Ruby Basics:

A Guide to Ruby in Swift

DATE	06/11/2019; 19/04/2022
PRINCIPAL AUTHORS	Darrell James
OTHER AMENDS	Justin Avery
VERSION	1.0; 1.1
CHANGELOG	Updating styles 19/04/2022

RT software

Unit 6, Hurlingham Business Park, Sulivan Road, London SW6 3DU

Contents

Overview

In this guide we will take an overview of how to write user code for Swift, what it can be used for and how it works. This guide will also demonstrate some concepts that will help the reader understand the ruby code that Swift produces when saving a graphic. This will not be a guide on how to write comprehensive Ruby but it will contain basic information that is necessary for understanding Swift. For more information on how to get started with Ruby, check the Ruby reference:

https://www.ruby-lang.org/en/documentation/

While not necessary, it is best not to use a production project while following this document, it is best to use a blank project for testing.

Feel free to try out some of the code snippets inside of this document, they have been tested and work in either a ruby interpreter standalone or inside of Swift. Code that can run inside of Swift has been labeled with the word Swift next to it, please always exercise caution when working on existing scripts.

Installation

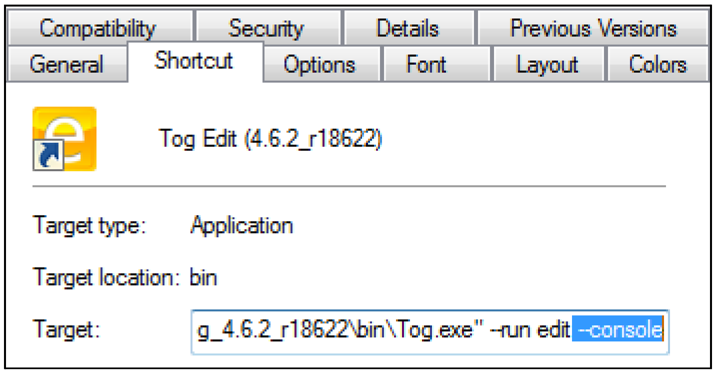
Ruby is installed alongside Swift as it's own standalone interpreter for running graphic scripts, however immediately starting in Swift is not the most convenient place to test scripts. In order to get started with Ruby scripting and testing out ideas it is a good idea to get a copy of Ruby from the ruby website and install it separately of Swift, so that it is no longer necessary to run Swift to run our scripts.

For Windows the ruby installer is available from here (note Swift uses version 1.9.3):

http://rubyinstaller.org/

When installing, it is a good idea to check the box that installs ruby executables to the environment PATH of Windows. Now, when typing "ruby" inside of command prompt, the ruby interpreter will open.

Alternatively, use a text editor which has a ruby interpreter to run the scripts, for example Sublime Text.



Tog Edit (4.6.2_r18622)

Target type: Application

Target location: bin

Target: g_4.6.2_r18622\bin\Tog.exe" --run edit --console

In order to see the output of "puts" (Ruby's print function) from Swift, it is necessary to launch the Editor with the console. Locate the Editor shortcut and add "--console" at the end of the target and hit OK.

First Script

Open up a text editor and type the following into it:

```ruby
puts "Hello World"
```

If using Sublime Text it is now possible to press CTRL+SHIFT+B to build the script and a console will open displaying "Hello World", this is how to print in Ruby and is our really simple first script.

If using a standalone installation of Ruby, then first save the ruby file to a simple location with the extension ".rb", for example: "C:\Scripts\myFirstScript.rb". Next, open up a command prompt in this directory by typing 'cmd' in the address bar. Type "ruby" followed by the name of the script, for example "ruby myFirstScript.rb".

The console window will output "Hello World".

Let's break down what we did:

```ruby
puts
```

This is a standard function inside of ruby that will write text to a console or application. Typing this tells Ruby to output the following information to the screen.

```ruby
"Hello World"
```

This information is called a String Literal (see Types and Literals for more information). It is denoted by the quotation marks either side of the words. It is important to use the quotation marks so that Ruby knows that this is a String type, otherwise it would give an error about there being no such thing as Hello or World.

Note: Do not use the puts keyword in production Swift see Debugging for information on this.

