C# aide memoire for SYSPRO VBScripters

Version 3.0

SYSPRO
Say Yes to Next

Contents

Introduction

These notes have been compiled as an aide memoire for SYSPRO VBScripters migrating from using VBScript as the SYSPRO scripting language to using C#.

The aim is to capture commonly used VBScript techniques that are used in SYSPRO VBScripting and show the equivalent syntax in C#.

These notes are aimed at developers who are competent in VBScript and have a working knowledge of C#. They are not intended as a C# basic training document.

Some simple gotchas

1. Case sensitivity

VBScript is not case sensitive.

C# is case sensitive.

In C# a variable name of XMLDoc is not the same as XMLdoc and a compile error will be thrown.

When the compiler indicates that a variable doesn't exist, check the spelling of the variable first.

2. Assignment operator and the equals condition operator

In VBScript the assignment operator and the equals condition operator are both a single equal sign =.

But in C# the assignment operator is a single equal sign = and the equals condition operator is a double equal sign ==.

This type of VBScript code If(myname = "andy") will throw a C# compile error "Cannot implicitly convert type 'string' to 'bool'".

In C# it should be If(myname == "andy")

C# syntax examples

1. Variables

In VBScript a variable has only one fundamental data type and that is variant.

In C# it is necessary to specify the type of the variable and ideally assign a value to it when it is declared.

For example:

string myName = “andy”;

or

int lineCount = 0;

or

var mySundayName = “andrew”;

The “=” sign means that a value is being assigned to that variable.

Note that in C# “var” does not indicate a variant type but instructs C# to infer the type when the variable is initialized. In this case it would infer it is a string.

2. Constant variables

VBScript uses the “const” key word to define a variable with a fixed unchangeable value.

C# also uses “const” to define an immutable variable, but as with the variable declaration it needs a type defining.

For example:

const string Tbar_PressMe = “01001”;

3. Newline characters

In VBScript the newline character was typically represented as “\n” or “\r\n”.

In C# use Environment.NewLine

For example:

string msg = “Hello Andy” + Environment.NewLine + “Hello Phil”;

4. If statement

The “if” statement is very similar to VBScript with
if, else, else if, style syntax.

And it has the following conditions it can check

- Less than: a < b
- Less than or equal to: a <= b
- Greater than: a > b
- Greater than or equal to: a >= b
- Equal to a == b
- Not Equal to: a != b

For example:

```
int time = 20;
if (time < 18)
{
  Console.WriteLine("Good day.");
}
else
{
  Console.WriteLine("Good evening.");
}
```

Watch out for the == sign.
`If(time = 10)` will throw an error “Cannot implicitly convert type ‘int’ to ‘bool’”
`If(time == 10)` is checking if the variable `time` has a value of 10.

5. If statement – ‘or’ / ‘and’ logical operator

If statement – “or” logical operator

The “or” logical operator is a double pipe sign ||.

Either side of the operator must be conditions.

e.g.

int time = 23;

if (time < 6 || time > 22)

{

  Console.WriteLine("It must be nighttime");

}

If statement – “and” logical operator

The “and” logical operator is a double ampersand &&;

Either side of the operator must be conditions.

e.g.

int time = 20;

if (time > 6 && time < 22)

{

Console.WriteLine("It must be daytime");

}

6. VBScript “select case” statement

The equivalent C# command to the VBScript “select/case” is the “Switch/case” statement.

But note the colon at the end each case and the break; at the end of the code block.

