Paradigm C++
Object Scripting Guide

Version 5.0

Paradigm Systems

Version 5.0

August 9, 1999

Paradigm Systems
3301 Country Club Road
Suite 2214
Endwell, NY 13760
USA

(607)748-5966
(607)748-5968 (FAX)
Sales information: info@devtools.com
Technical support: support@devtools.com
Web: http://www.devtools.com
FTP: ftp://ftp.devtools.com

For prompt attention to your technical questions, contact our technical support team via the Internet at support@devtools.com. Please note that our 90 days of free technical support is only available to registered users of Paradigm C++. If you haven't yet done so, take this time to register your products under the Paradigm C++ Help menu or online at http://www.devtools.com.

Paradigm's SurvivalPak maintenance agreement will give you unlimited free technical support plus automatic product updates for an additional 12 months. Call (800) 537-5043 to purchase this protection today.

Table of Contents

Chapter 4 cScript Class Reference

Chapter 1

Using Object Scripting

About Object Scripting

With Object Scripting, you can customize Paradigm C++ using built-in classes and a C++-like scripting language called cScript. Almost all elements of the Paradigm C++ Integrated Development Environment (IDE) are represented in the IDE scripting classes. Through an object called IDEApplication which is instantiated when Paradigm C++ first starts up, you can access most parts of the IDE, such as the editor, the debugger, keyboard, and the project manager, and change them to suit you.

Object Scripting quick start

The following topics give you some practical tips to help you start writing and running scripts:

- Running script statements interactively
- Writing and loading a script file
- Developing and testing scripts

For more detailed information on loading and unloading scripts and setting Scripting environment options, see "Working with scripts" on page 14.

Running script statements interactively

The simplest way to see the results of your script statements is to enter them interactively, one at a time in the Script Command window.

1. Choose Script | Run. A text entry box opens at the bottom of the screen.
2. Enter the complete syntax of the command you want to run and press *Enter*.

The following two examples show how to enter statements interactively:

- Using the print command
- Using the IDE Message dialog

Using the print command: a simple command entry example

To run a simple Hello script that uses the **print** command,

1. Choose View | Messages and click the Script tab to open the Message window's Script page, which is where the output of all **print** statements goes.

   If you want to start with a clear page, you can delete the messages generated by the IDE startup by right clicking in the Script page and choosing Delete All from the popup menu.
2. Choose Options | Environment | Scripting and click Diagnostic Messages so the script processor will send all scripting messages to the Script page (in case you make an error entering a statement).
3. Choose Script | Run and enter the following statement:

```
'print 'Hi';
```

4. Press *Enter*.

If you then click the Message window and scroll to the end, you see `Hi`.

Using the IDE Message dialog: a simple command entry example

To display output in a dialog box, the IDE object, an instantiation of the standard IDEApplication class (one of the classes that provide access to IDE functionality), provides a Message method that does just that. You can use the Script Command window as follows:

1. Choose View | Messages and click the Script tab to open the Message window's Script page, which is where the output of all **print** statements goes.

   If you want to start with a clear page, you can delete the messages generated by the IDE startup by right clicking in the Script page and choosing Delete All from the popup menu.

2. Choose Options | Environment | Scripting and click Diagnostic Messages so the script processor will send scripting error messages to the Script page (in case you make an error entering a statement).
3. Choose Script | Run and enter the following statement:
   ```
   IDE.Message('Hello World');
   ```
4. Press *Enter*.

You see `Hello World` appear in a message window. As with the print example, if you made an error entering the statement, error messages appear in the Script page of the Message window.

The reason you can simply use the IDE object has to do with the way scripts are loaded during Paradigm C++ integrated development environment (IDE) initialization. On startup, PCW loads a script called STARTUP.SPP that, among other things, instantiates an instance of *IDEApplication* and assigns it to the global variable IDE.

Writing and loading a script file

Typically you write a script in an editor, like the IDE editor, and save it to a file with an .SPP extension. You then load and run the script file by entering its name in the Script Command window. The following instructions show you how to write and run a program that displays `Hello World` in a message window:

1. Choose Options | Environment | Scripting and add your script directory path to the Script Path so the IDE can find your scripts. For example, if your path already contains .;C:\PC5\SCRIPT, it would look like this after you add a directory called C:\MYSCRPTS:
   ```
   .;C:\PC5\SCRIPT;C:\MYSCRPTS
   ```

   Be sure not to insert any spaces before your path name. Doing so will stop the search at the previous path.

2. While you're on the Scripting options page, click Diagnostic Messages so the script processor will send scripting error messages and **print** statement output to the Script page.
3. Press *Enter* to exit the environment settings dialog.
4. Choose View | Messages and click the Script tab to open the Message window's Script page, then scroll to the end of the window.

If you want to start with a clear page, you can delete the messages generated by the IDE startup by right clicking in the Script page and choosing Delete All from the popup menu.

5. Choose File | New | Text Edit to open a new file in the IDE editor, then enter the following script:

```
import IDE;   //Use the IDE object and any of its methods
hello()
{
   IDE.Message ('Hello World');
}
```

6. Choose File | Save and save the file with an .SPP extension in a directory of your choice (for example, C:\MYSCRPTS\HELLO.SPP).
7. Choose Script | Run File to compile and run the script.

Any statements that aren't in a function or other block will simply execute the first time you load the program. If you have a function called_init(), that function will also run when you load the program. If you have function with the same name as the file, that will be the default function that runs when you load the program after any _init() function runs.

When you load the script HELLO.SPP for the first time, it displays Hello World in the message window and then stays in memory. If you subsequently choose hello from the Script | Commands window, the script processor calls the function hello(), which displays `Hello World` in the message window.

Developing and testing scripts

As described in steps 1-4 of "Writing and loading a script file" on page 12, before you start writing scripts, you should set your scripting environment options to add your script directory to the script search path and to output messages to the Messages window. Then open the Messages window's Script page. Additionally, make sure that the Scripting option Create Token Files is not checked. (It should be off by default.)

While you're developing a script in an edit window, you can,

- Test single statements or short lists of statements in the Script Command window, as described in Running script statements interactively.
- Compile and run an .SPP file that is the active editor file by choosing Script | Run File.

If there are syntax errors in the script, error messages are displayed in the Script page of the Messages window. If your script doesn't run when you try to load it or you hear beeping, click the Messages window and scroll to the end of the Script page to see what happened.

After you get a script working and you save it to a file, you can,

- Load existing script files by choosing Script | Modules, selecting the module you want to run, and clicking Load. All loaded modules and all modules on your script path are listed in the Modules window. If you want to load an already loaded script again, unload it first (Choose Script | Module and click Unload.)
- Run a loaded script has either an *_init*() function or a function with the same name as the script file by choosing the function name from the list of commands. (If it has *_init()* function, *_init* will be one of the commands in the list window.)

Working with scripts

The following topics provide details about working with scripts, such as different methods of loading and unloading them, what Paradigm C++ does when it loads a script file, and what all the options mean on the Scripting dialog page (View | Environment | Scripting).

Writing scripts

Scripts are simply ASCII files. You can write a script in the IDE editor or another editor, then save it as an SPP (with an .SPP extension). Header files for scripts typically have the extension .H.

Loading scripts

When you load a script, it also runs. If the script affects the display (for example, it contains *print* statements), you see something happen on screen immediately. If you define new behavior for the IDE, you see that behavior when you use that part of the IDE. The script remains loaded until you unload it.

If you're developing and testing scripts, instead of loading scripts, use the Script | Run File command.

You can load a script in any of the following ways:

- Use the Script | Module dialog to choose and load a script file.
- Open the script in an edit window and choose Script | Run File (useful if you want to change the script).
- Specify a script on the PCW command line with the **-s** switch. It is processed after the complete processing of scripts specified in the Startup Scripts entry on the Scripting dialog page (Options | Environment | Scripting). Simple script names require no quotation marks. If you include script parameters, put the parameters in parentheses. To pass string parameters, enclose the strings in backslash-quotation combinations. To start multiple scripts, use the **-s** parameter for each script.

Examples

```
pcw -sScript1 -sScript2 -sScript3(Param1, Param2)
pcw -sMyScript(\"string\", \"parameters\")
```

Pro — The script won't be affected whenever you update to a new version of Paradigm C++.

Con — To start Paradigm C++ with the script, either you have to have an icon for Paradigm C++ that uses this command line or you have to remember to type it each time.

- Enter the name of the script file in the Startup Scripts field of the Scripting dialog page (Options | Environment | Scripting). For example, enter test. You can specify multiple scripts here separated with spaces. For example,

  ```
  test MyScript bar
  ```

Pro — Since the script names are stored in the configuration file, they can be shared across multiple PCW users.

Con — You have to reenter the script names every time you install a new version of Paradigm C++.

- Modify the source code of the STARTUP.SPP file (or any of the files that it loads.)

Pro	Regardless of where you run the script, you get the same results.
Con	When you update to a new version of Paradigm C++, you have to redo the changes to STARTUP.SPP.

By convention, the source files for scripts have the extension .SPP. When you load a script for the first time, it is compiled into an interpreted tokenized format called pcode. By default, a tokenized file is created that has the same name and the extension .SPX in the same directory as the script file.

After a script has been successfully loaded, it is scanned for the existence of two macros. One is named *_init()*, and the other is a macro with the same name as the script file just loaded. If these macros exist, they will be automatically called, *_init* first, then the other. If a series of scripts are loaded at the same time (on startup or from the command line), first all the *_init()*s are processed (left to right), then the named macros are processed.

Unloading scripts

Scripts are not unloaded automatically. To unload a script, choose Script | Module, then in the dialog box choose the script name and click Unload.

Setting Object Scripting options

The Scripting dialog (Options | Environment | Scripting) provides control over the script environment. The following table describes the choices on this dialog:

Option	Description
Stop at breakpoint	If the keyword **breakpoint** appears in the script, stop the script when it is encountered and load the script debugger.
Diagnostic Messages	Specifies whether or not to display all script processor messages in the IDE Message window's Script page (View | Messages | Script). This option by default is off.
Script path	Describes the path to search when loading a script file. During a load, every entry on the path will be searched for a file with the .SPX extension. If that fails, the same directories will be searched a second time for files with the .SPP extension. Starting the path with .; causes the current directory to be searched first.
Startup Scripts	Specifies the script files to load and execute as part of the IDE startup procedure. (Paradigm C++ always tries to load STARTUP.SPP from the SCRIPT subdirectory or any path you specify for scripts.) Use spaces to separate multiple script names. You can specify script parameters by enclosing the script name and its arguments in quotation marks. For example, `MyStartup DisplayCurProj 'Ascript Param1'`

Chapter 2

Using cScript

About cScript

The cScript language is a late-bound, object-oriented language that supports syntax and constructs familiar to the C++ developer.

cScript offers C++ programmers a familiar environment for customizing the IDE. It has many of the same constructs as C++ and on the surface looks and feels like C++. But under the hood the two languages are very different: They address two separate problem domains, the early-bound environment versus late-bound, and as a result there are some major semantic differences.

For more information, choose one of the following topics:

- "The advantages of a late-bound language" on page 17
- "cScript and types" on page 19
- "Comparing cScript and C++ on page 18

The advantages of a late-bound language

cScript is a *late-bound*, object-oriented language, which is roughly analogous to being an interpreted language. This gives cScript programs more flexibility than *early-bound* programs, such as those written in C++. In C++, everything about a program is known at compile time. The types of the variables, the return types and number of parameters to functions, the classes that will be used as well as all their properties and behaviors are all known when the program is compiled.

cScript is very different. While the syntax looks very similar to C++, you cannot declare a variable's type at compile time. Variables are generic and can hold any type of data needed at run-time. In fact, the same variable can hold different types of data as the program executes.

Just as in C++, you create classes with properties and methods and create objects which are instances of those classes. But in cScript, you are free to override the methods for a given object (not the *class*, just the *object* itself) at run-time with a new implementation of the method or a method "borrowed" from another object.

This means that an object of one class can use the methods of an object of another class without having to know anything about the second object at compile time. Existing objects can have their functionality extended without the need for the source code to the object's class, and without recompiling.

The benefits of late-binding

Late-binding provide important practical benefits. Let's say that you want to create a program to extend the functionality of the Paradigm C++ IDE. For example, you want to create a script that automatically saves changed source files to a central repository on the network as well as in your project directory. You want to add this functionality to the IDE and have it behave like a built-in feature.

The Paradigm C++ IDE is represented by a cScript object called IDE of the cScript class IDEApplication. If the object IDE was instead created from a C++ class, you would have to alter that C++ class and add your repository methods to it directly, through multiple inheritance, function pointers, or through some other mechanism. Then you would need to recompile the source for the class to create the extended object IDE. In cScript, you do not need to touch the definition of class IDEApplication at all. You can use cScript to attach your repository methods to the IDE object at *run-time*. There are no changes to the IDEApplication class and no recompilation is necessary.

So late-binding means that you can alter and extend the behavior of objects without having to know the details of how they are implemented, without having access to the source code, and without having to recompile.

Comparing cScript and C++

cScript differs from C++ in the following ways:

- All class members are public. There is no way to make members private or protected as part of their declaration. You can use **on** statements to make members inaccessible.
- cScript programs have no main() or WinMain() function.
- Globally scoped statements are allowed and will be executed when the script is run.
- Executable statements are allowed within a class definition, and in conjunction with optional initialization arguments passed when the class is instantiated, constitute the class's constructor. There is no constructor function per se in cScript.
- The implementation of a class's methods are defined within the class.
- The definition (not just the declaration) of a member function must always occur in the class declaration.
- Arrays are objects in cScript. When deallocating an array with the **delete** command, the square brackets "[]" are not needed.
- Functions may have varying numbers of parameters. cScript truncates or pads argument lists as necessary.
- Compound logical expressions do not short circuit. For example, in the expression if(TRUE || Foo())..., the function Foo() will always be called even though the constant TRUE insures that the expression will always evaluate to true.
- cScript does *not* have the following C++ features (this is not a complete list):
  - type checking (but there are type conversions with some operations)
  - type casting
  - multiple inheritance
  - C++-style exceptions
  - class constructor functions
  - function overloading
  - character arrays (cScript directly supports strings)
  - default arguments to functions
  - templates
  - default parameters in method declarations
  - pointers
  - direct memory access
  - function declarations that support default parameters

- enums
- unions
- structs or typedefs
- bitfields
- operator overloading
- **const** keyword (except in DLL imports)
- **static** keyword
- global scope resolution. You can access globally scoped variables, using the module function.
- The **#if** preprocessor directive
- The following operators: -> * ->* .*

cScript comments

cScript supports C++ comment syntax, including:

```
// This is a comment to the end of the physical line
/* This is a comment to the closing */
```

Nested comments are permitted in cScript.

cScript identifiers

Identifier names are made up of letters, digits and underscores (_). The first character of an identifier name cannot be a digit. Identifier names can be up to 64 characters in length. cScript is case-sensitive. Therefore foo, Foo, and FOO are three different identifiers. Keywords, operators, and intrinsic function names are also case-sensitive.

cScript and types

cScript is not an explicitly typed language and does not allow you to declare variables with C++ base types. When the parser encounters an unknown identifier, it makes it a new variable (unless the identifier is immediately followed by an open parenthesis, which might indicate it's a function). New variables created this way are local to the current scope.

The only declarators you can use are **declare**, **import**, and **export**, which are not types but declarators that indicate a new variable. "cScript modules and scope", page 20 discusses declare, import, and export.

Identifiers do have types, but the type of an identifier is determined by its value. For example, *x* in the following code is an integer because it is assigned an integer:

```
declare x = 25;
```

x can become any other cScript native type, depending on what is assigned to it. In the following example, *x* is of type IDEApplication because an object of that class is assigned to it:

```
declare MyIDE = new IDEApplication;
x = MyIDE;
```

Use the intrinsic function typeid to determine the type of an identifier.

Types also come into play when you use operators with variables of different types, the simple conversion rule with binary operators (such as + and /) is that the variable on the left determines the type of the expression. For example,

```
declare x = 4;
declare y = 4.0;
print x/3; // output is 1
print y/3; // output is 1.333333
```

The rule becomes more complicated with conversions between strings and numbers because cScript does some interpretation. When converting from a number to a string, cScript represents digits as numeric strings (3 becomes "3"). When converting from a string to a number, the string is converted to a number if the string can be interpreted as a number. If the string evaluates to anything but a number, it is converted to zero ("33" becomes 33, "33abc" also becomes 33, but "abc33" becomes 0).

If an object is converted to a string, it becomes the string "[OBJECT]". For example,

```
declare a = new IDEApplication;  // create a new
                                 // IDEApplication object
declare b = "Hello";             // create a new string variable
                                 // "add" the object to the string,
                                 // converting the object to a
                                 // string
declare c = b + a;
print c;                         // prints "Hello[OBJECT]"
```

cScript modules and scope

A cScript source file (an .SPP file) is a module. A variable declared or used for the first time at the module level is global to that module, and a variable declared or used for the first time inside a block is local to that block.

Because you don't have to declare variables as you do in C++, it's easy to mistakenly use a global variable in a function or class when you intend it to be local. It's safest to use **declare** with variables that you intend to be local. For example,

```
declare X = 2;          // Module scope X
declare Y = 4;          // Module scope Y

Func1(X){               // Parameter (local variable) X
  Y = "hello";          // modifies global Y.
}
Func2(X){               // Parameter (local variable) X
  declare Y = "hello"; // New local variable Y created
                        // and set to "hello".
}
```

Variables created at the module level (not in a function, method, class, control structure, or block) are global variables of the module. They are not normally accessible to other modules. To access a variable defined in module A from module B, three things must occur:

- Both Module A and Module B must be loaded.
- The variable must be declared **export** in Module A, at module scope.
- Module B must contain an **import** statement for the variable, at module scope.

Example

```
Module A
  declare varOne;  //A global variable accessible only in Module A.
  export varTwo;   //A variable accessible outside Module A.

Module B
  import varOne;   //Trying to link with exported varOne from
                   //another module (not varOne in Module A)
  import varTwo;   //Trying to link with varTwo in Module A.

  varOne = 33;     //Causes the run-time warning "Cannot locate
                   //external variable varOne".
  varTwo = 33;     //Changes the value of varTwo in Module A to
                   //33.
```

cScript statements

As in C++, statements must terminate with a semicolon. You can group multiple statements by surrounding them with braces. Variables declared within braces are local to those braces and go out of scope when the closing brace is reached. You can chain expressions with the comma operator.

cScript operators

The standard C operators are evaluated using the same precedence rules as in C. Operations may be grouped by using parentheses. The operators supported by cScript are in the Precedence of Operators.

Strings may be concatenated with + and +=. As in C++, you can use the colon (:) to derive a new class from an existing class. There is a new operator in cScript that defines a closure, the :> operator. You can use the operator ?? to test for elements of classes and arrays.

cScript strings

cScript strings (note the lowercase "s") work much the same as C++ strings. A string is a series of characters delimited by quotation marks. In cScript, a string's length is limited to 4096 bytes. cScript automatically keeps track of the ends of strings; appending '\0' (NULL) is unnecessary.

cScript recognizes many C++ formatting characters within strings such as new line (\n) and horizontal tab (\t).

Besides the alphanumeric and other printable characters, you can designate hexadecimal and octal escape sequences much as you can in C++. These escape sequences are interpreted as ASCII characters, allowing you to use characters outside the printable range (ASCII decimal 20-126).

The format of a hexadecimal escape sequence is \x<hexnum>, where <hexnum> is up to 2 hexadecimal digits (0-F). For example, the string "R3" can be written as "\x523" or "\x52\x33".

Octals are a backslash followed by up to three octal digits (\ooo). For example, "R3" in octal could be written "\1223" or "\122\063".

cScript arrays

cScript supports two types of arrays in cScript, bounded and unbounded (associative).

- Bounded arrays
- Associative arrays

Bounded arrays

cScript bounded arrays are similar to C++ arrays and are created with a size specifier. Run-time warnings will occur if you attempt to access a bounded array out of bounds. Bounded arrays use a zero-based index; that is, the first element of an array is element 0 and the last element is element *size* - 1.

You can declare a bounded array by using either of the following syntax variations:

```
x = new array [10];
array x[10];
```

Access is then as you would expect:

```
x[0] = 5;
x[1] = "a string";
x[2] = Foo;
x[3] = x[2];
```

You can also create a bounded array using the following initialization list syntax:

```
z[] = {"one", "two", x};  //Note the use of braces, {},
                          //rather than brackets, [].
```

In this case, "one", "two", and the value of *x* are the values in the array, and the array indexes start at 0 and go to 2. For example,

```
print z[0];   //Prints one
print z[1];   //Prints two
print z[2];   //Prints the value of x
```

You cannot initialize variables in an array initialization list: You must initialize them elsewhere. For example, you *cannot* define an array as follows:

```
z = {x=1, y=3, slogan="No more woe"} //Illegal syntax
```

In this array definition, assignments to *x*, *y*, and *slogan* must be elsewhere in your code.

An attempt to assign values beyond array bounds causes a run-time warning, but it works. Such an assignment doesn't increase the size of the bounded array to match the new index, but rather creates an associative array that is attached to the original bounded array. You can use any value as the new index. For example,

```
x[15] = "New index";           //Run-time warning,
                               //creates new associative array.
x[16] = "Another new index ";  //Run-time warning, adds to
                               //associative array.
print x[15];                   //Prints New index
print x[14];                   //Prints UNINITIALIZED (index not in
                               //array, so no value is assigned)
```

You cannot use a negative number to index into an array. Doing so causes a syntax error.

Associative arrays

You create associative arrays without a size specifier and access them *on demand*. They grow as required. Associative arrays are typically sparse and do not perform as well as bounded arrays.

To create a new associative array, use one of the following syntax variations:

```
z = new array[];
array z[];
```

Associative arrays can take string as their indexes as well as numbers. Typically, the index of an associative array element is something that is related to the data the element holds. For example:

```
History = new array[];
History["President"] = "Bill Clinton"
History["Vice President"] = "Al Gore"
History[1776] = "U.S. Independence"
History[1789] = "U.S. Constitution"
```

You also create an associative array when you make assignments beyond the bounds of a bounded array. See the previous section for more information.

cScript prototyping

Forward referencing for functions and methods is not supported. Because scripts are interpreted in a single pass at run-time, classes and methods and the methods in them must be defined before they can be used. There is no function prototype mechanism. This is because when the parser sees a function call, it needs to know the implementation at that time, whereas at compile-time, a C++ compiler only needs to be able to match the name, number of parameters, the types of the parameters and the return values. It does not really need to know anything internally about the function.

Parameter counting and type conversions are performed at run-time. cScript will pad (with NULLS) or truncate the argument list as necessary at run-time to ensure that the correct number of arguments is available on the stack.

cScript flow control statements

The following flow control statements work in cScript as they do in C++:

break	continue	do
else	if	for
return	while	

The behavior of **switch** is slightly different. Because cScript is not a compiled language, the expression is checked against each case exactly as if evaluating an **if-else if** construct. This means that the cases need not be constants - they may be any expression (including function calls). It also means that if a **default** case is desired, it must be the last case.

Switch example

```
switch( someNumber ) {
   case 3:      //Execution continues to case bar()
   case MyFunc():
      DoSomeStuff();
      // No break. Even if this case executes,
      // the next case is still evaluated.
   case W.Y.Z:
      DoSomethingElse();
      break;    // If this case executes, switch ends here.
   case 42:
      DoItAll();
   default:
   // Anything not matching previous cases comes through here
}
```

cScript classes

cScript supports single inheritance. There is no support for overloaded methods (member functions). In addition, there is no hiding of members: all properties (member data) and methods are public and virtual. You can override an instance of a class (an object) with **on** and **pass**, and you can bind objects' events (function calls) together in an event handling chain using **attach**. See "cScript event handling" on page 26 for more information.

All methods must be defined entirely in the class definition. A class definition may be nested in another class definition. The name of that nested class exists in the scope of the outer class, and is thus protected from accidental collision with identifier names in the module and global scopes. You can instantiate a nested class with the following syntax:

```
// Class Inner is nested in class Outer
innerObject = new Inner from Outer;
```

There are two ways to modify the behavior of methods in script classes:

- Derive a new class from the script class, overriding the methods whose behavior you want to change. Use this technique when you want to provide new behavior for a collection of objects.
- Override an instance of a class by using an **on** handler or **attach** to hook one of the object's methods. Use this technique when you want to tweak the behavior of a particular instance of a class.

By the same token, there are two ways to modify the behavior of properties in script classes:

- Derive a new class from the script class, overriding the properties whose values you want to change. Use this technique when you want to provide new behavior data values for a collection of objects.
- Override an instance of a property by using getters and setters. Use this technique when you want to tweak the behavior of a particular instance of a class.

Declaring a class

There are no C++ -like constructors in cScript. (Defining a method with the same name as the class, as you do in C++, does not make it a constructor.) Instead, code embedded in the class declaration that is not part of a method declaration is considered constructor code. For this reason, constructor arguments must be defined in the class declaration.

Member functions must be defined entirely in the class declaration. You cannot declare a member function in a class and then define it later in the program.

There are destructors in cScript, and they work as they do in C++. (Defining a method that starts with a tilde (~) and has the same name as the class makes it a destructor.) Destructors are called when the object is being destroyed.

For example, the following class is declared without parameters:

```
class noParams{
  declare aMember;
  declare anotherMember;
  Func1();    // constructor code
  for (y = 1; y < 10; y++) // more constructor code
     print "hello";
     ~noParams(){
     print "A noParams has been destroyed.";
  }
};
```

The following class is declared with parameters:

```
class Base(parmOne, parmTwo) {
  declare X = parmOne;  // a member variable
  declare Y = parmTwo;  // a member variable
  MethodOne() {
    X = X + Y;
  }
  AnotherMethod() {
  }
};
```

The following class is inherited from the class *Base*:

```
// aParm and cParm are passed through to
// Base as parmOne and parmTwo.
class Derived(aParm, bParm, cParm): Base(aParm, cParm) {
  declare Z = bParm;
};
```

Initialization arguments must be explicitly passed to the base class. They must also be stated in the derived class parameter list because that is the list referenced when a derived class object is instantiated.

Creating instances of cScript classes

Objects in script are created in one of two ways (assuming an already defined class *Foo*):

```
x = new Foo();
```

or

```
Foo x();
```

As with any declaration, you can use the **declare** and **export** keywords when you create objects. For example,

```
declare x = new Foo();
export Foo y();
```

cScript has automatic garbage collection. When an object goes out of scope, it is deallocated. Objects can be explicitly deallocated using the **delete** command. For example:

```
declare x = new Foo(); // allocate new object
delete x;              // explicitly delete object
```

Because cScript is untyped, you can destroy an object by assigning it another value. For example, cScript does not complain when you assign 0 to the object x as follows:

```
declare x = new MyClass();// create an object of class MyClass
x = 0;                    // object overwritten and replaced with 0
```

Discovering cScript class and array members

You can use **??** and **iterate** to discover the contents of classes and associative arrays. With **??** you can test individual members to see if they exist or if a certain value appears in a class or array. With **iterate**, you can see all members of a class or array.

About closures

Closures provide a means to obtain a reference to a method or property without invoking it. They provide functionality analogous to function pointers in C++.

Use the closure operator (:>) to bind a class instance (an object) with one of its methods in a single reference. You typically use the closure operator in on handlers and attach and detach statements to handle a method call. You can assign a closure to a variable and use that variable anywhere you would use the closure. An object method call hooked using an on handler, or attached in an attach statement, need not even exist in expanded or modified dynamically at run-time without effecting the objects' underlying class definition. It is important to note where event handlers are defined for specific object instances, these handlers in no way effect any other existing objects of that type. Only when handlers are defined within a class definition itself using the **this** reference do all objects of that type inherit that event handling behavior.

Closures are powerful features of cScript. You can pass a closure as a function argument, for example. Since it represents a member of a class instance (an object), it carries a **this** pointer for that object with it and has all of the object's context information.

Another use for closures is to declare arrays of closures to use like arrays of function pointers, only the functions need not do anything unless they happen to be defined. Calling an undefined closure is not an error - nothing happens because there's nothing to call.

cScript event handling

cScript uses an event handling model to override class behavior. Given an instance of a class, you can modify its behavior by "hooking" a specified method and supplying an alternative implementation. You can use either an on handler or attach and detach to accomplish this.

- Using on handlers
- Using attach and detach

Using on handlers

You can use an **on** handler to hook a method call for an instance of a class and override, or enhance, its functionality. You need not call the hooked method inside the **on** handler: Any code in the **on** handler will be executed instead of the hooked method. If you want to invoke the original method, use **pass()**. If the hooked method returns a

value, that or any other value can be returned by assigning the return value of pass to a local variable, including a return statement in the event handler.

In the on handler header, you use the closure operator (:>) to bind a class instance (an object) with a method of the object as a closure reference.

```
Declare AClass MyObject;  // or MyObject = new Aclass;

// Given this instance of class AClass, you can intercept one
// of its methods.

on MyObject:>Method1(parm1){
  // …
  // Programmer may provide some preprocessing here.
  // Programmer may delegate to original implementation
  // or get original return value with pass().

  Declare rv = pass(parm1); // call MyObject.Method1(parm1)

  // Programmer may provide some postprocessing here.

  return rv;
}
```

In order to be bound to an existing object method, the number of parameters in the on handler definition must match the hooked method. Once invoked, however, pass() will call the hooked event regardless of how many arguments it passes. As with all function calls, cScript will ensure that the proper number of arguments are passed, truncating or padding as needed.

While inside an **on** handler, there are a few things to keep in mind:

- You aren't actually in a method of the object. Simple function calls resolve to their global counterparts, not to the object's methods. If you want to call the method *bar()* from the *Method1()* **on** handler, you must explicitly denote the object. For example,

  ```
  on MyObject:>Method1(){
     MyObject.bar();
   }
  ```

- Another way to explicitly denote a method of this object is to use the shorthand *dot notation*, which relies on the fact that, in an **on** handler, the dot is a shortcut for the controlling object. For example (given an object MyObject that has methods Method1() and bar()),

  ```
  on MyObject:>Method1(){
     .bar();
  }
  ```

Using attach and detach

An **on** handler is not dynamic, but stays in effect once established. If you want to make dynamic changes to class instances, you can set up dynamic **on** handlers using the closure operator with **attach** and **detach**. Attached closures are used to set up a linkage between any member (method or property) of an instance of one class with any member from an instance of another class.

Example

```
x = new Foo();        // Create an instance of Foo called x
                      // and assume Color() is a method.
x.Color();            // Call x.Color().
y = new Bar();        // Create an instance of Bar called y
                      // and assume Notify() is a method.
y.Notify();           // Call y.Notify().
attach y:>Notify to x:>Color; // When x.Color() is called,
                      // instead call y.Notify().
x.Color();            // Call y.Notify().

// NOTE: In y.Notify() a pass() will
// now delegate back to x.Color().

detach y:>Notify from x:>Color; // unlink the two objects
x.Color();                      // Call x.Color().
```

Controlling access to cScript properties

You can use **on** handlers to control what happens when users get (read) or set (write) the values of properties. These two types of **on** statements are called *getters* and *setters*. This feature allows you to execute some code when a property is accessed instead of having to implement the property as a method.

- Using getters
- Using setters

Using getters

The syntax for a getter is:

```
on object:>property{
  [optional pre-processing statement(s)]
  return [pass()|SomeValue];
}
```

Since no value is passed to the handler, no parameter is needed. There must be a `return` statement because a getter is always invoked when the object's property is used in a statement that needs to obtain its current value. When the user accesses the property (for example, on the right hand side of an assignment operator or as an argument in a print statement), the on-read property event handler is called and its statements are executed.

You can use a getter for various purposes, including:

- Restricting access to a property.
- Executing related methods or modifying related properties.
- Performing computations on a value before returning it.

Example

The following getter hides the property *Hidden1*:

```
// GETTER.SPP
import IDE;   //Import IDE, an IDEApplication object

class MyClass () {
 declare Hidden1 = "Hidden: can't see this one";
 declare Public1 = "Public: can see this one";

// Getter
 on this:>Hidden1 {
   return NULL;
 }
} // End MyClass declaration

getter() {
 declare MyClass myobj;
 IDE.Message (myobj.Hidden1);
      //Prints nothing
 IDE.Message (myobj.Public1);
      //Prints "Public: can see this one"
}
```

Using setters

The syntax for a setter is

```
on ClassInstance:>property(parameter){
 [optional pre-processing statement(s)]
 [pass(parameter | SomeValue);]
 [optional post-processing statement(s)]
}
```

Unlike the getter syntax, parentheses and a parameter are required for the setter to obtain the value intended to be assigned to the hooked object property. If you want the handler to be able to set the property (rather than simply block write access to it), there must be a pass(); statement that sets the property's value. When the user tries to set the property (for example, when the property is used on the left hand side of the assignment operator `object.property = 1`), the **on** handler code executes.

Some uses for a setter are

- To restrict values of a property to a certain range.
- To limit access to a property (or even make it read-only).
- To execute related methods or modify related properties.
- To perform computations on a value before setting it.
- To convert user-supplied data to an internal format.

