X# from Scratch

Eric Selje
Salty Dog Solutions, LLC
@EricSelje
www.SaltyDogLLC.Com
eric@saltydogllc.com

X# is on your radar but you're a little unsure how to get started? This whitepaper will walk you through building your first X# application. We'll take a sample FoxPro application and convert it piece by piece to X# and demonstrate transferring our existing VFP skills to X#'s paradigms.

You will learn:

- How to take your first steps with X#
- How to access DBF files in X#
- How to do classes, forms, and more in X#

Introduction

In 2019 at Southwest Fox, I presented a high-level overview of X#, covering its origins and evolution up to its present state. X# is a very stable and mature product at this point, and perfectly capable of creating sophisticated datacentric Windows applications, or being the middleware for web-based ASP.Net apps. If you haven't read my whitepaper from that session, I think it'd give you a good foundation. It's at http://saltydogllc.com/wp-content/uploads/SELJE-Look-at-X-Sharp.pdf.

While X# has been around for years, what's newer and exciting in X# is its support for the Visual FoxPro *dialect*. This support makes learning X# from a Visual FoxPro developer's perspective as easy as, say, learning Spanish once you know Italian (I'm speculating here – I know neither!) There's a lot in common and many cognates so you should be able to translate your skills to a product that is still supported and takes advantage of the .Net Framework rather than the old Win32 classes.

Putting together this session, the most difficulties I had were not with the language itself but navigating the differences between Visual FoxPro's development environment and Visual Studio's. If you're experienced with Visual Studio that will be one less barrier for you to hurdle. And if you're experienced with developing C# applications in Visual Studio, you'll probably find X# to be extremely easy to pick up.

Aside: If you're wondering, "Why should I even learn X# if I'm an experienced C# developer," it's because X# adds DBF handling capabilities natively into the language. You can create data handling classes in X# that are referenced by your existing C# classes.

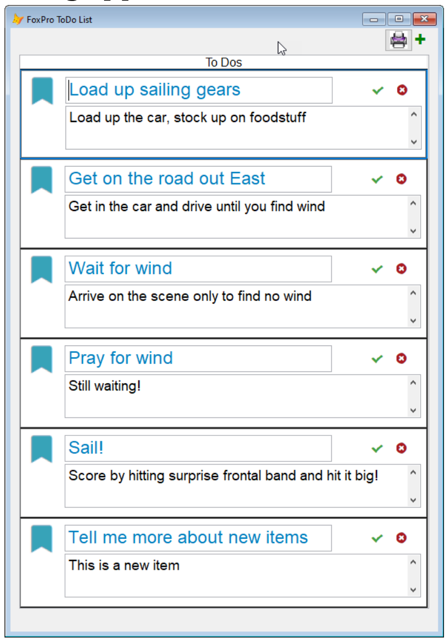
Let's get started! In this session we're going to start with what we already know – a Visual FoxPro application that I put together. It's not a real functioning application because I wanted an example that was small enough to translate but also included a lot of the features that we use in FoxPro and will want to use in X# as well.

Our Sample Application

The original FoxPro app is a simple ToDo list manager, FoxToDos. If it looks familiar it's because I borrowed the UI heavily from Rick Strahl's Vue session. [Thanks Rick!] I even used the same DBF free table that he used because his To Do list is much, much cooler than mine.

You can grab the source code for FoxToDos at my GitHub account, https://github.com/eselje/FoxToDos.

Under the covers, our application consists of the parts shown here.



