SOAPScript

"For squeaky-clean code, use SOAPScript"

Language Summary

Bob Nisco

Version 0.5 β

This page intentionally left blank?

It's a paradox more so than a question.

1. Introduction

SOAPScript, or more formally, Simple Objective And Procedural Script, is exactly what it sounds like: a modern strongly-typed language best used as in an objective and procedural environment. It also has a tagline to accompany it, "For squeaky-clean code, use SOAPScript." When writing code you should be repeating the secondary mantra to yourself, "wash rinse, don't repeat yourself."

The main theory behind SOAPScript is that code is read more often than it is written; therefore, the language should foster the creation of the most beautiful and readable code possible. SOAPScript is also focused on programmer happiness leading to increased productivity. Therefore, readability and writability are the most important criteria of SOAPScript. When beautiful, elegant code is written, it is a positive side effect that maintainability is also improved. In order to stay within the "squeaky-clean" ideology that underlies SOAPScript, there is a certain style guideline that will be outlined throughout this language summary document.

SOAPScript is mostly based on Python with the intention of making it strongly typed by adding some flavors of Java, but differing in the following ways:

1. SOAPScript is strongly typed. Variables must be one of the included data types (see section 3 for more information).
2. SOAPScript is compiled.
3. SOAPScript explicitly states the scope of classes, methods, and attributes.
4. SOAPScript is statically scoped and ergo utilizes early binding.

1.1 Hello World

```
public void HelloWorld():
    System.print("Hello World")
```

1.2 Program Structure

The key organizational concepts in SOAPScript are as followed:

1. Every class must be inside a module (the SOAPScript equivalent to Java's packages or C++'s namespaces). Modules can have multiple modules underneath it in a hierarchy, which is delimited by a period (eg: GrandparentModule.ParentModule.ChildModule).
2. Execution of instructions is sequential.
3. Blocks of code are bounded by meaningful whitespace and colons, much like Python. The user has the choice to use spaces or tabs to delimit whitespace, but it must be the same within a single file. The preferred SOAPScript style is to utilize hard tabs.
4. Getters and Setters for all class attributes are created on the fly with the prefixes get and set followed by the attributes capitalized name (much like Objective-C's synthesize property).
5. Assignment is delimited by the equal "=" sign.
6. Test for equality is delimited by two equals signs ("==") and the test for object identity is delimited by three equals signs ("==="). On the converse, the test for inequality is delimited

by an exclamation mark and an equal sign ("!=") and the test for object identity inequality is delimited by an exclamation mark and two equal signs ("!==").

The following program example will serve as an introduction to SOAPScript programs.

```
module Marist:
      public class Student:
            private String firstName, lastName, major

            # Default constructor
            public Student():
                  self.setFirstName("")
                  self.setLastName("")
                  self.setMajor("")

            # Overloaded constructor
            public Student(String fn, String ln, String mjr):
                  self.setFirstName(fn)
                  self.setLastName(ln)
                  self.setMajor(mjr)

            public Boolean hasTechnologyMajor():
                  if (self.getMajor() == "Computer Science" ||
                        self.getMajor() == "Information Systems" ||
                        self.getMajor() == "Information Technology"):
                        return True
                  else:
                        return False
```

This example declares a class called Student, under the module Marist. The fully qualified name of this class is Marist.Student. The class contains three privately scoped attributes, firstName, lastName, and major. It contains a default, no-arguments constructor and an overloaded constructor which takes arguments. The accessor and modifier methods ("getters and setters") are created on the fly, with no programmer input necessary. This class also has a single method, hasTechnologyMajor that will return a Boolean if the instance of the Student class has a major that is equal to Computer Science, Information Systems, or Information Technology.

1.3 Types and Variables

In sticking with the simplicity of SOAPScript, everything is treated as an object. For example, the programmer will never have to deal with worrying about a class Integer differing from a primitive int. Not only will all types be children of Object, but all programmer-defined classes will be subclassed from the parent class Object as well (this distinction does not need to be explicitly stated). All types and variables will store a reference to their data. Each type will have methods defined to cast to different types but will never do so unless explicitly stated (e.g.: an instance of Decimal can be cast to an Integer, but not without the programmer explicitly stating a cast, and understanding that there may be a loss of precision).

