.)

Samples

In seisdk\samples\debug, you can find a modified version of 'simple0' that is debuggable. Run the application from an MS-DOS prompt after first setting 'REMOTE_ADDR' as described above.

Example: Modifying your JSEOPT.H file for debugging

Any application that uses the debugger must have the following lines in its JSEOPT.H file:

```
#define JSE_DEBUGGABLE 1
#define JSE_DEBUG_RUN
```

Set these flags if you will be using the debugger remotely:

```
#define JSE_DEBUG_TCPIP
#define JSE_DEBUG_MASTER
#define JSE_DEBUG_FILES
#define JSE_DEBUG_REMOTE
#define JSE_DEBUG_PASSWORD
```

Set this flag if you will be using the debugger locally:

```
#define JSE_DEBUG_MEMORY
```

With the options described above, the debugger will launch the application in order to debug it. For the reverse, in which the application launches the debugger, define the following:

```
#define JSE_DEBUG_MASTER
```

Language Objects & Libraries

ScriptEase Global Functions

The global functions described in this section are unique to the ScriptEase implementation of JavaScript. In other words, they are not part of the ECMAScript standard, but they are useful. Avoid using these functions in a script if it will be used with a JavaScript interpreter that does not support these unique functions.
Like other global items these functions are actually methods of the global Object and can be called with function or method notation. The two following lines of code are equivalent.

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

General

defined(value)

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().

```
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 a value 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"

getArrayLength(array[, MinIndex])

This function should be used with dynamically created arrays, that is, with arrays that were **not** created using the Array() constructor and new operator. When working with arrays created using the Array() constructor and new operator, use the .length property of

the arrays. The .length property is not available for dynamically created arrays which must use the functions, getArrayLength() and setArrayLength(), when working with array lengths.
The getArrayLength() function returns the length of a dynamic array, which is one more than the highest index of an array, if the first element of the array is at index 0, which is most common. If the parameter MinIndex is passed, then it is used to set to the minimum index, which will be zero or less. You can use this function to get the length of an array that was not created with the Array() constructor function.
This function and its counterpart, setArrayLength(), are intended for use with dynamically created arrays, that is, arrays not created with the Array() constructor function. Use the .length property to get the length of arrays created with the constructor function and not getArrayLength().

getAttributes(variable)

This function gets and returns the variable attributes for the parameter variable. Variable attributes may be set using the function setAttributes(). See setAttributes() for more information and descriptions of the attributes of variables that can be set.

setArrayLength(array[, MinIndex], length])

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.

setAttributes(variable, attributes)

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.

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.

undefine(value)

This function undefines a variable, Object property, or value. If a value was previously defined so that its use with the function defined() returns true, then after using undefine() with the value, defined() returns false. Undefining a value is different than setting a value to null.

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);
```

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 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 the functions.

ToBoolean(value)

The following table lists how different data types are converted by this function.

Data type	Return
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

ToBuffer(value)

This function converts value to a buffer in a manner similar to ToString() except that the resulting array of characters is a sequence of ASCII bytes and not a unicode string.

ToBytes(value)

This function converts value to a buffer and differs from ToBuffer() in that the conversion is actually a raw transfer of data to a buffer. The raw transfer does not convert unicode values to corresponding ASCII values. For example, the unicode string "Hit" may be stored in a buffer as "\0H\0\i\0t", that is, as the hexadecimal sequence: 00 48 00 69 00 74.

ToInt32(value)

This function is the same as ToInteger() except that if the return is an integer, it is in the range of -2^{31} through $2^{31} - 1$.

ToInteger(value)

This function converts value to an integer type. First, call ToNumber(). If result is NaN, return +0. If result is +0, -0, +Infinity or -Infinity, return result. Else return floor(abs(result)) with the appropriate sign. For example, the value -4.8 is converted to -4.

ToNumber(value)

The following table lists how different data types are converted by this function.

Data type	Return
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

ToObject(value)

The following table lists how different data types are converted by this function.