Load up sailing gears
Load up the car, stock up on foodstuff

Get on the road out East
Get in the car and drive until you find wind

Wait for wind
Arrive on the scene only to find no wind

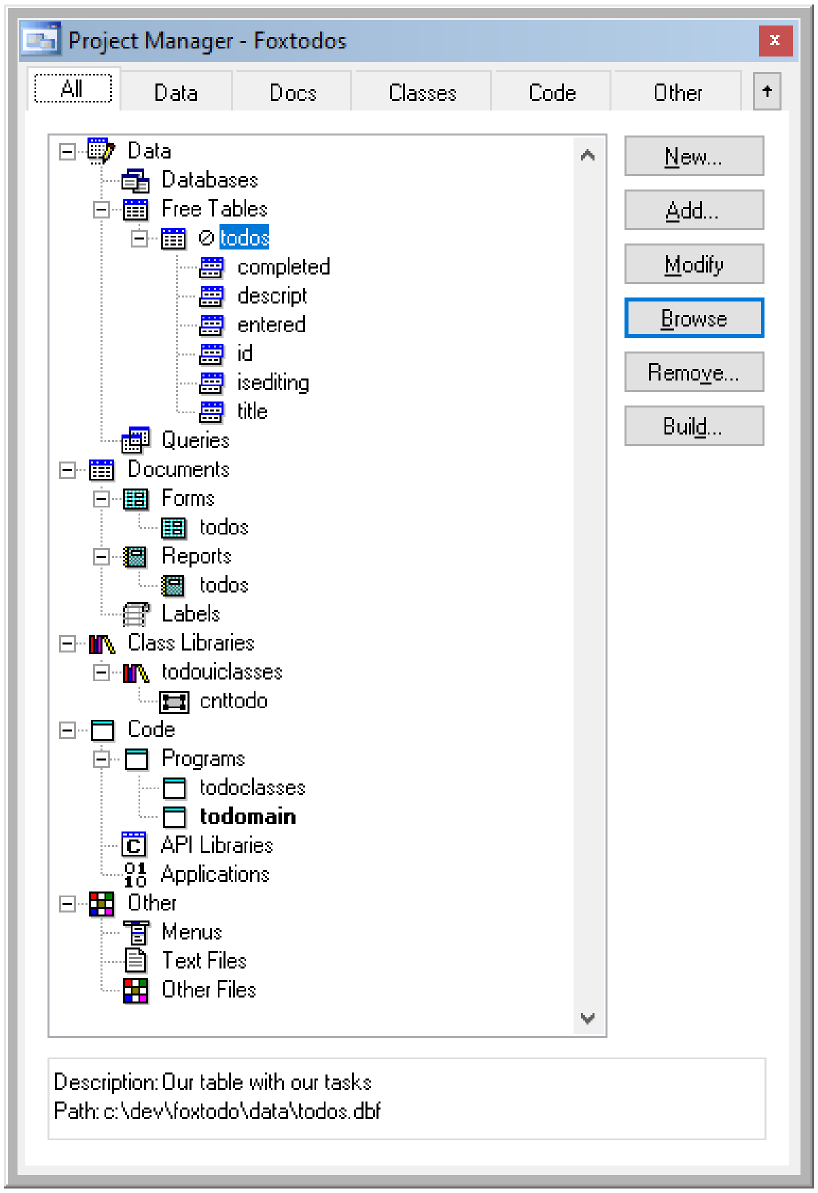
Pray for wind
Still waiting!

Sail!
Score by hitting surprise frontal band and hit it big!

Tell me more about new items
This is a new item

Figure 1: FoxToDos

FoxToDos was not based on any application framework, so it's simpler and less robust than any real application would ever be. It contains (in order):



Description: Our table with our tasks
Path: c:\dev\foxtodo\data\todos.dbf

Figure 2: FoxToDos Project Structure

- ToDos.dbf, the free table with our tasks.
- ToDos.scx, a form that serves as the user interface and includes a grid that contains one custom control, cntToDo.
- ToDos.frx, a simple, wizard-generated report for those that like their tasks on paper.
- ToDoUIClasses.vcx, a visual class library , that has the cntToDo control we use in the grid on our form.
- ToDoClasses.prg contains our non-visual business objects. while has one user interface control, cntToDo, for presenting each task on the form cohesively. Lastly, we have a simple wizard-generated report Converting these to X# should be enough to give us a good feel for what the experience is like.
- ToDoMain.prg. A simple startup program that gets us going.

Conversion Strategy

The steps we're going to follow to convert this Visual FoxPro application to X# is:

1. Create a new solution in Visual Studio
2. Rewrite the classes in ToDoClasses.prg as X# classes
3. Unit test those business objects inside of Visual Studio
4. Create a form in Visual Studio that uses those business objects to interact with the database, and also contains the equivalent of our user-interface composite control.
5. Create an application that has code to setup, run our form, and shutdown.
6. Look at possibilities for creating a report to output the tasks.

The Development Environment

For developing in X#, you have three choices:

1. Use any editor you like (ahem, VI), and compile using the command-line compiler. I'll leave it to the reader to explore this option.
2. XIDE, the X# integrated development environment that may be downloaded along with the rest of X#. XIDE is a perfectly serviceable environment and has a lot in common with Visual FoxPro's IDE. It's written in X# itself, so it provides a dramatic example of what the language can do in the right hands.
3. Visual Studio, either the Professional (ie paid) or Community (ie free) Edition. The big advantage of Visual Studio is that it's used by a lot of developers all over the world, so it's well-supported by its developer and the community. It has a ton of features but in my experience it's also a bit of a dog performance-wise and is a resource hog. It's a hog and a dog –a hot dog if you will. (If you laughed at that joke, shoot me an email and I'll buy you a drink the next time I see you.)