Example

In the following example, the setter uses the value set in *radius* to calculate and set the values of *circumference* and *area*. It then passes the user's value on to *radius*.

```
// SETTER.SPP

import IDE;    //Import IDE, an IDEApplication object
declare PI = 3.141592654;

class Circle(rad) {
 declare radius = rad;
 declare circumference;
 declare area;
```

```
// Setter
 on this:>radius(x) {
 if (x > 0) {
   circumference = PI * 2 * x;
   area = PI * x * x;
   pass(x);
  }
  else
   IDE.Message("Error: Radius must be greater than zero.");
 }


// Methods

}

 ShowProperties() {
   IDE.Message("radius = " + radius +
               ",     circumference = "
             + circumference +
               ",     area = " + area);
 }
}     // End of Circle class declaration

declare Circle obj(1);    //Initialize radius to 1.
obj.ShowProperties();

//Call the IDEApplication method SimpleDialog to prompt
//the user for input and get a value for radius.
Declare radius = IDE.SimpleDialog("Enter a radius", "10");

obj.radius = 0 + radius; //Convert string to integer
obj.ShowProperties();
```

cScript pass by reference

Parameters passed to methods and functions are passed by value unless explicitly made to be passed by reference. (Passing by value does not allow changes to the value of the caller's variable, while passing by reference does.) For example,

```
PassByValueFunction(aValueParameter){
  aValueParameter = 100; // Value of aValueParameter changed to
                         // 100. Caller's value unmodified.
}

PassByReferenceFunction(&aReferenceParameter){
  aReferenceParameter = 100; // Value of aReferenceParameter
}                            // changed to 100. Caller's value
                             // also updated
```

If you want to pass a variable by value in a pass-by-reference parameter, put it in parentheses. For example,

```
x = 10;
PassByReferenceFunction((x));
print x;   // Prints 10
PassByReferenceFunction(x);
print x;   // Prints 100
```

cScript built-in functions

The cScript language provides the following built-in functions:

- attach
- call
- detach
- initialized
- load
- module()
- pass
- print
- reload
- run()
- select
- typeid
- unload
- yield

cScript reserved identifiers

cScript reserves identifier names starting with two underscores as internal to the language. There are additional identifiers reserved for future use. The following identifiers are reserved and are not available for use in your scripts:

__break	Factory
__const	false
__cdecl	FALSE
__error	library
__pascal	method
__refc	NULL
__rundebug	object
__runimmediate	property
__stack	system
__stdcall	true
__warn	TRUE

event

cScript named arguments

If you don't know the order of arguments of a function, you can use *named arguments* when you call the function and cScript will ensure that the function receives the arguments in the correct order. To use named arguments, call the function with the parameters named and indicate the values you want to pass with a colon (:), as follows: *argument1:value1*[,*argument2:value2*[,…]]. You can put these parameters in any order. For example,

```
SetPos (row, column) {
  print "row = ", row, ", column = ", column;
}
SetPos (5,10);               //Outputs "row = 5, column = 10"
SetPos (column:5, row:10);   //Outputs "row = 10, column =5"
```

cScript error handling

cScript uses a mechanism similar to **on** handlers and closures to provide error handling with proper object cleanup. The technique requires that you create an error handling class in some module. An error is raised by invoking an exception method on an object of the error class. That method can do whatever you want (display a message, or attempt to correct the problem), and then it can **return**. If it **returns**, control is returned to the point at which the caller was invoked, thus exiting the caller immediately. This effectively terminates the entire call tree. For example, a module providing error handling contains the following code:

```
class ErrorHandler{
}
errhand = new ErrorHandler();
onerror FileOpen(terminate) from errhand {
  print "FileOpen must have failed";
  if(terminate)
   return
}
```

In some other module there is the following code:

```
import errhand;
OpenIt(){
  //Must be able to find the data file
  TryToOpen("foo.dat", TRUE);

  // Don't have to have a config file
  TryToOpen("foo.cfg", FALSE);
  print "No problem opening anything";
}
TryToOpen(filename, continueORAbortIfFailed) {
  declare fileObject = FileOpen(filename);
  if(fileObject == NULL)
   errhand.FileOpen(continueOrAbortIfFailed);
}
TestIt(){
  OpenIt();
  print ("So far so good");
}
```

At run-time, when someone calls TestIt(), if the file FOO.DAT cannot be opened, control will never be returned to OpenIt() or TestIt(), the open on FOO.CFG will never be attempted, and "So far so good" will never be printed. Instead the message "quitting" will be printed. If FOO.DAT can be opened, an attempt is made to open FOO.CFG. If file OpenIt() was called, TestIt() will immediately print "So far so good". If FOO.CFG can be opened, the message "No problem opening anything" is printed, and control is passed (as is normally the case) back to TestIt().

cScript access to exported DLL functions

Because all needed functionality is not directly available through the language or exposed by an object in the system, cScript allows you to access a function in a DLL directly through script by using code similar to the following:

```
// expose DLL entrypoints
import "foo.dll" {
 int __pascal FooFunc(short, char, unsigned, long);
 void DoIt();
}
// directly access the DLL calls
if (FooFunc(1, "hello there",2,3))
  print "FooFunc() succeeded";
else
  DoIt();
```

This DLL call uses the data type keywords **short**, **char**, **unsigned**, and **long**. Other data type keywords available for use in DLL calls are **void**, **int**, **bool**, and **const**.

cScript supports the calling conventions **__cdecl**, **__pascal**, and **__stdcall**.

cScript and OLE2

cScript to OLE2 interaction

If an automatable object has been exposed in the OLE2 registry, its functionality may be accessed from cScript by using the special *OleObject* class. For example,

```
// Creates an object with all the methods of Microsoft Word BASIC
wordBasic = new OleObject("word.basic");
// Call the Word BASIC function AppInfo() to find out what version
// of Word is installed on the system
print wordBasic.AppInfo(2);   // Returns "7.0" for Word version 7.0
```

OLE2 to cScript Interaction

The IDE registers the automation name *ParadigmIDE.Application* with the OLE2 registry during initialization. From any automation controller, the IDE's functionality may be accessed by creating a ParadigmIDE.Application object and using it. For example, from a *Visual* dBASE program you could do the following:

```
ParCppIDE = NEW OleAutoClient("ParadigmIDE.Application")
ParCppIDE.ProjectOpenProject("foo.ide")
IF(ParCppIDE.ProjectBuildAll())
  ParCppIDE.FileSend("success notification")
ELSE
  ParCppIDE.FileSend("failure notification")
ENDIF
```

About the IDE Class Library

In the Paradigm C++ integrated development environment (IDE), all user commands are directly mapped to a corresponding script. Every IDE window that uses the keyboard API has each keystroke mapped to a script. All main menu commands have a mapping to a script. These scripts are supplied by Paradigm and provide standard behavior that you can enhance to provide your own custom environment. If you want to modify the behavior of the IDE, you can write scripts that interact with the exposed IDE components.

The primary exposed component is an object called IDE. It is an IDEApplication object.

While the IDE object is an instance of the IDEApplication class, it has capabilities far beyond those of a normal instantiation of IDEApplication. Many additional properties and methods have been added to the IDE which greatly increase its power and flexibility. Since these additional features are implemented in cScript, you are free to use and exploit these additional capabilities. To learn about these capabilities, and the syntax to use them, study the cScript source code for the IDE in the SCRIPT subdirectory.

Manipulating the keyboard

You can access keyboard features through a keyboard manager, implemented by the global KeyboardManager object. The keyboard manager manipulates *Keyboard* objects (instantiations of the class Keyboard).

KeyboardManager manages individual component keyboards, such as that of the editor, the project view, and various other subsystems. This implementation allows support of BRIEF functionality through script simulation without predefined classes for each of the individual IDE components. Each component has a defineable keyboard. The desktop has a keyboard assignment that acts as a global assignment. If a key isn't found in the local keyboard, the desktop keyboard is searched. If the key assignment isn't in the desktop's keyboard, the default internal mapping is used.

The keyboard manager operates on the assumption of a *set context*. A derived class is used in a call to *SetContext()* to specify the current object to be used as a local scope. Since different macros may mean different things to different components, this mechanism provides a simple, straightforward approach to localizing functionality. For example, classes *A* and *B* both have a member function called *Search()*. If class *A* is the current context, class *A's Search()* member is called. The same goes with class *B*. If no context is set, then a global *Search()* function is accessed.

Another keyboard manager responsibility is key recording. A call to the *StartRecord* and *StopRecord* members populates a *Record* object with key sequences. An unlimited number of *Record* objects can be named and iterated. Because *Record* objects are constructed outside the context of the keyboard manager and may be built programatically, recordings can be saved to disk and restored, and keyboard sequences can be simulated through script.

The IDE object contains a *ReadOnly* member that holds the value of the *KeyboardManager*. New script instances may be created; however, the will all reference the same internal data and changes to one will be reflected in all.

Manipulating the IDE editor

The *IDE* editor's functionality is accessible at a low enough level that you can mimic in script the behavior of popular editors (such a BRIEF, Epsilon, vi, and WordStar). The editor itself is accessed through an object instantiated from the Editor class. Because the IDE instantiates an *Editor* object itself, any *Editor* objects you instantiate points to this internal IDE object; therefore, modifications in one *Editor* object's options are reflected in all Editor objects.

Further editor access is provided through the following classes:

BufferOptions Controls characteristics of the *EditBuffer*, such as margin, tab rack, syntax highlighting, and bookmarks.

EditBlock Cut, copy, delete, dimensions, and style.

EditBuffer	Access status, save, describe, time/date stamp.
EditOptions	Holds characteristics of a global nature, such as the insert/overtype setting, optimal fill, and scrap settings (how to handle blocks cut or copied from Editor buffers).
EditPosition	Location-dependent operations in a view or buffer: cursor movement, text rip, search, insert.
EditStyle	Provide named styles that override settings in a buffer or the entire editor.
EditView	Access to buffer, visual cursor manipulations, zoom.
EditWindow	Pane control, access to views.

Chapter 3

cScript Language Reference

cScript keywords and functions

Keywords are reserved for use in the cScript language and cannot be used as names of variables, methods, or classes or as any other identifier names.

array cScript

Use this keyword to declare an array.

Syntax 1

```
array_var = new array[ [size] ];
 array array_var[ [size] ];
```

size The number of elements in the bounded array. If *size* is omitted, the array is associative.

Syntax 2

```
array array_var[ [size] ] = {element1[, element2[, ...]] };
```

size An array created with this syntax always takes the number of elements in the declaration list. *size* is ignored.

element1... Creates a bounded array with contents *element1*, *element2*, and so on. Element numbering starts at 0 and continues to *size* - 1. The number of elements determines the size of the array and overrides *size* if it is specified.

Description

In cScript, you can create two types of arrays, bounded and associative. Bounded arrays are similar to C++ arrays. As in C++, they use a zero-based index. (The first element is 0 and the last is *size* -1.) Associative arrays are grown as necessary. If you assign more members to a bounded array than its *size*, the rest of the array becomes an associative array. If you create an array with a list of elements (Syntax 2), it is a bounded array and its size is the number of elements. Arrays can contain data of any cScript type, including objects and other arrays. An array with other arrays as elements is multi-dimensional. Elements of the contained arrays are accessed using additional sets of square brackets as shown in the example below.

Example

```
// Creates a bounded array of 10 elements
declare myArray;
myArray = new array[10];
myArray[1] = "Hello";
myArray[2] = "World";
print myArray[0], myArray[1];      // prints "Hello World"

// Creates an associative array
```

```
declare myAssocArray;
myAssocArray = new array[]         // no size declared
myAssocArray["Element1"] = "One";
myAssocArray["Element2"] = "Two";
print myAssocArray["Element2"]     // prints "Two"

// Creates a multidimensional array
declare array multiArray[] = {{1,2,3}, myArray, myAssocArray};
print multiArray[0][2], multiArray[1][0], multiArray[2]["Element2"];
// Prints: 3 Hello Two
```

attach — cScript

Use **attach** to link a method of an instance of one class to a method of another instance.

Syntax

```
attach ClassInst1:>method1 to ClassInst2:>method2
```

break — cScript

Use the **break** statement within **do**, **while**, for, and **iterate** loops, or within a **switch** construct, to pass control to the first statement following the innermost enclosing brace. The implementation of **break** in cScript is identical to the implementation in C++.

Syntax

```
break;
```

breakpoint — cScript

Syntax

```
breakpoint;
```

Description

Use the **breakpoint** statement to stop the program and pass control to the script debugger. If the debugger is not active, **breakpoint** is ignored.

call — cScript

This function directly invokes a closure.

Syntax

```
call ClosureName(argumentList);
```

Description

call permits you to directly execute a closure. The closure is invoked using the same arguments as the method normally uses. There is no method for obtaining a return value when calling through closures. If the method returns a value, it will be ignored.

Example

```
// Shows creating a closure and assigning it to a
// variable, then calling the closure directly.

Class MyClass {
   method1(p1, p2)
   {
```

```
      print p1, p2;
   }
};

declare MyClass instance;
declare closure = instance:>method1; // declare the closure
call closure("Hello", "world");      // output is Hello world
```

case — cScript

Use the **case** statement in a **switch** statement to determine which statements execute.

Syntax

```
switch ( switch_expression ){
  case expression :
    [statement1;]
    [statement2;]
    ...
    [break;]
  [default :
    [statement1;]
    [statement2;]
    ...]
}
```

switch_expression Any valid cScript expression, including a function call. Unlike C++, the *switch_expression* is evaluated for each **case** in a top-down fashion until a match is found or no more **case** statements remain.

expression Any valid cScript expression, including a function call.

statement One or more statements to execute.

Description

A **case** statement is the branch condition of a **switch** statement. If the value of the *expression* following **case** matches the value of *switch_expression*, the statements up to the next **break** or the end of the **switch** execute. Note that because cScript is a late-bound language, *expression* does not have to be a literal as in C++, nor does the expression have to be numeric. Other than this difference, **case** behaves exactly as it does in C++.

class — cScript

Use the **class** keyword to define a cScript class.

Syntax

```
class className [(initializationList)]
   [:baseClassName[(init_expression_list)]] { member_list}[;]
```

className The name of the class. className - any name unique within its scope

baseClassName The class that this class derives from (optional). **of** is a synonym for the : separator preceding this identifier.

initializationList The initial constructor values for the class, if any.

initExpressionList The initialization for the class instance.

memberList Declarations of the class's properties, methods, and events.

Description

A class declaration in cScript is similar to a class declaration in C++, with a few key differences.

There are no C++-like constructors in cScript. (Defining a method with the same name as the class, as you do in C++, doesn't make it a constructor.) Instead, executable statements embedded in the class declaration that is not part of a method declaration is considered constructor code. For this reason, initialization parameters must be defined in the class declaration. The base class is always initialized first, before the child class.

Only one base class can be initialized in a derived class declaration since cScript does not support multiple inheritance. Where a class is defined as being derived from a base class and the base class requires initialization values, they must be passed to the base class through the derived class's declaration. The base class initializer is essentially an implicit constructor call, and as such, expressions are allowed for its arguments.

When instantiated, the number and type of initializers is not checked (that is, function overloading is not supported). Arguments are padded and/or truncated the same as they are with functions.

Methods must be defined entirely in the class declaration. You can't just declare a member function in a class and then define it later in the program. All properties and methods of the class are public.

Destructors in cScript work as they do in C++. (Defining a method that starts with a tilde (~) and has the same name as the class makes it a destructor.) Destructors are called when the object is being destroyed. Destructors may not have parameters.

Where inheritance is used, the access method for base class members is the same as for those of the derived class. However, if a derived class member has the same name as one of the base class, you must use **super** must be used to clearly specify the reference.

A class cannot be instantiated as part of its declaration as in traditional C structs, so a semicolon is optional at the end of the declaration.

Example

```
//The following class is declared without parameters:
class noParams{
  declare aMember;
  declare anotherMember;
  Func1();         // constructor code
  for (y = 1; y < 10; y++) // more constructor code
   print "hello";
  ~noParams(){
   print "A noParams has been destroyed.";
  }
}
// The following class is declared with parameters:
class Base(parmOne, parmTwo){
  declare X = parmOne;  // a member variable
  declare Y = parmTwo;  // a member variable
  MethodOne(){
   X = X + Y;
  }
  AnotherMethod(){
  }
}
```

```
// The following class is inherited from the class Base:
// aParm and cParm are passed through to
// Base as parmOne and parmTwo.
class Derived(aParm, bParm, cParm) : Base(aParm, cParm) {
  declare Z = bParm;
}
// example using the Derived class:
declare obj = new Derived(1, 2, 3)  // 1&3 passed to Base
                                    // Base constructed before
```

continue — cScript

Syntax

```
continue;
```

Description

Use the **continue** statement within **do**, **while**, **for** and **iterate** loops to pass control to the end of the innermost enclosing brace, allowing the loop to skip intervening statements and re-evaluate the loop condition immediately. The behavior of **continue** in cScript is identical to C++.

declare — cScript

Use **declare** to ensure that a variable is local to the current scope and does not override a variable from an enclosing scope.

Syntax

```
declare identifier [optional identifier_syntax][, identifier...];
```

Description

The scope of a variable is the block in which it is first used in and any blocks nested in that block. If you are in a nested block, it is possible that a variable you think you are using for the first time has already been used in the enclosing block. What happens in that case is that you override the enclosing block's variable value (and possibly its type as well) with what you mistakenly think is a local variable. To ensure that this side effect doesn't occur, use **declare** with any variables you intend to be local to a block. Although not needed, **declare** can also be used in conjunction with the **export** and **import** declarators. Although multiple variables, objects, and arrays can be declared in a single statement, they cannot be mixed in the same statement.

default — cScript

This keyword indicates the statements to process in a **switch** if none of the **case** conditions apply. It is optional. If you include a **default**, it must be the last condition in the **switch**. If you do not include a **default** statement and none of the **case** conditions apply, none of the statements in the **switch** are executed. The behavior of **default** in cScript is the same as C++.

Syntax

```
switch ( switch_expression ){
  case expression :
    [statement_list;]
    ...
    [break;]
```

```
  [default :
    [statement_list;]
    ...]
}
```

delete cScript

This command deletes an object or array and causes an object's destructor, if any, to be called. Deleing an array does *not* require "[]" in the delete command, as it does in C++. Unlike C++, cScript has automatic garbage collection. Therefore objects are automatically deleted when there are no longer any references to them, or when they go out of scope. Use **delete** only when you need to explicitly deallocate an object before the references to that object have been destroyed.

Syntax

```
delete object_name;
```

object_name The name of the object to delete.

Description

Unlike C++, cScript has automatic garbage collection. Therefore, objects are automatically deleted when there are no longer any references to them, or when they go out of scope. Use delete only when you need to explicitly deallocate an object before the references to that object have been destroyed.

detach cScript

Use **detach** to detach a method instance of one class from a method instance of the same or another class when the two were previously linked using **attach**.

Syntax

```
detach ClassInst1:>method1 from ClassInst2:>method2
```

Description

If you want to make dynamic changes to class instances, you can set up dynamic **on** handlers using the closure operator with **attach**. This technique allows you to supply an alternative implementation for an instance method. In other words, you can override an object's method and provide an alternate implementation of that method at run-time, without affecting the class from which the object was instantiated. The override remains in effect for the lifetime of the object or until the link is broken using **detach**.

do cScript

The **do** statement executes until condition becomes FALSE. Since the condition tests after each the loop executes statement, the loop executes at least once. The behavior of **do** in cScript is the same as C++. **break** will cause loop execution to be terminated, while **continue** will cause the *condition* to be evaluated immediately without any intervening statements being executed.

Syntax

```
do statement while ( condition );
```

Description

statement Executed repeatedly as long as the value of *condition* remains **TRUE**.

export cScript

To make it possible to access a variable across modules, it must be declared **export** in the module that declares it and **import** in another module that needs access to it.

Syntax

```
export variable_name;
```

Description

Variables created at the module level (not in a function, method, class, or control structure) are global variables of the module, but are not accessible to any other modules. To access module scope variables defined in module A from module B, three things must occur:

- Both module A and module B must be loaded.
- The module scope (global) variable must be declared **export** in Module A.
- Module B must contain an **import** statement for the variable.

Example

```
// Example of export and import
// FILE1.SPP
export myExVar;   // export variable for use in other modules
myLocal = 10;
myExVar = 10;

// FILE2.SPP
import myExVar;   // import variable exported by another module
print myLocal;    // prints [UNINITIALIZED]
print myExVar;    // prints 10
```

for cScript

The **for** statement executes until *condition* becomes **FALSE**. Since the condition tests before each the loop executes *statement*, the loop may never execute. The behavior of **for** in cScript is the same as C++.

Syntax

```
for ( [initialization] ; [condition] ; [expression] ) statement
```

Description

The cScript **for** statement works the same as a C++ **for** statement. The *statement* is executed repeatedly until **condition** is **FALSE**. Before the first iteration of the loop, *initialization* initializes variables for the loop. After each iteration of the loop, *expression* executes (most commonly to increment or decrement the initialization variable in some way). *initialization* can be an expression or a declaration.

The scope of any identifier declared within the for loop extends to the end of the script module.

All the expressions are optional. If *condition* is left out, it is assumed to be always **TRUE**. **break** will cause loop execution to be terminated, while **continue** will cause the *condition* to be evaluated immediately without any intervening statements being executed.

from cScript

from is used in a detach statement or when instantiating nested classes.

Syntax 1

```
innerObject = new Inner from ClassInstance;
```

Inner	The nested class.
ClassInstance	Instance of the enclosing class.

Syntax 2

```
detach ClassInst1:>method1 from ClassInst2:>method2;
```

if cScript

Syntax 1

```
if ( condition )  statement;
```

Syntax 2

```
if ( condition )  statement;
   else  statement2;
```

Description

if works exactly as it does in C++. Use it to implement a conditional statement. The *condition* statement must evaluate to **TRUE** or **FALSE**.

When *condition* evaluates to **TRUE**, *statement1* executes. If *condition* is **FALSE**, *statement2* executes. *statement2* can be another **if** statement.

The **else** keyword is optional. If you use nested **if** statements, any **else** statement is associated with the closest preceding **if** unless you force association with braces.

import cScript

To make it possible to access a variable across modules, the variable must be declared **export** in the module that declares it and **import** in the module that requires access to it.

Syntax

```
import variableName;
```

Description

Variables created at the module level (not in a function, method, class, or control structure) have module scope. They are not accessible to any other modules. To access a variable defined in module A from module B, three things must occur:

- Both module A and module B must be loaded.
- The variable must be declared **export** in Module A.
- Module B must contain an **import** statement for the variable.

initialized cScript

This intrinsic function indicates if a variable has ever been initialized. It provides a means for determining the state of a variable before using it. Using an uninitialized variable is rarely dangerous (as in C++), but is also not usually what is intended. It is

particularly useful in determining the state of arguments passed to a function, and in class instantiation. It can also be useful to prevent divide by zero errors.

Syntax

```
initialized(x);
```

Return values

TRUE if the value has ever been initialized, **FALSE** otherwise.

Example

```
// Example of initialized
declare x, y;  // declares variables,
               // but does not initialize them!
x = 10;        // initialized!
print initialized(x);   // returns TRUE
print initialized(y);   // returns FALSE
```

iterate — cScript

Use an **iterate** loop to cycle through the members of a class object or an associative array in first to last order.

Syntax

```
iterate(outputvar; object[;keyvar]) [statement];
```

outputvar	A variable to hold a copy of the contents of the array or class data member.
object	The array or class object to iterate.
keyvar	Variable to hold the index or key into the array, or class object data member name.

Description

iterate is a loop structure that allows some action to be performed on each member of the array or property of a class object, such as printing it out. You can use **continue** and **break** to control execution inside the loop. Like a **for** loop, curly braces ({}) must be used to enclose multiple loop statements. **iterate** can also be used to determine the number of properties in an object or the number of elements in an array.

Examples

To print all the members of associative array z using the variable x,

```
iterate( x; z ) {
 print x;
}
```

To print all the members and the key values of associative array z using the variable x,

```
iterate( x; z; k ) {
 print "Key = " + k + "Value = " + x;
}
```

load — cScript

This function opens and parses the specified script file which can be subsequently executed using **run()**. Although classes and functions defined in the module come into

existence when the module is loaded, variables declared in the module are not defined nor are any other statements executed until the script is **run()**. If there is an **_init()** function, the module executes that code first. If there is a function with the same name as the module, that function is then executed.

Syntax

```
moduleHandle = load(filename);
```

filename A string, the module to load.

Return value

A module handle (module object reference) if successful, or **NULL** if not.

Example

```
declare myModule;
myModule = load("demo.spp");  // loads module and gets a handle
if (myModule) {               // if loaded
  run(myModule);              // run the module
unload(myModule);             // unloads the module
}
```

module command cScript

Use *module* to provide an alternative internal name, or alias, for a module.

Syntax

```
module ["newname"];
```

Return value

None

Description

After being parsed, every script file loaded into the IDE is assigned a module name. The name defaults to the file name without its path or file extension. This name may be used by other modules to explicitly access functionality in the module. You can alter a module's name by embedding `module "newname";` anywhere in the file.

module function cScript

Use *module()* to get access to any loaded module.

Syntax

```
module (["modulename"]);
```

Return value

A reference to an object, the module handle associated with the named module, or to the current module if no *moduleName* is specified. If a *moduleName* is specified and no matching module is found, it returns NULL.

Description

Use *module()* to get access to any loaded module. If you use it with the current module, *moduleName* has the same value as this used at the module level.

One use for this function is to access a globally scoped variable from a local scope. For example,

```
// Modtest.spp
module "modtest";
declare x = 1;
declare ModRef = this;
local x = 2;
print (module()).x; // prints 1
print ModRef.x;     // prints 1
```

new cScript

Use **new** to create a new object or array.

Syntax 1

```
objectname = new className[([initializerList])]
            [from outerClassName[([initializerList])]]
```

Syntax 2

```
arrayname = new array [[arraysize]];
```

Description

Use **new** as an alternate syntax for creating new class objects or arrays. See class, array, and **declare**.

Unlike C++, cScript does not distinguish between static and dynamic memory allocation. The difference between the standard declaration syntax and that using **new** is syntactic only. cScript has automatic garbage collection. Therefore, objects created with **new**, or otherwise, are automatically deleted when there are no longer references to them. Use **delete** only when you need to explicitly destroy an object before the references to that object have been destroyed.

of cScript

This keyword is a synonym for the colon (:) separator used when defining a class that derives from a base class.

Syntax

```
class classname [ (initialization_list) ]
   [of baseclass[ (initialization_list) ]] { member_list }
```

on cScript

Use **on** to set up a dynamic object method call event handler, or an object read-property getter, or a write-property setter.

Syntax 1

```
on ClassInstance:>{xe ">"}Method([argumentList]){
      [pre-processing statement(s)]
      [pass([argumentList]);]
      [post-processing statement(s)]
      [return value;]
}
```

This syntax is for an object method call event handler. This form of dynamic event handling allows processing to occur both before and after the optional call, through **pass()**, to the hooked method, and allows alternate values to be both passed to the hooked method and returned by the event handler.

In order to be bound to an existing object method, the number of parameters in an **on** handler definition must match the hooked method. Once invoked, **pass()** will call the hooked method regardless of how many arguments it passes. As with all function calls, cScript will insure that the proper number of arguments are passed, truncating or padding as needed.

Syntax 2

```
on ClassInstance:>property{
      [pre-processing statement(s)]
      return [pass() | value];
}
```

Where `pass()` returns the actual value, or, alternatively, any specified value. This syntax is used for a property getter and would be triggered by any subsequent statement that references that object's property for read access, such as on the right hand side of an assignment statement.

Syntax 3

```
on ClassInstance:>property(parameter){
      [pre-processing statement(s)]
      [pass(parameter | value);]
      [post-processing statement(s)]
}
```

This syntax is used for a property setter. The setter is triggered when the object's property is used as an lvalue, such as on the left hand side of an assignment statement. The value to be assigned to the property is what is passed to the setter as its parameter. The value passed in `pass()` sets the value of the property.

Description

Use object method call event handlers (also referred as "**on** handlers") to create new methods, or redefine existing methods, on an object of a given class. Unlike **attach**, methods overridden with **on** cannot be detached. To call the original method from within the overridden version with the same name, invoke the **pass()** function. If the global reference variable *selection* has been set using **select**, its reference will not be affected, but is superseded with the **with** block.

Example

```
import editor;
// create a new Debugger object called debug
declare debug = new Debugger();

// create a new method called RunToCurrent()
// on the object debug (not the class!)
on debug:>RunToCurrent()
{
   declare fileName = editor.TopBuffer.FullName;
   declare row = editor.TopBuffer.TopView.Position.Row;
   .RunToFileLine(fileName, row);
```

```
}
```

onerror cScript

Use **onerror** to set up an error handler.

Syntax 1

```
onerror Method(argumentList) from errorHandlerObject
   {[error_handler_code][resumeLabel]}
```

Syntax 2

```
onerror errorHandlerObject.>Method(argumentList)
   {[error_handler_code][resumeLabel]}
```

pass cScript

Use **pass** in an on handler to invoke the original function that is being overridden.

Syntax

```
varname = pass([param1[,param2[,...]]);
```

print cScript

This function prints the expression that is passes to it in the Script tab page of the IDE Message window. (Choose View | Messages to display the window, then click the Script tab.) If nothing is passed to it, it does nothing.

Syntax

```
print [expression_list];
```

Description

The **print** function takes any string, expression, or variable as a parameter. To concatenate expressions, separate them with commas. For example:

```
print "hello world";
print "the number is", x;
print "My name is", name, "and I'm", years, "years old";
```

A space is printed for each comma in the expression list:

- An uninitialized value outputs [UNINITIALIZED].
- A variable initialized to NULL outputs [NULL].
- An object outputs [OBJECT].

printf cScript

This function prints the string and integer expressions using the format string to the Script tab of the IDE message window (using the internal print function).

Syntax

```
printf( formatString, arg1, arg2, …, arg 10);
```

Description

This function is similar to the C run-time library function of the same name but only %c, %d, %i, %s, %u, %x, %X format specifiers are allowed. Optional flags and width fields are supported. This function is defined in the file MISCSCR.SPP.

Examples

```
printf( "Port: %05X Data: %04X", port, data) ;
printf( "The current Paradigm C++ caption is %s", IDE.Caption) ;
```

reload — cScript

This function does an unload followed by a load. It searches the module list for a matching module. If found, it removes it and then loads it again. If it doesn't find a module to unload, it simply loads the module for the first time.

If there are references to global objects in the module when it is reloaded, these references continue to refer to the older objects. (The module is not destroyed, but is stored to maintain these references.) Global module values that are not part of an object are destroyed and then reloaded.

Syntax

```
reload (moduleName);
```

Return value

A module handle if successful or **NULL** if not.

resume — cScript

Use **resume** to exit from the current onerror statement and jump to a labeled location in the same module.

Syntax

```
resume label;
```

return — cScript

Use **return** to exit from the current function, **on** handler, or module, optionally returning a value. A module, by default, returns **TRUE** if successfully run. However, an explicit **return** statement can be provided to return a customized return value, or simply to terminate execution prior to the end of the script.

Syntax

```
return [ expression ];
```

Example

```
sqr(x)
{
  return (x*x);
}
```

run — cScript

This function loads and runs the module indicated, or simply runs it if it is already loaded. The module remains loaded until explicitly unloaded using **unload()**.

Syntax

```
run (moduleName)
```

moduleName The string or a module handle.

Return value

By default, **run** returns **TRUE** if successful or **FALSE** if not. If the module has a global **return** statement, **run** returns that value if the module successfully runs and then displays a warning that the standard return value for **run** has been overridden.

select cScript

This command creates a special global variable, selection, that refers to the selected variable. Because the variable is global to all loaded scripts, only one **selection** can be active in an IDE session at a time.

Syntax

```
select objectName;
```

Description

You can call **select** on any variable that is loaded in any script. Doing so sets **selection** to reference that variable for all scripts in the session. You then have access to that object from any script by using the alias **selection** as the name of the variable. Variables so selected can also be referenced using the shorthand dot (".") notation. If you call **select** and there is already a **selection**, you override the current **selection** with your new one.

selection cScript

This keyword is a special global reference variable created by calling select on a variable. Because the selection variable is global to all loaded scripts, only one **selection** can be active in an IDE session at a time.

Syntax

Selection can be used in the same way as any other variable.

Description

Once the **selection** has been made, you can use **selection** in any way that you normally use the variable it refers to. You can access members of the referenced object with **selection**.*member*. The dot (".") shorthand syntax can also be used instead of **selection** outside a **with** or **iterate** block or an **on** handler. In those situations, the dot has local context and refers to the controlling variable for that block (usually an object).

Example

```
// SELECT1.SPP
class C0 (p1, p2, p3) {
   declare v1 = p1;
   declare v2 = p2;
   declare v3 = p3;
}
class C1 (p1, p2, p3) {
   declare v1 = p1;
   declare v2 = p2;
   declare v3 = p3;
}
declare C0 obj1("One", "Two", "Three");
declare C1 obj2(1, 2, 3);

// Select the first object
```

```
select obj1;

// Iterate across the selected object
// using selection, then dot notation.
iterate(iterator; selection; key)
   print typeid(selection), "property", key, "=", iterator;

iterate(iterator; . ; key)
   print typeid(.), "property", key, "=", iterator;

// Note that the dot within the with
// block refers to its own local selection.
with(obj2)
   iterate(iterator; . ; key)
      print typeid(.), "property", key, "=", iterator;

// But the global selection has not changed.
print .v1;
print selection.v2;
print ". and selection still refer to", typeid(.);
```

strtol — cScript

This function parses the supplied string in the specified radix and returns an integer result.

Syntax

```
strtol(numericString[ , radix ]);
```

Description

This function is similar to the C run-time library function of the same name. If no radix is specified, hexadecimal (base 16) is used. This function is defined in the file MISCSCR.SPP.

strtoul — cScript

This function parses the supplied string in the specified radix and returns an unsigned result.

Syntax

```
strtoul(numericString[ , radix ]);
```

Description

This function is similar to the C run-time library function of the same name. If no radix is specified, hexadecimal (base 16) is used. This function is defined in the file MISCSCR.SPP.

super — cScript

This function gives you access to a member of the base class with the same name as a member of a derived class. Base class members can be directly accessed without using **super** where the member name is unique within the class definition.

Syntax

objectName.super[.super...].*member*

Description

cScript doesn't support function overloading or the :: operator. However, you can use **super** to get access to overridden class members as follows:

```
class C1 {
  declare x = "C1";
  Method1() {
   print x;
  }
 }
class C2:C1 {
  Method1() {
   print "C2 derived from ", x;
  }
}
MyObj = new C2;
MyObj.Method1();       //Prints C2 derived from C1
MyObj.super.Method1(); //Prints C1
```

If a base class is itself a derived class and you want to access one of its overridden members, use **super.super** (and so on for further access up the inheritance hierarchy). For example,

```
class C3:C2 {
  Method1() {
   print "C3 derived from C2";
  }
}
MyObj3 = new C3;
MyObj3.Method1();              //Prints C3 derived from C2
MyObj3.super.Method1();        //Prints C2 derived from C1
dMyObj3.super.super.Method1(); //Prints C1
class C3:C2 {
  Method1() {
    print "C3 derived from C2";
  }
}
MyObj3 = new C3;
MyObj3.Method1();             //Prints C3 derived from C2
MyObj3.super.Method1();       //Prints C2 derived from C1
MyObj3.super.super.Method1(); //Prints C1
```

switch cScript

Use a switch statement to choose one of several alternatives. If *switch_expression* matches one of the cases, that case's statements execute. If you don't use **break** as the last statement in the case that executes, all the remaining statements (except **case** or **default**) in the **switch** execute until either a break is encountered or the end of the **switch** is reached. If you do use a **break** that executes, the switch statement ends there.

Syntax

```
switch (switch_expression){
  case expression :
    [statement1;]
    [statement2;]
    ...
    [break;]
  [default :
    [statement1;]
```

```
    [statement2;]
    ...]
}
```

switch_expression Any valid cScript expression, including a function call. Unlike C++, the *switch_expression* is evaluated for each **case** in a top-down fashion until a match is found or no more **case** statements remain.

expression Any valid cScript expression, including a function call.

Description

The value of the *switch_variable* is checked against the value of each **case** *expression* until a match is found or until either **default** or the end of the **switch** statement is reached. As in C++, all statements but **case** or **default** following the matching **case** are executed until **break** or the end of the **switch** statement is reached. If no **case** *expression* matches *switch_variable*, the statements following **default**, if any, are executed.

If you insert a **default** case, it must be the last case.

this cScript

Description

The cScript **this** keyword is analogous to the C++ **this** pointer. It is used to provide an object reference within a class definition. It is primarily needed to define closures used in event handlers that will apply to all instances of that class. For example, given the class definition:

```
class MyClass {
  method1() {}
  on this:>method1() {}
}
```

All objects of that class will have a default method call event "**on**" defined (rather than on a per-instance basis as when the **on** handler is defined outside of the class).