Data type	Return
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

ToPrimitive(value)

This function does conversions only for parameters of type Object. An internal default

value of the Object is returned.

ToString(value)

The following table lists how different data types are converted by this function.

Data type	Return
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"

ToUint16(value)

This function is the same as ToInteger() except that if the return is an integer, it is in the range of 0 through 2^{16} - 1.

ToUint32(value)

This function is the same as ToInteger() except that if the return is an integer, it is in the range of 2^{32} - 1.

The Buffer Object

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

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

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

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

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

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

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

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

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

Buffer Object Properties

.size

The size of the Buffer object. This property may be assigned to, such as "foo.size = 5". If a user changes the size of the buffer to something larger, then it is filled with *NULL* bytes. If the user sets the size to a value smaller than the current position of the cursor, then the cursor is moved to the end of the new buffer.

.cursor

The current position within a buffer. This value is always between 0 and .size. It can be assigned to as well. If a user attempts to move the cursor beyond the end of a buffer, than the buffer is extended to accommodate the new position, and filled with *NULL* bytes. If a user attempts to set the cursor to less than 0, then it is set to the beginning of the buffer, to position 0.

.unicode

This property is a boolean flag specifying whether to use unicode strings when calling .getString() and .putString(). This value is set when the buffer is created, but may be changed at any time. This property defaults to the unicode status of the underlying ScriptEase engine.

.bigEndian

This property is a boolean flag specifying whether to use bigEndian byte ordering when calling .getValue() and .putValue(). This value is set when a buffer is created, but may be changed at any time. This property defaults to the state of the underlying OS and processor.

.data

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

Buffer Object Methods

.putValue(value [, valueSize] [, valueType])

This method puts the specified value into a buffer. The value must be a number. ValueSize or both valueSize and valueType may be passed as additional parameters. ValueSize is a positive number describing the number of bytes to be used and defaults to 1. Acceptable values for valueSize are 1,2,3,4,8, and 10, providing that it does not conflict with the optional valueType flag. (See listing below.)
The parameter valueType must be one of the following: "signed", "unsigned", or "float". It defaults to "signed." The valueType parameter describes the type of data to be read. Combined with valueSize, any type of data can be put. The following list describes the acceptable combinations of valueSize and valueType:

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

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

```
var oldCursor = 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 sig. dig. 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).

.getValue([valueSize] [, valueType])

This method returns a value from the specified position in a buffer object. This call is similar to the putValue() function, except that it gets a value instead of puts a value.

.putString(string)

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. To put a *NULL* terminated string, the following can be done.

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

.getString([length])

This method returns a 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 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.

.toString()

This method returns a string equivalent of the current state of the buffer. Any conversion to or from unicode is done according to the .unicode flag of the object.

.subBuffer(beginning, end);

This method returns another Buffer object consisting of the data between the positions specified by the parameters: beginning and end. If the parameter beginning is less than 0, then it is treated as 0, the start of the buffer. If the parameter end is beyond the end of the

buffer, then the new sub-buffer is extended with *NULL* bytes, but the original buffer is not altered. The .unicode and .bigEndian flags are duplicated in the new buffer. The size of the new buffer is set to the beginning and end parameters. If the cursor in the old buffer is between beginning and end, then it is converted to a new relative position in the new buffer. If the cursor was before beginning, then it is set to 0 in the new buffer, and if it was past end, then it is set to the end of the new buffer.

Buffer[offset]

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

The 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 .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 that is set to a date and time. The following lines all demonstrate ways to get and set dates and times.

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

The first syntax returns a date and time represented by the number of milliseconds since midnight, January 1, 1970. This representation by milliseconds is a standard way of representing dates and times that makes it easy to calculate the amount of time between one date and another. 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 contains one or more of the following fields, in any order:

```
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 PM, which, expressed in 24 hour time, is 13:13 hours and 15 seconds. The time specification is optional and if included, the seconds specification is optional.
The third and fourth syntaxes are self-explanatory. All parameters passed to them are integers.

