

Scribe Scripting Library

Scribe is a C# based scripting library. I have decided to implement it over using another library for the fact that I can simply compile it against Mono for cross-platform support (Windows, Mac, Android, and iOS) as well as it does not inherently have any features and so I do not need to worry about sandboxing it. It is also designed around a resource system like XNA/MonoGame and so must be compiled into an XNB file.

Scribe Scripting Library

Contents

Comments.....	3
Supported Operators.....	4
Conditionals.....	5
Loops.....	6
Variables.....	7
Commands.....	8
Labels.....	10
Packages.....	13
Random.....	13
Console.....	13
IO.....	14
BIO.....	14

Comments

Comments are very simple and must follow these rules:

- Start with a //
- Start at the beginning of the line
- They cannot span multiple lines

Comments and blank lines are ignored during parsing and compilation.

Multi-line comments are on the wish-list.

Scribe Scripting Library

Supported Operators

Scribe supports most of your standard arithmetic operators:

Arithmetic Operators

Op	Name	Example
=	Assignment	@index = 10
+	Add	@index = 10 + 1
+=	Add-assign	@index += 1
-	Subtract	@index = 10 - 1
-=	Subtract-assign	@index -= 1
*	Multiply	@index = 10 * 2
*=	Multiply-assign	@index *= 2
/	Divide	@index = 10 / 2
/=	Divide-assign	@index /= 2
%	Modulus	@index = 10 % 2
%=	Modulus-assign	@index %= 2
^	Exponent	@index = 10 ^ 2
^=	Exponent-assign	@index ^= 2

Conditional Operators

Op	Name	Example
<	Less than	if 10 < 2
<=	Less than or equal to	if 10 <= 2
==	Equal to	if 10 == 2
!=, <>	Not equal to	if 10 != 2
>	Greater than	if 10 > 2
>=	Greater than or equal to	if 10 >= 2
&&	And	if 10 > 2 && 10 < 3
	Or	if 10 > 2 10 < 3

As of version 1 iteration 467 we have preliminary support for tertiary assignment. Currently the following format is acceptable: "@[destination variable] = [lhs] [conditional operator] [rhs] ? [on true value] : [on false value]". In a future version any "conditional statement" (such as "@boolVar" or "@boolVar || 10 == 2" or "2 < 10 && 3 > 1".)

Scribe Scripting Library

Conditionals

Scribe supports the if-elseif-else conditional branch system. Conditions can be a single value/variable (translates to not null and not zero), [value 1] [conditional op] [value 2], [value 1] [conditional op] [value 2] [&&/|] [value 3] [conditional op2] [value4], and any combination of those. Below are a few examples:

```
if @hasBeenLoaded && @score > 0
    // Do something here
elseif @hasBeenLoaded && @score < 0 && @onMap3
    // Do something else here
else
    // Do anything
```

All operations following the command statement that start with a tab are considered part of the branch. Blank lines will not change scope though.

```
if @hasBeenLoaded && @score > 0
    // This line is part of the branch
    // And so is this line
// But this line is not.
if @hasBeenLoaded && @score < 0 && @onMap3
    // This line is part of the branch
    // And so is this line

else
    // This is a broken statement as the line above
    // else does not have a tab.
    // This now works properly, as the line above
    // else will be ignored.
```

~~Nested conditionals are not currently supported, but are on the wish list.~~ Nested loops and conditionals are supported as of version 1, revision ~~371~~ 467. You can also still call yield inside of loops and conditionals.

Scribe Scripting Library

Loops

There are two commands available for loops: while and yield. The first, while, continues to run the statements below it until it's condition is false. Here is an example:

```
@index = 0
while @index < 10
    @index += 1
```

Yield is simply a loop without any child. This is a touchy command that is best used when the variable is being modified from the program. Here is an example:

```
MovePlayer 10 100
// Using yield:
yield @playerState == Moving

// Using loop:
// For this to be a well formed script, the block should do SOMETHING
while @playerState == Moving
    @block = 0
```

~~Nested loops are not currently supported, but are on the wish list. You can have a yield inside of a loop though.~~ Nested loops and conditionals are supported as of version 1, revision ~~371~~ 467. You can also still call yield inside of loops and conditionals.

Scribe Scripting Library

Variables

Variables are identified by the @ symbol and do not have an 'undefined' state; all variable interactions will react as if the variable is null until it has been set. Variables can be used in place of any value and can be used with any of the operators.

If an operation or command is expected to assign its result to a variable, you must pass the name of the variable only (without the @ sign); you are basically supplying the name of the variable as a string. For example, the extra package Console contains "Console.ReadLine" which reads user input and assigns it to a variable. You would call it as "Console.ReadLine userInput" and not "Console.ReadLine @userInput".

Here are some examples of variables:

```
if @myName != null
    // This line will never be reached on the first run as @myName is
    always null on the first run
    Console.WriteLine 'Hello there ' @myName
else
    // This line will never be reached after the first run as @myName will
    be set
    Console.WriteLine 'Good morrow to you sir'
@myName = 'Donny'
@age = 17
@age = @age + 4
@age += 5
// Age is now 26
```

Scribe Scripting Library

Commands

Commands are the heart of the library; they are how your script and program interact. The library does not have any built in commands*; it is meant to support the absolute minimal and then you implement the rest.