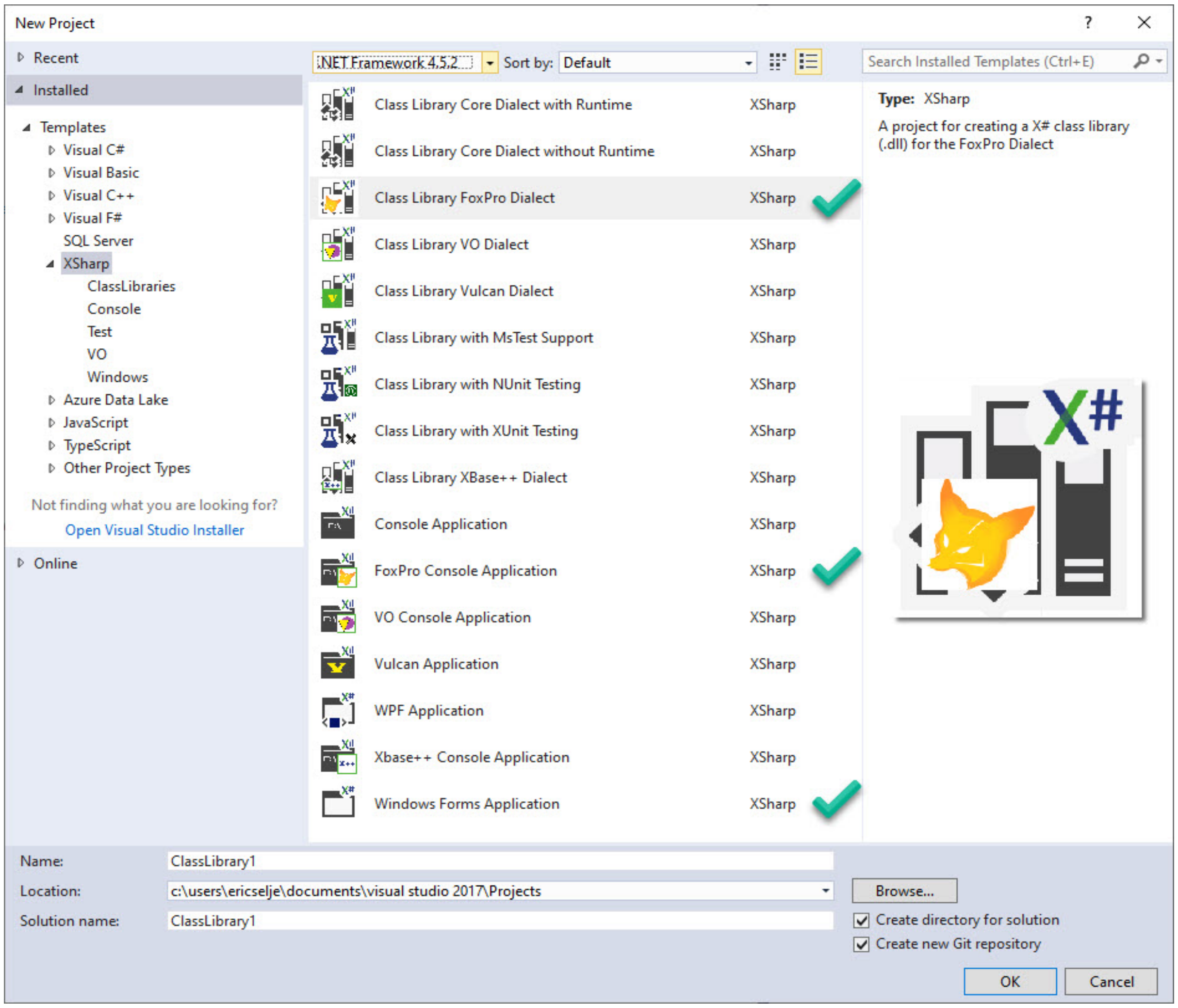
Visual Studio Professional 2017 is the environment I'll be using for this session. If you're not familiar with Visual Studio, the X# Help File has an introduction on using X# in Visual Studio.

Create a New Solution

In Visual Studio's parlance, a "Solution" is the main structure for an application. It's a collection of Projects, which are the main work units. It's good practice to put business objects in their own Project and keep user interface elements in their own separate Project, because then the business objects could be separated and re-used (i.e. "referenced") in multiple Solutions.

To create a new Solution from scratch, choose File, New, *Project* from the menu. The dialog lets you specify the name of the Solution that will contain your new Project, and will create the solution for you. (If you wanted to create a new project as part of an *existing* Solution, you must open that Solution, and choose Add, New Project. See **Figure 4**).

Because we've installed X# already (an exercise left to the reader, but it's a straightforward download and install from http://www.xsharp.info), we have XSharp Templates available to us.