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

For example, the following line of code:

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

creates a Date object containing the date, October 12, 1492.
The following is a brief description of the methods of the Date object. Instance methods are shown with a period, ".", at their beginnings. A specific instance of a variable should be put in front of the period to call a method.
For example, the Date object aDate was created above, and, to call the .getDate() method, the call would be: aDate.getDate(). Static methods have "Date." at their beginnings, since these methods are called with a literal call, such as Date.parse(). These methods are part of the Date object itself instead of instances of the Date object.

Instance Date methods

.getDate()

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.

.getDay()

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.

.getFullYear()
This method returns the year, as a number with four digits, of a Date object.

.getHours()
This method returns the hour, as a number from 0 to 23, of a Date object. Midnight is 0, and 11 PM is 23.

.getMilliseconds()
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.

.getMinutes()
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.

.getMonth()
This method returns the month, as a number from 0 to 11, of a Date object. January is 0, and December is 11.

.getSeconds()
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.

.getTime()
This method returns the milliseconds representation of a Date object, in the form of an integer representing the number of seconds from midnight on January 1, 1970, GMT, to the date and time specified by a Date object.

.getTimezoneOffset()
This method returns the difference, in minutes, between Greenwich Mean Time (GMT) and local time.

.getUTCDate()
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.

.getUTCDay()
This method returns the UTC day of the week, as a number from 0 to 6, of a Date object. Sunday is 0, and Saturday is 6.

.getUTCFullYear()
This method returns the UTC year, as a number with four digits, of a Date object.

.getUTCHours()
This method returns the UTC hour, as a number from 0 to 23, of a Date object. Midnight is 0, and 11 PM is 23.

.getUTCMilliseconds()
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.

.getUTCMinutes()
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.

.getUTCMonth()
This method returns the UTC month, as a number from 0 to 11, of a Date object. January is 0, and December is 11.

.getUTCSeconds()
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.

.getYear()
This method returns the year, as a number with two digits, of a Date object.

.setDate(DayOfMonth)
This method sets the day, as a number from 1 to 31, of a Date object to the parameter DayOfMonth. The first day of a month is 1, and the last is 28, 29, 30, or 31.

.setFullYear(year[, month[, date]])
This method sets the year of a Date object to the parameter year. The parameter year is expressed with four digits.
If the parameter month is passed, use data format for .setMonth().
If the parameter date is passed, use data format for .setDate().

.setHours(hour[, minute[, second[, millisecond]]])
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 PM is 23.
If the parameter minute is passed, use data format for .setMinutes().
If the parameter second is passed, use data format for .setSeconds().
If the parameter millisecond is passed, use data format for .setMilliseconds().

.setMilliseconds(millisecond)
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.

.setMinutes(minute[, second[, millisecond]])
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.
If the parameter second is passed, use data format for .setSeconds().
If the parameter millisecond is passed, use data format for .setMilliseconds().

.setMonth(month[, date])
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.
If the parameter date is passed, use data format for .setDate().

.setSeconds(second[, millisecond])
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.
If the parameter millisecond is passed, use data format for .setMilliseconds().

.setTime(milliseconds)

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.

.setUTCDate(DayOfMonth)

This method sets the UTC day, as a number from 1 to 31, of a Date object to the parameter DayOfMonth. The first day of a month is 1, and the last is 28, 29, 30, or 31.

.setUTCFullYear(year[, month[, date]])

This method sets the UTC year of a Date object to the parameter year. The parameter year is expressed with four digits.
If the parameter month is passed, use data format for .setUTCMonth().
If the parameter date is passed, use data format for .setUTCDate().

.setUTCHours(hour[, minute[, second[, millisecond]]])

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 PM is 23.
If the parameter minute is passed, use data format for .setUTCMinutes().
If the parameter second is passed, use data format for .setUTCSeconds().
If the parameter millisecond is passed, use data format for .setUTCMilliseconds().

.setUTCMilliseconds(millisecond)

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.