Commands are called from scripts via "[command_name] [arg1] [arg2] ... [argN]" (though arguments are not required) and support variable arguments, strong-typed arguments, and arguments with default values. Here are a few examples:

```
// Simple example
public void Write(string value)
{
    Console.Write(value ?? string.Empty);
}
// Scribe chunk: Write 10
// Output: 10
// Scribe chunk: Write Donny Beals
// Output: [An exception will be thrown. Write is expecting 1 value, but 2
are provided.]

// Variable arguments
public void Write(params string[] values)
{
    foreach(var value in values)
        Console.Write(value == null ? "" : value.ToString());
}
// Scribe chunk: Write Hello world
// Output: Helloworld
// Scribe chunk: Write 'Hello world'
// Output: Hello world

public void Write(string value, bool addQuotes = false)
{
    if(addQuotes)
        Console.Write(string.Format("\"{0}\"", value ?? string.Empty));
    else
        Console.Write(value ?? string.Empty);
}
// Scribe chunk: Write Donny
// Output: Donny
// Scribe chunk: Write Donny true
// Output: "Donny"
```

Scribe Scripting Library

* Though there are no built in commands, several packages of commands are included in the library that can be registered to a virtual machine. This way you start with a sandboxed environment rather than trying to create a sandboxed environment. The a section below covers these packages.

Scribe Scripting Library

Labels

Labels were born from the fact that the first version of Scribe did not support nested loops, nested conditionals, and functions/routines. This allows for better script control without adding much complication.

Labels are created by giving them a name and putting the colon (:) after them and can be jumped to using the goto keyword:

```
my_label:
// This code falls under a label
@iterations += 1
if @iterations < 10
    goto my_label
// we're no longer looping :D

another_label:
// This is another label
@iterations = 0
goto my_label
```

During processing label definitions are completely ignored (the system continues on to the next line.) The above example demonstrates this nicely: it creates an infinite loop. Once @iterations is 10 the loop will break, but then the code under another_label is hit and will set @iterations to 0, jump to my_label, and repeat.

As of version 1 iteration 412 you can call labels using variables that are storing strings. The following is a legal script:

```
@currentState = initialize
loop:
goto @currentState

initialize:
// We are initializing here
@currentState = state2
goto loop

state1:
// Do some stuff here
@currentState = exit
goto loop

state2:
// Do some other stuff here
```

Scribe Scripting Library

```
@currentState = state1  
goto loop  
  
exit:  
// Just a placeholder to exit
```

Wish List

These are the features that I would like to eventually implement. These are not promised in any way, they are just features that I believe would be useful. If an item from the wish list is added, a new line will be added with the version/revision it was added and any comments.

1. Multiline comments
2. Comments at the end of lines
3. Functions/methods/subroutines
4. Break command (need the parser to track the current loop so that it can break properly when break is called from other structures like a conditional.)
5. Script importing (import scripts in-place into the file, similar to PHP's include system.)
6. Object system (something similar to LUA's metatable system)

Scribe Scripting Library

Packages

As stated previously, there are no built in commands that come with the library. If you simply create a virtual machine, it will only be able to create/modify variables and do branching/looping. However, there are a few packages included that provide extra functionality. To register the commands in one of these packages, you must call the static C# method Register(), passing in your virtual machine instance and any necessary values.

Random

Random provides access to the random number generator (provided via System.Random.) When the package is registered it is seeded using DateTime.Now.Ticks; you can access this seed via the script command Random.GetSeed [variable_name] and you can set it to a new value via Random.SetSeed [new_seed].

There are 4 commands provided to generate a random number:

- **Random.Next destinationVariableName** - Generate a random, non-negative number and stores it in the specified variable.
- **Random.NextInRange destinationVariableName, min, max** - Generates a random number between min and max (including min, but excluding max; so NextInRange var 0 10 can return 0, but cannot return 10) and stores it in the specified variable.
- **Random.RollDie destinationVariableName, faceCount** - Generates a random number between 1 and faceCount (including 1 and faceCount) and stores it in the specified variable.
- **Random.RollDice destinationVariableName, dieCount, faceCount** - Generates random numbers between 1 and faceCount (including 1 and faceCount) dieCount times and stores the sum in the specified variables. So Random.RollDice var 3 6 will generate 3 random numbers and store their sum in var.

To register the random commands call Scribe.Packages.Random.Register(virtualMachine).

Console

Console provides access to the System.Console class. It allows you to clear the console (on Windows), write values & lines, read keys (on Windows) and lines, and change the background (on Windows) and text color (on Windows).

There are 7 commands provided:

- **Console.Clear** - Clears the console and places the cursor back at the top left cell. **This command only works on Windows machines.**
- **Console.Write [...]** - Accepts any number of arguments and writes them, in sequence, to the console.
- **Console.WriteLine [...]** - Accepts any number of arguments and writes them, in sequence, to the console. A single newline is written to the console after all of values.