Class Library Core Dialect with Runtime	XSharp
Class Library Core Dialect without Runtime	XSharp
Class Library FoxPro Dialect	XSharp ✓
Class Library VO Dialect	XSharp
Class Library Vulcan Dialect	XSharp
Class Library with MsTest Support	XSharp
Class Library with NUnit Testing	XSharp
Class Library with XUnit Testing	XSharp
Class Library XBase++ Dialect	XSharp
Console Application	XSharp
FoxPro Console Application	XSharp ✓
VO Console Application	XSharp
Vulcan Application	XSharp
WPF Application	XSharp
Xbase++ Console Application	XSharp
Windows Forms Application	XSharp ✓

Type: XSharp
A project for creating a X# class library (.dll) for the FoxPro Dialect

Name: ClassLibrary1
Location: c:\users\ericselje\documents\visual studio 2017\Projects
Solution name: ClassLibrary1
✓ Create directory for solution
✓ Create new Git repository

Figure 3: New Project dialog, with interesting Project Templates noted

What do Project Templates do for you?

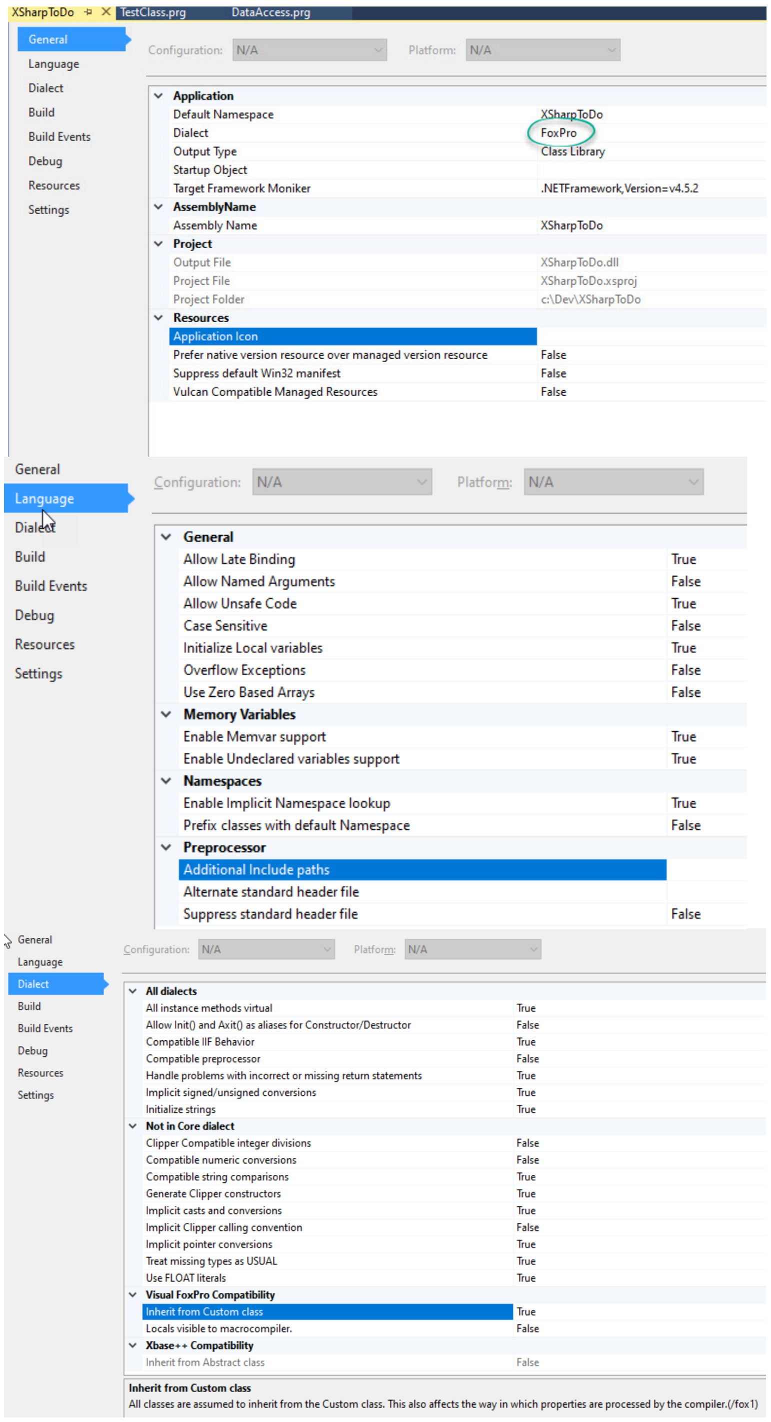
Project Templates set certain properties (see **Figure 4**) of the project that are tweaked for that project type. They may also include code files and other assets that that type of project will typically use. For example, notice how the Dialect is automatically set to **FoxPro** after we create a Project based on the **Class Library FoxPro Dialect** template. Projects based on this template also contains one barebones PRG library to get us started (**Figure 5**) , but they could contain include as much as a complete application framework.