Functions

Warning: Defining a function in the User Code tab in Swift will create a "Stack Level Too Deep" error. This is due to Ruby being unable to create a nested function inside of the Swift script structure, see Lambdas below for a "workaround".

Function Basics

A function is a piece of code that is designed to be re-usable. They should be thought of as building blocks of software. A function in Ruby is written as follows:

```
def sayHello()
    puts "Hello, World"
end

sayHello()    # Result: "Hello World"
```

We start by defining our function by using the def keyword, following that we give it a name, in this case it is called "sayHello". Next we use parenthesis, these are used to pass information into the function (see below) and since we are not passing any information we keep them blank. Next, we write our function code, in this case we output "Hello, World" and then we end it to complete the block.

In order for our function to actually do something we must Call it. We do this by typing it's name, as if it were a variable. For clarity, we use the parenthesis again to show that we are calling a function and that this is not a variable.

Note: Notice how the function code is indented, this makes it clearer to see what code belongs to what building block.

Function Parameters

A function can be useful when it is necessary to perform an action many times. In the following example, we make use of the parameter "name" inside of the parenthesis, to pass information into the function :

```
def printName(name)
    puts "My name is " + name.to_s
end

printName("Steve")    # Result: "My name is Steve"
printName("Chris")    # Result: "My name is Chris"
printName(2)          # Result: "My name is 2"
printName()           # Result: Error.
```

In this example we have created a function called printName and a variable named "name" and it is being printed inside of the function. The variable "name" has no value yet, it only gets created when the function gets called. We can give the variable "name" a value by supplying one in the function call, for example "Steve".

Functions can have multiple parameters separated by commas and are dependant on the order in which they were written, for example:

```
def printName(age, name)
    x = age - 18
    if x >= 0
        puts "I am over 18 years old."
    end
    puts "My name is " + name.to_s + "."
end

printName(33, "Steve")
# Result: "I am over 18 years old. My name is Steve."

printName("Chris", 16)
# Result: Error, Chris is not a number.
```

For some more information about defining functions see the Definitions and Declarations section.

Lambdas and Procedures

Lambdas and Procedures are an advanced topic and this section should be revisited after completing the rest of the guide and having written a few scripts.

Lambdas are anonymous functions, which means that they have no name given to them. They are denoted by using the lambda keyword, using curly brackets (first bracket must be on the same line) and then inserting code inside of the brackets, the following is an example of a lambda:

```
lambda {                                                  tOG
    puts "FooBar"
}
```

Using this by itself won't produce any result though, there must be a way to call it. Lambda's should explicitly be given some memory, such as a variable.

```
myFunction = lambda {                                     tOG
    puts "FooBar"
}
myFunction.call()    # Result: "FooBar"
```

A lambda attaches itself as a method to a variable object, see Variables and Methods for more information about these concepts.

However, it is probably best to use a Procedure (Proc) instead of a Lambda inside of Swift, since if no parameters get passed to a Procedure then it will default the type to Nil rather than fire an error. Procedures use similar syntax to Lambdas and are written as such:

```
myProcedure = Proc.new {
    puts "I am a Proc!"
}
myProcedure.call()    # Result: "I am a Proc!"
```

A Procedure (Proc) is a class and we are creating a new Object out of it by using the new keyword. In order to pass Swift parameters to Lambdas and Procedures, a new syntax has to be used:

```
param = Proc.new {
    |a, b, c|
    puts "Hello " + a.to_s + b.to_s + c.to_s
}

param.call("A", "c", "e")    # Result: "Hello Ace"
param.call()                 # Result: "Hello "
```

The |a, b, c| are “yielded” variables to the procedure object. Similarly to functions, “A”, “c” and “e” all get fed into a, b and c in the same order. The result is such that it is possible to create a function with optional values for intended results. The variables a, b and c do not exist outside of the param Object.

Variables

Standard Variables

Ruby's standard variables are basic names that have no special keywords or definitions. They can be defined simply such as:

```
myVariable = 1.0
```

Ruby has a few special characters that it uses to denote what type of variable it is though. One useful one is the global variable, which is denoted using the $ symbol:

```
$myGlobalVariable = 5.0
```

Global variables can be dangerous if another variable shares it's name. It is encouraged to check out the Ruby reference and to experiment to see exactly how the variables change depending on scope and location.