Scribe Scripting Library

- **Console.ReadKey [destinationVariableName:null, intercept:false]** - Accepts two optional variables; reads a single key from the console, stores it in the specified variable (if non-null) and stops it from displaying on screen (if intercept is true.) **This command only works on Windows machines.**
- **Console.ReadLine destinationVariableName** - Reads a line of input from the Console and stores it in the specified variable.
- **Console.SetBackgroundColor colorName** - Accepts a color name and assigns it to the console's background color. **This command only works on Windows machines. Acceptable values are: Black, DarkBlue, DarkGreen, DarkCyan, DarkRed, DarkMagenta, DarkYellow, Gray, DarkGray, Blue, Green, Cyan, Red, Magenta, Yellow, and White.**
- **Console.SetTextColor colorName** - Accepts a color name and assigns it to the console's text/foreground color. **This command only works on Windows machines. Acceptable values are: Black, DarkBlue, DarkGreen, DarkCyan, DarkRed, DarkMagenta, DarkYellow, Gray, DarkGray, Blue, Green, Cyan, Red, Magenta, Yellow, and White.**

IO

The IO package provides commands to check for a file, open a file and write information to it. The IO library is sandboxed in the directory supplied when registering; directory names are stripped in every command and special characters are omitted.

The IO package does not currently support reading from a file.

There are 5 commands provided:

- **IO.CheckForFile destinationVariableName, fileName** - Checks to see if the file specified exists and stores the result in the specified variable.
- **IO.OpenFile fileName [, append:false]** - Opens the specified file for writing, appending to it if append is true.
- **IO.WriteToFile fileName, value** - Writes the specified value to the specified file. The file must be opened via IO.OpenFile first.
- **IO.WriteLineToFile fileName, value** - Writes the specified value to the specified file, followed by a newline character. The file must be opened via IO.OpenFile first.
- **IO.CloseFile fileName** - Closes the specified file. The file must be opened via IO.OpenFile first.

To register the IO commands call `Scribe.Packages.IO.Register([file directory], virtualMachine)`.

BIO

The BIO library provides commands to check for a binary file, open a binary file, write to a binary file, and read from a binary file. Like the IO library, the BIO library is sandboxed in the directory supplied when registering; directory names are stripped in every command and special characters are omitted.

There are 5 commands provided:

Scribe Scripting Library

- **BIO.CheckForFile destinationVariableName, fileName** - Checks to see if the file specified exists and stores the result in the specified variable.
- **BIO.OpenFile fileName, fileMode** - Opens the specified file for reading/writing using the specified file mode. **Acceptable values for fileMode are: CreateNew (throws an exception if the file exists), Create (the file is created or overwritten), Open (throws an exception if the file doesn't exist), OpenOrCreate (the file is created or opened), Truncate (write only; file must exist), Append (write only, creates the file if it doesn't exist.)**
- **BIO.WriteToFile fileName, value** - Writes the type of value and the specified value to the specified file. The file must be opened via BIO.OpenFile first. **Only null, booleans, numbers, and strings are supported at the moment.**
- **BIO.ReadFromFile fileName [, destinationVariableName]** - Reads a value type from the file and then reads the value from the file. The value is then stored in destinationVariableName (if provided.) The file must be opened via BIO.OpenFile first.
- **BIO.CloseFile fileName** - Closes the specified file. The file must be opened via BIO.OpenFile first.

To register the BIO commands call `Scribe.Packages.BIO.Register([file directory], virtualMachine)`.

Array

The Array library provides commands to create and manipulate arrays. There are two options when creating an array: static (implemented as a normal array) or resizable (implemented as a list.) **This package has not been fully tested as of yet.**

There are 8 methods provided:

- **Array.Create destinationVariableName, size, canResize [, values...]** - Creates an array (or list if canResize is true) with the specified size, setting the nodes using the optional specified values.
- **Array.GetValue destinationVariableName, array index** - Looks up the value at the specified index of array and places it in the specified variable. **Index is zero-based.**
- **Array.SetValue array, index, newValue** - Assigns the new value to the specified index of the supplied array. **Index is zero-based.**
- **Array.AddValue array, newValue** - Adds a new value on to the end of a list (only valid if the passed array was created with canResize = true or was set via C# as a List<object>.)
- **Array.Fill array, value** - Sets every node in an array to the specified value (valid for both arrays and lists, but really only useful for arrays.)
- **Array.Clear array** - If the array is a list, it removes all elements, otherwise it sets every node of the array to null.
- **Array.Remove array [, index = -1]** - **Only valid for lists.** Removes either the element at the specified index or the very last element (if index is -1.)
- **Array.Count destinationVariableName, array** - Counts the number of elements in the supplied array, placing the count in destinationVariableName.

To register the Array commands call `Scribe.Packages.Array.Register(virtualMachine)`.

Samples

Samples have been removed and will be made available on the web once a page is set up.