Visual Studio allows developers to create their own Project templates, much like Word and Excel allow you to create your own templates for documents and spreadsheets.

There are three Project Templates shown in **Figure 3** that are of specific interest to us as FoxPro developers:

- **Class Library FoxPro Dialect.** A project of this type will have the FoxPro dialect pre-set in the General properties (see **Figure 4)** and include a barebones starter class definition.
- **FoxPro Console Application.** This also sets the dialect to FoxPro, but also sets the "Output Type" property to "Console Application". This allows us to create command line utilities in a FoxPro dialect with proper Exit Codes, which was difficult using Visual FoxPro. Command line applications play well with development pipelines such as Jenkins, a Continuous Integration tool.
- Windows Forms (or WPF) Application

These aren't FoxPro-specific, but we'll be explore using these to create the user interface for our task list.



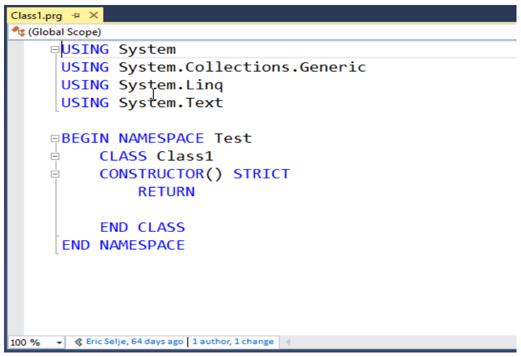
Application	
Default Namespace	XSharpToDo
Dialect	FoxPro
Output Type	Class Library
Startup Object	
Target Framework Moniker	.NETFramework,Version=v4.5.2
AssemblyName	
Assembly Name	XSharpToDo
Project	
Output File	XSharpToDo.dll
Project File	XSharpToDo.xsproj
Project Folder	c:\Dev\XSharpToDo
Resources	
Application Icon	
Prefer native version resource over managed version resource	False
Suppress default Win32 manifest	False
Vulcan Compatible Managed Resources	False

General	
Allow Late Binding	True
Allow Named Arguments	False
Allow Unsafe Code	True
Case Sensitive	False
Initialize Local variables	True
Overflow Exceptions	False
Use Zero Based Arrays	False
Memory Variables	
Enable Memvar support	True
Enable Undeclared variables support	True
Namespaces	
Enable Implicit Namespace lookup	True
Prefix classes with default Namespace	False
Preprocessor	
Additional Include paths	
Alternate standard header file	
Suppress standard header file	False

All dialects	
All instance methods virtual	True
Allow Init() and Axit() as aliases for Constructor/Destructor	False
Compatible IIF Behavior	True
Compatible preprocessor	False
Handle problems with incorrect or missing return statements	True
Implicit signed/unsigned conversions	True
Initialize strings	True
Not in Core dialect	
Clipper Compatible integer divisions	False
Compatible numeric conversions	False
Compatible string comparisons	True
Generate Clipper constructors	True
Implicit casts and conversions	True
Implicit Clipper calling convention	False
Implicit pointer conversions	True
Treat missing types as USUAL	True
Use FLOAT literals	True
Visual FoxPro Compatibility	
Inherit from Custom class	True
Locals visible to macrocompiler.	False
Xbase++ Compatibility	
Inherit from Abstract class	False

Inherit from Custom class
All classes are assumed to inherit from the Custom class. This also affects the way in which properties are processed by the compiler.(/fox1)

Figure 4: Project Properties, as set by the Project Templates



```
USING System
USING System.Collections.Generic
USING System.Linq
USING System.Text

BEGIN NAMESPACE Test
    CLASS Class1
    CONSTRUCTOR() STRICT
        RETURN

    END CLASS
END NAMESPACE
```

Figure 5: Starter Code from the Project Template

What's interesting about the starter class from the template is that it uses syntax that doesn't really look like FoxPro's. The X# dev team has updated its understanding of VFP syntax now, but I suspect the templates were written before that was officially supported. We'll see the updated syntax used in our translations later.

Pay extra special attention to the **Visual FoxPro Compatibility / Inherit from Custom Class** setting. If you want your code to behave the way it does in VFP, you want that set to True. Among other things, that causes X# to fire an `Init()` method (which wasn't a thing in X# before, as they use `Constructor()`) and create virtual _access and _assign methods for our "properties" – more on that later.

The code for the classes we're going to migrate is in **Appendix A**, for readability's sake. You can download the FoxPro source code from my GitHub repository at https://github.com/eselje/FoxToDos and the final X# solution at https://github.com/eselje/XToDos.