.setUTCMinutes(minute[, second[, millisecond]])

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.
If the parameter second is passed, use data format for .setUTCSeconds().
If the parameter millisecond is passed, use data format for .setUTCMilliseconds().

.setUTCMonth(month[, date])

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.
If the parameter date is passed, use data format for .setUTCDate().

.setUTCSeconds(second[, millisecond]])

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.
If the parameter millisecond is passed, use data format for .setUTCMilliseconds().

.setYear(year)

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.

.toGMTString()
This method converts a Date object to a string, based on Greenwich Mean Time.

.toLocaleString()
This method returns a string representing the date and time of a Date object based on the time zone of the user.

.toSystem()
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.

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

Static Date 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(time)
This method converts the parameter time, which is in the same format as returned by the Clib.time(), to a standard JavaScript Date object. 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(datestring)
This method converts the string datestring to a Date object. The string must be in the following format:

```
Friday, October 31, 1998 15:30:00 -0500
```

This format is used by the .toGMTString() method and by email and Internet applications. The day of the week, time zone, time specification or seconds field may be omitted.

```
var theDate = Date.parse(datestring);
```

is equivalent to:

```
var theDate = new Date(datestring);
```

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

This method interprets its parameters as a date and returns the number of milliseconds from midnight, January 1, 1970, to the date and time specified. The parameters are interpreted as referring to Greenwich Mean Time (GMT).

The 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.

Properties

Math.E

The number value for e, the base of natural logarithms. This value is represented internally as approximately 2.7182818284590452354.

Math.LN10

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

Math.LN2

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

Math.LOG2E

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.

Math.LOG10E

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.

Math.PI

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.

Math.SQRT1_2

The number value for the square root of ½, which is represented internally as approximately 0.7071067811865476. The value of Math.SQRT1_2 is approximately the reciprocal of the value of Math.SQRT2.

Math.SQRT2

The number value for the square root of 2, which is represented internally as approximately 1.4142135623730951.

Methods

Math.abs(x)
Returns the absolute value of x. Returns NaN if x cannot be converted to a number.

Math.acos(x)
Returns the arc cosine of x. 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.

Math.asin(x)
Returns an implementation-dependent approximation of the arc sine of the argument. 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.

Math.atan(x)
Returns an implementation-dependent approximation of the arc tangent of the argument. The return value is expressed in radians and ranges from -pi/2 to +pi/2.

Math.atan2(x, y)
Returns 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. 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.

Math.ceil(x)
Returns the smallest number that is not less than the argument and is equal to a mathematical integer. If the argument is already an integer, the result is the argument itself. Returns NaN if x cannot be converted to a number.

Math.cos(x)
Returns an implementation-dependent approximation of the cosine of the argument. The argument is expressed in radians. Returns NaN if x cannot be converted to a number.

Math.exp(x)
Returns an implementation-dependent approximation of the exponential function of the argument, that is, 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.

Math.floor(x)
Returns the greatest number value that is not greater than the argument and is equal to a mathematical integer. If the argument is already an integer, the return value is the argument itself.

Math.log(x)
Returns an implementation-dependent approximation of the natural logarithm of x.

Math.max(x, y)
Returns the larger of x and y. Returns NaN if either argument cannot be converted to a number.

Math.min(x, y)
Returns the smaller of x and y. Returns NaN if either argument cannot be converted to a number.

Math.pow(x, y)
Returns the value of x to the power of y.

Math.random()
Returns a number which is positive and pseudo-random and which is greater than or equal to 0 but less than 1. This method takes no arguments.

Math.round(x)
Returns the 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.

Math.sin(x)
Returns the sine of x, expressed in radians. Returns NaN if x cannot be converted to a number.

Math.sqrt(x)
Returns the square root of x. Returns NaN if x is a negative number or cannot be converted to a number.

Math.tan(x)
Returns the tangent of x, expressed in radians. Returns NaN if x cannot be converted to a number.

The String Hybrid

The String data type is a hybrid that shares characteristics of primitive data types, Boolean and Number, and of composite data types, Object and Array. The String 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.