Since a script module can actually be treated as an object, **this** when used outside of a class definition refers to the current module object. You can use it to create an event handler for a global function. For example,

```
DoNothing (){}                      //Globally scoped function
on this:>DoNothing() {              //method of current object
 print "Did something else first";
 pass();
}
```

Calls to module scope functions for which an event handler has been defined will only trigger the handler when they are called in the same way as defined in the **on** handler. For example:

```
this.DoNothing();  // Triggers the event handler
DoNothing();       // Does not trigger an event
```

typeid cScript

Use **typeid** to get run-time identification of variables or the resulting value of expressions.

Syntax

```
typeid(name_expn);
```

name_expn Any legal variable name or expression

Return value

A string representing the type. Possible return values are:

[ARRAY]
classname
[CLOSURE]
[INTEGER]
[NULL]
[REAL]
[STRING]
[UNINITIALIZED]

Description

If the variable or expression value is a built-in type, **typeid** returns the type in brackets [ ]. If it is an object, **typeid** returns the class name. If the expression is a function or method, **typeid**() indicates the type of the return value of the function.

unload cScript

This function searches the module list for a matching module. If found, it removes it, causing all functions, classes, and local variables that were defined in it to become invalid. However, if an object within the script is referenced from another active script (for example where a function in the unloaded script returned a reference to an object), that object will not be destroyed.

Syntax

```
unload (moduleName);
```

moduleName A string or module handle.

Return value

TRUE if successful, otherwise **FALSE**.

while cScript

Use a **while** loop to repeat one or more statements until *condition* is **FALSE**.

Syntax

```
while [( condition )] [{statement_list}]
```

Description

If no condition is specified, the **while** clause is equivalent to **while**(**TRUE**). Because the test takes place before any statements execute, if *condition* evaluates to **FALSE** on the

first pass, the loop does not execute. **break** will cause loop execution to be terminated, while **continue** will cause the *condition* to be evaluated immediately without any intervening statements being executed.

Example

```
i = 0
while (p[i] < 50) {
  p[i] += 10;
  i += 1;
}
```

with cScript

Use **with** to create a shorthand reference to a variable. This is particularly useful when the variable is a deeply nested object.

Syntax

```
with (variable){member_list}
```

Description

For example, assume an object `z`, which is contained within an object `y`, which is contained within an object `x`. Access to `z`'s members can be cumbersome. For example,

```
x.y.z.DoSomething();
x.y.z.DoSomethingElse();
x.y.z.NowDoThis();
```

You can decrease syntactical complexity by assigning `x.y.z` to another variable. For example,

```
p = x.y.z;              // Assignment lookup
p.DoSomething();        // 1 lookup
p.DoSomethingElse();    // 1 lookup
p.NowDoThis();          // 1 lookup
```

If you use **with**, referencing can be made even simpler:

```
with (x.y.z){           // 1 lookup
  .DoSomething();       // No lookup
  .DoSomethingElse();   // No lookup
  .NowDoThis();         // No lookup
}
```

Scoping of **with** statements in functions is handled as you would expect: the scope is local to the current function and the correct member gets called. For example,

```
WFunc1(){
  with (x.y.z){
   .DoSomething();
  }
}

WFunc2(){
  with (MyClass){
   Wfunc1();            // WFunc1 calls x.y.z.DoSomething()
   .Func2();            // This call is to MyClass.Func2()
  }
}
```

Using the dot operator in a **with** block refers to the current **with** assignment. If the global reference variable *selection* has been set using **select**, its reference will not be affected, but is superseded with the **with** block.

yield cScript

Syntax

```
yield;
```

yield forces cScript to check if the abort (Escape) key has been pressed. Imbedding **yield** in a time consuming process, such as a loop that executes many times, allows the user to be able to break out of the process if desired.

Return value

None

About cScript operators

Operators are tokens that trigger some computation when applied to variables and other objects in an expression. cScript uses many of the C++ operators. For the most part, these operators have the same precedence, associativity, and functionality as in C++.

Because cScript has no structs, unions, or references to memory locations, the following C++ operators do not exist in cScript:

```
->  *  ->*  .*
```

For the same reason, the & operator can be used only to declare function parameters as pass-by-reference parameters (not to dereference variables).

Additionally, cScript does not provide the following C++ operators:

```
::  sizeof  const_cast  reinterpret_cast
```

cScript does provide two new operators:

:>	The closure operator,cScr_closure_op typically used in oncScr_on statements to override functions
??	The in operator,cScr_QmarkQmark_Op used to test members of arrays and classes

The following list groups the operators by type (the first item shows all operators ordered by precedence):

Table of Operator Precedence
Arithmetic
Assignment
Bitwise
Comma
Conditional
Logical
Object-oriented
Enclosing
Preprocessor
Relational

Depending on context, the same operator can have more than one meaning. For example, the minus (-) can be interpreted as:

- subtraction (x - y)
- a unary negative (-y)

No spaces are allowed in compound operators (such as :>). Spaces change the meaning of the operator and will generate an error.

cScript precedence of operators

Operators on the same line have equal precedence.

Operators	Associativity
(),[]	left to right
.	left to right
:>, ??	left to right
!, ~, +, -, ++, --, &	right to left
*, /, %	left to right
+, -	left to right
<<, >>	left to right
<, <=, >, >=	left to right
= =, ! =	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
| |	left to right
?:	left to right
=, *=, /=, %=, +=, -=, &=, ^=, |=, <<=, >>=	right to left
,	left to right

Binary operators cScript

The binary cScript operators are as follows:

Arithmetic	+	Binary plus (add)
	-	Binary minus (subtract)
	*	Multiply
	/	Divide
	%	Remainder (modulus)
Bitwisec	<<	Shift left
	>>	Shift right
	&	Bitwise AND
	^	Bitwise XOR (exclusive OR)
	|	Bitwise inclusive OR
Logical	&&	Logical AND
	| |	Logical OR
Assignment	=	Assignment
	*=	Assign product

	/=	Assign quotient
	%=	Assign remainder (modulus)
	+=	Assign sum
	-=	Assign difference
	<<=	Assign left shift
	>>=	Assign right shift
	&=	Assign bitwise AND
	^=	Assign bitwise XOR
	| =	Assign bitwise OR
Relational	<	Less than
	>	Greater than
	<=	Less than or equal to
	>=	Greater than or equal to
	= =	Equal to
	!=	Not equal to
Conditional	? :	Actually a ternary operator
	a ? x : y	"if a then x else y"
Comma	,	Evaluate

Arithmetic operators cScript

The arithmetic operators are

```
+  -  *  /  %  ++  --
```

Syntax

```
+ expression
- expression
expression1 + expression2
expression1 - expression2
expression1 * expression2
expression1 / expression2
expression1 % expression2
postfix-expression ++       (postincrement)
++ unary-expression         (preincrement)
postfix-expression --       (postdecrement)
-- unary-expression         (predecrement)
```

Description

Use the arithmetic operators to perform mathematical computations. *expression1* determines the type of the result when variables of different types are used.

The unary expressions of + and - assign a positive or negative value to *expression*.

+ (addition), - (subtraction), * (multiplication), and / (division) perform their basic algebraic arithmetic on all data types.

% (modulus operator) returns the remainder of integer division.

++ (increment) adds one to the value of the expression. Postincrement adds one to the value of the expression after it evaluates; while preincrement adds one before it evaluates.

-- (decrement) subtracts one from the value of the expression. Postdecrement subtracts one from the value of the expression after it evaluates; while predecrement subtracts one before it evaluates.

Assignment operators cScript

The assignment operators are

```
=     *=      /=      %=     +=     -=
<<=   >>=     &=      ^=     |=
```

Syntax

unary-expr assignment-op assignment-expr

Description

The = operator is the only simple assignment operator, the others are compound assignment operators.

In the expression E1 = E2, E1 must be a modifiable lvalue. The assignment expression itself is not an lvalue.

The expression

```
E1 op = E2
```

has the same effect as

```
E1 = E1 op E2
```

except the lvalue E1 is evaluated only once. The expression's value is E1 after the expression evaluates.

For example, the following two expressions are equivalent:

```
x += y;
x = x + y;
```

Any assignment can change the cScript native type of the value on the left of the assignment, depending on the type of the value assigned. See "cScript and types" on page 2-19 for more information.

Do not separate compound operators with spaces (+<space>=). Doing so generates errors.

Bitwise operators cScript

Use bitwise operators to modify individual bits of a number rather than the whole number.

Syntax

```
AND-expression & equality-expression
exclusive-OR-expr ^ AND-expression
inclusive-OR-expr exclusive-OR-expression
~expression
shift-expression << additive-expression
shift-expression >> additive-expression
```

Operator	What it does
&	Bitwise AND: compares two bits and generates a 1 result if both bits are 1; otherwise, it returns 0.
|	Bitwise inclusive OR: compares two bits and generates a 1 result if either or both bits are 1; otherwise, it returns 0.
^	Bitwise exclusive OR: compares two bits and generates a 1 result if the bits are complementary; otherwise, it returns 0.
~	Bitwise complement: inverts each bit. (~ is also used to create destructors.)
>>	Bitwise shift right: moves the bits to the right, discards the far right bit and assigns the leftmost bit to 0.
<<	Bitwise shift left: moves the bits to the left, it discards the far left bit and assigns the rightmost bit to 0.

Both operands in a bitwise expression must be of an integral type.

Bit value		Result of &, ^, | operations		
E1	**E2**	**E1 & E2**	**E1 ^ E2**	**E1 | E2**
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	0	1

Reference operator — cScript

In cScript as in C++, the default function calling convention is to pass by value. The reference operator can be applied to parameters in a function definition header to instead pass the argument by reference.

Syntax

```
methodName(&parameter[,...]){statementList}
```

Description

cScript reference types created with the & operator, create aliases for objects and let you pass arguments to functions by reference.

When a variable *x* is passed by reference to a function, the matching formal argument in the function receives an alias for *x*, (similar to an address pointer in C++). Any changes to this alias in the function body are reflected in the value of *x*.

When a variable *x* is passed by value to a function, the matching formal argument in the function receives a copy of *x*. Any changes to this copy within the function body are not reflected in the value of *x* itself. Of course, the function can return a value that could be used later to change *x*, but the function cannot directly alter a parameter passed by value.

The reference operator is only valid when used in function definitions as applied to one or more of its parameters. The address of operator is not supported in cScript as it is in C++, where it can be used to obtain the address of (create a pointer to) a variable.

& Example

```
func1 (i){i=5;}
func2 (&Ir){i=5;}
// Ir is a reference variable
    ...
sum = 3;
func1(sum);     // sum passed by value
print sum;      // Prints 3
func2(sum);     // sum passed by reference
print sum;      // Prints 5
```

sum, passed by reference to *func2*, has its value changed when the function exits. *func1*, on the other hand, gets a copy of the *sum* argument (passed by value), so *sum* itself cannot be altered by *func1*.

Object-oriented operators cScript

The cScript object-oriented operators are:

.	Access a class object member.
:>	Closure operator (binds a class instance and a method as a single closure reference.
??	Tests for the existence of a class object property or array index.

In addition, there is a colon (:) punctuator:

:	Refers to a base class in a derived class declaration.

Closure (:>) operator cScript

Use the closure operator (:>) in an **on** handler, an **attach** statement, or a **detach** statement to bind a class instance with a class member as a single closure reference.

Syntax 1 (on handler)

on *ClassInstance:>Method{[code_to_replace_method_code]}*

Syntax 2 (attach)

attach ClassInst1:>method1 to ClassInst2:>method2;

Syntax 3 (detach)

detach ClassInst1:>method1 from ClassInst2:>method2;

Syntax 4 (getter)

```
on ClassInstance:>property{
  // your code here
  return [pass()|SomeValue];
```

Syntax 5 (setter)

```
on ClassInstance:>property(parm){
 // your code here
 [pass(SomeValue);]
}
```

Syntax 6 (closure variable)

```
declare closureVar = classInstance:>methodName;
```

☞ A closure variable as declared above can subsequently be used wherever a closure is needed. For example, an alternative to the attach statement (Syntax 2) using closure variables would be:

```
declare closureVar1 = classInst1:>method1;
declare closureVar2 = classInst2:>method2;
attach closureVar1 to closureVar2;
```

:> Example

```
import scriptEngine;
import IDE;
...
modList = new ListWindow(50, 5, 100, 300, "Module List",
          TRUE, FALSE, loadedModules);


// Hook the Accept event in order to do nothing.
// Default behavior is to put the list away.
on modList:>Accept(){}
```

Member(.) selector — cScript

Use the selector operator (.) to access class members.

Syntax

```
class-instance.class-member
```

Description

Suppose that the object *s* is of class S and *m* is a property declared in S. The expression:

```
s.m
```

represents the property *m* in *s*.

☞ Although the precedence of the . operator is the same as C++ in most respects, one place where it doesn't work as you would expect is in cScript native function calls that do not use parentheses. For example, `print module "MyModule".Data1` does not print the Data1 member of MyModule. To get this reference to work properly, you must use parentheses with the **module** function, as follows:

```
print module ("MyModule").Data1
```

Example

```
class myClass {
   i = 0;
}
s = new myClass();
s.i = 3;              // assign 3 to the i property of myClass s
```

?? operator — cScript

You can use ?? (read "in") either to test for the existence of an object property or for an array index.

Syntax 1

string-expression | *"string"* ?? *objectname* |*arrayname*

Syntax 2

integer-expression | *integer* ?? *arrayname*

Description

Use a quoted string, or an expression that evaluates to a string, to test for the existence of an object property or an associative array index. Use an integer, or an integer expression, to test for the existence of an index value in a numerically indexed array. For example,

```
class MyClass {
  declare property1 = 0;
  declare property2 = 1;
}

declare MyClass instance;
if ("property1" ?? instance)
  print "property1 is a property of instance.";

declare array a1[];
a1[0] = "a";
a1["Hello"] = 1;
if (0 ?? a1)
  print "Array a1 has an index 0.";
if ("Hello" ?? a1)
  print "Array a1 has an index \"Hello\".";
```

Comma (,) punctuator and operator — cScript

A comma separates elements in a function argument list. It is also used as an operator in comma expressions. Mixing the two uses of comma is legal, but you must use parentheses to distinguish them.

Syntax

```
expression , assignment-expression
```

Remarks

The left operand E1 is evaluated as a void expression, then the right operand E2 is evaluated to give the result and type of the comma expression. By recursion, the expression:

```
E1, E2, ..., En
```

results in the left-to-right evaluation of each Ex, with the value and type of En giving the result of the whole expression.

To avoid ambiguity with the commas in function argument and initializer lists, use parentheses. The following example calls func with three arguments: (i, 5, and k).

```
func(i, (j = 1, j + 4), k);
```

Conditional (?:) operator — cScript

The conditional operator `?:` is a ternary operator used as a shorthand for **if-else** statements.

Syntax

```
logical-OR-expr ? expr : conditional-expr
```

Remarks

This operator allows you to use a shorthand for

```
if (expression)
   statement1;
else
   statement2;
```

In the expression `E1 ? E2 : E3`, *E1* evaluates first. If its value is nonzero (**TRUE**), then *E2* evaluates and *E3* is ignored. If *E1* evaluates to zero (**FALSE**), then *E3* evaluates and *E2* is ignored. The result of the statement is the value of either *E2* or *E3*, depending upon which evaluates.

?: example

If statement:

```
if (x < y)
   z = x;
else
   z = y;
```

Ternary equivalent:

```
z = (x < y) ? x : y;
```

Logical operators — cScript

Use logical operators to evaluate an expression to **TRUE** or **FALSE**.

Syntax

```
logical-AND-expr && inclusive-OR-expression
logical-OR-expr || logical-AND-expression
! expression
```

Description

Operator	Description
&&	Logical AND returns **TRUE** (1) only if both expressions evaluate to a nonzero value; otherwise it returns **FALSE** (0). Unlike C++, if the first expression evaluates to **FALSE**, the second expression is still evaluated.
‖	Logical OR returns **TRUE** (1) if either of the expressions evaluates to a nonzero value; otherwise it returns **FALSE** (0). Unlike C++, if the first expression evaluates to **TRUE**, the second expression is still evaluated.
!	Logical negation returns **TRUE** (1) if the entire expression evaluates to a nonzero value; otherwise it returns **FALSE** (0). The expression *!E* is equivalent to (`0 == E`).

Enclosing operators — cScript

The enclosing operators are parentheses( ), braces { }, and brackets), [ ].

Syntax

```
(expression-list)
function (arg-expression-list)
array-name [expression]
{statement-list}
compound-statement {statement-list}
```

Description

Operator	Description
( )	Use to groups expressions, isolate conditional expressions, or indicate function calls and function parameters.
[ ]	Use to indicate single and multi-dimensional array subscripts.
{ }	Use as the start and end of compound statements and indicate a code block.

Array subscript operator cScript

Syntax
```
[expression-list]
```

Brackets ([ ]) indicate single and multi-dimensional array subscripts. Use brackets to declare an array or to access individual array components:

```
declare myArray = new array [10];
myArray[0] = 5;
myArray[1] = "Cheers";
declare array multiArray[] = {myArray};
print multiArray[0][1]; // prints "Cheers"
```

Parentheses operator cScript

Use parentheses () to

- Group expressions.
- Isolate conditional expressions.
- Indicate function calls and function parameters.

Syntax 1
(*expression-list*)

Description
This syntax groups expressions or isolates conditional expressions.

Syntax 2
postfix-expression (*arg-expression-list*)

arg-expression-list A comma-delimited list of expressions of any type representing the actual (real) function arguments.

Description
This syntax describes a call to the function given by the postfix expression. The value of the function call expression, if it has a value, is determined by the **return** statement in the function definition.

Preprocessor operator cScript

The # (pound sign) indicates a preprocessor directive when it occurs as the first non-whitespace character on a line. It signifies a compiler action not necessarily associated with code generation.

Relational operators cScript

Use relational operators test equality or inequality of expressions.

Syntax

```
equality-expression   == relational-expression
equality-expression   != relational-expression
relational-expression <  shift-expression
relational-expression >  shift-expression
relational-expression <= shift-expression
relational-expression >= shift-expression
```

Description

If the statement evaluates to **TRUE** it returns a nonzero value; otherwise, it returns **FALSE** (0).

Operator	Description
= =	equal
!=	not equal
>	greater than
<	less than
>=	greater than or equal
<=	less than or equal

Unary operators cScript

Syntax

```
unary-operator unary-expression
```

Remarks

cScript provides the following unary operators:

Operator	Description
!	Logical negation
++	Increment
~	Bitwise complement
--	Decrement
-	Unary minus
+	Unary plus

Increment and decrement operators cScript

The increment and decrement operators are ++ and --. They can be used either to change the value of the operand expression before it is evaluated (pre) or change the value of the whole expression after it is evaluated (post). The increment or decrement value is appropriate to the type of the operand.

Syntax 1 (pre)

```
postfix-expression ++       (postincrement)
postfix-expression --       (postdecrement)
```

Description

The value of the whole expression is the value of the postfix expression before the increment or decrement is applied. After the postfix expression is evaluated, the operand is incremented or decremented by 1.

Syntax 2 (post)

```
++ unary-expression          (preincrement)
-- unary-expression          (predecrement)
```

unary-expression The operand, which must be a modifiable lvalue.

Description

The operand is incremented or decremented by 1 before the expression is evaluated. The value of the whole expression is the incremented or decremented value of the operand.

Plus and minus operators — cScript

The plus (+) and minus (-) operators can operate in either a unary or binary fashion on any type of variable.

Syntax 1 (Unary)

```
+ unary-expression
- unary-expression
```

+ *unary-expression* Value of the operand after any required integral promotions.

- *unary-expression* Negative of the value of the operand after any required integral promotions.

Syntax 2 (Binary)

expression1 + *expression2*
expression1 - *expression2*

expression1 This expression determines the type of the result.

expression2 This expression converted if necessary to a type matching *expression1*, and then the operation is carried out.

Multiplicative operators — cScript

Syntax

```
multiplicative-expr * unary-expression
multiplicative-expr / unary-expression
multiplicative-expr % unary-expression
```

Remarks

There are three multiplicative operators:

* (multiplication)
/ (division)
% (modulus or remainder)

The usual type conversions are made on the operands.

`(op1 * op2)` Product of the two operands
`(op1 / op2)` Quotient of the two operands (op1 divided by op2)
`(op1 % op2)` Remainder of the two operands (op1 divided by op2)

For / and %, op2 must be a nonzero value. If op2 is zero, the operation results in an error. Note that division of a number by a string can result in this divide by zero error.

When op1 is an integer, the quotient must be an integer. If the actual quotient would not be an integer, the following rules are used to determine its value:

- If op1 and op2 have the same sign, op1 / op2 is the largest integer less than the true quotient, and op1 % op2 has the sign of op1.
- If op1 and op2 have opposite signs, op1 / op2 is the smallest integer greater than the true quotient, and op1 % op2 has the sign of op1.


Rounding is always toward zero.

Punctuators cScript

The cScript punctuators (also known as separators) are:

( )	Parentheses
{ }	Braces
,	Comma
;	Semicolon
:	Colon
=	Equal sign
#	Pound sign

Most of these punctuators also function as operators.

Braces ({ }) punctuator cScript

Braces ({ }) indicate the start and end of a compound statement.

Semicolon (;) punctuator

The semicolon (;) is a statement terminator.

Any legal cScript expression (including the empty expression) followed by ; is interpreted as a statement. The expression is evaluated and its value is discarded. If the statement has no side effects, cScript can ignore it. Semicolons are often used to create an empty statement.

Colon (:) punctuator cScript

Use the colon when declaring a child class or a class with a label.

Syntax 1

```
class childClass:parentClass
```

Use this version to indicate the parent class when declaring a child class. For an example of this syntax, see the **class** keyword.

Syntax 2

```
case expression:
```

Use this version to indicate the end of a case expression. For example,

```
switch (a) {
   case 1:
      print "One";
      break;
   case 2:
      print "Two";
      break;
   default: print "None of the above!";
}
```

Equal sign (=) punctuator cScript

The equal sign (=) separates variable declarations from initialization lists and determines the type of the variable.

```
array x[] = { 1, 2, 3, 4, 5 } ;
x = 5;
```

In cScript, declarations of any type can appear (with some restrictions) at any point within the code. In a cScript function argument list, the equal sign indicates the default value for a parameter:

```
MyFunc(i = 0){...} //Parameter i has default value of zero
```

The equal sign is also used as the assignment operator.

Lvalues and rvalues cScript

Lvalues

An *lvalue* is an identifier or expression that can be accessed as an object and legally changed in memory. A constant, for example, is not an lvalue. A variable, array member, or property is an lvalue.

Historically, the l stood for left, meaning that an lvalue could legally stand on the left (the receiving end) of an assignment statement. Only modifiable lvalues can legally stand on the left of an assignment statement.

For example, if a and b are variables, then they are both modifiable values and assignments. The following are legal:

```
a = 1
b = a + b
```

rvalues

An *rvalue* (short for "right value") is an expression that can be assigned to an lvalue. It is the "right side" of an assignment expression. While an lvalue can also be an rvalue, the opposite is not the case. For example, the following expression cannot be an lvalue:

```
a + b
```

`a + b = a` is illegal because the expression on the left is not related to an object that can be accessed and legally changed in memory.

However, `a = a + b` is legal, because a is a variable (an lvalue) and a + b is an expression that can be evaluated and assigned to a variable (an rvalue).

cScript preprocessor directives

Preprocessor directives are usually placed at the beginning of your source code, but they can legally appear at any point in a program. The cScript preprocessor, unlike a C++ preprocessor, supports preprocessor directives in the expansion side of a macro definition. It detects the following preprocessor directives and parses the tokens embedded in them:

- #ifndef
- #include
- #undef
- #warn

Any line with a leading # is considered as a preprocessor directive unless the # is part of a string literal, is in a character constant, or is embedded in a comment. The initial # can be preceded or followed by one or more spaces (excluding new lines).

#define cScript

Syntax

```
#define macro_identifier <token_sequence>
```

Description

The **#define** directive defines a *macro*. Macros provide a mechanism for token replacement with or without a set of formal, function-like parameters. Unlike C++ preprocessors, cScript allows you to continue a line with a backslash (\). You cannot use cScript keywords as macros.

Each occurrence of *macro_identifier* in your source code following the **#define** is replaced with *token_sequence* (with some exceptions). Such replacements are known as *macro expansions*. The token sequence is sometimes called the *body* of the macro.

If you use an empty token sequence, the macro identifier is removed wherever it occurs in the source code.

After each individual macro expansion, the preprocessor scans the newly expanded text to see if there are further macro identifiers that are subject to replacement (nested macros).

There are restrictions on macro expansion:

- Any occurrences of the macro identifier found within literal strings, character constants, or comments in the source code are not expanded.
- A macro is not expanded during its own expansion (so #define A A won't expand indefinitely).

Example

```
#define HI "Have a nice day!"
#define empty
#define NIL ""
#define GETSTD #include <stdio.h>
```

#ifdef, #ifndef, #else, and #endif cScript

Syntax

```
#ifdef/#ifndef identifier [logical-operator identifier [...]]
```

```
<section-1>
[#else
<final-section>]
#endif
<next-section>
```

Description

The **#ifdef** and **#ifndef** conditional directives let you test whether an identifier is currently defined or not; that is, whether a previous #define command has been processed for that identifier and is still in force. You can combine identifiers with logical operators.

#ifdef tests **TRUE** for the defined condition; so the line

```
#ifdef identifier
```

means that if *identifier* is defined, include the code follows up to the next **#else** or **#endif**. If *identifier* is not defined, ignore that code and skip to the next **#else** or **#endif**.

#else in this case means that if *identifier* is not defined, include the code that follows up to the next **#endif**.

#ifndef tests **TRUE** for the not-defined condition; so the line

```
#ifndef identifier
```

means that if *identifier* is not defined, include the code follows up to the next **#else** or **#endif**. If *identifier* is defined, ignore that code.

#else in this case means that if *identifier* is defined, include the code that follows up to the next **#endif**.

An identifier defined as NULL is considered to be defined.

cScript supports conditional compilation by replacing with blank lines the lines that are not to be compiled as a result of the directives. All conditional compilation directives must be completed in the source or include file in which they begin.

In the **TRUE** case, after *section-1* has been preprocessed, control passes to the matching **#endif** (which ends this conditional sequence) and continues with *next-section*. In the **FALSE** *case*, control passes to the next **#else** line (if any), which is used as an alternative condition for which the previous test proved false. The **#endif** ends the conditional sequence.

The processed section can contain further conditional clauses, nested to any depth; each **#ifdef** or **#ifndef** must be matched with a closing **#endif**.

The net result of the preceding scenario is that only one section (possibly empty) is passed on for further processing. The bypassed sections are relevant only for keeping track of any nested conditionals, so that each **#ifdef** or **#ifndef** can be matched with its correct **#endif**.

#include cScript

Syntax

```
#include <file_name>
#include "file_name"
#include macro_identifier
```

Description

The **#include** directive pulls other cScript files into the source code. The syntax has three versions:

- The first and second formats imply that no macro expansion will be attempted; in other words, *file_name* is never scanned for macro identifiers. *file_name* must be a valid file name with an optional path name and path delimiters.
- For the third version, neither < nor " can appear as the first non-whitespace character following **#include**. What must follow the **#include** is a macro definition that expands the macro identifier into a valid delimited file name with either of the <*file_name*> or *"file_name"* formats.

The preprocessor removes the **#include** line and replaces it with the entire text of the cScript source file at that point in the source code. The source code itself is not changed, but the compiler processes the enlarged text. The placement of the **#include** can therefore influence the scope and duration of any identifiers in the included file.

If you place an explicit path in the *file_name*, only that directory will be searched.

Unlike the C++ **#include**, there is no difference between the <*file_name*> and *"file_name"* formats. With both versions, the file is sought first in the current directory (usually the directory holding the source file being compiled). If the file is not found there, the search continues in the script directories in the order in which they are defined (as set up in the Options|Environment|Scripting|Script Path dialog box.) If the file is not located in any of the default directories, an error message is issued.

Example

This **#include** statement causes the preprocessor to look for MYINCLUD.H in the standard include directory.

```
#include <myinclud.h>
```

This **#include** statement causes the preprocessor to look for MYINCLUD.H in the current directory, then in default directories

```
#include "myinclud.h"
```

After expansion, this **#include** statement causes the preprocessor to look in C:\PARADIGM\SCRIPT\INCLUDE\MYSTUFF.H. Note that you must use double backslashes in the **#define** statement.

```
#define myinclud "C:\\PARADIGM\\SCRIPT\\INCLUDE\\MYSTUFF.H"
#include  myinclud
/* macro expansion */
```

#undef cScript

Syntax

```
#undef macro_identifier
```

Description

#undef detaches any previous token sequence from the macro identifier; the macro definition is forgotten, and the macro identifier is undefined. No macro expansion occurs within #undef lines.

The state of being defined or undefined is an important property of an identifier, regardless of the actual definition. The #ifdefcScr_ifdef and #ifndefcScr_ifdef

conditional directives, used to test whether any identifier is currently defined or not, offer a flexible mechanism for controlling many aspects of a compilation.

After a macro identifier is undefined, it can be redefined with #define,cScr_define using the same or a different token sequence.

Attempting to redefine an already defined macro identifier will result in a warning unless the new definition is exactly the same token-by-token definition as the existing one. The preferred strategy where definitions might exist in other header files is as follows:

```
#ifndef BLOCK_SIZE
  #define BLOCK_SIZE 512
#endif
```

The middle line is bypassed if BLOCK_SIZE is currently defined; if BLOCK_SIZE is not currently defined, the middle line is invoked to define it.

No semicolon (;) is needed to terminate a preprocessor directive. Any character found in the token sequence, including semicolons, will appear in the macro expansion. The token sequence terminates at the first non-backslashed new line encountered. Any sequence of whitespace, including comments in the token sequence, is replaced with a single-space character.

Example

```
#define BLOCK_SIZE 512
  .
  .
  .
#undef BLOCK_SIZE
/* use of BLOCK_SIZE now would be an illegal "unknown" identifier */
  .
  .
  .
#define BLOCK_SIZE 128   /* redefinition */
```

#warn cScript

Syntax

#warn *warning_level*

Description

The #warn directive sets the warning level. Warning levels range from 0 (suppress all warnings) to 3 (show all warnings).

For example, the following statement causes all warnings to be shown when the script is compiled:

```
#warn 3
```

Macros with parameters

The following syntax is used to define a macro with parameters:

```
#define macro_identifier(<arg_list>) token_sequence
```

Any comma within parentheses in an argument list is treated as part of the argument, not as an argument delimiter.

There can be no whitespace between the macro identifier and the (. The optional *arg_list* is a sequence of identifiers separated by commas, not unlike the argument list of a C function. Each comma-delimited identifier plays the role of a *formal argument* or *placeholder*.

Such macros are called by writing

```
macro_identifier<whitespace>(<actual_arg_list>)
```

in the subsequent source code. The syntax is identical to that of a function call. However, there are some important semantic differences, side effects, and potential pitfalls.

The optional *actual_arg_list* must contain the same number of comma-delimited token sequences, known as actual arguments, as found in the formal *arg_list* of the #define line. There must be an actual argument for each formal argument. An error will be reported if the number of arguments in the two lists is different.

A macro call results in two sets of replacements. First, the macro identifier and the parenthesis-enclosed arguments are replaced by the token sequence. Next, any formal arguments occurring in the token sequence are replaced by the corresponding real arguments appearing in the *actual_arg_list*.

As with simple macro definitions, rescanning occurs to detect any embedded macro identifiers eligible for expansion.

The similarities between function and macro calls can obscure their differences. A macro call can give rise to unwanted side effects, especially when an actual argument is evaluated more than once.

Nesting parentheses and commas

The *actual_arg_list* can contain nested parentheses provided that they are balanced; also, commas appearing within quotes or parentheses are not treated like argument delimiters.

Using the backslash (\) for line continuation

A long token sequence can straddle a line by using a backslash (\). The backslash and the following newline are both stripped to provide the actual token sequence used in expansions.

Chapter 4

cScript Class Reference

This chapter documents the built-in properties, methods, and events of the pre-defined cScript classes. The Paradigm C++ IDE is built using instances of these classes. Please note, however, that the objects which make up the IDE often have a number of additional properties, methods, and events added to them to make them even more powerful and flexible. These additional features are implemented in cScript. You are free to use and exploit these additional capabilities as you work with these objects. To learn more about these additional features, study the cScript source code and cScript examples in the SCRIPT subdirectory.

BufferOptions class

This class is one of the editor classes. *BufferOptions* objects hold data controlling the characteristics of edit buffers.

Syntax

```
BufferOptions()
```

The properties are initializaed during construction to match the global defaults.

Properties

`bool CreateBackup`	*Read-write*
`bool CursorThroughTabs`	*Read-write*
`bool HorizontalScrollBar`	*Read-write*
`bool InsertMode`	*Read-write*
`int LeftGutterWidth`	*Read-write*
`int Margin`	*Read-write*
`bool OverwriteBlocks`	*Read-write*
`bool PersistentBlocks`	*Read-write*
`bool PreserveLineEnds`	*Read-write*
`bool SyntaxHighlight`	*Read-write*
`string TabRack`	*Read-write*
`string TokenFileName`	*Read-write*
`bool UseTabCharacter`	*Read-write*
`bool VerticalScrollBar`	*Read-write*

Methods

```
void Copy(BufferOptions source)
```

BufferOptions class description

This class holds buffer options settings, such as scroll bars, right margin setting, tab rack, syntax highlighting, cursor shape, gutter width, block style and tabbing modes.

An instance of this class exists as a member of the global editor options accessible via Editor.Options. This class controls the settings of all edit buffers. Any change to this object changes the settings of all edit buffers.

You can instantiate a member of this class to store buffer options. They are not applied to any edit buffers until you copy them into Editor.Options, at which point the settings affect all edit buffers.

For example, you can store in a *BufferOptions* object a set of options that you want to apply to a buffer when it is activated (such as tab stops, syntax highlighting, and color). Applying these values to Editor.Options sets the buffer options for the new buffer and all other edit buffers as well.

CreateBackup property — BufferOptions

This is a read-write property.

Type expected
```
boolCreateBackup
```

CursorThroughTabs property — BufferOptions

This is a read-write property.

Type expected
```
bool CursorThroughTabs
```

HorizontalScrollBar property — BufferOptions

Set this property to **TRUE** if a horizontal scroll bar is to be associated with the buffer or **FALSE** if it is not. It is a read-write property.

Type expected
```
bool HorizontalScrollBar
```

InsertMode property — BufferOptions

This property indicates if the buffer is to be in insert (**TRUE**) or overstrike (**FALSE**) mode. It is a read-write property.

Type expected
```
bool InsertMode
```

LeftGutterWidth property — BufferOptions

This property indicates the width of the left gutter in pixels. It is a read-write property.

Type expected
```
int LeftGutterWidth
```

Margin property — BufferOptions

This property indicates the column to use to display the right margin. It is a read-write property.

Type expected
```
int Margin
```

OverwriteBlocks property — BufferOptions

This is a read-write property.