1.4 Statements Differing from Python and Java

Statement	Example

Constructor	`public void MyClass():`
	`  self.setAttribute1("")`
	`  self.setAttribute2(42)`
	
	`public void MyClass(String a, Integer b):`
	`  self.setAttribute1(a)`
	`  self.setAttribute2(b)`

If/Else-If/Else Block	`if (x < 0):`
	`  System.print("Negative Number")`
	`else if (x == 0):`
	`  System.print("Zero")`
	`else:`
	`  System.print("Positive Number")`

Casting	`private Decimal pi = 3.14`
	`private Integer x = 0`
	`# Programmer understands lack of precision by`
	`# casting a decimal to integer`
	`x = pi.toDecimal()`
	`private String y = ""`
	`y = x.toString()`

Defining Arrays & Dictionaries	`private Integer[] intArray = [1, 2, 3, 4]`
	`private Integer{} intDict = {`
	`  'one' : 1,`
	`  'two' : 2,`
	`  'three' : 3,`
	`  'four' : 4,`
	`}`

2. Lexical Structure

2.1 Programs

A SOAPScript program consists of one or more source files. A source file is an ordered sequence of (probably Unicode) characters.

Conceptually speaking, a SOAPScript program is compiled using three steps:

1. Transformation, which converts a file from a particular character repertoire and encoding scheme into a sequence of Unicode characters.
2. Lexical analysis, which translates a stream of Unicode input characters into a stream of tokens.
3. Syntactic analysis, which translates the stream of tokens into executable code.

2.2 Grammars

This specification presents the syntax of the SOAPScript programming language where it differs from Python and Java.

2.2.1 Lexical grammar where different from Python and Java

The lexical grammar of SOAPScript is very similar to Python and Java mixed together. Since Python more heavily influences SOAPScript's syntax, the proceeding grammars show the differences between Python and SOAPScript.

<null value> -> Null
<access modifier> -> private
-> public
-> protected

2.2.2 Syntactic ("parse") grammar where different from Python and Java

The syntactic grammar for SOAPScript is similar to a blending of Python and java. The SOAPScript syntactic grammar is outlined below.

<module declaration> -> module <identifier>
<class declaration> -> <access modifier> class <identifier>
<method declaration> -> <access modifier> <object type> <identifier> <parameter list>
-> <access modifier> <object type> <identifier>
<parameter list> -> <parameter> <parameter list>
-> <parameter>
<parameter> -> <object type> <identifier>

2.3 Lexical Analysis

2.3.1 Comments

There are two types of comments supported by SOAPScript. **Single-line comments** start with a the character "#" and extend to the end of the source line. **Delimited comments** start with the characters "##" and end at the end of the source line that begins with "##". Delimited comments may span multiple lines.

It is not necessary, but it is the preferred SOAPScript style for each line within the delimited comment block to also start the line with a "#." Examples and usage of these comments can be seen in section 4, Examples.

2.4 Tokens

There are several kinds of tokens: identifiers, keywords, literals, operators, and punctuators. White space and comments are not tokens, though they act as separators for tokens.

Tokens:

- identifier
- keyword
- integer-literal
- real-literal
- character-literal
- string-literal
- operator-or-punctuator

2.4.1 Keywords different from Python or Java

A keyword is an identifier-like sequence of characters that is reserved, and cannot be used as an identifier.

New keywords: Null, public, protected, private, static, function

Removed keywords: None, @staticmethod, new, is, is not, def

3. Types

All data types in SOAPScript are Object types. They are all subclassed from the parent class, Object. This hierarchy makes arrays and dictionaries of mixed types possible in this statically typed language.

3.1 Object types

Integer – a number that can be written without a fractional or decimal component.

```
private Integer gradeForThisAssignment = 100
```

Decimal – a general purpose, floating-point number that uses double precision as defined by IEEE 754.

```
private Decimal aSliceOfPi = 3.14159265
```

Char – a Unicode character.

```
private Char firstLetterOfAlphabet = "A"
```

String – a set of Chars.

```
private String fruitFlies = "Like an apple."
```

Array – a systematic arrangement of data.

```
private String[ ] worthwhileMajors = ["Computer Science", "Information Systems", "Information Technology"]

private Object[ ] mixedArray = [42, "Rush", 3.14]
```

Dictionary – an associative array where the keys are of String type.

```
private Integer[ ] myRatingOfLanguages = {
        'COBOL' : -4,
        'Fortran' : 2,
        'Pascal' : 7,
        'Scala' : 9,
        'LISP' : 4,
        'ML' : 3,
        'Erlang' : 7,
}

private Object[ ] mixedDictionary = {
        'name' : "Bob Nisco",
        'major' : "Computer Science",
        'graduationYear' : 2014,
}
```