For example:

```
if (EventType == ToolbarEvent)
{
    switch (EventID)
    {
       case Tbar_Help:
            //code block

            break;

       case Tbar_Stockcode:
           // code block

            break;
       default:
            SysproMessageBox("Unknown EventID: " + EventID);
            break;
     }
  }
```

7. Strings

Trimming white space from strings

Sometimes, data returned from the SYSPRO business objects can have leading or trailing spaces which need to be removed.

```
string myTextWithSpaces = "    andy   ";
```

To remove leading and trailing white space

```
myTextWithSpaces.Trim()
```

To remove leading white space

```
myTextWithSpaces.TrimStart()
```

To remove trailing white space

```
myTextWithSpaces.TrimEnd()
```

To remove a specified number of characters from a specified position

```
myTextWithSpaces.Remove(int startindex[,endindex])
```

Make sure that any string that is populated with data from a SYSPRO callout function is trimmed.

e.g.

```
dataretrievalmethod     =     CallSYSPROFunction(ToolbarGetValue,     "SAMDTZTB",
TBar_DataRetrievalMethod);

if (dataretrievalmethod.Trim() == "S")
{
    //do something
}
```

Or to trim a string from an App Designer toolbar.

```
logfiledirectory      =      CallSYSPROFunction(ToolbarGetValue,      "SAMDTZTB",
"01004").Trim();
```

Padding strings with leading characters

It is sometimes necessary to pad a string with leading characters. For example, pad a customer number with leading zeroes to pass to a SQL query.

Example of padding with zeroes and returning a 15 character long string.

```
string longCustomerNumber = CustomerNumber.PadLeft(15,char.Parse("0"));
```

Handling large strings

In VBScript a string can be built up with this type of syntax:

```
dim mystring
mystring = "hello andy"
mystring = mystring & " it is a nice day"
```

In C# this would be:

```
string mystring = "hello andy";
mystring += " it is a nice day";
```

However, in C# a string is immutable. This means that the variable cannot be changed. What the code above does is create a copy of the variable, appends the original text to the new text and deletes the original variable.

If large strings are being created, maybe by looping over a record set or some such array, then this method can be very slow.
When creating large strings the StringBuilder class should be used.

Using statement required:

```
using System.Text;
```

Example code:

```
StringBuilder sb = new StringBuilder();
sb.AppendLine("Hello Andy");
sb.AppendLine(" it is a nice day");

MessageBox.Show(sb.ToString());
```

There are a series of StringBuilder methods to manipulate text including
Append, append formatted, insert, replace, delete, etc

Converting strings to other types (int, decimal, date)

When using Business Objects and data sources in SYSPRO the data is usually returned as a string. This could be an Xml string, or a return value from App Designer.

As an example:
To get the number of rows populating a list view in App Designer the following code might be used.

```
ReturnValue = CallSYSPROFunction(ListviewGetRowCount, "SAMPAZL1", " ");
```

This will return the number of rows in the listview, but the value will be a string. E.g "10";

If you need to do arithmetic on the value, then it needs to be converted to an Integer.
e.g.

```
var NoLinesInListview = Int32.Parse(ReturnValue);
```

or you can use TryParse() to check if the conversion has worked.

```
int NoLinesInListview = 0;
bool StringToIntOk = Int32.TryParse(ReturnValue, out NoLinesInListview);
```

This technique can be used for decimal using `Decimal.Parse()`

Dates can be particularly tricky. But converting a string to a `DateTime` object means that date manipulation is very easy,

The `DateTime` class has two methods
`DateTime.Parse()` which is quite forgiving in the syntaxes it can handle and `DateTime.ParseExact()` where it is possible to define the exact format of the string that needs converting. This is particularly useful when handling data coming from web services.

In SYSPRO the date format returned in business objects is typically YYYY/MM/DD and `DateTime.Parse()` can recognise this format.

For web services the date format is often the 24 hour clock. E.g 2024-05-08 14:40:52,531
To convert this to a `DateTime` object.

```
//The day and time of the forecast
//is in the format 2024-12-09T12:00:00
//we need the day and time of the forecast for the chart
DateTime dt = DateTime.ParseExact(m_timefrom, "yyyy-MM-ddTHH:mm:ss",null);

//now need to get at short day name
m_dayofweek = dt.ToString("ddd");
m_timehhmm = dt.ToString("hh:mm");
```