Another variable type is called Constant (Const), which is intended to be a hard-set value that will never change. Unlike in other languages, Ruby can and will re-assign constants but may give a warning, for the sake of predictable code it is best to avoid re-assignments. A const can be defined by using a capital letter at the beginning of the variable name. For example:

```
MyConst = 5.0           # Defined a Constant
CONSTANTVALUE = 10.0
```

Swift Variables

Swift has uses some of these special character symbols to reference variables too. They will be explained below:

Node Variables

To reference a node in the scene you can use the @ symbol:

```
myReference = @n1_NODE
```

This will make the variable “myReference” reference the @n1_NODE in the SceneGraph. If a value of myReference is changed then @n1_NODE will update as well. As stated in the documentation, the advantage of this is the ability to change the myReference variable based upon a condition. Example:

```
if(redColour == 1.0)
      myNode = @n1_NODE
else
      myNode = @second_NODE
end
```

If you need to set some basic parameters then it will be easier to set it directly, without storing it as a variable:

```
@n1_NODE.setDisplay(true)
```

The @ symbol is actually a special symbol inside of Ruby, it denotes a class instance variable. Information pertaining to this is is out of the scope of our guide.

Input Variables

The underscore “_” character makes it possible to access an input, if it is declared in the same method. For example, if the input is called “textInput” then the value can be accessed and changed inside of user code:

```
_textInput = "Hello, Swift!"
```

Refer to the Structure section to see how User Code interacts with Inputs.

Changing Data Using Inputs and User Code

Rather than use user code to directly affect items in the

SceneGraph such as @n1_NODE.setDisplay(true), it's far better to assign to an input:

```
_myGraphicDisplay = true
```

The benefit to this is that it won't break if the node in the SceneGraph gets renamed. Swift handles referencing the correct node and method via properties. The inputs can also be hidden from MOS and other data inputs so that it doesn't clutter up a control desk.

Definitions and Declarations

Unlike in "statically typed" programming languages, especially C based languages, there is no concept of Definitions and Declarations inside of Ruby and nor is there any "hoisting" like there is in Javascript. In order for a variable to be used it must be assigned to first, for example:

```
myVar = "Hello"
puts myVar        # Result: Prints "Hello"

puts myVar2       # Result: Error, no variable named
myVar2 = "World"  #         myVar2 exists yet.

## Functions
def foo()
    return "foo"
end

puts foo()        # Result: Prints "foo"

################################################

puts bar()        # Result: Error, no function
                  #         called bar exists yet.
def bar()
    return "bar"
end
```

In ruby the rule is simple: If a variable is being declared then it must be given a value, even if it is Nil, in order for it to be used.

Types

Standard Types

Swift uses Ruby's out-of-the-box types, the most common are:

Int	|	A whole number. E.g. 5
Float	|	A decimal number. E.g. 1.245, 2.0
String	|	A string of text. E.g. "Hello"

It is possible to get a variable's type by using the class call on it. Here is an example Getting a variable's type:

```
myVar = 1.0
puts myVar.class     # Result: Float

# Or simply:
puts "Hello".class   # Result: String

# Getting a true or false value:
myVar2 = 6.kind_of? Integer
puts myVar2     # Result: True
```

To convert between types use the methods:

```
myVal.to_s        # myVal as a String
myVal.to_i        # myVal as an Integer
myVal.to_f        # myVal as a Float
```

Literals

A literal is a value that has no assigned memory (it has no variable name), it is the most basic data type that can exist since it is raw data. A literal was used above in Standard Types and even in our very first script! Here is an example:

```
puts "I am a Literal String"
```

```ruby
var = "I am not a literal anymore, since I belong to var"
puts var
```

Nil

A special "type" that you might run across is the type "nil". It is not specifically a type, but it contains no information whatsoever and just points to empty memory. You can test to see whether an object or variable is nil by using the nil? Method:

```ruby
myVar = nil
myVar.nil? ? puts "Foo" : puts "Bar"
# The above prints Foo.
```

A nil check can be important to test whether a value exists correctly (depending on the method of input). If doing work with user input values it is important to validate the data exists otherwise Ruby will throw an error and the script will not run correctly. An example of validation can be seen below:

```ruby
if (!userString.nil? && !userString.empty?)
      # Valid Input
else
      # Invalid Input
end
```

Note: The ! character is the "not" symbol/operator. The && is the "and" comparison operator. A full list of operators can be found in the ruby documentation. A list of the common operators can be found in the glossary at the end of this document.