The String as data type

A string is an ordered series of characters. The most common use for strings is to represent text. To indicate that text is a string, it is enclosed in quotation marks. For example, the first statement below puts the string "hello" into the variable word. The second sets the variable word to have the same value as a previous variable hello:

```
var word = "hello";
```

```
word = hello;
```

Escape sequences for characters

Some characters, such as a quotation mark, have special meaning to the interpreter and must be indicated with special character combinations when used in strings. This allows the interpreter to distinguish between a quotation mark that is part of a string and a quotation mark that indicates the end of the string.

The table below lists the characters indicated by escape sequences:

\a	Audible bell	
\b	Backspace	
\f	Formfeed	
\n	Newline	
\r	Carriage return	
\t	Tab	
\v	Vertical tab	
\'	Single quote	
\"	Double quote	
\\	Backslash character	
\0###	Octal number	(example: '\033' is the escape character)
\x##	Hex number	(example: '\x1B' is the escape character)
\0	*NULL* character	(example: '\0' is the *NULL* character)
\u####	Unicode number	(example: '\u001B' is the escape character)

Note that these escape sequences cannot be used within strings enclosed by back quotes, which are explained below.

Single quote strings

You can declare a string with single quotes instead of double quotes. There is no difference between the two in JavaScript, except that double quote strings are used less commonly by many scripters. In functions declared with the cfunction keyword, the difference is more important. For more information, see the section on cfunction.

Back quote strings

ScriptEase provides the back quote "`", also known as the back-tick or grave accent, as an alternative quote character to indicate that escape sequences are not to be translated. Any special characters represented with a backslash followed by a letter, such as "\n", cannot be used in back tick strings.
For example, the following lines show different ways to describe a single file name:

```
"c:\\autoexec.bat" // traditional C method
'c:\\autoexec.bat' // traditional C method
'c:\autoexec.bat'  // alternative ScriptEase method
```

Back quote strings are not supported in most versions of JavaScript. So if you are planning to port your script to some other JavaScript interpreter, you should not use them.

Long Strings: Using + to concatenate or join strings

You can use the + operator to concatenate strings. The following line:

```
var proverb = "A rolling stone " + "gathers no moss."
```

creates the variable proverb and assigns it the string "A rolling stone gathers no moss." If you try to concatenate a string with a number, the number is converted to a string.

```
var newstring = 4 + "get it";
```

This bit of code creates newstring as a string variable and assigns it the string "4get it". The use of the + operator is the standard way of creating long strings in JavaScript. In ScriptEase, the + operator is optional. For example, the following:

```
var badJoke =  "I was standing in front of an Italian "
"restaurant waiting to get in when this guy "
"came up and asked me, \"Why did the "
"Italians lose the war?\" I told him I had "
"no idea. \"Because they ordered ziti"
"instead of shells,\" he replied."
```

creates a long string containing the entire bad joke.

The String as 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

.length

The length of a string can be obtained by using the length property. For example:

```
var string = "No, thank you.";
Screen.write(string.length);
```

displays the number 14, the number of characters in the string.

String instance methods

.charAt()

This method returns a character at a certain place in a string. To get the first character in a string, use index 0, as follows:

```
var string = "a string";
string.charAt(0);
```

To get the last character in a string, use:

```
string.charAt(string.length - 1);
```

.charCodeAt(index)

This method returns a number representing the unicode value of the character at position index of a string. Returns NaN if there is no character at the position.

.indexOf(substring [, offset])

This method returns the index of the first appearance of a substring in a string. For 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, the 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.

.lastIndexOf(substring [, offset])

This method is similar to .indexOf(), except that it finds the last occurrence of a character in a string instead of the first.

.split([substring])

This method splits a string into an Array of strings based on the delimiters in the

parameter substring. The parameter substring is optional and if supplied, determines where the string is split. If no delimiters are specified, the method returns an Array with one element which is the original string.