Converting back to strings
Each of these objects has a `ToString()` method which will output the data in a string. Most of the `ToString()` methods also have a format capability.
e.g This example takes a decimal number and outputs it to 2 decimal places.

```
var probablyDecimalString = "0.4351242134";
decimal value;
if (Decimal.TryParse(probablyDecimalString , out value))
    Console.WriteLine ( value.ToString("0.##") );
else
    Console.WriteLine ("not a Decimal");
```

And to get at parts of the date (see above)

```
m_dayofweek = dt.ToString("ddd");
m_timehhmm = dt.ToString("hh:mm");
```

8. Dialog/message boxes

VBScript uses `MsgBox` to display a dialog box.
e.g. `MsgBox("hello andy")`

To simplify references and using statements SYSPRO have introduced some SYSPRO methods to help in this area. See the help page for more information.

9. Exiting methods

In VBScript to exit a function the `exit function` key words were used.

In C# `return` has the same effect.

10. Returning values from a C# method

In VBScript to return a value from a function, the function name is set to the value to be returned and the function exited.

e.g. VBScript

```
Function GetMyName()
      GetMyName = "Andy"
End Function
```

In C# a return value must be of the same type as the method.

e.g. C#

```
private string GetMyName()
{
    //return a string
      return "Andy";
}
```

11. File handling

Writing to text files

In VBScript code like this is used to write to a file.

```
Set fso = CreateObject("Scripting.FileSystemObject")

set fout = fso.CreateTextFile(OutputFile, true)
fout.WriteLine(xml)

fout.close
set fso = nothing
```

In C# to create a file and then append more lines to it:

Using statement required:

```
using System.IO;


var filename = @"c:/temp/andylogtest.txt";
var msg = @"This is some text to write to a file";

//creates file and writes string to the file
using (StreamWriter sw = new StreamWriter(filename))
{
    sw.WriteLine(msg);
}


msg = @"This is another line of text" + Environment.NewLine + "and more";

//appends text to existing file
using (StreamWriter sw = new StreamWriter(filename, append: true))
{
    sw.WriteLine(msg);
}
```

Does file or directory exist

Using statement required:

```
using System.IO;


if(!Directory.Exists("C:\temp")
{
   //doesn't exist - create it
   Directory.CreateDirectory("C:\temp");
}


If(!File.Exists("c:\temp\andylogtest.txt")
{
   //doesn't exist - do something
}
```

NB. Note the exclamation mark. This means "does not".

Filename parts

To get at the component parts of a local file URL use the Path class.

Using statement required:

```
using System.IO;
```

Example:

```
string myFileName = @"C:\temp\andylogtest.txt";

string msg = "";

string filename = Path.GetFileName(myFileName);
string filenamenoext = Path.GetFileNameWithoutExtension(myFileName);
string ext = Path.GetExtension(myFileName);
string directory = Path.GetDirectoryName(myFileName);

msg += "Filename: " + filename + Environment.NewLine;
msg += "Filename with out extension: " + filenamenoext + Environment.NewLine;
msg += "Extension: " + ext + Environment.NewLine;
msg += "Directory:" + directory + Environment.NewLine;

MessageBox.Show(msg);
```

12. Arrays, Dictionaries, Lists

Arrays

Arrays are used to store several pieces of data in a single variable.
An Array must be of a specific type.
e.g.

```
string[] names;
```

Arrays are typically populated when they are initialised.
e.g.

```
string[] names = {"Andy", "Phil", "Benita", "Cameron"};
```

(N.B. The brackets are squiggly brackets.)