Arrays

Arrays in Ruby are lists of data. They can be created like so:

```
myArray = ["a", "b", "c"]
```

Swift's nodes have some methods that work with arrays but they can be unreliable. It is best to get the values you need from a Swift "get" method and return it into a variable.

Arrays can be accessed using the square brackets ('[', ']') as seen above. For example, getting the Y coordinate from a position could look like this:

```
yPos = Swift.Position[1]
```

Which will return the second value (since 0 is the first), in this case, Y. Note that this code is an example and does not work in Swift.

Custom Types (aka Classes) and Methods

Note: Classes are an advanced topic, it may be worth revisiting this topic at a later date.

Ruby is an Object Oriented Programming (OOP) language and for that reason it has a concept of Classes and Objects. An object is an instance of a class and a class can be considered a template for many child objects. It is worth noting that this concept is not necessary for scripting with Swift but being aware of it while reading the .rb script files will help understand the structure.

For example, consider a person. A person is a "class" of object, it has a name, age and many other different types of data. At the same time, there are many people, with many different names ages and more. In order to manage this data we create a "data type" which manages the data. Below is an example of the class and object relationship, it is possible to see that Person is the class and john is an Object of this class. John is also known as being a type of Person.

```
class Person
    def initialize(name, age, gender)
        @person_name = name
        @person_age = age
        @person_gender = gender
    end

    def age()
        puts "I am " + @person_age.to_s + "."
    end
end

john = Person.new("John", 43, "Male")

john.age()    # Result: "I am 43."
```

Classes make use of methods. These are similar to functions and in fact they share the exact same syntax in Ruby. In the above example, initialize and age are both methods of the Person class. "initialize" is a special keyword method that Ruby uses to create the new object, in our example: The object variable "john" is created by initializing the Person with the name John, age of 43 and gender of Male.

Methods are called using the dot notation, for example "john.age()" will call the age method from the john Object. This dot notation has been used in previous areas of this guide such as Types, using the .to_s method. Refer to the Finding Methods section for information pertaining to discovering methods.

Structure

Methods

There are some already defined methods in a Swift graphic when it is first run, they are:

1. Initialize
2. Construct
3. Main

Initialize

The initialize(RootNode) method is the constructor for the script. It gets called when the script is loaded. Loading and running a script are two separate functions within Swift. A script may be loaded well before the script is run. It is used to register the script with Swift so it knows about it. For example, in the Menu interface, initialize get called when the script is stacked into the stack list. The rootnode parameter is the root node at the top of the SceneGraph. The Super() call is a constructor for Swift to initialize the graphic as a new object. The self.setIsLibrary(false) method tells Swift that this graphic is not a library.

Construct

The construct method is the method that constructs the Scenegraph. It gets called when the script is run. As we progress the user will see that this contains commands to create and initialise the nodes within the scenegraph. The parameter passed into this method is data . Data is a parameter that is used to pass information between Swift and Ruby. It contains name value pairs obtained from network, database and Gui's. The user will see examples of this later. As shown above this method only contains the self.swap() command. This is an internal method again contained within GMScript and is used for the object->object transfer described in . This always appears at the top of Construct.

Main Method

The Main method is, as it sounds, the main method for this script. It gets called immediately after the Construct method when the script is run. Again, Main gets passed the data parameter. As the user will see as this section progresses, it contains input definitions, blocks and animations that get run by default. However, before this there is the call to self.transfer() . This method does the actual object->object transfer described in section . Thus, any animate off/on or between methods will get called during this transfer.

After this method, there can be many more user-added methods created either through code or via the Swift editor by clicking the “New Method” button above the timeline.

User Code

To see the user code block inside of the .RB script graphic, open up the User Code tab inside of Swift and begin to type some code. A comment will also generate it, such as:

```
# Hello World
```

Now we can see where the user code exists inside of Swift.