This FoxPro class library includes two class definitions:

- ToDo, which manages an individual task, and
- ToDos, which manages a collection of the ToDo objects

Though I could have written this class library in a VCX (Visual Class Library), I chose to do it in straight code because it makes it easier to illustrate the transition to X#, which doesn't have the concept of "Visual Classes" as we know them. All coding is done in text, which is a huge advantage when it comes to source control because there's no need for any of the myriad workarounds that we had to implement in VFP to serialize our binaries. As we'll see when we talk about forms, menus, and custom controls, there is a "visual" element to Visual Studio, but the source code itself is all text.

Disclaimer: Any imperfections or questionable design decisions that you find in this code may have been purposely included to illustrate some important point. Or they may just be an error. It's hard to say.

Converting Classes

ToDo => XtoDo

We'll start with the ToDo class, which reads and writes individual tasks to the DBF file.

In FoxPro, we begin defining the class with

```
DEFINE CLASS ToDo AS Custom
  Name = "ToDo"
  cId = ""
  oData = .null.
  lNew = .f.
  lSaved = .f.
  lLoaded = .f.
  oException = .null.
```

in XSharp, we begin with

```
USING System
USING System.Collections.Generic
USING System.Text

BEGIN NAMESPACE XSharpToDo
   DEFINE CLASS XToDo as Custom
    public id as string
    public title as string
    public descript as string
    public entered as datetime
    public completed as boolean
    private isEditing as boolean
```

Using

The initial X# `using` statements are somewhat akin to FoxPro's `set classlib to` in that they tell the program, "Hey I have some code stored somewhere else that I'm maybe going to use here so make this available to me, and if I make a call to a function that you don't recognize it may be in one of these classes." There's no requirement that you actually *do* use them, but if you refer to them the compiler is going to include them. In other .Net languages I've used, you could remove any unneeded `using` by right+clicking and choosing "Remove Unused Usings" (an awkward phrasing, in my opinion), but that doesn't seem to work on .prg files so you'll want to remove any that you don't use manually to shrink the size of your codebase.

.Net has a minimalist concept, along the lines of "If you want it, you gotta include it or it's not going to be available." `System` is the root namespace and includes what we might refer to as the "Base Classes" in VFP, but not the visual base classes that were included in _Classes.vcx, but the native data types such as strings, integers, etc. If you didn't include `System` explicitly, you'd have to preface any of the classes that are in system each time you called them, so `Console.Write()` would have to be called as `System.Console.Write()`.

Also notice that `USING System` does NOT include every sub-namespace of System. You have to explicitly use any library that you want access to. For a complete list of .Net's native class libraries, see the (excellent) docs at https://docs.microsoft.com/en-us/dotnet/standard/class-library-overview.

Begin Namespace

Namespaces aren't new to FoxPro developers, although we maybe didn't refer to them as such. If you created an OLEPUBLIC class in Visual FoxPro, the name of the DLL you created would be its Namespace. You could then use that class from another program with

Best Practice

USE PRIVATE FIELDS AND PUBLIC PROPERTIES

`CreateObject("NameSpace.ClassName")`.

By specifying the namespace in code with `BEGIN NAMESPACE,` .Net allows you to spread the code for classes among multiple files which makes it *much* easier to manage the source control. No more conflicts because one developer was refactoring an entire class when you only wanted to make a minor tweak to another class in the same class library!

The next few lines are very similar. .Net is strictly typed, so when you specify the class property you must specify the type too. Also, in VFP we have the very powerful `SCATTER` and `GATHER` command, allowing us to use one property, oData in my example, to dynamically store the field values, while in the X# class we don't have that yet (UPDATE: This was released on Sept 20, 2020, in version 2.6, too late for me to implement!), so must explicitly name the fields.

Properties v. Fields

There is a fundamental difference in .NET classes vs Visual FoxPro classes. In VFP when we added what we called a "property" to a class, we could immediately assign values to that property without going through any hoops. This is bad, because there were no checks on the input at all, and anyone could read the value. We got around that by adding `_access` and `_assign` methods to the properties. The "visibility" of the property (public, protected, hidden) affected how whether other objects could see the properties, but had no effect on what values were visible within the class itself.

.Net classes call those "Fields" rather than properties, and their visibility is determined by whether they're Public or Private. Public fields are akin to our Properties, but this isn't recommended because all the reasons mentioned above.

Properties on a .Net class are the public-facing interface, akin to our `_access` and `_assign`, they have get() and set() methods, which filter the input to the fields or restrict the output.

In order to emulate Visual FoxPro's class behavior, there's an option for X# classes to "Inherit from Custom Class", which is set to True by default in the FoxPro Project Templates. Under the covers, this Custom class emulates FoxPro Properties in .NET's Fields.