Please note, if the array is declared without initialising it with data, then when it is populated with data the 'new' keyword must be used to initialise it.

e.g.

```
string[] names;

names = new string[] {"Andy", "Phil", "Benita", "Cameron"};
```

On several of the App Designer callout functions the response is a tab delimited string of data.
This string needs to be broken into its constituent parts so that each part can be accessed.
The easiest way to do this is to split the string and populate a string array.
For example:

```
string CustomerNumber = CallSYSPROFunction(GetVariable, "customer", " ");

//build task dialog with OK and Cancel buttons
string returnValue = CallSYSPROFunction(TaskDialog, " ", "<Msg><Title>Confirm
Deletion</Title><Text>Ok to delete customer '"
+ CustomerNumber + "'</Text><Standard OK='true' Cancel='true' /></Msg>");

//create new string array. Split the TaskDialog response on a tab character
//and populate the array
string[] TaskDialogArray = returnValue.Split("\t");

//write some debug info to the tracer
CallSYSPROFunction(WriteToTracer, " ", "TaskDialogArray[0] value : '" +
TaskDialogArray[0] + "'");

//act on the response
if (TaskDialogArray[0] == "001")
{
      SysproMessageBox("OK pressed");
}
else
{
      SysproMessageBox("something else pressed");
}
```

Dictionaries

In VBScript dictionaries (<key, value> pair lists) are created like this:

```
Dim m_titles_dic
set m_titles_dic = CreateObject("Scripting.Dictionary")
```

and entries are added like this:

```
call m_titles_dic.add(1, "andy")
```

In C#

Using statement required:

```
using System.Collection.Generic
```

The `System.Collection.Generic` class supports `Dictionary<TKey,TValue>`, and `List<T>` types.

To create a dictionary

```
Dictionary datadic = new Dictionary<int, string>();
```

To add an entry

```
datadic.Add(1,”andy”);
datadic.Add(2,”phil”);
datadic.Add(3,”cameron”);
```

And to loop over the dictionary

```
foreach(KeyValuePair<int,string> elem in datadic)
{
      Console.Writeline(“{0} and {1}”, elem.Key, elem.Value);
}
```

Lists

Lists are extremely flexible.
They expand automatically when items are added and have numerous methods for adding, deleting, sorting. Also, properties that give you the number of items in the list
The key difference with VBScript is that they are strongly typed.

To create a list and add items

```
// Dynamic ArrayList with no size limit
List<int> numberList = new List<int>();
numberList.Add(32);
numberList.Add(21);
numberList.Add(45);
numberList.Add(11);
numberList.Add(89);
```

```
To iterate over a list
Console.Writeline(“There are ” + numberList.Count() + “ items in the list”);
Console.Writeline(“They are:”);

foreach(int i in numberList)
{
   Console.Writeline(i);
}
```

`List<T>` can be sorted using the `Sort()` method.

For example:

```
//Alphabetical list of titles
booknodes = XMLDoc.XPathSelectElements("//title");
List<string> alltitles = new List<string>();
foreach(XElement elem in booknodes)
 {
    //build list of titles
    alltitles.Add(elem.Value);
 }
alltitles.Sort(); //sort the list

//now loop over the list (see above)to display the sorted list
```

Handling XML

The following section describes a way to handle XML in C#.
There are numerous ways to do this in C#, and most developers will have their own preference but using the XDocument class to manipulate XML is probably the most straight forward.

This section describes one way of loading XML strings and getting data from them using the XDocument class

There is an App Designer sample called SAMXMZ that has a working example of these techniques.

1. Basics

To use the XDocument class several "using" statements are required.
These are:

```
using System.Xml.Linq       //enables use of XDocument
using System.Xml.Xpath            //enables XPath queries with XDocument
using System.Linq                 //reqd. for the Count() method on XDocument
```

These statements should be put at the beginning of the C# code.

2. Create XML Document from an XML string – Parse()

```
XDocument XMLDoc = new XDocument();

try
{
   //read xml string and create an XML Doc
   XMLDoc = XDocument.Parse(XMLStructure);
}
catch (Exception ex)
{
   SysproMessageBox(ex.ToString(), "Creating XML doc", "", SysproButtons.Ok);
}
```

Where `XMLStructure` is an XML string from a Business Object

3. Create XML Document from a file – Load()

The "Load" method of XDocument creates a new XDocument from a file specified by a URI, from a TextReader, or from an XmlReader.

e.g

```
XDocument XMLDoc = new XDocument();

try
{
   //read xml string and create an XML Doc
   XMLDoc = XDocument.Load("C:\Syspro\work\temp\myxmldoc.xml");
}
catch (Exception ex)
{
   SysproMessageBox(ex.ToString(), "Creating XML doc", "", SysproButtons.Ok);
}
```

4. Querying an XML Document

The easiest way to query the XML document is to use XPath syntax.