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(' ');
```

.substring()

This method retrieves a section of a string. For example, to get the first ten characters in string, use something like the following code fragment:

```
var string = "a string with many words in it";
var substring = string.substring(0, 10);
```

.toLowerCase()
.toUpperCase()

These two methods change the case of a string. .toLowerCase() returns a copy of a string with all of the letters changed to lower case. .toUpperCase() returns a copy of a string with all of the letters changed to upper case.

String static methods

String.fromCharCode(char1, char2...)

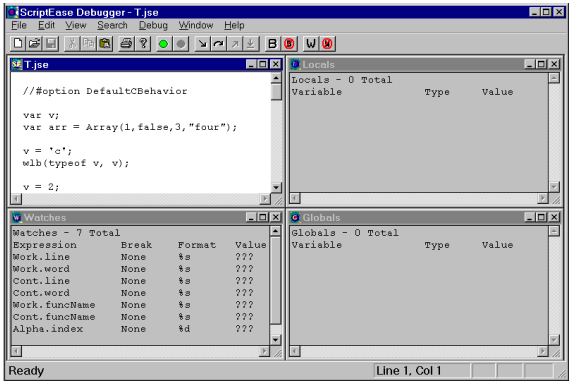
This method returns a string created from the character codes that are passed to it as parameters. The identifier String is used with this static method, instead of a variable name as with instance methods. The arguments passed to this method are assumed to be unicode characters. The following line:

```
var string = String.fromCharCode(0x0041,0x0042)
```

set the variable string to be "AB".

Using the Integrated Debugger

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



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

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

Using the ScriptEase Debugger

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

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

Components of main MDI window

Menu bar

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

Tool bar

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

Document window

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

Status bar

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

MDI windows

Source

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

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

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

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

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

Globals

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

Locals

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

Watches

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

the following code:

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

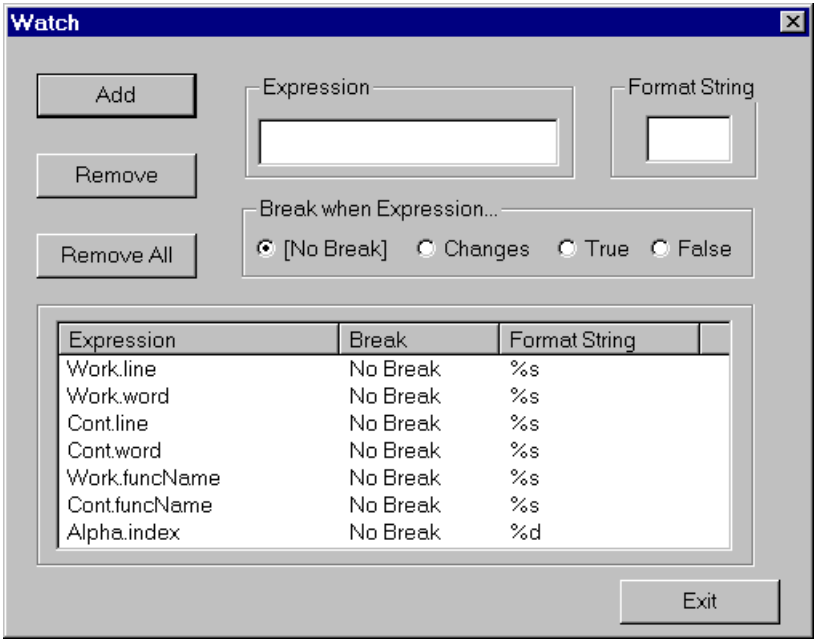
creates an array with four elements. In the Locals and Globals windows, the array `arr` is shown as type object with no value shown.

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

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

Setting watches

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



Add

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

Remove

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

Remove All

The Remove All button removes all expressions to be watched.

Expression

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

Format String

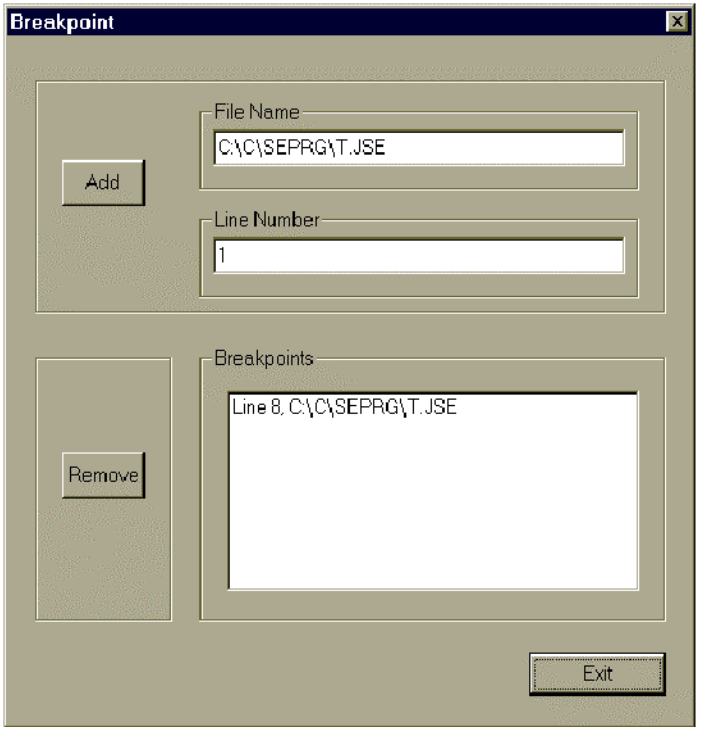
The Format String edit box allows some control over the format of expression, that is, how an expression value will appear.

Break when Expression

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

Setting breakpoints

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



Add

The Add button adds a breakpoint at the line specified, in the Line Number edit box, to

the script specified in the File Name edit box. Of course, the script itself is not altered since scripts are plain text files. Breakpoints are retained as settings within the ScriptEase debugger.

Remove

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

File Name

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

Line Number

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

Breakpoints

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

Main menu bar

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

File menu

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

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

New In toolbar and Ctrl+N

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

Open... In toolbar and Ctrl+O

Open dialog to open a script file.

Close Ctrl+W

Close the currently active script file.

Save In toolbar and Ctrl+S

Save the currently active script file.

Save As...

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

Print... In toolbar and Ctrl+P

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

Print Preview

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

Print Setup...

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

(Recent files list)

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

Exit

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

Edit menu

Undo Ctrl+Z

Undo the last editor operation in the script window.

Cut In toolbar and Ctrl+X

Cut selected text from the script window.

Copy In toolbar and Ctrl+C

Copy selected text from the script window.

Paste In toolbar and Ctrl+V

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

Options

Font...

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

Tabs...

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

Trace On

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

Trace Speed

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

Trace over

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

Source Mark

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

Default Interpreter...

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

View menu

Toolbar

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

Status Bar

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

Search menu

Find... Ctrl+F

Find text in the script window using a find dialog.

Replace... Ctrl+R

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

Debug menu

Start Debug Session

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

Restart

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

Run in Debugger In toolbar and F5

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

Go Ctrl+F5

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

Stop In toolbar

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

Step Into In toolbar and F9

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

Step Over In toolbar and F10

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

Step to Cursor In toolbar and F11

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

Step Out In toolbar and F12

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

Parameters...

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

Breakpoint

Toggle current In toolbar and F8

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

Add/Remove...

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

Remove all In toolbar

Removes all breakpoints in the current script.

Watch

Add/Remove... In toolbar

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

Remove all In toolbar

Remove all watches from the current script and debugging session.

Change Variables

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

Window menu

Cascade

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

Tile

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

Arrange Icons

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

Global... Ctrl+Shft+G

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

Local... Ctrl+Shft+L

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

Watch... Ctrl+Shft+W

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

user.

(Open windows list)

A list of the currently open windows in the debugger.

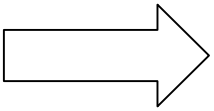
Help menu

Help Topics... F1

Display a help file for the debugger.

About ScriptEase Debugger... In toolbar

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



Index