Init() vs Constructor()

FoxPro's classes all come with an `Init()` method that accepts parameters . You get exactly one Init per class and you must code around the possible combinations of parameters that were sent in. X# classes have a Constructor method, and you can overload them with different "*signatures*": different combinations of parameters, which is awesome.

FoxPro

```
PROCEDURE Init
LPARAMETERS cId
This.cId = cId
IF EMPTY(cId)
    This.New()
ELSE
    This.Load(cId)
ENDIF
ENDPROC
```

X#

```
public FUNCTION Constructor()
// No Parameter. New Task.
    This.New()

public FUNCTION Constructor(cId AS String)
// Parameter. Existing Task
    This.cId = cId
    This.Load(cId)
```

In the FoxPro dialect, class definition in X# do have an `init()` method which you can use exactly like FoxPro's.

When you create a task object, you either send it the ID of the task, or leave it blank if you want to create a new task. FoxPro's `init()` handles that by inspecting the parameter and branching based on whether anything was passed in, but X# has the overloaded constructor that either takes an ID parameter or doesn't. I find that much more intuitive.

The `new()` function highlights how similar X# can be to FoxPro, but also showcases the extra functionality you can get. The code we need to create GUIDs in .NET is available to us (in this cases, from System which we always include, but we have the entire universe of .NET libraries available to us and if for some reason we didn't like this GUID library we could have referenced some other one instead.)

FoxPro

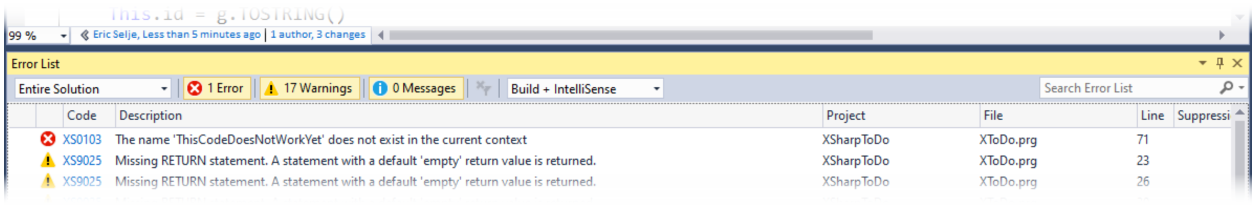
```
PROCEDURE New
LOCAL lUsed, oGUID
lUsed = This.OpenToDos()
oGUID = CreateObject("scriptlet.typelib")
SCATTER BLANK NAME This.oData MEMO
This.oData.Id =Strextract(oGUID.GUID, "{", "}" )
This.oData.Entered = DATETIME()
This.lNew = .t.
RETURN This.oData
```

X#

```
PROCEDURE New
LOCAL lUsed
This.Clear()
VAR g = GUID.NEWGUID()
This.id = g.TOSTRING()
this.entered = DateTime.Now
this.isEditing = true
this.isNew = .t.
RETURN This.oData
```

Testing our Code

Now that we've got a couple of methods written for our X# class, we can check if it has errors in that time-honored tradition: Does it compile? Hit Ctrl+Shift+B to Build the solution and the Output window will display any issues it finds. DoubleClicking the row that has the issue will take you right to that code in the editor, or click on the error code to take you to a web page that can give you more information about that error.



This.id = g.TOSTRING()

99 % | Eric Selje, Less than 5 minutes ago | 1 author, 3 changes

Error List

Entire Solution | 1 Error | 17 Warnings | 0 Messages | Build + IntelliSense | Search Error List

	Code	Description	Project	File	Line	Suppressi
✖	XS0103	The name 'ThisCodeDoesNotWorkYet' does not exist in the current context	XSharpToDo	XToDo.prg	71	
⚠	XS9025	Missing RETURN statement. A statement with a default 'empty' return value is returned.	XSharpToDo	XToDo.prg	23	
⚠	XS9025	Missing RETURN statement. A statement with a default 'empty' return value is returned.	XSharpToDo	XToDo.prg	26	

Once it compiles without any errors, we want to find a way to make sure it actually works.

Command Window?

We FoxPro developers love to “test” our code by opening the Command Window, instantiating an instance of our class, and invoking the methods manually. If they didn’t do what we expected, we’d set a Breakpoint and walk through the code in debug mode.

Visual Studio doesn’t have a Command Window though. If you installed the XIDE environment you will get something *like* a Command Window called XSI, the X# Interpreter (there’s more on XSI in last year’s whitepaper.) Since we are using Visual Studio for this demonstration, we’ll instead create a quick Console Application that we can use to “test” our code.