Get a specific element value – XpathSelectElement()

```
var m_node = XMLDoc.XPathSelectElement("/BomQuery/Parent");

if (m_node == null)
{
SysproMessageBox("Unable to find <Parent> node");
return "";
}
string key = "";
key = m_node.XPathSelectElement("StockCode").Value;
```

Get a collection of elements – XpathSelectElements()

```
var componentnodes = XMLDoc.XPathSelectElements("//BomQuery/Parent/Component");

if (componentnodes.Count() == 0)
{
SysproMessageBox("No <Component> elements found", "Looking for       components",
"", SysproButtons.Ok);
          return "";
}
```

Iterate over an element collection

See above on how to create the collection

```
foreach (XElement element in componentnodes)
{

    //do something for each element
```

```
}
```

Get an attribute from an element

```
//example XML data element
//<people>
//....
//<person name="Fred Smith" hobby="art"/>
//....
//</people>

//get all elements with a hobby of 'art'
var hobbyelements = doc.XPathSelectElements("//person[@hobby='art']");

if (hobbyelements != null && hobbyelements.Count() > 0)
{
    //get the names of the people who have art as a hobby
    foreach (XElement element in hobbyelements)
    {
        msg = msg + element.Attribute("name").Value + Environment.NewLine;

    }
    MessageBox.Show(msg);
}
else
{
    MessageBox.Show("Couldn't find any hobbies");
}
```

Get Form/Field attribute data

App Designer data source can format an XML response for auto populating a form.
This is an example of retrieving an attribute value based on the value of another attribute.

e.g. Get the value of the form field based on its column name

```
//Xpath query to pick up a field with a column name of city_name
//this returns a single XElement object
var dataelem = XMLDoc?.XPathSelectElement("/Form/Field[@Column ='city_name']");


if (dataelem != null)
{
   SysproMessageBox("city:    '"    +    dataelem.Attribute("Value").Value    +
"'","","",SysproButtons.Ok);
}
```

Create an XDocument and add elements

- Create new XDocument and add a root element.

```
XDocument xd = new XDocument();
XElement root = new XElement("ContactList");
xd.Add(root);
```

- Create a new element with data
  This code creates an element called “Contact” with child elements

```
XElement AContact = new XElement("Contact",
    new XElement("Name", "Gretchen Rivas"),
    new XElement("Phone", "206-555-0163"),
    new XElement("Address",
    new XElement("Street1", "123 Main St"),
    new XElement("City", "Mercer Island"),
    new XElement("State", "WA"),
    new XElement("Postal", "68042")),
    new XElement("NetWorth", "11"));
```

- Add new contact to the XDocument root element
  `root.Add(AContact);`
- An alternative syntax would be
  `xd.Element(“ContactList”).Add(AContact);`

Delete an element based on the value of a child element

To delete all Contact elements which have an address state of WA.

```
xd.Descendants("Contact")
    .Where(x => x.Element("Address").Element("State").Value == "WA").Remove();
```

5. Convert XML to Json using Newtonsoft

To use `Newtonsoft` in App Designer the first thing that needs to be done is add `Newtonsoft` as a Reference to the App.

Newtonsoft.Json.dll is located in BASE\ManagedAssemblies\SYSPROSA_ScriptingHost

Then add the `using` statement.

```
using Newtonsoft.Json;
```

Then finally some code that parses an XML string and converts it to Json

```
private void XmlToJsonExample()
{
    string xml = @"<?xml version='1.0' standalone='no'?>
<root>
  <person id='1'>
```

```
  <name>Andy</name>
  <url>http://www.google.com</url>
  </person>
  <person id='2'>
  <name>Cameron</name>
  <url>http://www.yahoo.com</url>
  </person>
</root>";

     XDocument XMLDoc = new XDocument();
     XMLDoc = XDocument.Parse(xml);

     string json = JsonConvert.SerializeXNode(XMLDoc);

     SysproMessageBox(json);
 }
```

Please note that Newtonsoft has numerous options to control how the Xml is converted to Json including formatting (prettifying), omitting root elements, etc.