4. Examples

4.1 Caesar Cipher

```
module Caesar:
      public class Cipher:
            # Make a class with one attribute, string
            private String str

            # Default constructor
            public Cipher():
                  self.setStr("")

            # Overloaded constructor which takes in a string
            public Cipher(String s):
                  self.setStr(s)

            ##
            # An encrypt class method that will set the str variable of
            # this class to the encrypted version of the string based
            # on the Integer shftAmt passed to it
            ##
            public void encrypt(Integer shftAmt):
                  private String result = ""
                  ##
                  # SOAPScript knows that a string is just an array of
                  # Chars under the hood, so no need to use any fancy
                  # notation for iterating over the string,
                  # char by char.
                  ##
                  for (Char c in self.str):
                        result += self.shiftChar(c, shftAmt)
                  self.setStr(result)

            ##
            # An overloaded method that not only accepts different
            # parameters, but is static so that it can be used in other
            # programs that import the Caesar module, but don't
            # necessarily want to instantiate an instance of
            # the Cipher class.
            ##
            public static String encrypt(String str, Integer shftAmt):
                  private String result = ""
                  for (Char c in str):
                        result += self.shiftChar(c, shftAmt)
```

```
        return result

    public void decrypt(Integer shftAmt):
        self.setStr(self.encrypt(shftAmt * -1))

    public static String decrypt(String str, Integer shftAmt):
        return self.encrypt(str, shftAmt * -1)

    private Char shiftChar(Char c, Integer shftAmt):
        ##
        # - The first toASCII() call is actually calling the
        #   Char.toASCII() function.
        # - The toASCII() call at the end is technically
        #   calling the Integer.toASCII() casting method.
        #   SOAPScript is smart enough to infer that we were
        #   dealing with integers previously, due to the
        #   toInteger() call and that no other operation was
        #   happening with a variable with a data type other
        #   than Integer.
        ##
        return (((c.toUpper().toASCII() - 65 + shftAmt) % 26)
             + 65).toASCII()
```

This Caesar cipher example shows off the power of the type system within SOAPScript. Since all data types are handled as objects, the iteration over a String does not take any extra boilerplate code, nor does the conversion from a Char to an ASCII value. The flexibility of having all data types as objects allows for some really powerful, built-in features leading to increased programmer happiness and overall clean code.

4.2 Generic Linked List

```
module LinkedList:
      ##
      # A simple LinkedListNode class to show how
      # generics works within SOAPScript
      ##
      public class LinkedListNode<E>:
            private E data
            private LinkedListNode<E> next

            # Default constructor
            public LinkedListNode<E>():
                  self.setData(Null)
                  self.setData(Null)

            # Overloaded constructor
            public LinkedListNode<E>(E data):
                  self.setData(data)
                  self.setNext(Null)

            # Overloaded constructor
            public LinkedListNode<E>(E data, LinkedListNode<E> next):
                  self.setData(data)
                  self.setNext(next)

            public void insertAfter(LinkedListNode<E> next):
                  LinkedListNode<E> oldNext = self.getNext()
                  next.setNext(oldNext)
                  self.setNext(next)

      public class LinkedList<E>:
            private LinkedListNode<E> head

            public LinkedList<E>():
                  self.setHead(LinkedListNode<E>())

            public LinkedList<E>(LinkedListNode<E> first):
                  self.setHead(first)

            public LinkedListNode<E> getLast():
                  LinkedListNode<E> current = self.getHead()
                  while (current.getNext() != Null):
                        current = current.getNext()
                  return current
```

```
public void insertAtEnd(E data):
     self.getLast().setNext(LinkedListNode<E>(data))

public void insertAtEnd(LinkedListNode<E> next):
     self.getLast().setNext(next)

public static void main(String[] args):
     public LinkedList<String> mySemester =
          LinkedList<String>()

     mySemester.insertAtEnd("Theory of Programming
          Languages")
     mySemester.insertAtEnd("Database Management Systems")
     mySemester.insertAtEnd("Software Development")
     mySemester.insertAtEnd("Computer Organization &
          Architecture")
     mySemester.insertAtEnd("Systems Analysis & Design")

     for (String s in mySemester):
          System.print(s + "\n")
```

The generic linked list example shows just how cleanly SOAPScript handles generics. It is nearly effortless to define your own generic class, and it is still fully readable, statically typed, and does not rely on duck typing for generic classes like Python does.

4.3 Fibonacci

```
module Math:
      public class Fibonacci:
            public static Integer fib(Integer i):
                  if (i == 0):
                        return 0
                  else if (i == 1):
                        return 1
                  return fib(i - 1) + fib(n - 2)
```

A simple Fibonacci example that shows how SOAPScript handles recursive functions. This example outlines the ease of recursion by defining 2 base cases and the recursive call.

4.4 Convert Array to Dictionary

```
module Example:
      public class ArrToDict:
            private Integer[ ] arr = [1, 2, 3, 4]

            public ArrToDict():
                  public Integer{} dict = {}
                  for (Integer i in arr):
                        dict[i.toString()] = i
                  return dict
```

A relatively simple example of showing off the powers of converting an array to a dictionary.