Type expected
```
bool OverwriteBlocks
```

PersistentBlocks property — BufferOptions

This is a read-write property.

Type expected
```
bool PersistentBlocks
```

PreserveLineEnds property — BufferOptions

This is a read-write property.

Type expected
```
bool PreserveLineEnds
```

SyntaxHighlight property — BufferOptions

This property indicates if the syntax is to be highlighted. It is a read-write property.

Type expected
```
bool SyntaxHighlight
```

TabRack property — BufferOptions

This property indicates the buffer's tab settings: a space-delimited sequence of tab stops in ascending order, for example, "3 7 12". It is a read-write property.

Type expected
```
string TabRack
```

TokenFileName property — BufferOptions

This property indicates the syntax highlighter file to use. It is a read-write property.

Type expected
```
string TokenFileName
```

UseTabCharacter property — BufferOptions

This is a read-write property.

Type expected

```
bool UseTabCharacter
```

VerticalScrollBar property — BufferOptions

Set this property to **TRUE** if a vertical scrollbar is to be associated with the buffer or **FALSE** if it is not. It is a read-write property.

Type expected
```
bool VerticalScrollBar
```

This method copies the values from the source *BufferOptions* object into this *BufferOptions* object.

Types expected
```
void Copy(BufferOptions source)
```

Return value
None

Tokenized cScript definition
After the cScript parser successfully parses a script (.SPP file), it converts it to interpreted pcode (*tokenizes* it) and saves it in a file with a .SPX extension. When the script is run in the future, the .SPX file is used if the script source file has not been changed (does not have a later time stamp).

Debugger class

Syntax
```
Debugger()
```

Properties
`bool HasProcess` *Read-only*

Methods
```
bool AddBreakpointAtCurrent()
bool AddBreakpoint()
bool AddBreakpointFileLine(string fileName, int lineNum)
bool AddWatch(string symbolName)
bool Animate()
bool Attach(string processID)
bool BreakpointOptions()
string Evaluate(string symbol)
bool EvaluateWindow(string symbol)
bool FindExecutionPoint()
bool Inspect(string symbol)
bool InstructionStepInto()
bool InstructionStepOver()
bool IsRunnable(int processID)
bool Load(string exeName)
bool PauseProgram()
bool Reset()
bool Run()
bool RunToAddress(string addr)
bool RunToFileLine(string fileName, int lineNum)
bool StatementStepInto()
bool StatementStepOver()
bool TerminateProgram()
bool ToggleBreakpoint(string fileName, int lineNum)
bool ViewBreakpoint()
bool ViewCallStackDebugger_ViewCallStack()
bool ViewCpuDebugger_ViewCpu([address])
bool ViewCpuFileLineDebugger_ViewCpuFileLine(string fileName, int
     lineNum)
bool ViewProcessDebugger_ViewProcess()
```

```
bool ViewWatchDebugger_ViewWatch()
```

Events

```
void DebugeeAboutToRunDebugger_DebugeeAboutToRun()
void DebugeeCreatedDebugger_DebugeeCreated()
void DebugeeStoppedDebugger_DebugeeStopped()
void DebugeeTerminatedDebugger_DebugeeTerminated()
```

HasProcess property — Debugger

This property is **TRUE** when the debugger has a process loaded, and is **FALSE**, otherwise.

Types expected

```
bool AddBreakpoint()
```

AddBreakAtCurrent method — Debugger

Adds a breakpoint to the current line (indicated by the cursor) of the file in the topmost edit buffer. If no file is open, this method opens the Add Breakpoint dialog.

Types expected

```
bool AddBreakAtCurrent()
```

Return value

TRUE if successful, **FALSE** otherwise.

AddBreakpoint method — Debugger

This method opens the Add Breakpoint dialog.

Types expected

```
bool AddBreakpoint()
```

Return value

TRUE if successful, **FALSE** otherwise.

AddBreakpointFileLine method — Debugger

This method adds a breakpoint on the specified line of the specified file. If the arguments are **NULL**, this method opens the Add Breakpoint dialog.

Types expected

```
bool AddBreakpointFileLine(string fileName, int lineNum)
```

Return value

TRUE if successful, **FALSE** otherwise

AddWatch method — Debugger

This method adds a watch on the specified *symbolName*. If argument is **NULL**, this method opens the Add Watch dialog.

Types expected

```
bool AddWatch(string symbolName)
```

Return value

TRUE if successful, **FALSE** otherwise.

Animate method — Debugger

This method lets you watch your program's execution in "slow motion."

Types expected

```
bool Animate()
```

Return value

TRUE if successful, **FALSE** otherwise

Description

Animate performs a continuous series of Statement Step Into commands. To interrupt animation, invoke one of the following methods either by menu selections or by keystrokes tied to the script:

Debugger.Run
Debugger.RunToAddress
Debugger.RunToFileLine
Debugger.PauseProgram
Debugger.Reset
Debugger.TerminateProgram
Debugger.FindExecutionPoint

Attach method — Debugger

This method invokes the debugger for the currently executing process identified by *processID*

Types expected

```
bool Attach(string processID)
```

Return value

TRUE if successful, **FALSE** otherwise

BreakpointOptions method — Debugger

This method opens the Breakpoint Condition/Action Options dialog.

Types expected

```
bool BreakpointOptions()
```

Return value

TRUE if successful, **FALSE** otherwise

Evaluate method — Debugger

This method evaluates the given expression, such as a global or local variable or an arithmetic expression.

Types expected

```
string Evaluate(string expression)
```

Return value

Returns the result of the evaluation.

EvaluateWindow method — Debugger

This method opens the Evaluator view with *expression* pasted into the Expression field of the view.

Types expected

```
bool EvaluateWindow(string expression)
```

Return value

TRUE if successful, **FALSE** otherwise

FindExecutionPoint method — Debugger

This method displays the current execution point.

Types expected

```
bool FindExecutionPoint()
```

Return value

TRUE if successful, **FALSE** otherwise

Inspect method — Debugger

This method attempts to open an inspector for the specified *symbol*. The inspector window opens using the specified *view* at the given *row* and *column* positions.

Types expected

```
bool Inspect(string symbol, EditView view, int row, int column)
```

Return value

TRUE if successful, **FALSE** otherwise

InstructionStepInto method — Debugger

This method executes the next instruction, stepping into any function calls. If a process is not loaded, *InstructionStepInto* first loads the executable for the current project.

Types expected

```
bool InstructionStepInto()
```

Return value

TRUE if successful, **FALSE** otherwise

InstructionStepOver method — Debugger

This method executes the next instruction, running any functions called at full speed. If a process is not loaded, *InstructionStepOver* first loads the executable for the current project.

Types expected

```
bool InstructionStepOver()
```

Return value

TRUE if successful, **FALSE** otherwise

IsRunnable method — Debugger

This method indicates if the specified process can be run or single stepped.

Types expected

```
bool IsRunnable(int processID)
```

processID The process you wish to query. If that process is not runnable or does not exist, the current process is used.

Return value

TRUE if the EXE is runnable or can be single stepped; **FALSE** otherwise

Load method — Debugger

This method loads the specified executable into the debugger.

Types expected

```
bool Load(string exeName)
```

Return value

TRUE if successful, **FALSE** otherwise.

Description

Upon loading, the process runs to the starting point specified in the Options|Environment|Debugger|Debugger Behavior dialog. If the parameter is **NULL**, this method opens the Load Program dialog.

PauseProgram method — Debugger

This method causes the debugger to pause the current process. It has an effect only if the current process is running or animated.

Types expected

```
bool PauseProgram()
```

Return value

TRUE if successful, **FALSE** otherwise

Reset method — Debugger

This method causes the debugger to reset the current process to its starting point (as specified in the Options|Environment|Debugger|Debugger Behavior dialog)

Types expected

```
bool Reset()
```

Return value

TRUE if successful, **FALSE** otherwise

Run method — Debugger

This method causes the debugger to run the current process. If no process is loaded, this method first loads the executable associated with the current project.

Types expected

```
bool Run()
```

Return value

TRUE if successful, **FALSE** otherwise

RunToAddress method — Debugger

This method runs the current process until the instruction at the given address is encountered. If no process is loaded, the method first loads the executable associated with the current project.

Types expected

```
bool RunToAddress(string address)
```

Return value

TRUE if successful, **FALSE** otherwise

RunToFileLine method — Debugger

This method runs the current process until the source at the specified line in the specified file is encountered. If no process is loaded, this method will first load the executable associated with the current project.

Types expected

```
bool RunToFileLine(string fileName, int lineNum)
```

Return value
TRUE if successful, **FALSE** otherwise

StatementStepInto method — Debugger

This method executes the next source statement and steps through the source of any function calls. If a process is not loaded, this method first loads the executable for the current project.

Types expected
```
bool StatementStepInto()
```

Return value
TRUE if successful, **FALSE** otherwise

StatementStepOver method — Debugger

This method executes the next source statement and does not step into any functions called, but rather runs them at full speed. If a process is not loaded, *StatementStepOver* first loads the executable for the current project.

Types expected
```
bool StatementStepOver()
```

Return value
TRUE if successful, **FALSE** otherwise

TerminateProgram method — Debugger

This method terminates the current process. If no process is loaded, this method has no effect.

Types expected
```
bool TerminateProgram()
```

Return value
TRUE if successful, **FALSE** otherwise

ToggleBreakpoint method — Debugger

This method either adds a breakpoint on the specified line of the specified file or deletes an existing breakpoint. If the arguments are **NULL**, this method opens the Add Breakpoint dialog.

Types expected
```
bool ToggleBreakpoint(string fileName, int lineNum)
```

Return value
TRUE if successful, **FALSE** otherwise

ViewBreakpoint method — Debugger

This method opens the breakpoint view.

Types expected
```
bool ViewBreakpoint()
```

Return value
TRUE if successful, **FALSE** otherwise

ViewCallStack method — Debugger

This method opens the Call Stack view. It works only if a process is loaded.

Types expected
```
bool ViewCallStack()
```

Return value
TRUE if successful, **FALSE** otherwise

ViewCpu method — Debugger

This method opens or selects the CPU view.

Syntax
```
ViewCPU()
```

Types expected
```
bool ViewCpu([address])
```

Return value
TRUE if successful, **FALSE** otherwise

Description
If the "Allow Multiple CPU Views" option is checked in the Options | Environment | Debugger | Debugger Behavior dialog, this method always opens a new CPU view. If the option is not checked, *ViewCpu* only opens a new CPU view if one is not already open. It works only if a process is loaded.

ViewCpuFileLine method — Debugger

This method opens or selects the CPU view.

Types expected
```
bool ViewCpu(string fileName, int lineNum)
```

Return value
TRUE if successful, **FALSE** otherwise

Description

If the "Allow Multiple CPU Views" option is checked in the Options | Environment | Debugger | Debugger Behavior dialog, *ViewCpuFileLine* always opens a new CPU view. If the option is not checked, it opens a new CPU view only if one is not already open. After opening or selecting a CPU view, the Disassembly pane is scrolled so that the disassembled code for the specified line of the specified file is visible. If the parameters are **NULL** or if the line doesn't generate code, the window displays an error message. This method works only if a process is loaded.

ViewProcess method — Debugger

This method opens the Process view.

Types expected

```
bool ViewProcess()
```

Return value

TRUE if successful, **FALSE** otherwise

ViewWatch method — Debugger

This method opens the watch view.

Types expected

```
bool ViewWatch()
```

Return value

TRUE if successful, **FALSE** otherwise

DebugeeAboutToRun event — Debugger

This event is raised just before a process is run.

Types expected

```
void DebugeeAboutToRun()
```

Return value

None

DebugeeCreated event — Debugger

This event is raised when a new process is created (loaded into the debugger).

Types expected

```
void DebugeeCreated()
```

Return value

None

DebugeeStopped event — Debugger

This event is raised when a process stops. A process can stop for any number of reasons: upon normal termination, after a step, when a breakpoint is hit, when an exception occurs, or when the user pauses, resets, or terminates a running application

Types expected

```
void DebugeeStopped()
```

Return value

None

DebugeeTerminated event — Debugger

This event is raised when a process is terminated.

Types expected

```
void DebugeeTerminated()
```

Return value

None

watch definition

A watch monitors the state of a variable. Whenever your program pauses, the debugger evaluates all watched variables and updates the Watches window with their values.

EditBlock class

This class is one of the editor classes. It provide area-marking features for an edit buffer or view.

Syntax

```
EditBlock(EditBuffer);
EditBlock(EditView);
```

Properties

`bool IsValid`	*Read-only*
`int EndingColumn`	*Read-only*
`int EndingRow`	*Read-only*
`bool Hide`	*Read-write*
`int Size`	*Read-write*
`int StartingColumn`	*Read-only*
`int StartingRow`	*Read-only*
`int Style`	*Read-write*
`string Text`	*Read-only*

Methods

```
void Begin()
void Copy([bool useClipboard, bool append])
void Cut([bool useClipboard, bool append])
bool Delete()
void End()
bool Extend(int newRow, int newCol)
bool ExtendPageDown()
bool ExtendPageUp()
bool ExtendReal(int newRow, int newColumn)
bool ExtendRelative(int deltaRow, int deltaColumn)
void Indent(int magnitude)
void LowerCase()
bool Print()
void Reset()
void Restore()
void Save()
bool SaveToFile([string fileName])
void ToggleCase()
void UpperCase()
```

EditBlock class description

EditBlock objects allow you to mark areas of text. Because *EditBlock* members exist in both the EditView and the EditBuffer, EditView and EditBuffer support different marked areas in different views on the same *EditBuffer*.

Although multiple *EditBlocks* can exist in script for an individual *EditBuffer* or *EditView*, they are mapped to the same internal representation of the *EditBlock*. Therefore, manipulations on one will affect the others. Use of the *Extend()* members will cause the *EditPosition* for the owner to be updated appropriately:

IsValid property — EditBlock

Becomes **FALSE** in any of the following cases:

- The owning *EditBuffer* or *EditView* is destroyed.
- A destructive operation has occurred on the block, such as delete or cut.
- The ending point is not greater than the starting point.

This is a read-only property.

Type expected

```
bool IsValid
```

EndingColumn property — EditBlock

Initialized to the current position in the *EditView* or *EditBuffer* upon construction. May be changed by a call to an external method. This is a read-only property.

Type expected

```
int EndingColumn
```

EndingRow property — EditBlock

Initialized to the current position in the *EditView* or *EditBuffer* upon construction. May be changed by a call to an external method. This is a read-only property.

Type expected

```
int EndingRow
```

Hide property — EditBlock

Visually disables the block without modifying its coordinates. This is a read-write property.

Type expected

```
bool Hide
```

Size property — EditBlock

If the area is not valid, the value is zero; otherwise, the value is the number of characters contained in the marked area. A newline (CR/LF) counts as one character. This is a read-write property.

Type expected

```
int Size
```

StartingColumn property — EditBlock

Initialized to the current position in the *EditView* or *EditBuffer* upon construction. May be changed by a call to an external method. This is a read-only property.

Type expected

```
int StartingColumn
```

StartingRow property — EditBlock

Initialized to the current position in the *EditView* or *EditBuffer* upon construction. May be changed by a call to an external method. This is a read-only property.

Type expected

```
int StartingRow
```

Style property — EditBlock

One of the following values:

INCLUSIVE_BLOCK
EXCLUSIVE_BLOCK
COLUMN_BLOCK
LINE_BLOCK
INVALID_BLOCK

This is a read-write property.

Type expected
```
int Style
```

Description
An *EditBlock* is initially set to the *Style* EXCLUSIVE_BLOCK. It is also set to this style after a *Reset* is called. If an *EditBlock* has a *Style* of INVALID_BLOCK, it is because it was retained after the *EditBuffer* or *EditView* to which it was attached was destroyed.

Text property — EditBlock

If the marked block is valid, *Text* returns the marked text. If it is invalid, Text returns the empty string. This is a read-only property.

Type expected
```
string Text
```

Begin method — EditBlock

Resets the *StartingRow* and *StartingColumn* values to the current location in the owning *EditBuffer* or *EditView*. This is a read-only property.

Type expected
```
void Begin()
```

Return value
None

Copy method — EditBlock

Copies the contents of the marked block to the Windows Clipboard. *append* defaults to **FALSE**. If *append* is **TRUE** the contents of the marked block are appended to the clipboard contents

Types expected
```
void Copy([bool append])
```

Return value
None

Cut method — EditBlock

Cuts the contents of the marked block to the Windows Clipboard and invalidates the marked block. *append* defaults to **FALSE**. If *append* is **TRUE** the contents of the marked block are appended to the clipboard contents.

Types expected
```
void Cut([bool append])
```

Return value
None

Delete method — EditBlock

This method deletes the current block (if valid). The return value indicates if any characters were deleted. The cursor's position is restored to the position it occupied prior to the delete.

Types expected

```
bool Delete()
```

Return value

None

End method — EditBlock

Resets the *EndingRow* and *EndingColumn* values to the current location in the owning *EditBuffer* or *EditView*.

Types expected

```
void End()
```

Return value

None

Extend method — EditBlock

Extends an existing EditBlock to encompass the text delimited by *newRow* and *newCol*.

Types expected

```
bool Extend(int newRow, int newCol)
```

Return value

TRUE if the Extend successfully completes; otherwise **FALSE**.

ExtendPageDown method — EditBlock

This method updates the starting or ending points of the existing mark to extend the mark to the specified location. It also causes the position in the owning EditBuffer or EditView to be updated to the new location. *ExtendPageDown* works only if the block is associated with an *EditView* and is ignored if the block is associated with an *EditBuffer*.

Types expected

```
bool ExtendPageDown()
```

Return value

TRUE if the cursor move is successful; otherwise **FALSE**.

ExtendPageUp method — EditBlock

This method updates the starting or ending points of the existing mark to extend the mark to the specified location. It also causes the position in the owning EditBuffer or

EditView to be updated to the new location. *ExtendPageUp* works only if the block is associated with an *EditView* and is ignored if the block is associated with an *EditBuffer*.

Types expected

```
bool ExtendPageUp()
```

Return value

TRUE if the cursor move is successful.

ExtendReal method — EditBlock

This method updates the starting or ending points of the existing mark to extend the mark to the specified location. It also causes the position in the owning EditBuffer or EditView to be updated to the new location.

Types expected

```
bool ExtendReal(int newRow, int newColumn)
```

Return value

TRUE if the cursor move is successful.

ExtendRelative method — EditBlock

This method updates the starting or ending points of the existing mark to extend the mark to the specified relative location. It also causes the position in the owning EditBuffer or EditView to be updated to the new location.

Types expected

```
bool ExtendRelative(int deltaRow, int deltaColumn)
```

Return value

TRUE if the cursor move is successful.

Indent method — EditBlock

Moves the contents of the block left/right the number of columns specified. Negative values move to the left, positive move to the right.

Types expected

```
void Indent(int magnitude)
```

Return value

None

LowerCase method — EditBlock

Converts all alphabetic characters enclosed within the EditBlock to lowercase.

Types expected

```
void LowerCase()
```

Return value
None

Print method — EditBlock

Prints the current block (if any).

Types expected
```
bool Print()
```

Return value
Returns **TRUE** if the print was successful or **FALSE** if there is no marked block or if the print failed.

Reset method — EditBlock

Implicitly invoked by the constructor, *Reset* may also be used to reset the style to EXCLUSIVE_BLOCK and the starting and ending points to be the same as the current position in the owning *EditBuffer* or *EditView*.

Types expected
```
void Reset()
```

Return value
None

Restore method — EditBlock

Restores a block from an internal stack. The block must have been saved with *Save*().

Types expected
```
void Restore()
```

Return value
None

Save method — EditBlock

Preserves the block attributes on an internal stack for future restoration with *Restore*().

Types expected
```
void Save()
```

Return value
None

SaveToFile method — EditBlock

This method causes the contents of the marked block to be saved to the file *fileName*. If *fileName* is not supplied, the user will be prompted for one.

Types expected

```
bool SaveToFile([string fileName])
```

Return value

Returns **TRUE** if the save was successful or **FALSE** if it wasn't.

ToggleCase method — EditBlock

Converts all the uppercase alphabetic characters enclosed within the EditBlock to lowercase, and the lowercase characters to uppercase.

Types expected

```
void ToggleCase()
```

Return value

None

UpperCase method — EditBlock

Converts all the lowercase alphabetic characters enclosed within the EditBlock to uppercase.

Types expected

```
void UpperCase()
```

Return value

None

EditBuffer class

This class is one of the editor classes. An edit buffer is associated with one file and any number of edit views.

Syntax

```
EditBuffer(string fileName [, bool private, bool readOnly])
```

fileName The name of the file associated with the edit buffer.

private Implies that the buffer is a hidden system buffer. Undo information is not retained, and the *EditBuffer* is never attachable to an EditView. The default value for *private* is FALSE.

readOnly Default value is **FALSE**.

Properties

`EditBlock Block`	*Read-only*
`TimeStamp CurrentDate`	*Read-only*
`string Directory`	*Read-only*
`string Drive`	*Read-only*
`string Extension`	*Read-only*

`string FullName`	*Read-only*
`TimeStamp InitialDate`	*Read-only*
`bool IsModified`	*Read-only*
`bool IsPrivate`	*Read-only*
`bool IsReadOnly`	*Read-only*
`bool IsValid`	*Read-only*
`EditPosition Position`	*Read-only*
`EditView TopView`	*Read-only*

Methods

```
void ApplyStyle(EditStyle styleToApply)
EditBlock BlockCreate()
string Describe()
bool Destroy()
EditBuffer NextBuffer(bool privateToo)
EditView NextView(EditView)
EditPosition PositionCreate()
bool Print()
EditBuffer PriorBuffer(bool privateToo)
bool Rename(string newName)
int Save([string newName])
```

Events

```
void AttemptToModifyReadOnlyBuffer()
void HasBeenModified()
```

EditBuffer class description

An *EditBuffer* is a representation of the contents of a file. An EditView is used to provide a visual representation of the *EditBuffer*. The same *EditBuffer* can be displayed simultaneously in different *EditViews* (for example, two edit windows open on the same file). *EditBuffer* objects provide functionality for a file being edited that is independent of the number of views associated with the buffer.

Buffers have methods allowing them to traverse the list of views containing the same *EditBuffer*. They maintain access to a list of bookmarks (position markers which track text edits). Buffers can be queried for their time and date stamps.

Buffer contains a Position member through which manipulation of the underlying *EditBuffer* is performed. Typically this member will be used when manipulating an *EditBuffer* through script.

A single *EditBuffer* object exists internally for each loaded filename. If you create additional representations for an edit buffer, they are attached to the existing *EditBuffer* object. Any changes to one of these representations changes the others, since they refer to the same object. All representations inherit the IsReadOnly and IsPrivate attributes of the original, because these properties are set only when the object is first created.

Buffers may be made private to provide raw data storage for script usage. No undo information is maintained for private buffers, nor are they attachable to an *EditView*. Buffers may also be specified as *ReadOnly*.

Block property — EditBuffer

This property contains a reference to the hidden EditBlock. This is a read-only property.

Type expected

```
EditBlock Block
```

CurrentDate property — EditBuffer

This property is originally set to the same value as InitialDate but is updated when the buffer's contents are altered. It is a read-only property.

Type expected

```
TimeStamp CurrentDate
```

Directory property — EditBuffer

This property is **NULL** if the *EditBuffer* is invalid; otherwise, it indicates the directory path in uppercase letters. It is a read-only property.

Type expected

```
string Directory
```

Drive property — EditBuffer

This property is **NULL** if the *EditBuffer* is invalid; otherwise, it indicates the uppercase drive letter with its associated colon (:). It is a read-only property.

Type expected

```
string Drive
```

Extension property — EditBuffer

This property is **NULL** if the *EditBuffer* is invalid; otherwise, it indicates the uppercase file extension including the period (.), if any. It is a read-only property.

Type expected

```
string Extension
```

FileName property — EditBuffer

This property is **NULL** if the *EditBuffer* is invalid; otherwise, it indicates the file name in uppercase letters. It is a read-only property.

Type expected

```
string FileName
```

FullName property — EditBuffer

This property indicates the name of the *EditBuffer* or **NULL** if the *EditBuffer* is invalid. It is a read-only property.

Type expected

```
string FullName
```

InitialDate property — EditBuffer

If the buffer is initialized from a disk file, *InitialDate* reflects the file's age. If the file doesn't reside on disk, *InitialDate* holds the time at which the buffer was created. It is a read-only property.

Type expected

```
TimeStamp InitialDate
```

IsModified property — EditBuffer

This method indicates if the buffer was changed since it was last opened or saved, whichever occurred most recently. It is a read-only property.

Type expected

```
bool IsModified
```

IsPrivate property — EditBuffer

This property is **TRUE** if the buffer was created with the *private* parameter set to **TRUE**, otherwise it is FALSE. It is a read-only property.

Type expected

```
bool IsPrivate
```

IsReadOnly property — EditBuffer

This property is **TRUE** if the buffer was created with the *readOnly* parameter set to **TRUE**, otherwise it is **FALSE**. It is a read-only property.

Type expected

```
bool IsReadOnly
```

IsValid property — EditBuffer

This property is **FALSE** if the *EditBuffer* is destroyed, otherwise it is **TRUE**. It is a read-only property.

Type expected

```
bool IsValid
```

Position property — EditBuffer

This property provides access to the *EditPosition* instance for this *EditBuffer*.

Type expected

```
EditPosition Position
```

TopView property — EditBuffer

This property indicates the topmost *EditView* that contains this *EditBuffer*, or **NULL** if no view is associated with the buffer. It is a read-only property.

Type expected

```
EditView TopView
```

ApplyStyle method — EditBuffer

This method updates the *EditBuffer's* member with the contents of *styleToApply*.

Types expected

```
void ApplyStyle(EditStyle styleToApply)
```

BlockCreate method — EditBuffer

Types expected

```
EditBlock BlockCreate()
```

Describe method — EditBuffer

This method invoked during buffer list creation by an *Editor* object. It returns a text description of the buffer, as in:

FOO.CPP(modified)
BAR.CPP

Types expected

```
string Describe()
```

Destroy method — EditBuffer

This method removes the buffer from the IDE's buffer list and does not save any changes.

Types expected

```
bool Destroy()
```

Return value

TRUE if the buffer was actually destroyed, or **FALSE** if views relying on it still exist.

NextBuffer method — EditBuffer

This method finds the next edit buffer in the buffer list. The list is circular, so a buffer will be found if one exists, unless all buffers are private and *privateToo* is **FALSE**.

Types expected

```
EditBuffer NextBuffer(bool privateToo)
```

privateToo Indicates if private buffers are to be included.

Return value

The edit buffer found or **NULL** if it finds none.

NextView method — EditBuffer

Returns the next EditView containing this **EditBuffer**.

Types expected

```
EditView NextView(EditView next)
```

next — The view to use in getting the next associated view for this edit buffer. Commonly you start walking the view list by passing the value of TopView to this method.

Description

An *EditBuffer* is a representation of the contents of a file. An *EditView* is used to provide a visual representation of the *EditBuffer*. The same *EditBuffer* can be displayed simultaneously to the user in different *EditViews* (for example, two edit windows can be open on the same file). This method enables you to cycle through all the *EditViews* representing this *EditBuffer*.

PositionCreate method — EditBuffer

Types expected

```
EditPosition PositionCreate()
```

Print method — EditBuffer

Prints this buffer and returns **TRUE** if the print was successful or **FALSE** if the print failed.

Types expected

```
bool Print()
```

PriorBuffer method — EditBuffer

This method finds the previous edit buffer in the buffer list. The list is circular, so a buffer will be found if one exists, unless all buffers are private and *privateToo* is **FALSE**.

Types expected

```
EditBuffer PriorBuffer(bool privateToo)
```

privateToo — Indicates if private buffers are to be included.

Return value

The edit buffer found or **NULL** if it finds none.

Rename method — EditBuffer

This method changes the edit buffer name to the one specified in *newName*. *Rename* fails when an *EditBuffer* with the new name is already in the buffer list. If a file with the new name already exists on disk, it is overwritten when this buffer is saved.

Types expected

```
bool Rename(string newName)
```

Return value

TRUE if the operation succeeded or **FALSE** if it failed.

Save method — EditBuffer

This method writes the file associated with the buffer to disk whether it was modified or not. It uses the current name of the file or *newName* if it is specified.

Types expected

```
int Save([string newName])
```

Return value

The number of bytes written or 0 if the save was unsuccessful.

AttemptToModifyReadOnlyBuffer event — EditBuffer

This event is triggered when an attempt is made to modify a read-only buffer.

Types expected

```
void AttemptToModifyReadOnlyBuffer()
```

Return value

None

AttemptToWriteReadOnlyFile event — EditBuffer

This event is triggered when an attempt is made to write the contents of an EditBuffer to a read-only file.

Types expected

```
void AttemptToWriteReadOnlyBuffer()
```

Return value

None

HasBeenModified event — EditBuffer

This event is triggered when a buffer has been modified for the first time.

Types expected

```
void HasBeenModified()
```

Return value
None

Editor class

This class provides access to the IDE's internal editor. Editor is associated with other classes which provide the editor with its functionality.

Syntax
```
Editor()
```

Properties
`EditStyle FirstStyle`	*Read-only*
`EditOptions Options`	*Read-only*
`SearchOptions SearchOptions`	*Read-only*
`EditBuffer TopBuffer`	*Read-only*
`EditView TopView`	*Read-only*

Methods
```
void ApplyStyle(EditStyle newOptions)
void BufferList()
BufferOptions BufferOptionsCreate()
bool BufferRedo(EditBuffer buffer)
bool BufferUndo(EditBuffer buffer)
void DestroyedStyle(EditStyle styleToRemove)
EditBuffer EditBufferCreate(string fileName [, bool private, bool
readOnly])
EditOptions EditOptionsCreate()
EditStyle EditStyleCreate(string styleName[,EditStyle toInheritFrom])
EditWindow EditWindowCreate(EditBuffer buffer)
string GetClipboard()
int GetClipboardToken()
EditWindow GetWindow([bool getLast])
bool IsFileLoaded(string filename)
EditStyle StyleGetNext(EditStyle)
bool ViewRedo(EditView view)
bool ViewUndo(EditView view)
```

Events
```
void BufferCreated(EditBuffer buffer)
void MouseBlockCreated()
void MouseLeftDown()
void MouseLeftUp()
string MouseTipRequested(EditView theView, int line, int column)
void OptionsChanged(EditorOptions newOptions)
void OptionsChanging(EditorOptions newOptions)
void ViewActivated(EditView view)
void ViewCreated(EditView newView)
void ViewDestroyed(EditView deadView)
```

Editor class description

The IDE instantiates an *Editor* object, which maintains undo and redo data and has methods allowing access to the list of all buffers and Edit windows. Editors have a member of type *EditOptions* that controls global editor characteristics such as scrap

manipulation (blocks cut or copied from Editor buffers), the default regular expression language, and destination paths for backups. Although multiple instances of *Editor* objects may be created in script, they all refer to the same instance of a single C++ object internal to the IDE; therefore, modification of one *Editor* object's options will be reflected in all *Editor* objects.

FirstStyle property — Editor

This property contains the first style in the list of editor styles. It is usually used in conjunction with the Editor.StyleGetNext() method. At least one *EditStyle* must exist for this property to contain a valid value. This is a read-only property.

Type expected

```
EditStyle FirstStyle
```

Options property — Editor

This property holds the buffer options settings for all edit buffers, Changing an option in this property affects all edit buffers. This is a read-only property.

Type expected

```
EditOptions Options
```

SearchOptions property — Editor

This property provides access to the instance of *SearchOptions* associated with this editor. This is a read-only property.

Type expected

```
SearchOptions SearchOptions
```

TopBuffer property — Editor

This property indicates the current edit buffer. This is a read-only property.

Type expected

```
EditBuffer TopBuffer
```

TopView property — Editor

This property provides a quick way to get at top view associated with the current edit buffer. This is a read-only property.

Type expected

```
EditView TopView
```

ApplyStyle method — Editor

This method updates the edit options with the contents of *newOptions*.

Types expected

```
void ApplyStyle(EditStyle newOptions)
```

Return value
None

BufferList method — Editor

This method presents to the user a list of buffers that comes from Edit.Buffer.Describe().

Types expected
```
void BufferList()
```

Return value
None

BufferOptionsCreate method — Editor

This method creates a new instance of the *BufferOptions* class.

Types expected
```
BufferOptions BufferOptionsCreate()
```

Return value
A *BufferOptions* object

BufferRedo method — Editor

This method undoes the last undo operation on the buffer or view regardless of whether the operation was performed on the *EditBuffer*, the *EditView*, an *EditBlock*, or an *EditPosition*.

Types expected
```
bool BufferRedo(EditBuffer buffer)
```

Return value
TRUE if there are more operations to redo, or **FALSE** if there are not.

BufferUndo method — Editor

This method undoes the last operation on the buffer or view regardless of whether the operation was performed on the *EditBuffer*, the *EditView*, an *EditBlock*, or an *EditPosition*.

Types expected
```
bool BufferUndo(EditBuffer buffer)
```

Return value
TRUE if there are more operations to undo or **FALSE** if there are not.

EditBufferCreate method — Editor

This method creates an edit buffer corresponding to *fileName*.

Types expected

```
EditBuffer EditBufferCreate(string fileName [, bool
           private, bool readOnly])
```

Return value

The edit buffer created, or **NULL** if none could be created.

EditOptionsCreate method — Editor

This method creates a new instance of the *EditOptions* class.

Types expected

```
EditOptions EditOptionsCreate()
```

Return value

An *EditOptions* object.

EditStyleCreate method — Editor

This method creates an edit style.

Types expected

```
EditStyle EditStyleCreate(string styleName[,EditStyle
          toInheritFrom])
```

Return value

The edit style created, or **NULL** if none could be created.

EditWindowCreate method — Editor

This method creates an edit window.

Types expected

```
EditWindow EditWindowCreate(EditBuffer buffer)
```

Return value

The edit window created, or **NULL** if none could be created.

GetClipboard method — Editor

This method returns the contents of the Windows Clipboard in a string.

Types expected

```
string GetClipboard()
```

GetClipboardToken method — Editor

This method returns the memory address of the Windows Clipboard contents.

Types expected

```
int GetClipboardToken()
```

GetWindow method — Editor

This method returns an *EditWindow*. If *getLast* is **FALSE**, it returns the top level window. If it is **TRUE**, *GetWindow* returns the last EditWindow in the Z-order. *getLast* defaults to **FALSE**.

Types expected

```
EditWindow GetWindow([bool getLast])
```

IsFileLoaded method — Editor

This method returns **TRUE** if a buffer by that name exists or **FALSE** if one doesn't.

Types expected

```
bool IsFileLoaded(string fileName)
```

StyleGetNext method — Editor

Use this method in conjunction with the *FirstStyle* property to access the circularly linked list representing all the editor styles.

Types expected

```
EditStyle StyleGetNext(EditStyle)
```

Return value

The editor style that was found, or **NULL** if no editor style is found.

ViewRedo method — Editor

This method reapplies the last operation that was undone on the buffer or view regardless of whether the operation was performed on the *EditBuffer* the *EditView*, an *EditBlock*, or an *EditPosition*.