User code in Swift exists inside of Methods. It’s possible to write user code in every type of method but the most useful ones will be user created methods. Due to the scope of the ruby script, it will not be possible to access data from other method blocks inside of user code, without editing the file directly.

Changing Data Using User Code and Inputs

The structure of the ruby script plays a vital part to the inputs and scenegraph systems. In order to get an idea for what is going on here, try:

1. Create an input on a node in the SceneGraph and target the display.
2. Set the input to 0 (or false) and play the scene to confirm that the node disappears.
3. In the usercode tab, type the input name and set it to 1 (or true).
4. Replay the scene and the node should re-appear. Therefore it is possible to change the values of user inputs after they have been set.
5. Change the user code to target the node in the SceneGraph rather than the input.
6. The input will override the user code and hide it again.

This is because of the structure of the ruby script file, it will pass data in at different times. The order goes like this:

1. Create SceneGraph
2. Define Inputs
3. Run User Code (This is where we changed the input in step 3)
4. Update Inputs
5. Update SceneGraph

Updating the inputs/updating the SceneGraph get run after the User Code and therefore the input will re-assign into the SceneGraph overriding the User Code value supplied just before it.

For the most control when changing values in methods, add a step animator to the timeline and target it via an input. The input can still be manipulated via a user, then the user code and then finally it will be able to be set manually on the timeline for the best timing control.

Debugging

Debugging User Code in Swift is fairly rudimentary. A very common method of debugging any Ruby script, is to output values to read using the “puts” command. When trying this in Swift without a console window, it will crash the editor. In order to view the output in the console, the Swift Edit Shortcut Target must be edited by and adding:

```
--console
```

This will keep the console window open after the initial load. Miscellaneous errors from Swift and script values printed with “puts” will appear within it. So now it is possible to do:

```
myValue = 5.0
puts myValue
```

tOG

WARNING: Remember to remove the debug “puts” code before passing the file along or using it live.

Logging messages safely:

Using self.logMessage(“Hello”) will allow you to log messages to the console and to the Swift generated logs without any issues. So long as a SwiftLogs folder exists in C:\

Finding Methods

If unsure what is possible with certain nodes, it is possible to run a method called “methods” on objects. This list will usually be too big to fit inside of the console so it's usually best to write it out to a file using the File.write command:

```ruby
File.write
(
"C:/users/[namehere]/desktop/[node]_methods.txt",
@n1_node.methods()
)

# or print a smaller range of options:

puts @n1_node.methods[0..30]
```

Replace the information where necessary. Windows will be fine with / in path names so there's no need to escape backslashes \\.

This will print a text file full of information about the node’s methods that look like this:

```
:addKeyframe, :setKeyframes, :setScaleKeyframes,
:setOrientationKeyframes, :clearKeyframes, :setFollow, :getFollow,
:setSplinePosition, :getSplinePosition, :getSpline,
:setConstantVelocity, :getConstantVelocity, :pKeyframes
```

Using the information above it is now possible to use additional methods like

```ruby
@n1_node.getFollow()
```

This will now be accessible in user code.

Glossary

Expression Operators

Add	+
Subtract	-
Divide	/
Multiply	*
Modulus (Gives the remainder)	%
Power	**

Assignment Operators

Assignment	=
Expression	(Expression Operator)=

The assignment operator can be modified by typing an expression operator before it. For example:

```
myVar = 1
myVar += 1
puts myVar # Prints 2
```

Logical Operators

And	&& or ‘and’
Or	|| or ‘or’
Not	! or ‘not’

Conditional Operators

Is Equal To	==
Is Greater Than	>
Is Equal To or Greater Than	>=
Is Less Than	<
Is Equal To or Less Than	<=
Not Equal To	!=

Range Operators

Start to End Point Inclusive	..
Start to End Point Exclusive	...

The difference between these can be explained by programming languages 0-index based nature. See a topic about Arrays for more information. An example of the differences here are:

```
puts 0..10      # Ouputs 0 to 10
puts 0...10     # Outputs 0 to 9.
```

Resources

http://ruby-doc.org/

Swift Documentation page 218