6. Convert Json to XML using Netwonsoft

To use `Newtonsoft` in App Designer the first thing that needs to be done is add `Newtonsoft` as a Reference to the App.

Newtonsoft.Json.dll is located in BASE\ManagedAssemblies\SYSPROSA_ScriptingHost

Then add the `using` statement.

```
using Newtonsoft.Json;
```

Then finally some code that parses an XML string and converts it to Json

```
private void JsonToXmlExample()
{
   string Json = @"
{
    ""SquidGame"": {
        ""Genre"": ""Thriller"",
        ""Rating"": {
            ""@Type"": ""Imdb"",
            ""#text"": ""8.1"",
        },
        ""Stars"": [""Lee Jung-jae"", ""Park Hae-soo""],
        ""Budget"": null
    }
}";

    var xmlDoc = new XDocument();
    xmlDoc = JsonConvert.DeserializeXNode(Json);

    SysproMessageBox(xmlDoc.ToString());
  }
```

The ‘using’ directive

The “using” directive imports types from a namespace and makes them available to the C# code so that you don’t have to specify a fully qualified namespace to use the type.
They are normally seen at the top of C# code above the namespace.
e.g.

```
using System;
using SYSPROSA_ScriptCommon;
```

The following is a list of useful “using” statements for use with App Designer. They can be imported into your code from the “Sample Code” tab in the text editor.

```
//Sample using statements - delete the ones you don't need
 //XML and XDocument
 using System.Linq;          //XDocument: reqd for Count() method
 using System.Xml.Linq;      //XDocument: this is to use XDocument
 using System.Xml.XPath;     //XDocument: this is reqd to use XPath with XDocument
 using System.Security;      //XDocument: to replace special chars in XML

 //string handling
 using System.Text;            //reqd for StringBuilder
 using System.Text.Json;       //reqd for Json
 using System.Text.Json.Nodes; //reqd to query Json
 using System.Text.RegularExpressions; //needed for Regex

 //general
 using System.IO;              //reqd for file handling e.g. File.Exists()
 using System.Collections.Generic;  //needed for the List<>

 //Threading and Tasks - see AppDesigner sample SAMCOZ
 using System.Threading;       //reqd to use sleep;
 using System.Threading.Tasks; //required to create Task<T> for async calls

 //Web API calls - see AppDesigner sample SAMCOZ
 using System.Net;             //used for WebClient
 using System.Net.Http;        //required for httpClient calls to web API
```

Error handling

This section describes the basics of error handling

In VBScript the `On Error Resume Next` syntax is used to trap an error in the `Err` object and then force execution of the next line of code where typically the `Err` object would be interrogated and the error dealt with. To turn off the error handling the `On error goto 0` syntax is used which forces the VBScript run time to trap and report errors.

In C# the `try...catch` syntax is used

e.g.

```
try
{
    //block of code to try
}
catch (Exception ex)
{
    //Block of code to handle errors
}
```

There can be multiple catch blocks trapping different exceptions

e.g.

```
try
{
    //block of code to try
}
catch (System.Xml.XmlException xmlex)
{
    //Block of code to handle XML parsing errors
}

catch (Exception ex)
{
    //Block of code to handle all other errors
}
```

SYSPRO inbuilt methods

SYSPRO has built in several methods that the C# scripting engine recognises. These become crucial to the usage of SYSPRO scripting. Functionality, such as the SysproMessageBox and SysproInputBox, become important and valuable to any SYSPRO script.

See the SYSPRO help page for more information regarding the usage of these methods.

syspro.com