Types expected

```
bool ViewRedo(EditView view)
```

Return value

TRUE if there are more operations to redo, or **FALSE** if there are not.

ViewUndo method — Editor

This method undoes the last operation on the buffer or view regardless of whether the operation was performed on the *EditBuffer*, the *EditView*, an *EditBlock*, or an *EditPosition*.

Types expected

```
bool ViewUndo(EditView view)
```

Return value

TRUE if there are more operations to undo, or **FALSE** if there are not.

BufferCreated event — Editor

This event is triggered when a new *EditBuffer* is created. *buffer* is the newly created buffer. The default action is to do nothing.

Types expected

```
void BufferCreated(EditBuffer buffer)
```

MouseBlockCreated event — Editor

This event is triggered when the user selects a block with the mouse in the top view.

Types expected

```
void MouseBlockCreated()
```

Return value

None

MouseLeftDown event — Editor

This event is triggered when the mouse left button is pressed in an Edit window.

Types expected

```
void MouseLeftDown()
```

Return value

None

MouseLeftUp event — Editor

This event is triggered when the mouse left button is released in an Edit window.

Types expected

```
void MouseLeftUp()
```

Return value

None

MouseTipRequested event — Editor

This event is raised when the mouse has remained idle over an editor window for a period of time.

Types expected

```
string MouseTipRequested(EditView theView, int line,
                         int column)
```

theView The *EditView* object describing the edit window that contains the idle mouse.

line, column The position in the edit buffer of the character the mouse cursor is on.

Return value

If this routine returns a string, it displays the string to the user as a help hint. The default implementation returns a NULL.

OptionsChanged event — Editor

This event is raised when the *OptionsChanging* event handler has completed and the global values have been changed. This event notifies a script that needs to update the global options that those options have changed.

Types expected

```
void OptionsChanged(EditorOptions newOptions)
```

Return value

None

OptionsChanging event — Editor

This event is raised when leaving one of the editor MPD pages with *accept*. The event contains a copy of the new values for the global editor options. An event handler may examine these values and determine if any of the values need to be overridden with any values from *newOptions*.

Types expected

```
void OptionsChanging(EditorOptions newOptions)
```

Return value

None

ViewActivated event — Editor

This event is triggered when the *EditView* represented by *view* is activated. There is no default action for this event.

Types expected

```
void ViewActivated(EditView view)
```

Return value

None

ViewCreated event — Editor

This event is triggered when the *EditView* represented by *newView* is created. There is no default action for this event.

Types expected

```
void ViewCreated(EditView newView)
```

Return value

None

ViewDestroyed event — Editor

This event is triggered when the *EditView* represented by *deadView* is destroyed. There is no default action for this event.

Types expected

```
void ViewDestroyed(EditView deadView)
```

Return value

None

EditOptions class

This class is one of the editor classes. It holds editor characteristics of a global nature such as the insert/overtype setting, optimal fill, and scrap settings (how to handle blocks cut or copied from Editor buffers).

Syntax

```
EditOptions()
```

The values are initialized from global defaults during construction.

Properties

`string BackupPath`	*Read-write*
`int BlockIndent`	*Read-write*
`BufferOptions BufferOptions`	*Read-only*
`string MirrorPath`	*Read-write*
`string OriginalPath`	*Read-write*
`string SyntaxHighlightTypes`	*Read-write*
`bool UseBRIEFCursorShapes`	*Read-write*
`bool UseBRIEFRegularExpression`	*Read-write*

Methods

None

EditOptions class description

The *EditOptions* object holds editor characteristics of a global nature, such as whether to create backups and the option settings for all buffers (in *BufferOptions*).

BackupPath property — EditOptions

This property contains the path where the editor stores backup files. This is a read-write property.

Type expected

```
string BackupPath
```

BlockIndent property — EditOptions

BlockIndent (set in the IDE in Options|Environment|Editor|Options|Block Indent indicates the number of characters to indent or outdent a block of characters. The value must be between 1 and 16. This is a read-write property.

Type expected

```
int BlockIndent
```

BufferOptions property — EditOptions

This property holds the buffer options settings for all edit buffers. This is a read-only property.

Type expected

```
BufferOptions BufferOptions
```

MirrorPath property — EditOptions

This property holds the path where the editor stores mirror copies of files in. This is a read-write property.

Type expected

```
string MirrorPath
```

OriginalPath property — EditOptions

This property holds the path where the editor stores the original files in. This is a read-write property.

Type expected

```
string OriginalPath
```

SyntaxHighlightTypes property — EditOptions

This property holds the file extensions, or file names, of the file types for which syntax highlighting is to be enabled in the editor. This is a read-write property.

Type expected

```
string SyntaxHighlightTypes
```

UseBRIEFCursorShapes property — EditOptions

When **TRUE**, the editor uses the default cursor shapes that Brief uses for insert mode and overtype mode. This is a read-write property.

Type expected

```
bool UseBRIEFCursorShapes
```

When **TRUE**, complex search and search/replace operations can be performed using the Brief regular expression syntax. This is a read-write property.

Type expected

```
bool UseBRIEFRegularExpression
```

EditPosition class

This is one of the editor classes. EditPosition class members provide positioning functionality related to the active location in an EditView or EditBuffer.

Syntax

```
EditPosition(EditBuffer)
EditPosition(EditView)
```

Properties

`int Character`	*Read-only*
`int Column`	*Read-only*
`bool IsSpecialCharacter`	*Read-only*
`bool IsWhiteSpace`	*Read-only*
`bool IsWordCharacter`	*Read-only*
`int LastRow`	*Read-only*
`int Row`	*Read-only*
`SearchOptions SearchOptions`	*Read-only*

Methods

```
void Align(int magnitude)
bool BackspaceDelete([int howMany])
bool Delete([int howMany])
int DistanceToTab(int direction)
bool GotoLine(int lineNumber)
void InsertBlock(EditBlock block)
void InsertCharacter(int characterToInsert)
void InsertFile(string fileName)
void InsertScrap()
void InsertText(string text)
bool Move([int row, int col])
bool MoveBOL()
bool MoveCursor(moveMask)
bool MoveEOF()
bool MoveEOL()
bool MoveReal([int row, int col])
bool MoveRelative([int deltaRow, int deltaCol])
string Read([int numberOfChars])
bool Replace([string pat, string rep, bool case, bool useRE, bool dir,
      int reFlavor, bool global, EditBlock block])
bool ReplaceAgain()
void Restore()
string RipText(string legalChars [,int ripFlags])
void SaveEditPosition_Save()
int SearchEditPosition_Search([string pat, bool case, bool useRE, bool
      dir, int reFlavor, EditBlock block])
int SearchAgain()
void Tab(int magnitude)
```

EditPosition class description

Most of the interesting editing activity takes place here, such as searching, text ripping, character reading, and text insertion and deletion.

Character property — EditPosition

Integer value of the character at this position, or one of the following values:

VIRTUAL_TAB
VIRTUAL_PAST_EOF
VIRTUAL_PAST_EOL

This is a read-only property.

Type expected

```
int Character
```

Column property — EditPosition

This property represents the current column position in the buffer. To change, use one of the *Move* methods. This is a read-only property.

Type expected

```
int Column
```

IsSpecialCharacter property — EditPosition

This property is **TRUE** if the character at the current edit position is not an alphanumeric or whitespace character; otherwise it is **FALSE**.

This is a read-only property.

Type expected

```
bool IsSpecialCharacter
```

IsWhiteSpace property — EditPosition

This property is **TRUE** if the character at the current edit position is a Tab or Space; otherwise it is **FALSE**.

This is a read-only property.

Type expected

```
bool IsWhiteSpace
```

IsWordCharacter property — EditPosition

This property is **TRUE** if the character at the current edit position is an alphabetic character, numeric character or underscore. Otherwise, the property is **FALSE**.

This is a read-only property.

Type expected

```
bool IsWordCharacter
```

LastRow property — EditPosition

The line number of the last line in the edit buffer. This is a read-only property.

Type expected

```
int LastRow
```

Row property — EditPosition

The property represents the current row position in the buffer. To change, use one of the *Move* methods. This is a read-only property.

Type expected

```
int Row
```

SearchOptions property — EditPosition

This property contains an instance of the *SearchOptions* class, the current search options in force.

This is a read-only property.

Type expected

```
SearchOptions SearchOptions
```

Align method — EditPosition

Positions the insertion point on the current line, aligning it with columns calculated from prior lines in the file.

Types expected

```
void Align(int magnitude)
```

Return value

None

Description

If *magnitude* is a positive value, then enough characters are inserted to align the character position as follows: Starting with a column defined by the current character position on the current line, the character position is aligned with the first character following the first white space on the previous line after the column position. If the previous line is too short to calculate a position on the current line, previous lines are scanned until finding one that is long enough to calculate a column position. If *magnitude* is negative, the column position is moved to the left.

EditPosition class, Align method example

At the start, the previous two lines contain the text "Leaning over the console, she stuck out her hand and said," and '"Hello there, buddy"', and the current line has the cursor (^)in column.

```
Leaning over the console, she stuck out her hand and said,
"How are you, buddy"
  ^
```

Calling Align(1) results in:

```
Leaning over the console, she stuck out her hand and said,
"How are you, buddy."
       ^
```

Calling `Align(1)` again results in:

```
Leaning over the console, she stuck out her hand and said,
"How are you, buddy."
              ^
```

Calling `Align(1)` again results in:

```
Leaning over the console, she stuck out her hand and said,
"How are you  buddy."
                 ^
```

Calling `Align(-1)` results in:

```
Leaning over the console, she stuck out her hand and said,
"Hello there, buddy."
              ^
```

BackspaceDelete method — EditPosition

Deletes characters to the left of the current position. The number of characters is indicated by *howMany* and defaults to 1. Returns **TRUE** if any characters are deleted or **FALSE** if there are no characters to the left.

Types expected

```
bool BackspaceDelete([int howMany])
```

Delete method — EditPosition

Deletes characters to the right of the current position. The number of characters is indicated by *howMany* and defaults to 1. Returns **TRUE** if any characters are deleted or **FALSE** if there are no characters to the left.

Types expected

```
bool Delete([int howMany])
```

DistanceToTab method — EditPosition

Retrieves the number of character positions between the current cursor position and the next/previous tab stop. *direction* is either SEARCH_FORWARD (default) or SEARCH_BACKWARD.

Types expected

```
int DistanceToTab(int direction)
```

GotoLine method — EditPosition

Moves the cursor to the line specified by *lineNumber*. Does not change the column position. Prompts the user if *lineNumber* is not supplied.

Types expected

```
bool GotoLine(int lineNumber)
```

Return value

TRUE if the move was successful, **FALSE**, otherwise

InsertBlock method — EditPosition

Inserts the last marked Editblock at the current cursor position.

Types expected

```
void InsertBlock(EditBlock block)
```

Return value

None

InsertCharacter method — EditPosition

Types expected

```
void InsertCharacter(int characterToInsert)
```

characterToInsert The integer value of the character that should be inserted.

Return value

None

InsertFile method — EditPosition

Inserts the contents of the specified file at the current cursor location.

Types expected

```
void InsertFile(string fileName)
```

Return value

None

InsertScrap method — EditPosition

Text to insert is taken from the Windows Clipboard.

Types expected

```
void InsertScrap()
```

Return value

None

InsertText method — EditPosition

Inserts the specified *text* at the current cursor position.

Types expected

```
void InsertText(string text)
```

Return value

None

Move method — EditPosition

Moves the cursor to the row and column indicated by *row* and *col*.

Types expected

```
bool Move([int row, int col])
```

Return value

The return value, **TRUE** or **FALSE**, indicates whether the position actually changed. Attempts to position either the column at 0 or less, or 1025 or more, or the line at 0 or less, or *MaxLineNumber* + 1 or more are invalid.

MoveBOL method — EditPosition

Moves the cursor to the first character on the current line.

Types expected

```
bool MoveBOL()
```

Return value

The return value, **TRUE** or **FALSE**, indicates whether the position actually changed. Attempts to position either the column at 0 or less, or 1025 or more, or the line at 0 or less, or *MaxLineNumber* + 1 or more are invalid.

MoveCursor method — EditPosition

Moves the current position forward or backward in the buffer as indicated in *moveMask*. The value of *moveMask* can be built from the one of the following: SKIP_WORD (default), SKIP_NONWORD, SKIP_WHITE, SKIP_NONWHITE, SKIP_SPECIAL, and SKIP_NOSPECIAL. These may be combined with SKIP_LEFT (default) or SKIP_RIGHT. SKIP_STREAM may also be used with any of these combinations if line ends should be ignored.

Types expected

```
bool MoveCursor(moveMask)
```

Return value

The return value, **TRUE** or **FALSE**, indicates whether the position actually changed. Attempts to position either the column at 0 or less, or 1025 or more, or the line at 0 or less, or *MaxLineNumber* + 1 or more are invalid.

MoveEOF method — EditPosition

Moves the current position to the last character in the file.

Types expected

```
bool MoveEOF()
```

Return value

The return value, **TRUE** or **FALSE**, indicates whether the position actually changed. Attempts to position either the column at 0 or less, or 1025 or more, or the line at 0 or less, or *MaxLineNumber* + 1 or more are invalid.

MoveEOL method — EditPosition

Moves the current position to the last character on the line.

Types expected

```
bool MoveEOL()
```

Return value

The return value, **TRUE** or **FALSE**, indicates whether the position actually changed. Attempts to position either the column at 0 or less, or 1025 or more, or the line at 0 or less, or *MaxLineNumber* + 1 or more are invalid.

MoveReal method — EditPosition

The position assumes that the file is unedited. If edits have been made to the file, the move is relative to the original, unedited file. For example, the original, unedited file is a two-line file with the word ONE on the first line and the word TWO on the second line. The user subsequently inserts 100 lines of text after line 1. `MoveReal(2,1)` moves the cursor to the "T" in "TWO".

Types expected

```
bool MoveReal([int row, int col])
```

Return value

The return value, **TRUE** or **FALSE**, indicates whether the position actually changed. Attempts to position either the column at 0 or less, or 1025 or more, or the line at 0 or less, or *MaxLineNumber* + 1 or more are invalid.

MoveRelative method — EditPosition

Moves the cursor *deltaRow* rows from the current row position and *deltaCol* columns from the current column position.

Types expected

```
bool MoveRelative([int deltaRow, int deltaCol])
```

Return value

The return value, **TRUE** or **FALSE**, indicates whether the position actually changed. Attempts to position either the column at 0 or less, or 1025 or more, or the line at 0 or less, or *MaxLineNumber* + 1 or more are invalid.

Read method — EditPosition

numberOfChars indicates the number of characters to read from the current position. If omitted, it reads to the end of the line.

Types expected

```
string Read([int numberOfChars])
```

Return value

Returns a string containing the characters read.

Replace method — EditPosition

This method searches in the current edit buffer and in the direction indicated for the search expression indicated in *pat* and replaced it with *rep*.

Types expected

```
bool Replace([string pat, string rep, bool case,
              bool useRE, bool dir, int reFlavor, bool
              global, EditBlock block])
```

pat	The string to search for.
rep	The string to replace with.
case	Indicates if the case of *pat* is significant in the search.
useRE	Indicates whether or not to interpret pat as a regular expression string.
dir	One of the following: SEARCH_FORWARD (default) SEARCH_BACKWARD
reFlavor	The type of regular expression being used; it may be one of the following: IDE_RE (default) BRIEF_RE BRIEF_RE_FORWARD_MIN BRIEF_RE_SAME_MIN BRIEF_RE_BACK_MIN BRIEF_RE_FORWARD_MAX BRIEF_RE_SAME_MAX BRIEF_RE_BACK_MAX
block	If given, restricts the search to the indicated block.

Return value

Either the string is replaced or the empty string ("") if nothing is found.

ReplaceAgain method — EditPosition

Repeats the most recently performed replace operation.

Types expected

```
bool ReplaceAgain()
```

Return value

Either the string is replaced or the empty string ("") if nothing is found.

Restore method — EditPosition

Restores the cursor position to the position saved by the last call to the Save method.

Types expected

```
void Restore()
```

Return value

None

RipText method — EditPosition

This method performs an edit rip operation. At most, this routine will rip an entire line. It returns the string copied from the edit buffer.

Types expected

```
string RipText(string legalChars [,int ripFlags])
```

legalChars Determines the delimiter characters that can halt the edit rip. If omitted, INCLUDE_ALPHA_CHARS, INCLUDE_NUMERIC_CHARS, INCLUDE_SPECIAL_CHARS are all automatically added to the ripFlags argument, making any character between ASCII decimal 32 and 128 a delimiter.

ripFlags A mask built by combining any or all of the following values:

BACKWARD_RIP	Rip from left to right.
INVERT_LEGAL_CHARS	Interpret the *legalChars* string as the inverse of the string you wish to use for *legalChars*. In other words, specify t to mean any ASCII value between 1 and 255 except t.
INCLUDE_LOWERCASE_ALPHA_CHARS	Append the characters `abcdefghijklmnopqrstuvwxyz` to the legalChars string.
INCLUDE_UPPERCASE_ALPHA_CHARS	Append the characters `ABCDEFGHIJKLMNOPQRSTUVWXYZ` to the legalChars string.

INCLUDE_ALPHA_CHARS	Append both ppercase and lowercase alpha characters to the *legalChars* string.
INCLUDE_NUMERIC_CHARS	Append the characters `1234567890` to the legalChars string.
INCLUDE_SPECIAL_CHARS	Append the characters `` `-=[]\;',./~!@#$%^&*()_+{}|:"<>? `` to the *legalChars* string.

Save method — EditPosition

Save the current cursor position. Use Restore to restore the cursor to this position later.

Types expected

```
void Save()
```

Return value

None

Search method — EditPosition

This method searches the edit buffer in the direction indicated for the search expression indicated in *pat*.

Types expected

```
int Search(string pat [, bool case, bool useRE,
           bool dir, int reFlavor, EditBlock block])
```

pat	The string to search for.
case	Indicates if the case of *pat* is significant in the search.
useRE	Indicates whether or not to interpret *pat* as a regular expression.
dir	One of the following: SEARCH_FORWARD (default) SEARCH_BACKWARD
reFlavor	The type of regular expression in use; it may be one of the following: IDE_RE (default) BRIEF_RE BRIEF_RE_FORWARD_MIN // same as BRIEF_RE BRIEF_RE_SAME_MIN BRIEF_RE_BACK_MIN BRIEF_RE_FORWARD_MAX BRIEF_RE_SAME_MAX BRIEF_RE_BACK_MAX
block	If given, restricts the search to the indicated block.

☞ If *case*, *useRE*, or *reFlavor* is not supplied, the value is determined by querying the *Editor* object.

Return value

The size (in characters matched) of the match.

SearchAgain method — EditPosition

Repeats the most recently performed search operation.

Types expected

```
int SearchAgain()
```

Tab method — EditPosition

Moves the current cursor location to the next or previous tab stop, depending on whether *magnitude* is positive (next) or negative (previous).

Types expected

```
void Tab(int magnitude)
```

Search expression definition — EditPosition

A search expression is a set of characters that the search engine will attempt to match against the text contained in an edit buffer. A search expression can be either a literal string or a regular expression.

- In a literal string, there are no operators: Each character is treated literally.
- In a regular expression, certain characters have special meanings: They are operators that govern the search.

A regular expression is either a single character or a set of characters enclosed in brackets. A concatenation of regular expressions is a regular expression.

Regular expressions have two formats, IDE and Brief.

Brief regular search expression symbols

The following table describes the symbols that can be used in a Brief search expression:

Symbol	Description
?	Any single character except a newline
*	Zero or more characters (except newlines)
\t	Tab character
\n	Newline character
\c	Position cursor after the match
\\	Literal backslash
< or %	Beginning of line
> or $	End of line
@	Zero or more of the last expression
+	One or more of the last expression

|	Either the last or the next expression
{}	Define a group of expressions
[ ]	Any one of the characters inside `[ ]`
[~]	Any character except those in `[~ ]`
[a-z]	Any character between a and z, inclusive

In replacement text, `\t`, `\n`, and `\c` are allowed, as well as

`\<n>` Substitute text matched by `<n>`th group (`0 <= n <= 9`)

IDE search expression symbols

The following table describes the symbols that can be used in an IDE search expression:

Symbol	Description
^	A circumflex at the start of the expression matches the start of a line.
$	A dollar sign at the end of the expression matches the end of a line.
.	A period matches any single character.
*	An asterisk after a string matches any number of occurrences of that string followed by any characters, including zero characters. For example, `bo*` matches `bot`, `boo`, and `bo`.
+	A plus sign after a string matches any number of occurrences of that string followed by any characters, except zero characters. For example, `bo+` matches `bot` and `boo`, but not `b` or `bo`.
{ }	Characters or expressions in braces are grouped to allow for controlling evaluation of a search pattern or referring to grouped text by number.
[ ]	Characters in brackets match any single character that appears in the brackets, but no others. For example `[bot]` matches `b`, `o`, or `t`.
[^]	A circumflex at the start of the string in brackets means **NOT**. For example, `[^bot]` matches any characters except `b`, `o`, or `t`.
[-]	A hyphen in brackets signifies a range of characters. For example, `[b-o]` matches any character from `b` through `o`.
\	A backslash before a special character indicates that the character is to be interpreted literally. For example, `\^` matches `^` and does not indicate the start of a line.

Edit rip definition

Edit rip is the process by which the editor detects the current cursor position and copies characters from the edit buffer. The IDE editor uses edit rip to find relevant Help topics when the user press *F1* in an edit window.

EditStyle class

This class is one of the editor classes. EditStyle applies styles that overrride settings for a buffer or for the entire editor.

Syntax

```
EditStyle(string styleName[,EditStyle styleToInitializeFrom])
```

Properties

`EditOptions EditMode` *Read-write*

`int Identifier` *Read-only*

`string Name` *Read-write*

EditStyle class description

EditStyle objects provide a mechanism to collect *EditOptions*, name them, and apply them across buffers, across the entire Editor, or both. *EditStyle* objects contain an *EditOptions* member, a name, and an internal filter that indicates the characteristics that the style controls. *EdStyles* are implicitly persistent. The list of available styles may be traversed from the *Editor* object.

EditMode property — EditStyle

This property contains an EditOptions object that defines the options for the style. This is a read-write property.

Type expected

```
EditOptions EditMode
```

Identifier property — EditStyle

A unique integer you can use to distinguish styles. This is a read-only property.

Type expected

```
int Identifier
```

Name property — EditStyle

A unique name for this EditStyle. The value is taken from the *styleName* parameter. This is a read-write property.

Type expected

```
string Name
```

EditView class

This class is one of the editor classes. It provides the visual representation of the *EditBuffer* to the user within the edit window. Each edit view has only one edit buffer and is in an edit window.

Syntax

```
EditView (EditWindow parent[, EditBuffer buffer])
```

If *buffer* is omitted, the parent's currently active *EditBuffer* is used.

Properties

`EditBlock Block` *Read-write*

`int BottomRow` *Read-only*

`EditBuffer Buffer` *Read-only*

`int Identifier` *Read-only*

`bool IsValid` *Read-only*

`bool IsZoomed`	*Read-write*
`int LastEditColumn`	*Read-only*
`int LastEditRow`	*Read-only*
`int LeftColumn`	*Read-only*
`EditView Next`	*Read-only*
`EditPosition Position`	*Read-only*
`EditView Prior`	*Read-only*
`int RightColumn`	*Read-only*
`int TopRow`	*Read-only*
`EditWindow Window`	*Read-only*

Methods

```
EditBuffer Attach(EditBuffer buffer)
bool BookmarkGoto(int bookmarkIDorPrevRef)
int BookmarkRecord(int bookmarkIDorPrevRef)
void Center([int row, int col])
void MoveCursorToView()
void MoveViewToCursor()
void PageDown()
void PageUp()
void Paint()
int Scroll(int deltaRow[, int deltaCol])
void SetTopLeft(int topRow, int leftCol)
```

EditView class description

EditView objects provide an editing window for a buffer being edited. Each *EditView* is associated with only one EditWindow and one EditBuffer. During creation, the *EditView's* Position member is initialized from the *EditBuffer's* Position member. Views have methods that allow traversal of their sibling views. They can also be queried to find the associated *EditWindow* or *EditBuffer*. Views have a *Position* member you can use to manipulate the underlying *EditBuffer*. Typically this member is used by scripts and primitives tied to the user interface. Although the underlying *EditBuffer* object owns the list of bookmarks, the *EditView* object provides access (via *record* and *goto*), thus providing access to a common list of bookmarks for the same buffer regardless of the view being used.

Block property — EditView

This property provides access to the instance of the *EditBlock* class attached to this *EditView*. This is a read-only property.

Type expected

```
EditBlock Block
```

BottomRow property — EditView

Row number displayed at the last line in the view. This is a read-only property.

Type expected

```
int BottomRow
```

Buffer property — EditView

Returns the EditBuffer to which the view is attached. This is a read-only property.

Type expected

```
EditBuffer Buffer
```

Identifier property — EditView

A unique identifier you can use to tell views apart. This is a read-only property.

Type expected

```
int Identifier
```

IsValid property — EditView

The view will be invalidated if it is destroyed by the user. This is a read-only property.

Type expected

```
bool IsValid
```

IsZoomed property — EditView

A zoomed *EditView* is an *EditView* that has been expanded to occupy the entire *EditWindow* client space. If an *EditView* is zoomed in an *EditWindow*, you can't manipulate sibling views. (You can't create, resize, or delete them.) This is a read-write property.

Type expected

```
bool IsZoomed
```

LastEditColumn property — EditView

LastEditRow and LastEditColumn describe the character position at which the last edit took place. An edit is defined as being a character or block insertion or a deletion (anything that modifies the contents of the buffer). This is a read-only property.

Type expected

```
int LastEditColumn
```

LastEditRow property — EditView

LastEditRow and LastEditColumn describe the character position at which the last edit took place. An edit is defined as being a character or block insertion or a deletion (anything that modifies the contents of the buffer). This is a read-only property.

Type expected

```
int LastEditRow
```

LeftColumn property — EditView

Column number displayed at the left edge of the view. This is a read-only property.

Type expected

```
int LeftColumn
```

Next property — EditView

The next *EditView* embedded in the same window. This is a read-only property.

Type expected

```
EditView Next
```

Position property — EditView

Provides access to the instance of the *EditPosition* class attached to this *EditView*. This is a read-only property.

Type expected

```
EditPosition Position
```

Prior property — EditView

The previous *EditView* embedded in the same window. This is a read-only property.

Type expected

```
EditView Prior
```

RightColumn property — EditView

Column number displayed at the right edge of the view. This is a read-only property.

Type expected

```
int RightColumn
```

TopRow property — EditView

Row number displayed at the first line in the view. This is a read-only property.

Type expected

```
int TopRow
```

Window property — EditView

Returns the window in which this view is embedded. This is a read-only property.

Type expected

```
EditWindow Window
```

Attach method — EditView

This method attaches the view to a new EditBuffer.

Types expected

```
EditBuffer Attach(EditBuffer buffer)
```

Return value

The previous edit buffer.

Description

You can *Attach* to replace the currently attached edit buffer. When a view is created, it is associated with an *EditBuffer*. The purpose of the view is to provide a visual representation of the edit buffer to which it is attached. Suppose you have a view that you like (for example, you like its position in the creation history, its size and color, its parent window, and everything else about it except that you want it to display a different edit buffer). Instead of having to remember all these things, you can destroy the current view, and recreate it all, and use *Attach* to switch its associated buffer to another edit buffer.

BookmarkGoto method — EditView

This method updates the EditBuffer's position with the value from the specified marker.

Types expected

```
bool BookmarkGoto(int bookmarkIDorPrevRef)
```

bookmarkIDorPrevRef — Either an index (range 0-19) to the list of bookmarks in the buffer or a reference to a bookmark that was returned from a previous call to BookmarkRecord.

Return value

TRUE if the marker is valid, or **FALSE** if it is not.

BookmarkRecord method — EditView

This method returns a value suitable for passing to BookmarkGoto() or zero if there was an error.

Types expected

```
int BookmarkRecord(int bookmarkIDorPrevRef)
```

bookmarkIDorPrevRef — Either an index (range 0-19) to the list of bookmarks in the buffer or a reference to a bookmark that was returned from a previous call to *BookmarkRecord*.

Description

You can use *BookmarkRecord*() to remember a known location in a buffer. The bookmark moves with edit inserts and deletes. For example, if you placed a bookmark using `BookMarkRecord(1)` at the a in "are" in the following line, you could move around and then return to that location with `BookmarkGoto(1)`:

```
hello how are you?
```

If the word "how" were deleted, you would still return to the a in "are".

Center method — EditView

Scrolls the *EditView* as necessary to center the character in the view window. Centers the character at the specified position vertically or horizontally or both. A zero in a parameter means to leave that orientation alone. If the character is already centered, nothing happens.

Types expected

```
void Center([int row, int col])
```

Return value

None

MoveCursorToView method — EditView

This method ensures that the cursor is visible in the view by altering the cursor's position, if necessary.

Types expected

```
void MoveCursorToView()
```

Return value

None

MoveViewToCursor method — EditView

This method ensures that the cursor is visible in the view by altering the view's coordinates, if necessary.

Types expected

```
void MoveViewToCursor()
```

Return value

None

PageDown method — EditView

This method advances the row position by the number of visible rows in the *EditView*.

Types expected

```
void PageDown()
```

Return value

None

PageUp method — EditView

This method moves the cursor toward the top of the buffer by the number of lines in the visible rows in the *EditView*.

Types expected

```
void PageUp()
```

Return value

None

Paint method — EditView

During normal script execution, screen updates are suppressed. Calling *Paint* forces a refresh.

Types expected

```
void Paint()
```

Return value

None

Scroll method — EditView

This method scrolls in the direction indicated and returns the number of lines actually scrolled. The parameter values indicate and magnitude: A value less than 0 means scroll up or left or both by that number of lines or columns or both, and a value greater than 0 means scroll down or right or both by that number of lines or columns or both.

Types expected

```
int Scroll(int deltaRow[, int deltaCol])
```

Return value

None

SetTopLeft method — EditView

Attempts to position the character at the specified position in the upper left corner of the *EditView*. Might fail if the position is outside the window's bounds. A zero in either *topRow* or *leftCol* means to ignore the position request in that dimension (just set the top or just set the left). A zero in both parameters causes the method to be ignored altogether.

Types expected

```
void SetTopLeft(int topRow, int leftCol)
```

Return value

None

EditWindow class

This class is one of the editor classes. It provides control of editor views.

Syntax

```
EditWindow(EditBuffer buffer)
```

Properties

`int Identifier`	*Read-only*
`bool IsHidden`	*Read-write*
`bool IsValid`	*Read-only*
`EditWindow Next`	*Read-only*
`EditWindow Prior`	*Read-only*
`string Title`	*Read-write*
`EditView View`	*Read-only*

Methods

```
void Activate()
void Close()
void Paint()
EditView ViewActivate(int direction[, EditView srcView])
EditView ViewCreate(int direction[, EditView srcView])
bool ViewDelete(int direction[, EditView srcView])
EditView ViewExists(int direction[, EditView srcView])
void ViewSlide(int direction[, int magnitude, EditView srcView])
```

EditWindow class description

EditWindow objects provide little functionality except for the management of panes (views). Creation of an *EditWindow* does not cause a window to appear; rather, it provides an object to which a view may be attached. As soon as the first view is attached to an *EditWindow*, it can be displayed. Views may be zoomed, in which case they expand to fill the client area of their *EditWindow*, hiding all sibling views (those embedded in the same *EditWindow*). As long as an *EditWindow* contains a zoomed view, views can't be created, destroyed or resized. *EditWindows* can be hidden and unhidden to allow the user to free screen space and preserve the view layout in the hidden *EditWindows*.

Identifier property — EditWindow

This property is a unique value you can use to tell windows apart. It is a read-write property.

Type expected

```
int Identifier
```

IsHidden property — EditWindow

This property indicates if the current *EditWindow* is hidden. It is a read-only property.

Type expected

```
bool IsHidden
```

IsValid property — EditWindow

This property is **TRUE** if the current *EditWindow* is ready for edit operations. It is **FALSE** if the window is not available (for example, it is closed). It is a read-only property.

Type expected

```
bool IsValid
```

Next property — EditWindow

This property indicates the next *EditWindow*, if any. It is a read-only property.

Type expected

```
EditWindow Next
```

Prior property — EditWindow

This property indicates the previous EditWindow, if any. It is a read-only property.

Type expected

```
EditWindow Prior
```

Title property — EditWindow

This property indicates the title of the current EditWindow. It is a read-write property.

Type expected

```
string Title
```

View property — EditWindow

This property indicates the current EditView. It is a read-only property.

Type expected

```
EditView View
```

Activate method — EditWindow

This method brings this window to the top and gives it focus.

Types expected

```
void Activate()
```

Return value

None

Close method — EditWindow

This method closes the current window.

Types expected

```
void Close()
```

Return value

None

Paint method — EditWindow

During normal script execution screen updates are suppressed. Calling Paint forces a refresh.

Types expected

```
void Paint()
```

Return value

None

ViewActivate method — EditWindow

This method makes an existing view the current, active view.

Types expected

```
EditView ViewActivate(int direction[, EditView srcView])
```

direction — Relative to the current *EditView* in an *EditWindow*. Can be one of the following values:

UP

DOWN

LEFT

RIGHT

srcView — If omitted, the *EditWindow's* current *EditView* is activated.

Return value

The newly activated view or NULL if no view exists.

ViewCreate method — EditWindow

This method creates an *EditView*.

Types expected

```
EditView ViewCreate(int direction[, EditView srcView])
```

direction — Relative to the existing *EditViews* in an *EditWindow* and ignored for the first view. Can be one of the following values:

UP
DOWN
LEFT
RIGHT

srcView The view to create. If omitted, the *EditWindow's* current *EditView* is used. The newly created *EditView* by default is not activated.

Return value

The new *EditView* or NULL if creation failed

ViewDelete method — EditWindow

This method deletes the view in the *direction* relative to the *srcView*, if any. If no *srcView* is specified, the currently active EditView in the EditWindow is deleted. The target view (if any) is then removed from the EditWindow. *srcView* is then resized to occupy the space previously held by the target view.

Types expected

```
bool ViewDelete(int direction[, EditView srcView])
```

direction Relative to the existing *EditViews* in an *EditWindow* and ignored for the first view. Can be one of the following values:

UP
DOWN
LEFT
RIGHT

srcView If omitted, the *EditWindow's* current *EditView* is deleted.

Return value

TRUE if the view was deleted or **FALSE** if it was not.

ViewExists method — EditWindow

Get a reference to an adjoining *EditView*, if the adjoining *EditView* exists.

Types expected

```
EditView ViewExists(int direction[, EditView srcView])
```

direction Relative to the current *EditView* in an *EditWindow*. Can be one of the following values:

UP
DOWN
LEFT
RIGHT

srcView If omitted, the *EditWindow's* current *EditView* is used.

Return value

The *EditView* or NULL if it doesn't exist.

ViewSlide method — EditWindow

Moves the view in the direction indicated.

Types expected

```
void ViewSlide(int direction[, int magnitude,
               EditView srcView])
```

direction — Relative to the existing *EditViews* in an *EditWindow*. Can be one of the following values:

UP
DOWN
LEFT
RIGHT

magnitude — The direction (+ or -) and amount to move

srcView — If omitted, the *EditWindow's* current *EditView* is used.

Return value

None

IDEApplication class

This class represents the Paradigm C++ Integrated Development Environment (IDE). An *IDEApplication* object called *IDE* is instantiated when Paradigm C++ starts up. You typically use this class to determine how to use or extend this IDE object.

Syntax

```
IDEApplication()
```

Properties

`string Application`	*Read-only*
`string Caption`	*Read-write*
`string CurrentDirectory`	*Read-only*
`string CurrentProjectNode`	*Read-only*
`string DefaultFilePath`	*Read-write*
`Editor Editor`	*Read-only*
`string FullName`	*Read-only*
`int Height`	*Read-write*
`int IdleTime`	*Read-only*
`int IdleTimeout`	*Read-write*
`int LoadTime`	*Read-only*
`string KeyboardAssignmentFile`	*Read-write*
`KeyboardManager KeyboardManager`	*Read-only*
`int Left`	*Read-write*
`string ModuleName`	*Read-only*
`string Name`	*Read-only*
`string Parent`	*Read-only*
`bool RaiseDialogCreatedEvent`	*Read-write*
`string StatusBar`	*Read-write*
`int Top`	*Read-write*
`bool UseCurrentWindowForSourceTracking`	*Read-write*
`int Version`	*Read-only*
`bool Visible`	*Read-write*
`int Width`	*Read-write*

Methods

```
void AddToCredits()
bool CloseWindow()
bool DebugAddBreakpoint()
bool DebugAddWatch()
bool DebugAnimate()
bool DebugAttach()
```

```
bool DebugBreakpointOptions()
string DebugEvaluate()
bool DebugInspect()
bool DebugInstructionStepInto()
bool DebugInstructionStepOver()
bool DebugLoad()
bool DebugPauseProcess()
bool DebugResetThisProcess()
bool DebugRun()
bool DebugRunTo()
bool DebugSourceAtExecutionPoint()
bool DebugStatementStepInto()
bool DebugStatementStepOver()
bool DebugTerminateProcess()
int DirectionDialog(string prompt)
string DirectoryDialog(string prompt, string initialValue)
void DisplayCredits()
bool DoFileOpen(string filename, string toolName [, ProjectNode node])
bool EditBufferList()
bool EditCopy()
bool EditCut()
bool EditPaste()
bool EditRedo()
bool EditSelectAll()
bool EditUndo()
void EndWaitCursor()
void EnterContextHelpMode()
void ExpandWindow()
bool FileClose()
string FileDialog(string prompt, string initialValue)
bool FileExit( [int IDEReturn] )
bool FileNew([string toolName, string fileName])
bool FileOpen([string name, string toolName])
bool FilePrint(bool suppressDialog)
bool FilePrinterSetup()
bool FileSave()
bool FileSaveAll()
bool FileSaveAs([string newName])
bool FileSend()
int GetRegionBottom(string RegionName)
int GetRegionLeft(string RegionName)
int GetRegionRight(string RegionName)
int GetRegionTop(string RegionName)
bool GetWindowState()
void Help(string helpFile, int command, string  helpTopic)
bool HelpAbout()
bool HelpContents()
bool HelpKeyboard()
bool HelpKeywordSearch([string keyword])
bool HelpUsingHelp()
bool HelpWindowsAPI()
string KeyPressDialog(string prompt, string default)
string[ ] ListDialog(string prompt, bool multiSelect, bool sorted,
      string [ ] initialValues)
void Menu()
bool Message(string text, int severity)
int MessageCreate(string destinationTab, string toolName, int
      messageType, int parentMessage, string filename, int lineNumber,
      int columnNumber, stringtext, string helpFileName, int
      helpContextId)
bool NextWindow(bool priorWindow)
```

```
bool OptionsEnvironment()
bool OptionsProject()
bool OptionsSave()
bool OptionsStyleSheets()
bool OptionsTools()
bool ProjectBuildAll([bool suppressOkay, string nodeName])
bool ProjectCloseProject()
bool ProjectCompile([string nodeName])
bool ProjectGenerateMakefile([string nodeName])
bool ProjectMakeAll([bool suppressOkay, string nodeName])
bool ProjectManagerInitialize()
bool ProjectNewProject([string pName])
bool ProjectNewTarget( [string nTarget, int targetType, int platform,
int libraryMask, int modelOrMode] )
bool ProjectOpenProject([string pName])
void Quit()
bool SaveMessages(string tabName, string fileName)
bool ScriptCommands()
bool ScriptCompileFile(string fileName)
bool ScriptModules()
bool ScriptRun([string command])
bool ScriptRunFile([string filename])
bool SearchBrowseSymbol([string sName])
bool SearchFind([string pat])
bool SearchLocateSymbol([string sName])
bool SearchNextMessage()
bool SearchPreviousMessage()
bool SearchReplace([string pat, string rep])
bool SearchSearchAgain()
bool SetRegion(string RegionName, int left, int top, int right, int
      bottom)
bool SetWindowState(int desiredState)
string SimpleDialog(string prompt, string initialValue [, int
      maxNumChars])
void SpeedMenu()
void StartWaitCursor()
string StatusBarDialog(string prompt, string initialValue [, int
      maxNumChars])
bool Tool([string toolName, string commandstring])
void Undo()
bool ViewActivate(int direction)
bool ViewBreakpoint()
bool ViewCallStack()
bool ViewClasses()
bool ViewCpu()
bool ViewGlobals()
bool ViewMessage([string tabName])
bool ViewProcess()
bool ViewProject()
bool ViewSlide(int direction [, int amount])
bool ViewWatch()
bool WindowArrangeIcons()
bool WindowCascade()
bool WindowCloseAll([string typeName])
bool WindowMinimizeAll([string typeName])
bool WindowRestoreAll([string typeName])
bool WindowTileHorizontal()
bool WindowTileVertical()
string YesNoDialog(string prompt, string default)
```

Events

```
void BuildComplete(bool status, string inputPath, string OutputPath)
void BuildStarted()
void DialogCreated(string dialogName, int dialogHandle)
void Exiting()
void HelpRequested(string filename, int command, int data)
void Idle()
void KeyboardAssignmentsChanged(string newFilename)
void KeyboardAssignmentsChanging(string newFilename)
void MakeComplete(bool status, string inputPath, string outputPath)
void MakeStarted()
void ProjectClosed(string projectFileName)
void ProjectOpened(string projectFileName)
void SecondElapsed()
void Started(bool VeryFirstTime)
void SubsytemActivated(string systemName)
bool TransferOutputExists(TransferOutput output)
void TranslateComplete(bool status, string inputPath, string
outputPath)
```

Application property — IDEApplication

This property contains the IDEApplication object's internal name. The name is for use by Windows and its presence is required by Microsoft conventions. This is a read-only property.

Type expected

```
string Application
```

Caption property — IDEApplication

This property indicates the caption of the Paradigm C++ IDE main window. This is a read-write property.

Type expected

```
string Caption
```

CurrentDirectory property — IDEApplication

This property indicates the application's current directory. Whenever a project file is opened, the value of *CurrentDirectory* changes to the directory containing the project file. This is a read-only property.

Type expected

```
string CurrentDirectory
```

CurrentProjectNode property — IDEApplication

This property contains the name of the currently selected node in the Project window. If the Project window is not open, or if multiple nodes are selected in the Project window, *CurrentProjectNode* contains an empty string (""). This is a read-only property.

Type expected

```
string CurrentProjectNode
```

DefaultFilePath property — IDEApplication

This property indicates the default file path for the Paradigm C++ IDE. This is a read-only property.

Type expected

```
string DefaultFilePath
```

Editor property — IDEApplication

This property is an instance of the Paradigm C++ IDE editor. This is a read-only property.

Type expected

```
Editor Editor
```

FullName property — IDEApplication

This property contains the string, "Paradigm C++ for Windows, vers. 5.0". This is a read-only property.

Type expected

```
string FullName
```

Height property — IDEApplication

This property indicates the height of the Paradigm C++ IDE main window. This is a read-only property.

Type expected

```
int Height
```

IdleTime property — IDEApplication

This property indicates the number of seconds since the last user-generated event. This is a read-only property.

Type expected

```
int IdleTime
```

IdleTimeout property — IDEApplication

This property specifies the number of seconds the IDE must remain idle before an idle event will be generated. It defaults to 180 (3 minutes). This is a read-write property.

Type expected

```
int IdleTimeout
```

LoadTime property — IDEApplication

The number of milliseconds it takes for the IDE to load. The number reflects time up through the processing of the startup script. Thereafter it remains fixed. This is a read-only property.

Type expected

```
int LoadTime
```

KeyboardAssignmentFile property — IDEApplication

This property indicates the file name of the keyboard file (.KBD) last selected from the Options|Environment|Editor dialog. This is a read-write property.

Type expected

```
string KeyboardAssignmentFile
```

KeyboardManager property — IDEApplication

This property is an instance of the Paradigm C++ IDE keyboard manager. This is a read-only property.

Type expected

```
KeyboardManager KeyboardManager
```

Left property — IDEApplication

Indicates the left coordinate of the IDE's main window. This is a read-write property.

Type expected

```
int Left
```

ModuleName property — IDEApplication

This property indicates the module name of the running application, including its path. For example, "`c:\paradigm\bin\pcw.exe`". This is a read-only property.

Type expected

```
string ModuleName
```

Name property — IDEApplication

This property indicates the name of the Paradigm C++ IDE, "PCW". This is a read-only property.

Type expected

```
string Name
```

Parent property — IDEApplication

Contains a value required by Windows. Presence required by Microsoft conventions. This is a read-only property.

Type expected

```
string Parent
```

RaiseDialogCreatedEvent property — IDEApplication

This property's value is initialized to **FALSE**. Setting it to **TRUE** causes the *DialogCreated* event to be raised whenever a new dialog is created. This is a read-write property.

Type expected

```
bool RaiseDialogCreatedEvent
```

StatusBar property — IDEApplication

This property is used to gets or set the text displayed in the IDE's status bar. This is a read-write property.

Type expected

```
bool StatusBar
```

Top property — IDEApplication

This property contains the top coordinate of the IDE main window. This is a read-write property.

Type expected

```
int Top
```

UseCurrentWindowForSourceTracking property — IDEApplication

This property, if **TRUE**, the IDE replaces the contents of the active Edit window whenever a new file is loaded. If **FALSE**, the IDE opens a new Edit window. This is a read-write property.

Type expected

```
bool UseCurrentWindowForSourceTracking
```

Version property — IDEApplication

This property holds the value 500 for Paradigm C++ version 5.0. This is a read-only property.

Type expected

```
int Version
```

Visible property — IDEApplication

This property, if TRUE, makes the IDE visible to the user. If FALSE, the IDE is not visible on the screen. This is a read-write property.

Type expected

```
bool Visible
```

Width property — IDEApplication

This property contains the width of the IDE's main window. This is a read-write property.

Type expected

```
int Width
```

AddToCredits method — IDEApplication

This method adds a name to the list of developer credits in the About dialog box. It adds the new name to the end of the existing list. To display developer credits, choose Help|About and press Alt-I.

Types expected

```
void AddToCredits()
```

CloseWindow method — IDEApplication

This method closes the currently selected IDE child window.

Types expected

```
bool CloseWindow()
```

Return value

TRUE if the window closed, **FALSE** if unable to close the window.

DebugAddBreakpoint method — IDEApplication

This method opens the Add Breakpoint dialog.

Types expected

```
bool DebugAddBreakpoint()
```

Return value

TRUE if successful, **FALSE** otherwise

DebugAddWatch method — IDEApplication

This method adds a watch on the current symbol.

Types expected

```
bool DebugAddWatch()
```

Return value

TRUE if successful, **FALSE** otherwise

DebugAnimate method — IDEApplication

This method lets you watch your program's execution in "slow motion."

Types expected

```
bool DebugAnimate()
```

Return value

TRUE if successful, **FALSE** otherwise

Description

Animate performs a continuous series of Statement Step Into commands. To interrupt animation, invoke one of the following Debugger methods either by menu selections or by keystrokes tied to the script:

Debugger.Run
Debugger.RunToAddress
Debugger.RunToFileLine
Debugger.PauseProgram
Debugger.Reset
Debugger.TerminateProgram
Debugger.FindExecutionPoint

DebugAttach method — IDEApplication

This method invokes the debugger for the currently executing process.

Types expected

```
bool DebugAttach()
```

Return value

TRUE if successful, **FALSE** otherwise

DebugBreakpointOptions method — IDEApplication

This method opens the Breakpoint Condition/Action Options dialog.

Types expected

```
bool DebugBreakpointOptions()
```

Return value

TRUE if successful, FALSE, otherwise

DebugEvaluate method — IDEApplication

This method evaluates the current expression, such as a global or local variable or an arithmetic expression.

Types expected

```
string DebugEvaluate()
```

Return value

Returns the result of the evaluation.

DebugInspect method — IDEApplication

This method attempts to opens an inspector for the current symbol.

Types expected

```
bool DebugInspect()
```

Return value

TRUE if successful, **FALSE** otherwise.

DebugInstructionStepInto method — IDEApplication

This method executes the next instruction, stepping into any function calls. If a process is not loaded, *InstructionStepInto* first loads the executable for the current project.

Types expected

```
bool DebugInstructionStepInto()
```

Return value

TRUE if successful, **FALSE** otherwise.

DebugInstructionStepOver method — IDEApplication

This method executes the next instruction, running any functions called at full speed. If a process is not loaded, *InstructionStepOver* first loads the executable for the current project.

Types expected

```
bool DebugInstructionStepOver()
```

Return value

TRUE if successful, **FALSE** otherwise.

DebugLoad method — IDEApplication

This method loads the current executable into the debugger.

Types expected

```
bool DebugLoad([string fileToLoad])
```

Return value

TRUE if successful, **FALSE** otherwise.

Description

Upon loading, the process is run to the starting point as specified in the Options|Environment|Debugger|Debugger Behavior dialog. If the parameter is **NULL**, this method opens the Load Program dialog.

DebugPauseProcess method — IDEApplication

This method causes the debugger to pause the current process. It has an effect only if the current process is running or is animated.

Types expected

```
bool DebugPauseProcess()
```

Return value

TRUE if successful, **FALSE** otherwise.

DebugResetThisProcess method — IDEApplication

This method causes the debugger to reset the current process to its starting point as specified in the Options|Environment|Debugger|Debugger Behavior dialog.

Types expected

```
bool DebugResetThisProcess()
```

Return value

TRUE if successful, **FALSE** otherwise.

DebugRun method — IDEApplication

This method causes the debugger to run the current process. If no process is loaded, this method first loads the executable associated with the current project.

Types expected

```
bool DebugRun()
```

Return value

TRUE if successful, **FALSE** otherwise.

DebugRunTo method — IDEApplication

If this method is called while working with an *EditView*, the method runs the current process until the source at the current line in the current file is encountered. If the current object is not an *EditView* the method runs the current process until the instruction at the current address is encountered. If no process is loaded, this method will first load the executable associated with the current project.

Types expected

```
bool DebugRunTo()
```

Return value

TRUE if successful, **FALSE** otherwise.

DebugSourceAtExecutionPoint method — IDEApplication

This method displays the source code at the current execution point.

Types expected

```
bool DebugSourceAtExecutionPoint()
```

Return value

TRUE if successful, **FALSE** otherwise.

Description

The current execution point is indicated by the EIP register. If the current execution point is in source code, the execution point is shown in an edit window. (The appropriate source file is opened if necessary.) If the current execution point is at an address that has no source associated with it, the execution point is shown in a CPU view. (One is opened if necessary.)

DebugStatementStepInto method — IDEApplication

This method executes the next source statement and steps through the source of any function calls. If a process is not loaded, the method first loads the executable for the current project.

Types expected

```
bool DebugStatementStepInto()
```

Return value

TRUE if successful, **FALSE** otherwise.

DebugStatementStepOver method — IDEApplication

This method executes the next source statement and does not step into any functions called, but runs them at full speed. If a process is not loaded, *StatementStepOver* first loads the executable for the current project.

Types expected

```
bool DebugStatementStepOver()
```

Return value

TRUE if successful, **FALSE** otherwise.

DebugTerminateProcess method — IDEApplication

This method terminates the current process. If no process is loaded, this method has no effect.

Types expected

```
bool DebugTerminateProcess()
```

Return value

TRUE if successful, **FALSE** otherwise.

DirectionDialog method — IDEApplication

This method invokes a dialog that allows the user to specify a direction.

Types expected

```
int DirectionDialog(string prompt)
```

Return value

One of the following values: CANCEL, RIGHT, LEFT, UP, DOWN.

DirectoryDialog method — IDEApplication

This method invokes a directory-browsing dialog box that allows the user to choose a directory.

Types expected

```
string DirectoryDialog(string prompt, string initialValue)
                       [, string pathSpecifier])
```

prompt	The value to place in the caption of the dialog.
initialValue	The value to initialize the edit field with.
pathSpecifier	The directory in which to start browsing. If this parameter is not specified, the current directory is used.

Return value

If successful, this method returns a fully qualified directory name. If the user cancels, it returns the empty string ("").

DisplayCredits method — IDEApplication

This method displays the list of developer credits in the About dialog box. To display developer credits, choose Help|About and press Alt-I.

Types expected

```
void DisplayCredits()
```

DoFileOpen method — IDEApplication

This method is used by the *FileOpen* method to open files. If the specified *filename* does not exist, it is created. The *node* argument is passed if the file is to be associated with a specific node in the project.

Types expected

```
bool DoFileOpen(string fileName, string toolName
                [, ProjectNode node])
```

Return value

TRUE is successful, **FALSE** otherwise.

EditBufferList method — IDEApplication

This method displays the buffer list dialog for the user.

Types expected

```
bool EditBufferList()
```

Return value

TRUE if the buffer list was successfully edited or **FALSE** if no edit buffers exist.

EditCopy method IDEApplication

This method copies selected text from the current edit buffer to the Windows clipboard.

Types expected

```
bool EditCopy()
```

Return value

TRUE if the topmost window is an *EditView* with a valid marked block, **FALSE** otherwise.

EditCut method IDEApplication

This method copies selected text from the current edit buffer to the clipboard and deletes the selected text.

Types expected

```
bool EditCut()
```

Return value

TRUE if the topmost window is an *EditView* with a valid marked block, **FALSE** otherwise.

EditPaste method IDEApplication

This method copies selected text from the Clipboard to the current edit position in the current edit buffer.

Types expected

```
bool EditPaste()
```

Return value

TRUE if the topmost window is an *EditView* with a valid marked block, **FALSE** otherwise.

EditRedo method IDEApplication

This method reapplies the operation that was undone with the last *EditUndo*.

Types expected

```
bool EditRedo()
```

Return value

TRUE if the operation was successful or **FALSE** if it is not.

EditSelectAll method — IDEApplication

This method selects all the text in the current edit buffer.

Types expected

```
bool EditSelectAll()
```

Return value

TRUE if the select was successful or **FALSE** if it is not.

EditUndo method — IDEApplication

This method undoes the last edit operation.

Types expected

```
bool EditUndo()
```

Return value

TRUE if the operation was successful or **FALSE** if it is not.

EndWaitCursor method — IDEApplication

This method stops the display of the Windows wait cursor (by default, an hourglass).

Types expected

```
void EndWaitCursor()
```

Return value

None

EnterContextHelpMode method — IDEApplication

Calling this method puts the IDE in help context mode: The next click of the mouse generates a help event for whatever the mouse pointer is on.

Types expected

```
void EnterContextHelpMode()
```

Return value

None

ExpandWindow method — IDEApplication

This method increases the size of the currently selected window to its maximum *view managed* size, defined by calls to SetRegion. After the window has been expanded with this method, there is no way to decrease its size.

Types expected

```
void ExpandWindow()
```

Return value
None

FileClose method — IDEApplication

This method closes the file that is currently open and selected.

Types expected
```
bool FileClose()
```

Return value
TRUE if the file was successfully closed, or **FALSE** if the file could not be closed.

FileDialog method — IDEApplication

Invokes a File Open dialog and allows the user to choose a file.

Types expected
```
string FileDialog(string prompt, string initialValue)
```

Return value
This method returns a fully qualified file name if successful. If the user cancels, the method returns the empty string ("").

FileExit method — IDEApplication

This method closes the application after first ensuring that all files are saved.

Types expected
```
bool FileExit( [int IDEReturn] )
```

IDEReturn The return value of the IDE application when it exits. By default, this value is 0.

Return value
TRUE if the application was closed or **FALSE** if it could not be closed.

FileNew method — IDEApplication

This method creates a new file for the tool specified in *toolName*.

Types expected
```
bool FileNew([string toolName, string fileName])
```

Return value
TRUE if the file was created or **FALSE** if the file could not be created.

FileOpen method — IDEApplication

This method opens a file. If the file specified by *name* doesn't exist, the user is prompted for a file name. Internally, this method uses *DoFileOpen*.

Types expected

```
bool FileOpen([string name, string toolName])
```

Return value

TRUE if the file was opened or **FALSE** if the file could not be opened.

FilePrint method — IDEApplication

This method prints a file. If *suppressDialog* is set to **TRUE**, this method does not display the Printer Options dialog prior to performing the print operation but rather reuses the last print options specified.

Types expected

```
bool FilePrint(bool suppressDialog)
```

Return value

TRUE if the print operation was successful or **FALSE** if it was not.

FilePrinterSetup method — IDEApplication

Displays the Printer Setup dialog box to allow the user to set print options.

Types expected

```
bool FilePrinterSetup()
```

Return value

TRUE if the dialog sets the options or **FALSE** if the user exits with Cancel.

FileSave method — IDEApplication

This method saves the currently open editor file.

Types expected

```
bool FileSave()
```

Return value

TRUE if the file was saved or **FALSE** if the file could not be saved.

FileSaveAll method — IDEApplication

This method saves all open editor files.

Types expected

```
bool FileSaveAll()
```

Return value

TRUE if all files were saved or **FALSE** if a file could not be saved.

FileSaveAs method — IDEApplication

This method displays the standard File Save As dialog box so the user can save the currently active editor file. If *newName* is supplied, it attempts to save the file under that name in the current directory.

Types expected

```
bool FileSaveAs([string newName])
```

Return value

TRUE if the file was saved or **FALSE** if the file could not be saved.

FileSend method — IDEApplication

This method instructs the Windows MAPI to send files to another MAPI client.

Types expected

```
bool FileSend()
```

Return value

TRUE if the file was sent or **FALSE** if the file could not be sent.

GetRegionBottom method — IDEApplication

This method gets the bottom value of the specified region. It can be used in conjunction with *SetRegion* to position a window.

Types expected

```
int GetRegionBottom(string RegionName)
```

RegionName See the RegionName description.

Return value

The bottom value of the specified region in display units (0 - 10000) or -1 if no such region exists.

Region names

Available region names include:

"Breakpoint"
"CPU"
"Debugger"
"Editor"
"Evaluator"
"Event Log"
"Inspector"
"Message"
"Processes"
"Project"
"Stack"

"Thread Count"
"Watches"

GetRegionLeft method — IDEApplication

This method gets the left value of the specified region. It can be used in conjuction with *SetRegion* to position a window.

Types expected

```
int GetRegionLeft(string RegionName)
```

RegionName See the RegionName description.

Return value

The left value of the specified region in display units (0 - 10000) or -1 if no such region exists.

GetRegionRight method — IDEApplication

This method gets the right value of the specified region. It can be used in conjunction with *SetRegion* to position a window.

Types expected

```
int GetRegionRight(string RegionName)
```

RegionName See the RegionName description.

Return value

The right value of the specified region in display units (0 - 10000) or -1 if no such region exists.

GetRegionTop method — IDEApplication

This method gets the top value of the specified region. It can be used in conjunction with *SetRegion* to position a window.

Types expected

```
int GetRegionTop(string RegionName)
```

RegionName See the RegionName description.

Return value

The top value of the specified region in display units (0 - 10000) or -1 if no such region exists.

GetWindowState method — IDEApplication

This method retrieves the state of the currently focused window.

Types expected

```
bool GetWindowState()
```

Return value
SW_NORMAL, SW_MINIMIZE, or SW_MAXIMIZE

Help method — IDEApplication

This method invokes the Windows Help system with the specified Help file and context ID.*helpFile* specifies the name (with optional path) of the Windows Help file to open. *helpCommand* is a constant representing a command passes to the Windows Help engine. The *helpCommand* constants begin with **HELP_** and are defined in the C++ header file WINUSER.H. *helpTopic* is the name of the Help topic to display.

Types expected
```
void Help (string helpFile, int helpCommand, string helpTopic)
```

Return value
None

HelpAbout method — IDEApplication

This method displays the Help About dialog box.

Types expected
```
bool HelpAbout()
```

Return value
TRUE if the dialog box displays, **FALSE** if it cannot be displayed.

HelpContents method — IDEApplication

This method displays the default Help contents screen. For Windows 95 Help systems, this window is the Help Topics Contents page.

Types expected
```
bool HelpContents()
```

Return value
TRUE if the Help window can be displayed or **FALSE** if it cannot be displayed.

HelpKeyboard method — IDEApplication

This method displays a Help window describing how to map the keyboard in the IDE.

Types expected
```
bool HelpKeyboard()
```

Return value
TRUE if the Help window can be displayed or **FALSE** if it cannot be displayed.

HelpKeywordSearch method — IDEApplication

This method displays the Help Topics Index page with *keyword* as the default entry.

Types expected

```
bool HelpKeywordSearch([string keyword])
```

Return value

TRUE if the Help window can be displayed or **FALSE** if it cannot be displayed.

HelpUsingHelp method — IDEApplication

This method displays a Help window describing how to use Help.

Types expected

```
bool HelpUsingHelp()
```

Return value

TRUE if the Help window can be displayed or **FALSE** if it cannot be displayed.

HelpWindowsAPI method — IDEApplication

This method displays the Microsoft Windows API Help Topics dialog box.

Types expected

```
bool HelpWindowsAPI()
```

Return value

TRUE if the Help window can be displayed or **FALSE** if it cannot be displayed.

KeyPressDialog method — IDEApplication

This method displays a dialog and records the keys pressed by the user in a mnemonic format suitable for using with key assignments.

Types expected

```
string KeyPressDialog(string prompt, string default)
```

Return value

The mnemonic name of the key pressed by the user or the empty string ("") if the user presses *Esc* or *Cancel*.

ListDialog method — IDEApplication

This method dsplays a modal list dialog containing the strings as specified in the *initialValues* array. The list is sorted as indicated by *sorted* and supports multiple selection as indicated by *multiSelect*.

Types expected

```
string[ ] ListDialog(string prompt, bool multiSelect,
                     bool sorted, string [ ] initialValues)
```

Return value

An array containing the strings that were selected.

Menu method — IDEApplication

This method activates the main menu.

Types expected

```
void Menu()
```

Return value

None

Message method — IDEApplication

This method displays messages to the user.

Types expected

```
bool Message(string text, int severity)
```

text The message to display.

severity One of the following values: INFORMATION, WARNING, ERROR. The value specified also determines the text for the caption.

Return value

This method returns **TRUE** if the message box successfully opened, **FALSE** otherwise.

MessageCreate method — IDEApplication

This method adds messages to the Message window.

Types expected

```
int MessageCreate(string destinationTab, string toolName,
   int messageType, int parentMessage, string filename,
   int lineNumber, int columnNumber, string text,
   string helpFileName, int helpContextId)
```

destinationTab The name of the tab on the page of the MessageView on which this message should appear. The default supported values for this parameter are "Buildtime", "Runtime", and "Script". If a non-existant tab name is given, a new tab will be created.

toolName The name of the tool to be associated with the file to open. Tools can be standalone programs (like GREP, the Paradigm C++ integrated debugger, or an alternate editor), or they can be translators that are used for each file (or node) in a project. You can also use the tool name: AddOn. You can run a DOS program with the Windows IDE transfer. If toolName is not provided, a default is used.

messageType The severity to be associated with the message. (INFORMATION (default), WARNING, ERROR, or FATAL).

parentMessage The message that this message should be stored under. (Zero creates a new top-level message.)

fileName, *lineNumber*, and *columnNumber* provide source navigation for the message. By selecting the message the user can be taken to the specified *lineNumber* and *columnNumber* in the *filename* in an editor.

helpFile and *helpContext* specify where the user can find Windows Help of this message. If these parameters are set to valid values, the user will go to the given Help topic if *F1* is pressed while the message is selected.

Return value

The message ID of the generated message.

NextWindow method — IDEApplication

This method advances focus and activation to the next MDI child window from the currently selected window. If *priorWindow* is **TRUE**, focus and activation go to the previous window. *priorWindow* defaults to **FALSE**.

Types expected

```
bool NextWindow(bool priorWindow)
```

Return value

TRUE if focus changes to another window, **FALSE** if it does not.

OptionsEnvironment method — IDEApplication

This method displays the Environment Options dialog box.

Types expected

```
bool OptionsEnvironment()
```

Return value

TRUE if the dialog box can be displayed, **FALSE** if it cannot.

OptionsProject method — IDEApplication

This method displays the Project Options dialog box.

Types expected

```
bool OptionsProject()
```

Return value

TRUE if the dialog box can be displayed, **FALSE** if it cannot.

OptionsSave method — IDEApplication

This method opens the Options Save dialog box, which allows the user to save the contents of their project, desktop, messages and current settings for Environment.

Types expected

```
bool OptionsSave()
```

Return value
TRUE if the dialog can be opened, **FALSE** if it cannot.

OptionsStyleSheets method — IDEApplication

This method displays the Style Sheets dialog box.

Types expected
```
bool OptionsStyleSheets()
```

Return value
TRUE if the dialog box can be opened, **FALSE** if it cannot.

OptionsTools method — IDEApplication

This method displays the Tools dialog box.

Types expected
```
bool OptionsTools()
```

Return value
TRUE if the dialog box can be opened, **FALSE** if it cannot.

ProjectBuildAll method — IDEApplication

This method builds all the files in the current project, regardless of whether they are out of date (exactly the same as the Project|Build All menu selection.

Types expected
```
bool ProjectBuildAll([bool suppressOkay, string nodeName])
```

suppressOkay Builds the project without requiring the user to respond with *OK* to continue.

nodeName Build only the node specified.

Return value
TRUE if the build was successful or **FALSE** if it was not.

ProjectCloseProject method — IDEApplication

This method closes the current project.

Types expected
```
bool ProjectCloseProject()
```

Return value
TRUE if the project was successfully closed or **FALSE** if it was not.

ProjectCompile method — IDEApplication

This method compiles the current project. If a *nodeName* is indicated, what happens depends on the type of node:

A .CPP node causes the C++ compiler to be called.
A .RC node causes resource compiler to be called.
An .EXE node causes the linker to be called.
A .LIB node causes the librarian to be called.
An .SPP node causes the cScript compiler to be called.

Types expected

```
bool ProjectCompile([string nodeName])
```

Return value

TRUE if the project was successfully closed or **FALSE** if it was not.

ProjectGenerateMakefile method — IDEApplication

This method generates a make file for the current project. If the *nodeName* is specified, then the generated makefile contains only the commands necessary to build that node. Otherwise, commands are generated to build the entire project.

Types expected

```
bool ProjectGenerateMakefile([string nodeName])
```

Return value

TRUE if the makefile was successfully generated or **FALSE** if it was not.

ProjectMakeAll method — IDEApplication

This method makes all targets for the current project, rebuilding only those files that are out of date.

suppressOkay Makes the project without requiring the user to respond with *OK* to continue.

nodeName Makes only the node specified.

Types expected

```
bool ProjectMakeAll([bool suppressOkay, string nodeName])
```

Return value

TRUE if the targets were successfully made or **FALSE** if not.

ProjectManagerInitialize method — IDEApplication

This method is called once during IDE initialization to ensure that the IDE's Project Manager is in a stable state prior to the occurrence of any major events, such as the opening of files or creation of new targets.

Types expected

```
bool ProjectManagerInitialize()
```

Return value

TRUE if the Project Manager has successfully initialized or **FALSE** if it did not.

ProjectNewProject method — IDEApplication

This method creates a new project. If *pName* is specified, then the project is created with *pName* as its name, otherwise the user is prompted for a project name.

Types expected

```
bool ProjectNewProject([string pName])
```

Return value

TRUE if the project was successfully created or **FALSE** if it was not.

ProjectNewTarget method — IDEApplication

This method creates a new target for the node specified in *nTarget*.

Types expected

```
bool ProjectNewTarget ( [string nTarget, int targetType,
     int platform, int libraryMask, int modelOrMode] )
```

ProjectNewTarget parameter descriptions — IDEApplication

nTarget is the name of the node.

targetType must be one of the following target values:

TE_APPLICATION (default)
TE_STATICLIB

platform must be one of the following platform values:

TE_AXE

libraryMask indicates which libraries to link and is one or more of the following values:

TE_STDLIBS (default: same as TE_STDLIB_RTL | TE_STDLIB_EMU)
TE_STDLIB_EMU
TE_STDLIB_MATH
TE_STDLIB_NOEH
TE_STDLIB_RTL

modelOrMode is one of the following values:

TE_MM_LARGE(default if platform is TE_WIN32)
TE_MM_SMALL
TE_MM_MEDIUM
TE_MM_COMPACT
TE_MM_HUGE

ProjectOpenProject method — IDEApplication

This method opens a project. If *pName* is specified, it opens that project. If not, it displays the Open Project dialog box and prompts the user for a project name.

Types expected

```
bool ProjectOpenProject([string pName])
```

Return value

TRUE if the project opened or **FALSE** if it did not.

Quit method — IDEApplication

This method shuts down the IDE and exits.

Types expected

```
void Quit()
```

Return value

None

SaveMessages method — IDEApplication

This method saves the contents of the specified Message window tab page to the specified file.

Types expected

```
bool SaveMessages(string tabName, string fileName)
```

tabName One of the following values:

"Buildtime"
"Runtime"
"Script"

Return value

TRUE if the messages are saved or **FALSE** if it cannot be saved.

ScriptCommands method — IDEApplication

This method invokes the Script Commands dialog box. The dialog lists the commands currently available in the system.

Types expected

```
bool ScriptCommands()
```

Return value

TRUE if the user enters a command and selects *Run*, **FALSE** otherwise.

ScriptCompileFile method — IDEApplication

This method compiles the script file *fileName*.

Types expected

```
bool ScriptCompileFile(string fileName)
```

Return value

TRUE if the compile was successful, **FALSE** otherwise.

ScriptModules method — IDEApplication

This method invokes the Script Modules dialog box. The dialog lists the modules loaded or in the Script Path.

Types expected

```
bool ScriptModules()
```

Return value

TRUE if a module is selected, **FALSE** otherwise.

ScriptRun method — IDEApplication

Executes the script command given in *command*. If no *command* is passed, the Script Run dialog appears.

Types expected

```
bool ScriptRun(string command)
```

Return value

TRUE if the command is executed, **FALSE** otherwise.

ScriptRunFile method — IDEApplication

This method causes script file given in *fileName* to execute. If no *fileName* is given, the method attempts to execute the commands in the current EditView.

Types expected

```
bool ScriptRunFile([string fileName])
```

Return value

TRUE if a file is executed or an *EditView* is found, **FALSE** otherwise.

SearchBrowseSymbol method — IDEApplication

This method searches for the symbol indicated in *sName*. If *sName* is not provided, the user will be prompted for it.

Types expected

```
bool SearchBrowseSymbol([string sName])
```

Return value

TRUE if the symbol is found or **FALSE** if it cannot be found.

SearchFind method — IDEApplication

This method searches in the current edit buffer for the pattern supplied in *pat*. If *pat* is found, the cursor is moved to the occurrence of *pat*. This pattern can be a simple string or a search expression.

Types expected

```
bool SearchFind([string pat])
```

Return value

TRUE if the expression is found or **FALSE** if it cannot be found.

SearchLocateSymbol method — IDEApplication

This method searches through the current target of the current project and uses the Browser's symbol information to locate a symbol's definition.

Types expected

```
bool SearchLocateSymbol([string sName])
```

Return value

TRUE if the expression is found or **FALSE** if it cannot be found.

Description

On success, this method opens the source file and line where the symbol name *sName* is defined. If *sName* is NULL, *SearchLocateSymbol()* rips the current word out of the editor and searches for that symbol. *SearchLocateSymbo()l* works only with globally defined symbols.

For a function symbol, *SearchLocateSymbol()* locates the line where the function begins. For a class or typedef symbol, it locates the line where the typedef or class is defined. For a variable, it locates the line where the variable is defined.

SearchNextMessage method — IDEApplication

This method works only if a Message view is displayed on the users screen. It displays the next message listed in the Message view if there is a next message.

Types expected

```
bool SearchNextMessage()
```

Return value

TRUE if the next message is displayed, or **FALSE** if there is no message to display.

SearchPreviousMessage method — IDEApplication

This method displays an active Edit window and places the cursor on the line in your source code that generated the previous error or warning listed in the Message window. If the file is not currently loaded, the IDE opens it in a new Edit window.

Types expected

```
bool SearchPreviousMessage()
```

Return value

TRUE if the source line is found or **FALSE** if it cannot be found.

SearchReplace method — IDEApplication

This method searches in the current edit buffer for the pattern indicated in *pat* and replaces it with the string indicated in *rep*. If either parameter is not specified, the method opens the Replace Text dialog box and prompts the user for input. The pattern can be a simple string or a search expression.

Types expected

```
bool SearchReplace([string pat, string rep])
```

Return value

TRUE if the text is found or **FALSE** if it cannot be found.

SearchSearchAgain method — IDEApplication

This method repeats the last SearchFind.

Types expected

```
bool SearchSearchAgain()
```

Return value

TRUE if the text is found or **FALSE** if it cannot be found.

SetRegion method — IDEApplication

This method determines how windows tile and cascade on the IDE desktop, as well as their initial position when they are created.

Types expected

```
bool SetRegion(string RegionName, int left, int top, int right, int
               bottom)
```

RegionName See the RegionName description.

left, top, right, bottom The dimensions of the window in display units of 1-9999.

Return value

TRUE if the region was successfully set or **FALSE** if it was not.

Description

SetRegion is used in conjunction with *GetRegionBottom*, *GetRegionTop*, *GetRegionLeft*, *GetRegionRight* to change the area where windows are placed when tiled and cascaded.

For example, the default configuration of the IDE is to have all editor windows in the upper two-thirds of the screen when you tile, and the message window and the project window in the lower one-third. You could change this default with the script statement

```
IDE.SetRegion("Editor", 1, 1, 5000, 5000);
```

After executing this statement, the editors are in the upper left quarter of the IDE desktop after tiling. You can look at STARTUP.SPP for other examples.

SetWindowState method — IDEApplication

This method changes the style of the currently focused window.

desiredState One of the following values: SW_MINIMIZE, SW_MAXIMIZE, SW_RESTORE

Types expected

```
bool SetWindowState(int desiredState)
```

Return value

TRUE if the state was successfully set or **FALSE** if it was not.

SimpleDialog method — IDEApplication

This method invokes a simple dialog containing a single text field, an *OK* button, and a *Cancel* button.

Types expected

```
string SimpleDialog(string prompt, string initialValue
       [, int maxNumChars])
```

prompt The caption of the dialog.

initialValue The value that initializes the edit field.

Return value

The value in the edit field if the user clicks *OK* or presses *Enter* on the edit field, or the empty string ("") if the user clicks *Cancel*.

SpeedMenu method — IDEApplication

Activates the SpeedMenu for the current subsystem.

Types expected

```
void SpeedMenu()
```

Return value

None

StartWaitCursor method — IDEApplication

This method displays the Windows wait cursor.

Types expected

```
void StartWaitCursor()
```

Return value

None

StatusBarDialog method — IDEApplication

This method displays a dialog on top of the status bar.

Types expected

```
string StatusBarDialog(string prompt, string initialValue
       [, int maxNumChars])
```

prompt The caption of the dialog.

initialValue The value that initializes the edit field.

Return value

The value in the edit field if the user clicks *OK* or presses *Enter* on the edit field, or the empty string ("") if the user clicks *Cancel*.

Tool method — IDEApplication

This method runs the tool specified in *toolName* using the command string specified in *commandstring*. If no parameters are specified, it displays a dialog box prompting the user for a tool.

Types expected

```
bool Tool([string toolName, string commandString])
```

Return value

TRUE if the tool successfully ran or **FALSE** if it did not.

Undo method — IDEApplication

This method does the same thing as *EditUndo*. It is included for compliance with Microsoft conventions.

Types expected

```
void Undo()
```

Return value

None

ViewActivate method — IDEApplication

This method activates the IDE pane that is adjacent to the currently selected pane. The *direction* argument indicates the direction of the adjacent pane to activate, relative to the current pane. The supported values for *direction* are **UP**, **DOWN**, **LEFT** and **RIGHT**.

Types expected

```
bool ViewActivate(int direction)
```

Return value

Returns **TRUE** if there was a valid current pane and the method was able to activate an adjacent pane in the direction indicated by direction, otherwise **FALSE**.

ViewBreakpoint method — IDEApplication

This method opens a window showing all breakpoints.

Types expected

```
bool ViewBreakpoint()
```

Return value

TRUE if breakpoints can be found or **FALSE** if no breakpoints can be found.

ViewCallStack method — IDEApplication

This method opens the Call Stack window, which shows the sequence of functions your program called to get to the current execution point. Each entry displays the function name and the values of any parameters passed to it.

Types expected

```
bool ViewCallStack()
```

Return value

TRUE if the Call Stack window can be displayed or **FALSE** if it cannot be displayed.

ViewClasses method — IDEApplication

This method opens the Browsing Objects window, which displays all the classes in your application arranged in a horizontal tree that shows parent-child relationships.

Types expected

```
bool ViewClasses()
```

Return value

TRUE if the Browsing Objects window can be displayed or **FALSE** if it cannot be displayed.

ViewCpu method — IDEApplication

This method displays the Debugger's CPU view for the current application.

Types expected

```
bool ViewCpu()
```

Return value

TRUE if the CPU view can be displayed or **FALSE** if it cannot.

ViewGlobals method — IDEApplication

This method opens the Browsing Globals window, which lists every variable in the program in the current Edit window or the first file in the current project. If the program has not been compiled, the IDE first compiles it before invoking the Browser.

Types expected

```
bool ViewGlobals()
```

Return value

TRUE if the Browse Globals window can be displayed or **FALSE** if it cannot.

ViewMessage method — IDEApplication

This method displays the Message window which displays status, warning, and error messages for C++ compilations and links, ObjectScripting module compilations, run-time messages from the debugger, and output. If given, the page specified by *tabName* is selected when Message window is opened. If *tabName* is not found, the currently selected tab remains unchanged.

tabName may have one of the following values:

"Buildtime"
"Runtime"
"Script"

or *tabName* may be the name of a user-defined tab.

Types expected

```
bool ViewMessage([string tabName])
```

Return value

TRUE if the Message window can be displayed or **FALSE** if it cannot. If *tabName* is not found, the method returns **FALSE** even if the Message window is successfully displayed.

ViewProcess method — IDEApplication

This method displays the debugger's Process window.

Types expected

```
bool ViewProcess()
```

Return value

TRUE if the Process window can be displayed or **FALSE** if it cannot.

ViewSlide method — IDEApplication

This method moves the border of the currently selected IDE pane *amount* characters in the given *direction* on the screen The size of a character is determined by the number of pixels high and wide a character is in the font used by the pane. The default value for

amount is 1. The supported values for *direction* are the constants **UP**, **DOWN**, **LEFT** and **RIGHT**.

Types expected

```
bool ViewSlide(int direction [, int amount])
```

Return value

TRUE if there is a valid current IDE pane, and it was successfully moved *amount* characters in the given *direction*, otherwise returns **FALSE**.

ViewProject method — IDEApplication

This method displays the Project window for the currently open project.

Types expected

```
bool ViewProject()
```

Return value

TRUE if the Project window can be displayed or **FALSE** if it cannot.

ViewWatch method — IDEApplication

This method displays the Debugger's Watches window for the current program.

Types expected

```
bool ViewWatch()
```

Return value

TRUE if the Watches window can be displayed or **FALSE** if it cannot.

WindowArrangeIcons method — IDEApplication

This method rearranges any minimized window's icons on the desktop. The rearranged icons are evenly spaced, beginning at the lower left corner of the desktop.

Types expected

```
bool WindowArrangeIcons()
```

Return value

TRUE if there are icons to rearrange or **FALSE** if there are none.

WindowCascade method — IDEApplication

This method stacks all open windows and overlaps them, making all windows the same size and showing only part of each underlying window.

Types expected

```
bool WindowCascade()
```

Return value
TRUE if there are windows to cascade or **FALSE** if there are none.

WindowCloseAll method — IDEApplication

This method closes all open windows if no window type is specified in *typeName*. Otherwise, it closes the windows of the specified type.

typeName cam be one of the following values:

"Browser"
"Debugger"
"Editor"

Types expected
```
bool WindowCloseAll([string typeName])
```

Return value
TRUE if all windows successfully close or **FALSE** if at least one does not.

WindowMinimizeAll method — IDEApplication

This method minimizes all open windows if no window type is specified in *typeName*. Otherwise it minimizes the windows of the specified type.

typeName can be one of the following values:

"Browser"
"Debugger"
"Editor"

Types expected
```
bool WindowMinimizeAll([string typeName])
```

Return value
TRUE if all windows successfully minimize or **FALSE** if at least one does not.

WindowRestoreAll method — IDEApplication

This method restores all minimized windows if no window type is specified in *typeName*. Otherwise, it restores the window type specified in *typeName*.

Types expected
```
bool WindowRestoreAll([string typeName])
```

Return value
TRUE if all windows successfully restore or **FALSE** if at least one does not.

WindowTileHorizontal method — IDEApplication

This method stacks all open windows horizontally.

Types expected

```
bool WindowTileHorizontal()
```

Return value

TRUE if all windows successfully tile or **FALSE** if they do not.

WindowTileVertical method — IDEApplication

This method stacks all open windows vertically.

Types expected

```
bool WindowTileVertical()
```

Return value

TRUE if all windows successfully tile or **FALSE** if they do not.

YesNoDialog method — IDEApplication

This method displays a dialog box that prompts the user for a yes or no response.

Types expected

```
string YesNoDialog(string prompt, string default)
```

prompt The prompt that displays in the dialog box.

default The text displayed in the text entry box.

Return value

"Yes" or "No".

BuildComplete event — IDEApplication

This event is raised at the end of a build. The *status* parameter indicates if the build was successful; **TRUE** if successful, **FALSE** if there were errors. *inputPath* indicates the source directory. *outputPath* is the directory where files created as a result of the build are created.

Types expected

```
void BuildComplete(bool status, string inputPath,
                   string outputPath)
```

Return value

None

BuildStarted event — IDEApplication

This method is raised when one of this class's "Help" methods is invoked. If passes the appropriate parameters to the Windows Help engine. Default action is to do nothing.

Types expected

```
void BuildStarted()
```

Return value
None

DialogCreated event — IDEApplication

This event is raised as new dialogs are presented to the user.

Types expected
```
void DialogCreated(string dialogName, int dialogHandle)
```

Return value
None

Description
Use *DialogCreated* in conjunction with the *SendKeys* method of KeyboardManager to simulate user entries to dialogs and drive the dialog. The *dialogName* parameter contains the dialog's caption. The *dialogHandle* is an environment-specific identifier used by the system when referring to the dialog. Under Microsoft Windows the *dialogHandle* is the HWND of the dialog. This value is supplied in case you need your script to interact directly with the system.

Exiting event — IDEApplication

Raised as the IDE is closing. Default action is to do nothing.

Types expected
```
void Exiting()
```

Return value
None

HelpRequested event — IDEApplication

The method is raised when one of the class's "Help" methods is invoked. It passes the appropriate parameters to the Windows Help engine. Default action is to do nothing.

Types expected
```
void HelpRequested(string fileName, int command, int data)
```

Return value
None

Idle event — IDEApplication

Raised when the number of seconds specified by *IdleTimeout* has elapsed without a significant event occurring (like a user event). Default action is to do nothing.

Types expected
```
void Idle()
```

Return value
None

KeyboardAssignmentsChanging event — IDEApplication

Raised when the user is exiting the MPD after having modified the keyboard file choice.

Types expected
```
void KeyboardAssignmentsChanging(string newFileName)
```

Return value
None

KeyboardAssignmentsChanged event — IDEApplication

Raised after the keyboard file name has been changed.

Types expected
```
void KeyboardAssignmentsChanged(string newFileName)
```

Return value
None

MakeComplete event — IDEApplication

Raised at the end of a make. The *status* parameter indicates if the make was successful; **TRUE** if successful, **FALSE** if there were errors. *inputPath* indicates the source directory. *outputPath* is the directory where files created as a result of the make are created.

Types expected
```
void MakeComplete(bool status, string inputPath,
     string outputPath)
```

Return value
None

MakeStarted event — IDEApplication

Raised at the beginning of a make.

Types expected
```
void MakeStarted()
```

Return value
None

ProjectClosed event — IDEApplication

Raised when a project file has been successfully closed. *projectFilename* contains the absolute name of the project file. Since the IDE always has a project open (even if it is the default project: PCWDEF.IDE), this event will always precede the *ProjectOpened()* that it corresponds to. In other words, you get a *ProjectClosed* event followed by a *ProjectOpened* event.

Types expected

```
void ProjectClosed(string projectFileName)
```

Return value

None

ProjectOpened event — IDEApplication

Raised when a project file has been successfully opened. *projectFilename* contains the fully qualified name of the project file.

Types expected

```
void ProjectOpened(string projectFileName)
```

Return value

None

SecondElapsed event — IDEApplication

Raised once every second. Default action is to do nothing.

Types expected

```
void SecondElapsed()
```

Return value

None

Started event — IDEApplication

Raised after the IDE has been loaded and initialized and all startup scripts have been processed. The parameter *VeryFirstTime* indicates whether this is the first time the IDE has been loaded on a particular machine. Its value is determined by the presence or absence of the default configuration file (PCCONFIG.PCW). This file is created for you the first time you run the IDE and should be present only if the IDE has been run previously.

Types expected

```
void Started(bool VeryFirstTime)
```

Return value

None

SubsystemActivated event — IDEApplication

Raised when the active subsystem is changed (usually in response to the user clicking on another window type). *systemName* holds the name of the subsystem acquiring fonts. Default action is to do nothing.

Types expected

```
void SubsytemActivated(string systemName)
```

Return value

None

TransferOutputExists event — IDEApplication

Raised when a transfer tool has created output that needs processing (usually in a Make sequence). The return value signals an error code: **FALSE** if no error occurred, **TRUE** if there was an error parsing the data supplied by output. Default action is to do nothing.

Types expected

```
bool TransferOutputExists(TransferOutput output)
```

Return value

None

TranslateComplete event — IDEApplication

Raised at the end of a translation. The *status* parameter indicates if the translation was successful; **TRUE** if successful, **FALSE** if there were errors. *inputPath* indicates the source directory. *outputPath* is the directory where files created as a result of the translation are created.

Types expected

```
void TranslateComplete(bool status, string inputPath,
                       string outputPath)
```

Return value

None

KeyboardManager class

Syntax

```
KeyboardManager()
```

Properties

`bool AreKeysWaiting`	*Read-only*
`Record CurrentPlayback`	*Read-only*
`Record CurrentRecord`	*Read-write*
`int KeyboardFlags`	*Read-only*
`int KeysProcessed`	*Read-only*
`int LastKeyProcessed`	*Read-only*
`Record Recording`	*Read-only*
`string ScriptAbortKey`	*Read-write*

Methods

```
string CodeToKey( int KeyCode )
void Flush()
Keyboard GetKeyboard( [string ComponentName] )
int KeyToCode( string KeyName )
void PausePlayback()
int Playback( [Record RecordObject] )
Keyboard Pop( string ComponentName )
bool ProcessKeyboardAssignments( string fileName, bool unassign )
void ProcessPendingKeystrokes()
void Push( Keyboard keyboard, string ComponentName, bool transparent )
int ReadChar( void )
void ResumePlayback()
bool ResumeRecord( Record RecordObject )
bool SendKeys(string keyStream)
bool StartRecord( Record RecordObject )
void StopRecord( )
```

Events

```
void UnassignedKey (int keyCode)
```

AreKeysWaiting property — KeyboardManager

Returns **TRUE** if any keys are waiting to be processed. This is a read-only property.

Type expected

```
bool AreKeysWaiting
```

CurrentPlayback property — KeyboardManager

Only valid while in a *Playback()*. This is a read-only property.

Type expected

```
Record CurrentPlayback
```

CurrentRecord property — KeyboardManager

This property contains a reference to the Record object associated with this *KeyboardManager*. This is a read-write property.

Type expected
```
Record CurrentRecord
```

KeyboardFlags property — KeyboardManager

This property returns a value whose bits may be checked to determine the state of *NumLock*, *Caps*, *Ctrl*, *Alt*, and so on. The mask values are:

0x03 - Shift pressed
0x04 - Ctrl pressed
0x08 - Alt pressed
0x10 - Scroll Lock on
0x20 - Num Lock on
0x40 - Caps Lock on

This is a read-only property.

Type expected
```
int KeyboardFlags
```

KeysProcessed property — KeyboardManager

The total number of keys processed by any keyboard since the IDE was loaded. This is a read-only property.

Type expected
```
int LastKeyProcessed
```

LastKeyProcessed property — KeyboardManager

The *keyCode* of the last key that was processed by any keyboard. This is a read-only property.

Type expected
```
int LastKeyProcessed
```

Recording property — KeyboardManager

Only valid while in a *StartRecord()* until *StopRecord()* is called.

The return value matches cBrief's *inq_kbd_flags()*.

This is a read-only property.

Type expected
```
Record Recording
```

ScriptAbortKey property — KeyboardManager

Contains the KeySequence of the key which, when pressed, causes the currently running script to abort. The default value is <Escape>, except when Epsilon emulation is enabled in which case the default is <Ctrl-G>.

The KeySequence is a mnemonic key name made up of a key description, such as <a>. Key descriptions can be augmented with any (or all) of the following: "Shift", "Ctrl", "Alt", and "Keypad".

Keys that don't map to a single character have names associated with them. Keys in this category are: "Enter", "Backspace", "Tab", "Home", "End", "PageUp", "PageDown", "Left", "Right", "Up", "Down", "Insert", "Delete", "Escape", "Space", "PrintScreen", "Center", "Pause", "CapsLock", "ScrollLock", and "NumLock".

Modifiers and names are separated by a dash (-). For example:

```
<Ctrl-Enter>
```

To assign the dash character in a key sequence, use the keyname <Minus>. Use the keyname <Plus> for the '+' character.

This is a read-write property.

Type expected

```
string ScriptAbortKey
```

CodeToKey method — KeyboardManager

This method accepts the integer keycode representation and returns the textual description of the key.

Types expected

```
string CodeToKey(int KeyCode)
```

Return value

Matches the cBrief key naming conventions for *inq_assignment* and *assign_to_key*.

Flush method — KeyboardManager

This method removes all waiting keystrokes from the IDE's message queue.

Types expected

```
void Flush()
```

Return value

None

GetKeyboard method — KeyboardManager

This method finds the keyboard currently assigned to the IDE subsystem specified in ComponentName. Specifying "Default" returns the internal mapping, which cannot be remapped. If ComponentName is omitted, the method gets the current keyboard.

Valid subsystems are

Editor
Message
Project
Desktop

Types expected

```
Keyboard GetKeyboard ( [string ComponentName] )
```

Return value

The keyboard currently assigned to an IDE subsystem, or NULL if the subsystem is invalid.

KeyToCode method — KeyboardManager

This method accepts the textual name of a key and returns the integer keycode equivalent. It accepts single keystroke entries such as <F> and <Ctrl-B>, but not multikey sequences such as "Ctrl+K Ctrl+B".

Types expected

```
int KeyToCode (string KeyName)
```

Return value

The integer keycode of the key.

PausePlayback method — KeyboardManager

Pauses the playback of a recording object initiated with *Playback()*. To resume playback, call *ResumePlayback()*.

Types expected

```
void PausePlayback()
```

Return value

None

Playback method — KeyboardManager

This method replays the series of keystrokes assigned to a *Record* object. If no *Record* object is specified, the last recording is replayed.

Types expected

```
int Playback ( [Record RecordObject] )
```

Return value

One of the following values:

0	No sequence to play back
1	Playback successful
-1	Sequence is paused or being remembered
-2	Error loading disk file (macros will handle this)
-3	Canceled by user with ScriptAbortKey

ProcessKeyboardAssignments method — KeyboardManager

Converts a .KBD file into a .KBP file.

Types expected

```
bool ProcessKeyboardAssignments (string fileName, bool unassign)
```

fileName The name of the .KBD formatted file. Includes the path to the file.

unassign Specifies if the file contents should be used to unassign keys defined in the .KBD file. If **TRUE**, defined keys will be unassigned. If **FALSE**, defined keys will be assigned.

Return value

TRUE if a .KBP file is loaded and **FALSE** if it is not.

ProcessPendingKeystrokes method KeyboardManager

This method can be used fine tune the behavior of SendKeys. If one or more calls to *SendKeys* indicated that key processing was to be delayed, these keystrokes are not processed until *ProcessPendingkeystrokes* is called or until the script completes execution.

Types expected

```
void ProcessPendingKeystrokes()
```

Return value

None

Pop method KeyboardManager

This method restores the previously assigned keyboard mapping after a call to *Push*.

Types expected

```
Keyboard Pop(string ComponentName)
```

Return value

The keyboard that was restored or NULL, which indicates that no additional keyboard mappings were applied and the default keyboard desktop mapping is active.

Push method KeyboardManager

This method pushes a keyboard on the keyboard stack, making the new keyboard mapping current. A subsequent *Pop* operation restores the previously assigned keyboard mapping.

Types expected

```
void Push ( Keyboard keyboard, string ComponentName, bool
            transparent )
```

transparent Determines the run-time behavior of keystrokes not found in the keyboard. If *transparent* is set, the next keyboard on the stack is searched. Otherwise, the key is ignored.

Return value

None

ReadChar method — KeyboardManager

This method returns either -1 (no key is waiting) or the scan value for the key that was pressed.

Types expected

```
int ReadChar ( void )
```

Return value

The high-order byte is the scan code, and the low-order byte is the ASCII value.

Description

ReadChar() manages two queues, a local queue for *Push()* as well as the standard Windows messaging system. It first checks the local queue for any waiting keys. If no keys are available in the local queue, it checks the Windows message queue.

ResumePlayback method — KeyboardManager

This method resumes the playback of a recording object initiated with *Playback()* after the recording has been suspended by a call to *PausePlayback()*.

Types expected

```
void ResumePlayback()
```

Return value

None

ResumeRecord method — KeyboardManager

This method reinitiates record mode on a *Record* object, appending new keystrokes to the end of the record buffer and updating the *Recording* member.

Types expected

```
bool ResumeRecord ( Record RecordObject )
```

SendKeys method — KeyboardManager

This method simulates the pressing of the keys indicated in the *keyStream* parameter.

Types expected

```
bool SendKeys(string keyStream[, bool suppressImmediateProcessing])
```

keystream	A series of key presses. The limit on the number of characters in Windows 95 is 715. There is no limit in Windows NT.
suppressImmediateProcessing	The default behavior is to process the keys immediately, before the next line of script is processed. If you include this parameter and set it to **TRUE**, *SendKeys* delays processing of the keys until ProcessPendingKeystrokes is called or until the script completes execution.

Return value

TRUE if *keyStream* has valid syntax and can be interpreted or **FALSE** if *keyStream* could not be turned into a series of keypresses.

Description

SendKeys takes a key or series of keys as its parameter.

Simple displayable keys are just a string of characters that are the same as the keycaps. For example,

```
SendKeys("hello world");
```

Keys that do not have simple displayable counterparts, like *Alt+S*, have a special syntax.

The following list shows how to indicate Alt+keyname, Shift+keyname, and Ctrl+keyname:

Alt key modifier	Preface the key name with the percent character (%). For example, *Alt+s* is `%s`.
Shift key modifier	Either preface the key name with the plus character (+) or capitalize it. For example, *Shift+s* is either +s or S.
Ctrl key modifier	Preface the key name with the carat character (^). For example, *Ctrl+s* is `^s`.

The *SendKeys* parameter is case sensitive. `^s` is *Ctrl+S*, but ^S is *Ctrl+Shift+S*.

To indicate the %, +, or ^ key itself, precede the key name with a backslash (\) as below:

```
To indicate %, use "+\\%"
To indicate ^, use "+\\^"
To indicate +, use "+\\+"
```

To simulate non-displaying keys, use a key mnemonic (described in a table that follows) and enclose it in braces ({ }).

For example, to simulate the key sequence `Alt+s 1 + 2 [Enter]`, use the following syntax:

```
SendKeys("%s1\+2{VK_RETURN}");
```

The key mnemonics are:

```
VK_ADD
VK_BACK
VK_CANCEL
VK_CAPITAL
VK_CLEAR
VK_CONTROL
VK_DECIMAL
VK_DELETE
VK_DIVIDE
VK_DOWN
VK_END
VK_ESCAPE
VK_EXECUTE
VK_F1
VK_F2
VK_F3
VK_F4
VK_F5
VK_F6
VK_F7
VK_F8
VK_F9
VK_F10
VK_F11
VK_F12
VK_F13
VK_F14
VK_F15
VK_F16
VK_F17
VK_F18
VK_F19
VK_F20
VK_F21
VK_F22
VK_F23
VK_F24
VK_HELP
VK_HOME
VK_INSERT
VK_LBUTTON
VK_LEFT
VK_MBUTTON
VK_MENU
VK_MULTIPLY
```

```
VK_NEXT          VK_NUMPAD7       VK_SCROLL
VK_NUMLOCK       VK_NUMPAD8       VK_SELECT
VK_NUMPAD0       VK_NUMPAD9       VK_SEPARATOR
VK_NUMPAD1       VK_PAUSE         VK_SHIFT
VK_NUMPAD2       VK_PRINT         VK_SNAPSHOT
VK_NUMPAD3       VK_PRIOR         VK_SPACE
VK_NUMPAD4       VK_RBUTTON       VK_SUBTRACT
VK_NUMPAD5       VK_RETURN        VK_TAB
VK_NUMPAD6       VK_RIGHT         VK_UP
```

At the current time there are two separate keyboard parsers, one for processing key assignments and the other for processing *SendKeys()*. These processors accept different formats for the same keys: For example, *"<Alt-a>"* versus `"%a"`.

SendKeys example

```
x = new KeyboardManager();
x.SendKeys("^S");  /* Sends Ctrl+S and processes it immediately
z.SendKeys("^S", FALSE);  /* Sends Ctrl+S and processes it immediately
x.SendKeys("...", TRUE); /* Sends Ctrl+S and delays processing
x.ProcessPendingKeystrokes();  /* Processes the delayed keystrokes
```

StartRecord method — KeyboardManager

This method begins storing keystroke sequences in a *Record* object and updates the *Recording* member.

Types expected

```
bool StartRecord ( Record RecordObject )
```

Return value

TRUE if the key sequence is stored or **FALSE** if it is not.

Description

StartRecord replaces any key sequences already stored in the *Record* object.

You can record to only one *Record* object at a time. If you attempt a *StartRecord* before calling a matching *StopRecord* for a previous recording, the *StartRecord* fails.

StopRecord method — KeyboardManager

Halts recording keystrokes previously started with *StartRecord()*. This operation updates the *CurrentRecord* member and updates the *Recording* member with an object whose *IsValid* value is false.

Types expected

```
void StopRecord ( )
```

UnassignedKey event — KeyboardManager

This event is raised when a key having no recorded keystrokes is pressed. The default action is to do nothing.

Types expected

```
void UnassignKey (int keyCode)
```

Return value

None

Keyboard class

This class works with the *KeyboardManager* class to manage keyboards assigned to various IDE components, such as the Editor and the Project View.

Syntax

```
Keyboard([bool transparent])
```

If the *Keyboard* is created with the *transparent* attribute, keystrokes having no assignment in this keyboard are passes to the next one on the current keyboard stack. This value defaults to **FALSE** if not supplied.

Properties

`int Assignments`	*Read-only*
`string DefaultAssignment`	*Read-write*

Methods

```
void Assign(string KeySequence, string CommandName, int
           ImplicitAssignments)
void AssignTypeables(string CommandName)
void Copy(Keyboard SourceKeyboard)
int CountAssignments(string CommandName)
string GetCommand(string KeySequence )
string GetKeySequence(string CommandName [,int whichOne])
bool HasUniqueMapping(string KeySequence)
void Unassign(string KeySequence)
```

Keyboard class description

Keyboard objects administer key assignments and can be assigned to IDE components, pushed and popped from the keyboard manager's keyboard stack, and queried on individual key assignments.

Assignments property — Keyboard

This property indicates the number of key assignments contained in this keyboard. It is a read-only property.

Type expected

```
int Assignments
```

DefaultAssignment property — Keyboard

This property establishes the command to execute if no other commands are assigned to a keystroke. It returns an empty string ("") if no assignment exists. It is a read-write property.

Type expected

```
string DefaultAssignment
```

Assign method — Keyboard

This method assigns a script to a keystroke.

Types expected

```
void Assign (string KeySequence, string CommandName, int
             ImplicitAssignments)
```

Return value

None

Description

CommandName is the script to be executed when the key is pressed, as in

```
editor.MarkWord(TRUE) ;
```

KeySequence is a mnemonic key name made up of a key description, such as <a>. Key descriptions can be augmented with any (or all) of the following: "Shift", "Ctrl", "Alt", and "Keypad".

Keys that do not map to a single character have names associated with them. Keys in this category are: "Enter", "Backspace", "Tab", "Home", "End", "PageUp", "PageDown", "Left", "Right", "Up", "Down", "Insert", "Delete", "Escape", "Space", "Print Screen", "Center", "Pause", "CapsLock", "ScrollLock", and "NumLock".

Modifiers and names are separated by a dash (-). For example,

<Ctrl-Enter>.

To assign the dash character in a key sequence, use the keyname <Minus>. Use the keyname <Plus> for the + character.

implicitAssignments is one or more of the following values:

ASSIGN_EXPLICIT (default)	No implicit assignments should be created.
ASSIGN_IMPLICIT_KEYPAD	When an assignment is made to a sequence that has a numeric keypad ("Keypad") equivalent, such as PageUp, a second assignment is implicitly made for the equivalent. Assignments are made to both the shifted and non-shifted versions at the same time, but only if the implicit assignment doesn't overwrite an existing explicit assignment.
ASSIGN_IMPLICIT_SHIFT	<a> == <A>
ASSIGN_IMPLICIT_MODIFIER	<Ctrl-k><Ctlr-b> = = <Ctrl-k><b>

This method has no effect on the default keyboard, which is returned from a call to `KeyboardManager.GetKeyboard("Default")`.

Assign method examples — Keyboard

```
// Explicit assignment to <Home>. Implicit assignment
// assignment to <Keypad-Home>.
```

```
Assign("<Home>","ToStart();", ASSIGN_IMPLICIT_KEYPAD);

// Explicit assignment to <Keypad-End>.
Assign("<Keypad-End>","ToEnd();");

// Explicit assignment to <End>
Assign("<End>","ToEnd(TRUE);", ASSIGN_IMPLICIT_KEYPAD);

// Implicit assignment to <Keypad-End> thwarted due to
// existence of explicit assignment to <Keypad-End>.
```

AssignTypeables method — Keyboard

This method assigns either a script to the predefined typeable characters (all ASCII characters, *Enter*, *Delete*, and *Backspace*).

This method has no effect on the default keyboard, which is returned from a call to `KeyboardManager.GetKeyboard("Default")`.

Types expected

```
void AssignTypeables(string CommandName)
```

CommandName The command to assign and any parameters to the command.

Return value

None

Copy method — Keyboard

This method copies all assignments made from *SourceKeyboard* into this keyboard, replacing any that already exist.

This method has no effect on the default keyboard, which is returned from a call to `KeyboardManager.GetKeyboard("Default")`.

Types expected

```
void Copy(Keyboard SourceKeyboard)
```

Return value

None

CountAssignments method — Keyboard

This method returns the number of key assignments tied to the specified command.

Types expected

```
int CountAssignments(string CommandName)
```

GetCommand method — Keyboard

This method returns the command assigned to the specified key code. It returns the empty string ("") if no script has been assigned.

Types expected

```
string GetCommand (string KeySequence )
```

GetKeySequence method — Keyboard

This method returns the key sequence tied to the specified command. If *whichOne* is less than 1 or omitted, it is assumed to be 1.

Types expected

```
string GetKeySequence(string CommandName [,int whichOne])
```

HasUniqueMapping method — Keyboard

Determines if a key has no mapping or maps directly to a command, or is the non-terminating key of a multikey assignment.

Types expected

```
bool HasUniqueMapping(string KeySequence)
```

Return value

Returns **TRUE** if a key either has no mapping or maps directly to a command. Returns **FALSE** if the key is a non-terminating key of a multikey assignment. For example, WordStar <Ctrl-K> would be **FALSE** since <Ctrl-K> signifies the beginning of a multikey assignment, such as <Ctrl-K><Ctrl-B> or <Ctrl-K><Ctrl-K>.

Unassign method — Keyboard

This method restores a key assignment.

This method has no effect on the default keyboard, which is returned from a call to `KeyboardManager.GetKeyboard("Default").`

Types expected

```
void Unassign(string KeySequence)
```

Return value

None

ListWindow class

Syntax

```
ListWindow(int Top, int Left, int Height, int Width,
           string Caption, bool MultipleSelect, bool Sorted,
           string[] InitialValues)
```

Top, Left, Height, Width	Initial coordinates of the list.
Caption	Text to be displayed in the list title.
MultipleSelect	Determines whether the list will support multiple selections.

Sorted	Determines whether new additions to the list are put in their sorted order.
InitialValues	An array of strings specifying the initial contents of the list.

Properties

`string Caption`	*Read-write*
`int Count`	*Read-only*
`int CurrentIndex`	*Read-only*
`[]Data`	*Read-only*
`int Height`	*Read-write*
`bool Hidden`	*Read-write*
`bool MultiSelect`	*Read-only*
`bool Sorted`	*Read-only*
`int Width`	*Read-write*

Methods

```
void Add(string newEl, int offset)
void Clear()
void Close()
void Execute()
int FindString(string toFind)
string GetString(int offset)
void Insert()
bool Remove(int offset)
```

Events

```
void Accept()
void Cancel()
void Closed()
void Delete()
bool KeyPressed(string keyName)
void LeftClick(int xPos, int yPos)
void Move()
void RightClick(int xPos, int yPos)
```

Caption property — ListWindow

This property contains the title of the list window. This is a read-write property.

Type expected

```
string Caption
```

Count property — ListWindow

This property contains the number of elements in the list. This is a read-only property.

Type expected

```
int Count
```

CurrentIndex property — ListWindow

This property contains the zero-based index of the currently highlighted list element, or -1 if nothing is selected. This is a read-only property.

Type expected

```
int CurrentIndex
```

Data property — ListWindow

This property contains an array of strings that represent the contents of the list. This is a read-only property.

Type expected

```
[]Data
```

Height property — ListWindow

This property contains the height, in pixels, of the list window. This is a read-write property.

Type expected

```
int Height
```

Hidden property — ListWindow

This property determines if the list window can be removed from the display. This property only has meaning after the Execute() method has been called and before the list window is closed. This is a read-write property.

Type expected

```
bool Hidden
```

MultiSelect property — ListWindow

This property, if **TRUE**, allows multiple selections from the list. If **FALSE**, only a single selection can be made. This is a read-only property.

Type expected

```
bool MultiSelect
```

Sorted property — ListWindow

This property, if **TRUE**, the elements of the list are sorted as new elements are added. If **FALSE**, elements appear at the offset given in the call to the Add() method. This is a read-only property.

Type expected

```
bool Sorted
```

Width property — ListWindow

This property contains the width, in pixels, of the list window. This is a read-write property.

Type expected

```
int Width
```

Add method — ListWindow

Adds the string *newEl* to the list at the zero based offset position designated by *offset*. *offset* is ignored if the list is a sorted list.

Types expected

```
void Add(string newEl, int offset)
```

Return value

None

Clear method — ListWindow

Removes all elements from the list.

Types expected

```
void Clear()
```

Return value

None

Close method — ListWindow

Removes the *List* Window from the screen.

Types expected

```
void Close()
```

Execute method — ListWindow

Creates the *List* window to be created and displayed to the user.

Types expected

```
void Execute()
```

FindString method — ListWindow

Returns the one-based offset of the string or zero if not found.

Types expected

```
int FindString(string toFind)
```

GetString method — ListWindow

Returns the string at the specified offset or "" if an illegal offset.

Types expected

```
string GetString(int offset)
```

Insert method — ListWindow

This method is invoked when the user presses *Insert*. The default action is to do nothing.

Types expected

```
void Insert()
```

Return value

None

Remove method — ListWindow

Removes the element from the specified zero based offset.

Types expected

```
bool Remove(int offset)
```

Accept event — ListWindow

This event is raised when the user presses *Enter* or double-clicks on a list element. Default action is to close the list.

Types expected

```
void Accept()
```

Return value

None

Cancel event — ListWindow

This event is raised when the user presses *Escape*. Default action is to close the list.

Types expected

```
void Cancel()
```

Return value

None

Closed event — ListWindow

This event is raised when the *ListWindow* is destroyed.

Types expected

```
void Closed()
```

Return value

None

Delete event — ListWindow

This event is raised when the user presses *Delete*. Default action is to do nothing.

Types expected

```
void Delete()
```

Return value

None

KeyPressed event — ListWindow

This event is raised when the user presses a key other than *Delete*, *Insert*, *Accept*, or *Cancel* while the *ListWindow* is active.

Types expected

```
bool KeyPressed(string keyName)
```

keyName Indicates a key in the standard key format (<a> or <Ctrl-a>).

Return value

A return value of **TRUE** indicates that the script has processed the key and that no further processing is desired. A return value of **FALSE** indicates that normal processing (whatever that is) should take place.

LeftClick event — ListWindow

This event is raised when the user clicks the left mouse button on the list window. The parameters describe the mouse's position at the time.

Types expected

```
void LeftClick(int xPos, int yPos)
```

Return value

None

Move event — ListWindow

This event is raised whenever the selection in the list is changed. Default action is to do nothing.

Types expected

```
void Move()
```

Return value

None

RightClick event — ListWindow

This event is raised when the user clicks the right mouse button on the list window. The parameters describe the mouse's position at the time.

Types expected

```
void RightClick(int xPos, int yPos)
```

Return value
None

PopupMenu class

The class manages pop-up menus. In the Paradigm C++ IDE, pop-up menus are known as SpeedMenus.

Syntax
```
PopupMenu(int Top, int Left, string [] InitialValues)
```

Top, Left Initial coordinates of the pop-up menu.

InitialValues An array of strings specifying the initial contents of the pop-up menu.

Properties
`[] Data` *Read-only*

Methods
```
void Append(string newChoice)
int FindString(string toFind)
string GetString(int offset)
bool Remove(int offset)
string Track()
```

Events
None

Data property PopupMenu

This property contains an array of strings that specifies the choices that will be offered in the menu. This is a read-only property.

Type expected
```
[] Data
```

Append method PopupMenu

This method appends a new choice to the menu's options.

Types expected
```
void Append(string newChoice)
```

Return value
None

FindString method PopupMenu

This method looks for the string indicated in *toFind*.

Types expected
```
int FindString(string toFind)
```

Return value
The one-based offset of the string found or zero if not found.

GetString method — PopupMenu

This method returns the string at the specified zero based offset or "" if the offset is illegal.

Types expected
```
string GetString(int offset)
```

Remove method — PopupMenu

This method removes the specified zero based offset.

Types expected
```
bool Remove(int offset)
```

Return value
TRUE if the element is removed, FALSE, otherwise

Track method — PopupMenu

This method displays the pop-up menu to the user and tracks responses. It will return after the user has made a choice or cancelled the menu.

Types expected
```
string Track()
```

Return value
The string selected or the empty string ("") if the user cancels the menu.

ProjectNode class

Syntax
```
ProjectNode(nodeName, EditView associatedView)
```

nodeName A string indicating the full name of the node (as in MyProg.exe). If no name is specified, *ProjectNode* uses the top level IDE node.

Properties

`[] ChildNodes`	*Read-only*
`string IncludePath`	*Read-only*
`string InputName`	*Read-only*
`bool IsValid`	*Read-only*
`string LibraryPath`	*Read-only*
`string Name`	*Read-only*
`bool OutOfDate`	*Read-write*
`string OutputName`	*Read-only*
`string SourcePath`	*Read-only*
`string Type`	*Read-only*

Methods

```
bool Add(string nodeName [, string type])
bool Build(bool suppressUI)
bool Make(bool suppressUI)
void MakePreview()
bool Remove([string nodeName])
bool Translate(bool suppressUI)
```

Events

```
void Built(bool status)
void Made(bool status)
void Translated(bool status)
```

ChildNodes property — ProjectNode

This property indicates all the child nodes of the current node. It is an array of strings containing the InputNames of the child nodes. It is a read-only property.

Type expected

```
[] ChildNodes
```

IncludePath property — ProjectNode

This property indicates the path to use for include files for the currently loaded project. It is a read-only property.

Type expected

```
string IncludePath
```

InputName property — ProjectNode

The node's relative path name of the input file including extension, as in "Myfile.cpp" or "SOURCE\MYFILE.CPP". It is a read-only property.

Type expected

```
string InputName
```

IsValid property — ProjectNode

This property indicates if a node is valid. A node becomes invalid if the project file it is associated with is closed or if the node is deleted. It is a read-only property.

Type expected

```
bool IsValid
```

LibraryPath property — ProjectNode

This property indicates the path to use for libraries for the currently loaded project. It is a read-only property.

Type expected

```
string LibraryPath
```

Name property — ProjectNode

This property indicates the node's relative path name with an extension, as in "Myfile" or "SOURCE\MYFILE". It is a read-only property.

Type expected

```
string Name
```

OutOfDate property — ProjectNode

This property can be checked, or set, to determine the date of a node. This property is used by the make engine to determine if a node needs to be rebuilt. This is a read-write property.

Type expected

```
bool OutOfDate
```

OutputName property — ProjectNode

This property indicates the relative path name of the output file including extension, as in "MYFILE.CPP" or "Source\Myfile.cpp". It is a read-only property. You can always generate the absolute filename by prepending the result of the *IDEApplication's CurrentDirectory* property to *InputName*, as in

```
absName = IDE.CurrentDirectory + node.InputName;
```

Type expected

```
string OutputName
```

SourcePath property — ProjectNode

This property indicates the path where the source files for the currently loaded project reside. It is a read-only property.

Type expected

```
string SourcePath
```

Type property — ProjectNode

This property indicates the type of node (".CPP", ".H", "SourcePool", ".LIB", and so on). It contains the empty string ("") when the node is invalid. It is a read-only property.

Type expected

```
string Type
```

Add method — ProjectNode

This method adds a node to this project node with a specified name. If *type* is omitted, it is derived from the *nodeName*.

Types expected

```
bool Add(string nodeName [, string type])
```

Build method — ProjectNode

This method causes the node to be built, made, or translated by the IDE's Make engine according to the rules of the node. If *suppressUI* is **TRUE**, the build status dialog will not be displayed during the build process.

Types expected

```
bool Build(bool suppressUI)
```

Return value

TRUE if the node is built successfully, **FALSE** otherwise.

Make method — ProjectNode

This method causes the node to be built, made, or translated by the IDE's Make engine according to the rules of the node *if* the node's OutOfDate property is **TRUE**. If *suppressUI* is **TRUE**, the build status dialog will not be displayed during the make process.

Types expected

```
bool Make(bool suppressUI)
```

Return value

TRUE if the node is made successfully, **FALSE** otherwise

MakePreview method — ProjectNode

This method provides information about what files will be processed if you Make or Build this node. It performs the same dependency checks as a Make and generates a report to the Message window listing the nodes that need to be rebuilt to keep the project up to date.

Types expected

```
void MakePreview()
```

Return value

None

Remove method — ProjectNode

If *nodeName* is not specified, *Remove* removes the node from the project If *nodeName* is specified, *Remove* finds it and removes it from the project.

Types expected

```
bool Remove([string nodeName])
```

Translate method — ProjectNode

This method causes the node to be built, made, or translated by the IDE's Make engine according to the rules of the node. If *suppressUI* is **TRUE**, the build status dialog will not be displayed during the make process.

Types expected

```
bool Translate(bool suppressUI)
```

Return value

TRUE if the node is translated successfully, **FALSE** otherwise.

Built event — ProjectNode

This event is raised after a build has been performed on the node. *status* describes the result of the build. *status* is set to **TRUE** if the build completed successfully or with warnings, and **FALSE** if there were errors. Default behavior is to do nothing.

Types expected

```
void Built(bool status)
```

Return value

None

Made event — ProjectNode

This event is raised after a make has been performed on the node. *status* describes the result of the make operation. *status* is set to **TRUE** if the build completed successfully or with warnings, and **FALSE** if there were errors. Default behavior is to do nothing.

Types expected

```
void Made(bool status)
```

Return value

None

Translated event — ProjectNode

This event is raised after a translate has been performed on the node. *status* describes the result of the translate operation. *status* is set to **TRUE** if the build completed successfully or with warnings, and **FALSE** if there were errors. This event's default behavior is to do nothing.

When the user performs a make or a build, the *ProjectNode* object receives a *Translated* event before it receives the *Built* or *Made* event.

Types expected

```
void Translated(bool status)
```

Return value

None

Record class

This class creates an empty Record object into which keystrokes are saved and assigns it the name specified by the RecordName parameter. If no record name is specified, a default name is automatically assigned ("Record1", "Record2", and so on).

Syntax
```
Record([string RecordName])
```

Properties
`bool IsPaused`	*Read-only*
`bool IsRecording`	*Read-only*
`int KeyCount`	*Read-only*
`string Name`	*Read-write*

Methods
```
void Append( int KeyCode )
string GetCommand( int offset )
int GetKeyCode( int offset )
Record Next( void )
```

Events
None

IsPaused property — Record

This property is set to **TRUE** when KeyboardManager's PauseRecording method is called in order to allow users to enter keystrokes, which will not become part of the recording. This is a read-only property.

Type expected
```
bool IsPaused
```

IsRecording property — Record

This property is set to **TRUE** when the KeyboardManager begins storing keystrokes to the *Record* object in response to a call to KeyboardManager's StartRecord method. This is a read-only property.

Type expected
```
bool IsRecording
```

KeyCount property — Record

This property contains the number of keystrokes stored in this *Record* object. This is a read-only property.

Type expected
```
int KeyCount
```

Name property — Record

This property contains the name of the *Record* object. This is a read-write property.

Type expected
```
string Name
```

Append method — Record

This method appends a keycode to the record buffer. It allows empty record objects to be built programatically or added to through a script.

Types expected

```
void Append(int KeyCode)
```

Return value

None

GetCommand method — Record

This method returns information describing a key stored in the *Record* object. Keys are stored in the order which they are recorded. The first key in the recording is at offset 0.

Types expected

```
string GetCommand(int offSet)
```

Return value

The information returned is transitory, because the meanings of the stored keystrokes may have been altered by the execution of the recording. For instance, if the recording switches to another subsystem with a different key map. The method is intended to be used after a *Record* object has been executed. The answers returned reflect the values as of the last run.

GetKeyCode method — Record

This method returns information describing a key stored in the *Record* object. Keys are stored in the order which they are recorded. The first key in the recording is at offset 0.

Types expected

```
int GetKeyCode(int offset)
```

Return value

Returns the keystroke of the key at the specified offset, or zero if the offset is illegal.

Next method — Record

As *Record* objects are created, they are automatically linked together. This method provides a mechanism for iterating the recordings.

Types expected

```
Record Next(void)
```

Return value

Either the next *Record* object or **NULL** indicating the end of the list.

ScriptEngine class

Syntax
```
ScriptEngine()
```

Properties
`bool AppendToLog`	*Read-write*
`int DiagnosticMessageMask`	*Read-write*
`bool DiagnosticMessages`	*Read-write*
`string LogFileName`	*Read-write*
`bool Logging`	*Read-write*
`string ScriptPath`	*Read-write*
`string StartupDirectory`	*Read-only*

Methods
```
void Debug()
int Execute(string commandLine, bool temporary)
string Execute(string commandLine, bool temporary)
bool IsAClass(string className)
bool IsAFunction(string functionName)
bool IsAMethod(string className, string methodName)
bool IsAProperty(string className, string propertyName)
bool IsLoaded(string scriptFileName)
bool Load(string scriptFileName)
[] Modules(bool libraryOnly)
bool Reset(int resetWhat)
void Symbol(string fileName, string symbols)
bool Unload(string scriptFileName)
```

Events
```
void Loaded(string scriptFileName)
void Unloaded(string scriptFileName)
```

AppendToLog property — ScriptEngine

Used when *Logging* is on. This property determines whether the next message logged to the log file name should replace an existing log file (if one exists) before performing the write. Once the write has been completed, *AppendToLog* is set to **TRUE**, causing subsequent messages to be appended to the log. This is a read-write property.

Type expected
```
bool AppendToLog
```

DiagnosticMessageMask property — ScriptEngine

Controls which types of diagnostic messages to record. This bitmask can be any combination of

OBJECT_DIAGNOSTICS
METHOD_DIAGNOSTICS
MEMBER_DIAGNOSTICS
ARGUMENT_DIAGNOSTICS
LANGUAGE_DIAGNOSTICS

MODULE_DIAGNOSTICS
FULL_DIAGNOSTICS
NO_DIAGNOSTICS

This is a read-write property.

Type expected

```
int DiagnosticMessageMask
```

DiagnosticMessages property — ScriptEngine

Controls whether diagnostic messages should be recorded in the Message window. This is a read-write property.

Type expected

```
bool DiagnosticMessages
```

LogFileName property — ScriptEngine

The name of the log file. Defaults to "\SCRIPT.LOG". This is a read-write property.

Type expected

```
string LogFileName
```

Logging property — ScriptEngine

If set to **TRUE**, script messages will be stored in the log file. This is a read-write property.

Type expected

```
bool Logging
```

ScriptPath property — ScriptEngine

Holds a string containing the name(s) of the directory(s) to be searched for script files. Each directory path is separated from the others by a semicolon (;). This is a read-write property.

Type expected

```
string ScriptPath
```

StartupDirectory property — ScriptEngine

The name of the directory in which the file STARTUP.SPX was found in during initialization. This is a read-write property.

Type expected

```
string StartupDirectory
```

Debug method — ScriptEngine

This method launches the Script Debugger.

Types expected

```
bool Debugger()
```

Return value

TRUE if the Script Debugger can be launched, **FALSE** otherwise.

Execute method — ScriptEngine

Executes the string in *commandLine*, which must be a valid cScript command.

Types expected

```
int Execute(string commandLine, bool temporary)
string Execute(string commandLine, bool temporary)
```

Return value

Returns the value appropriate to whatever *commandLine* evaluates to. If that value is an object, it is converted to a string.

Description

If *temporary* is **TRUE**, the command is run within a new context and must therefore use **import** to access global variables declared in another module. Any global variables it creates will be used for the purposes of the command and then discarded.

If *temporary* is **FALSE** (the default), the command is executed with the scope of Immediate mode and has automatic access to globals from other modules. In this case, any variables created by the command continue to exist after the command has run and can be accessed from Immediate mode.

IsAClass method — ScriptEngine

This method is used to determine if ObjecttScript has seen the class declaration for the class indicated by *className*.

Types expected

```
bool IsAClass(string className)
```

Return value

TRUE if instances of the class can be constructed, **FALSE** otherwise.

IsAFunction method — ScriptEngine

This method is used to determine if ObectScript has seen the function declaration for the function indicated by *functionName*.

Types expected

```
bool IsAFunction(string functionName)
```

Return value

TRUE if the function can be called, **FALSE** otherwise.

IsAMethod method — ScriptEngine

This method is used to determine if the class *className* has as a member of the class the method *methodName*.

Types expected

```
bool IsAMethod(string className, string methodName)
```

Return value

TRUE if the method is a member of the class, **FALSE** otherwise.

IsAProperty method — ScriptEngine

This method is used to determine if the class *className* has as a member of the class the property *propertyName*.

Types expected

```
bool IsAProperty(string className, string propertyName)
```

Return value

TRUE if the property is a member of the class, **FALSE** otherwise.

IsLoaded method — ScriptEngine

This method is used to determine if the specified script file *scriptFileName* has been loaded, or if the file (either the source or the binary) can be found in the ScriptPath.

Types expected

```
bool IsLoaded(string scriptFileName)
```

Return value

TRUE if the file is loaded or can be loaded, **FALSE** otherwise.

Load method — ScriptEngine

This method is used to load the script file *scriptFileName*. If not already loaded, the file (either the source or the binary) is searched for using the ScriptPath.

Types expected

```
bool Load(string scriptFileName)
```

Return value

TRUE if the script was located and loaded or **FALSE** if the script file was not found.

Description

If the script file to be loaded has already been loaded into memory, *Load()* performs an in-place *Reset()*. (The module's position in the module chain is not affected, but all its variables are restored to their original state.) **On** handlers are disconnected or reconnected. All variables local to the module are released and reset. Any code at the module level scope is executed again.

Modules method — ScriptEngine

This method finds all the loaded modules.

Types expected

```
[] Modules(bool libraryOnly)
```

libraryOnly An optional parameter indicating that the method is to fetch only the library modules.

Return value

An array of strings containing the names of the loaded modules.

Reset method — ScriptEngine

This method resets the script session by discarding all modules that match the value of *resetWhat*. If no value is supplied, the method does nothing.

Types expected

```
bool Reset(int resetWhat)
```

resetWhat Can be either LIBRARY_MODULE or SCRIPT_MODULE.

Return value

TRUE if the session is reset or **FALSE** if it is not.

SymbolLoad method — ScriptEngine

This method provides hints about where the definition of a given symbol might be. For example, SymbolLoad("ScriptFile", "Foo, Bar, jump").

Types expected

```
void SymbolLoad(string fileName, string symbols)
```

fileName A script file that should be loaded if the lookup for any of the listed symbols fails.

symbols A comma delimited string of the symbols which may be resolved by loading fileName.

Return value

None

Description

At run time when the Script Engine tries to find a class, function, method, or global variable that it doesn't know about, it consults an internal table constructed by calls to this method.

Unload method — ScriptEngine

Tries to unload the specified script file. Future references from other scripts to variables, functions or classes defined in the unloaded script file are no longer valid.

Types expected

```
bool Unload(string scriptFileName)
```

Return value

FALSE when the script file is not found to have been loaded, **TRUE** otherwise.

Loaded event — ScriptEngine

This event is raised whenever a new script module is successfully loaded.

Types expected

```
void Loaded(string scriptFileName)
```

Return value

None

Unloaded event — ScriptEngine

This event is raised when a module has been unloaded.

Types expected

```
void Unloaded(string scriptFileName)
```

Return value

None

SearchOptions class

Syntax

```
SearchOptions()
```

Properties

`bool CaseSensitive`	*Read-write*
`bool FromCursor`	*Read-write*
`bool GoForward`	*Read-write*
`bool PromptOnReplace`	*Read-write*
`bool RegularExpression`	*Read-write*
`bool ReplaceAll`	*Read-write*
`string ReplaceText`	*Read-write*
`string SearchReplaceText`	*Read-write*
`string SearchText`	*Read-write*
`bool WholeFile`	*Read-write*
`bool WordBoundary`	*Read-write*

Methods

```
void Copy(SearchOptions optionsToCopyFrom)
```

Events

None

CaseSensitive property — SearchOptions

If **TRUE**, a case-sensitive search is performed. This is a read-write property.

Type expected

```
bool CaseSensitive
```

FromCursor property — SearchOptions

If **TRUE**, the search is made from the current cursor position. This is a read-write property.

Type expected

```
bool FromCursor
```

GoForward property — SearchOptions

If **TRUE**, the search is "forward" towards the end of the file. This is a read-write property.

Type expected

```
bool GoForward
```

PromptOnReplace property — SearchOptions

If **TRUE**, you are prompted to confirm each instance where the SearchReplaceText will be replaced by the ReplaceText before the replacements are made. This is a read-write property.

Type expected

```
bool PromptOnReplace
```

RegularExpression property — SearchOptions

If **TRUE**, regular expressions are used in matching the SearchText or SearchReplaceText with the text to be searched. This is a read-write property.

Type expected

```
bool RegularExpression
```

ReplaceAll property — SearchOptions

If **TRUE**, all text which matches the SearchReplaceText is replaced with the ReplaceText without any prompting for confirmation. This is a read-write property.

Type expected

```
bool ReplaceAll
```

ReplaceText property — SearchOptions

This property contains text which replaces instances of the SearchReplaceText string(s) found in the text being searched. This is a read-write property.

Type expected

```
string ReplaceText
```

SearchReplaceText property — SearchOptions

This property contains the text to search for in a search and replace operation (not a search-only operation). This is a read-write property.

Type expected

```
string SearchReplaceText
```

SearchText property — SearchOptions

This property contains the text to search for in a search operation (*not* a search and replace operation). This is a read-write property.

Type expected

```
string SearchText
```

WholeFile property — SearchOptions

If **TRUE**, the whole file is searched for SearchText or SearchReplaceText, regardless of the cursor position. This is a read-write property.

Type expected

```
bool WholeFile
```

WordBoundary property — SearchOptions

If **TRUE**, a match between SearchText or SearchReplaceText and the text being searched only occurs if the characters in *SearchText* make up an entire word (that is, they are surrounded by whitespace) and are not embedded in a larger word. This is a read-write property.

Type expected

```
bool WordBoundary
```

Copy method — SearchOptions

This method creates a copy of the current *SearchOptions*.

Types expected

```
void Copy(SearchOptions optionsToCopyFrom)
```

Return value

None

StackFrame class

Syntax

```
StackFrame(int howFarBack)
```

howFarBack refers to the number of stack frames to go back through. 0 gets information for this call. 1 retrieves the stack passed to this function's caller, and so on. When *howFarBack* is less than the depth of the stack, the object is not valid.

Properties

`int ArgActual`	*Read-only*
`int ArgPadding`	*Read-only*
`string Caller`	*Read-write*
`bool IsValid`	*Read-only*

Methods

```
StackElement GetParmStackFrame_GetParm(int parmNumber)
string InqTypeStackFrame_InqType(int arg)
bool SetParmStackFrame_SetParm(int parmNumber, newValue)
```

Events

None

ArgActual property — StackFrame

This property indicates the number of objects on the cScript stack belonging to this call frame. It is a read-only property.

Type expected

```
int ArgActual
```

Description

ArgActual is the number of arguments that were actually passed to a method. cScript either pads or truncates arguments as necessary, so it must keep track of the number actually passed.

For example, if you have a call in your code to

```
MyMethod("hi");
```

Its declaration shows the following:

```
MyMethod(first, second, third, fourth){
  print first, second, third, fourth;
}
```

If you were to insert `x = new StackFrame(0);` into the call to *MyMethod*, the value of *x.ArgActual* would be 1 since only one argument is passed.

ArgPadding property — StackFrame

This property indicates the number of objects cScript had to pad or truncate from the original call stack to resolve any discrepancy between the number of arguments in the declaration and the number of arguments in the call. This is a read-only property.

Type expected

```
int ArgPadding
```

Caller property — StackFrame

This property indicates the name of the method owning the stack frame. This is a read-write property.

Type expected

```
string Caller
```

Description

Caller contains the empty string ("") if the call is a top level one. When the value is set, it is reflected in subsequent *StackFrame* calls until the current stack frame is popped off, at which point the value of *Caller* is reset to its original value.

IsValid property — StackFrame

This property is **FALSE** if the object was constructed with an invalid stack frame depth or if the stack frame has gone out of scope. It is **TRUE** otherwise. This is a read-only property.

Type expected

```
bool IsValid
```

InqType method — StackFrame

This method returns a descriptor for the argument specified or, if *arg* is greater than or equal to ArgActual, for an argument that maps to "Out of range".

Types expected

```
string InqType(int arg)
```

GetParm method — StackFrame

This method returns the object at the specified stack frame offset.

Types expected

```
StackElement GetParm(int parmNumber)
```

SetParm method — StackFrame

This method sets the value of the object at the specified stack frame offset.

Types expected

```
bool SetParm (int parmNumber, newValue)
```

String class

Syntax

```
String(string theText)
String(String anotherString)
```

Properties

`int Character`	*Read-write*
`int Integer`	*Read-write*
`bool IsAlphaNumeric`	*Read-only*
`int Length`	*Read-only*
`string Text`	*Read-write*

Methods

```
String Compress()
bool Contains(string charactersToLookFor, int mask)
int Index(string substr[, int direction])
String Lower()
String SubString(int startPos[, int length])
String Trim([bool fromLeft])
String Upper()
```

Character property — String

This property indicates the integer value of character 0 of the string. It is a read-write property.

Type expected

```
int Character
```

Description

When the value of *Character* is set, it changes the whole string to the new value. For example, if you start with a string *Str* containing the text `"FOO"`, the value of *Str.Text* is `"FOO"` and the value of *Str.Character* is `'F'`. If you then set the value of *Str* with `Str.Character = 'X'`, the value of *Str.Text* is now `"X"` and not `"XOO"`.

Integer property — String

This property indicates the numerical equivalent of the character string that this object represents, or zero if the string does not contain numerals. This is a read-write property.

Type expected

```
int Integer
```

IsAlphaNumeric property — String

This property is **TRUE** if the text of the String is made up entirely of alphanumeric characters (determined by checking the system's current locale). Its value is **FALSE** otherwise. This is a read-only property.

Type expected

```
bool IsAlphaNumeric
```

Length property — String

This property indicates the length of the string (equivalent to *strlen*). This is a read-only property.

Type expected

```
int Length
```

Text property — String

The character string that this object represents. This is a read-write property.

Type expected

```
string Text
```

Compress method — String

This method returns a new *String* that mimics this one, but with redundant white space removed.

Types expected

```
String Compress()
```

Contains method — String

This method returns **TRUE** if the string contains one of the characters specified or **FALSE** if it does not.

Types expected

```
bool Contains(string charactersToLookFor, [int mask])
```

mask can be any of the following constants:

BACKWARD_RIP	Rip from left to right.
INVERT_LEGAL_CHARS	Interpret the *legalChars* string as the inverse of the string you wish to use for *legalChars*. In other words, specify "t" to mean any ASCII value between 1 and 255 except `'t'`.
INCLUDE_LOWERCASE_ALPHA_CHARS	Append the characters `abcdefghijklmnopqrstuvwxyz` to the *legalChars* string.
INCLUDE_UPPERCASE_ALPHA_CHARS	Append the characters `ABCDEFGHIJKLMNOPQRSTUVWXYZ` to the *legalChars* string
INCLUDE_ALPHA_CHARS	Append both uppercase and lowercase alpha characters to the *legalChars* string.
INCLUDE_NUMERIC_CHARS	Append the characters `1234567890` to the *legalChars* string.
INCLUDE_SPECIAL_CHARS	Append the characters `` `-=[]\;',./~!@#$%^&*()_+{}|:"<>? `` to the *legalChars* string.

Index method — String

This method scans the string for an embedded occurrence of the *substr* in the direction specified by *direction*. It defaults to SEARCH_FORWARD. It does not accept regular expressions.

Types expected

```
int Index(string substr[, int direction])
```

Return value

0 if *substr* is not found or, if found, the one based offset + 1 of the substring.

Lower method — String

This method returns a new string that mimics this one, but is all lowercase letters.

Types expected

```
String Lower()
```

SubString method — String

This method returns a new string consisting of the substring indicated by *startPos* and optionally *length*.

Types expected

```
String SubString(int startPos[, int length])
```

startPos	The starting point in the string of the substring.
length	The number of characters in the substring. Defaults to MAX_EDITOR_LINE_LEN (1024). If length is not specified, *SubString* continues to the end of the string.

Trim method — String

This method returns a new string that mimics this one, but either without trailing white space or without leading white space, as indicated by the *fromLeft* argument. The default is to trim trailing white space (*fromLeft* is **FALSE**).

Types expected

```
String Trim([bool fromLeft])
```

Upper method — String

This method returns a new string that mimics this one, but is all uppercase letters.

Types expected

```
String Upper()
```

TransferOutput class

Internally created by the IDE after processing a transfer tool, *TransferOutput* is passed to the IDE event TransferOutputExists().

Syntax
```
TransferOutput()
```

Properties
`int MessageId`	*Read-only*
`string Provider`	*Read-only*

Methods
```
string ReadLine()
```

Description
An object of type *TransferOutput* is internally created by the IDE whenever a transfer operation is performed.

When the IDE starts a transfer, it outputs a message to the Message window saying "Transferring to *ToolName*..."

When a transfer happens, the IDE captures all its output and stores it in an internal buffer. The contents of this buffer may be accessed by using *TransferOutput* object's *ReadLine* method. This method returns the next line of text until the stream is exhausted, at which point it returns **NULL**.

The IDE contains built-in processing for tools it commonly transfers to. These tools include PASM and GREP. The script sample files FILTSTUB.SPP and FILTERS.SPP show uses of this class in action.

Transfer definition
The term used when another application is spawned from within the IDE. Command-line tools such as the WinHelp compiler or a DOS box (COMMAND.COM) are commonly invoked from the IDE during a build process.

MessageId property — TransferOutput

This property is the "owning" message stored to the message system. This ID is intended to be used as the parentMessage parameter of IDE.MessageCreate so that the messages produced by the transfer can be grouped with the transfer message rather than at the same level. It is a read-only property.

Type expected
```
int MessageId
```

Provider property — TransferOutput

This property indicates the name of the tool that was spawned by the transfer; for example, "COMMAND.COM". It is a read-only property.

Type expected
```
string Provider
```

ReadLine method — TransferOutput

This method reads the next line of text that was produced by the transfer. When a transfer happens, the IDE captures all its output and stores it in an internal buffer. The contents of this buffer may be accessed by repeatedly calling *ReadLine*, which returns the next line of text until the stream has been exhausted, at which point it returns **NULL**.

Types expected

```
string ReadLine()
```

Return value

ReadLine returns the next line of text that was produced by the transfer. If the line is empty, *ReadLine* returns the empty string (""). If there is no more input to read, it returns **NULL**.

TimeStamp class

This class indicates the current time. It initializes to the system time at the time of construction.

Syntax

```
TimeStamp()
```

Properties

`int Day`	*Read-write*
`int Hour`	*Read-write*
`int Hundredth`	*Read-write*
`int Millisecond`	*Read-write*
`int Minute`	*Read-write*
`int Month`	*Read-write*
`int Second`	*Read-write*
`int Year`	*Read-write*

Methods

```
int Compare(TimeStamp tstamp)
string DayName()
string MonthName()
```

Day property — TimeStamp

This property indicates the current day in the range of 0 (Sunday) to 6 (Saturday). It is a read-write property.

Type expected

```
int Day
```

Hour property — TimeStamp

This property indicates the current hour in the range of 0 (Midnight) to 23 (11:00 PM). It is a read-write property.

Type expected

```
int Hour
```

Hundredth property — TimeStamp

This property indicates the current hundredth of an hour in the range of 0 to 99. It is a read-write property.

Type expected

```
int Hundredth
```

Millisecond property — TimeStamp

This property indicates the number of milliseconds after the current second in the range of 0 to 999. It is a read-write property.

Type expected

```
int Millisecond
```

Minute property — TimeStamp

This property indicates the number of minutes after the current hour in the range of 0 to 59. This is a read-write property.

Type expected

```
int Minute
```

Month property — TimeStamp

This property indicates the current month of the year in the range of 0 (January) to 11 (December). It is a read-write property.

Type expected

```
int Month
```

Second property — TimeStamp

This property indicates the number of seconds after the current minute in the range of 0 to 59. It is a read-write property.

Type expected

```
int Second
```

Year property — TimeStamp

This property indicates the current year. It is a read-write property.

Type expected

```
int Year
```

Compare method — TimeStamp

Compares the time properties of the calling TimeStamp object with those of the *tstamp* argument.

Types expected

```
int Compare(TimeStamp tstamp)
```

Return value

-1 if the calling TimeStamp is newer than *tstamp*, 0 if the calling TimeStamp is the same age as *tstamp*, and 1 if the calling TimeStamp is older than *tstamp*.

DayName method — TimeStamp

This method returns the name of the current day of the week.

Types expected

```
string DayName()
```

Return value

"Monday", "Tuesday:, and so on.

MonthName method — TimeStamp

This method returns the name of the current month.

Types expected

```
string MonthName()
```

Return value

"January", "February", and so on.

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y