To create the Console Application, Right+Click on the Solution, choose Add, New Project (See **figure 6**), base it on the **FoxPro Console Project** template (refer back to **Figure 3**), give it a name, add a reference to the library that we want to test (**Figure 7**), set our new project as the Startup project, and change the code to write our rudimentary test:

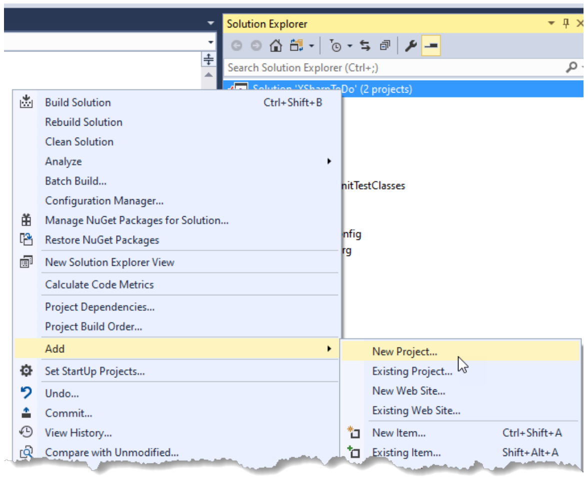
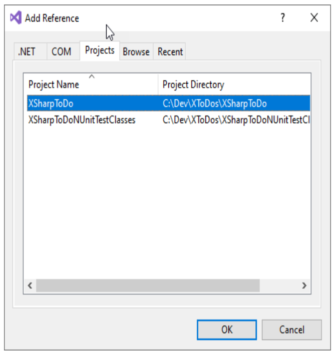


Figure 6: Add project to existing solution

```
USING System
USING XSharpToDo

FUNCTION Start() AS VOID STRICT
    LOCAL oToDo AS XToDo, cTestId AS String,
cDescript As String
    cTestId = "EDF53AEF-5C29-4DC4-A"
    oToDo = createObject("XToDo")
    IF oToDo.openToDos()
        SET DELETE ON
        SCAN
            Console.WriteLine("{0:00}: ID: {1}, {2}",  RECNO(), ToDos.id, ToDos.descript)
//            ? RECNO()  ToDos.id ToDos.descript
             IF ToDos.Id = cTestId
                  cDescript = ToDos.descript
             ENDIF
        ENDSCAN
        ? cTestId +": " + cDescript
        oToDo.Load(cTestId)
        oToDo.closeToDos()
    ELSE
```

```
            ? "Could not open ToDos.dbf"
            ? "Default folder is " + SET("DEFAULT")
    ENDIF
    WAIT
RETURN
```

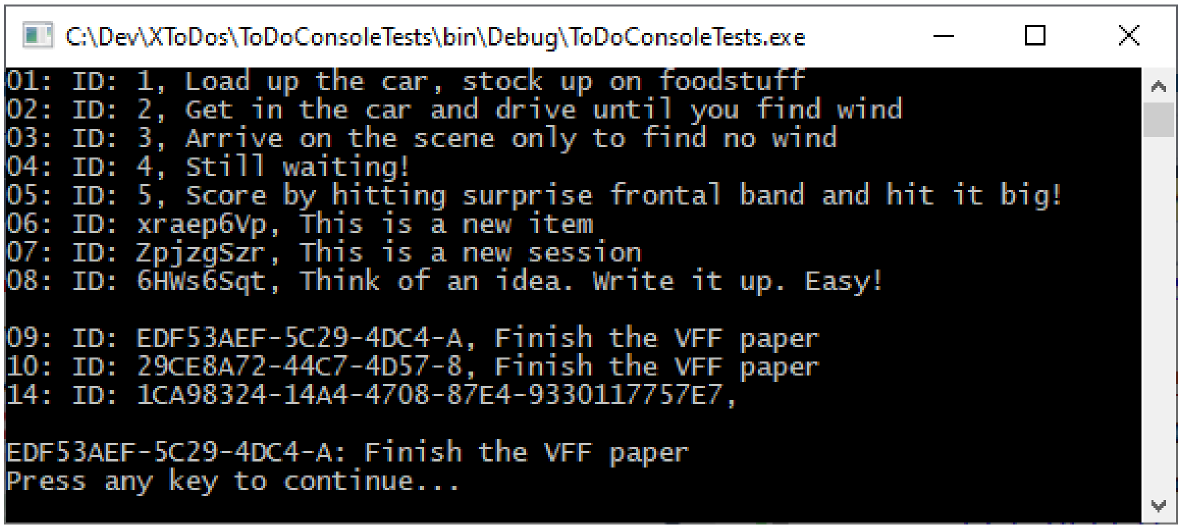


Project Name	Project Directory
XSharpToDo	C:\Dev\XToDos\XSharpToDo
XSharpToDoNUnitTestClasses	C:\Dev\XToDos\XSharpToDoNUnitTestCl

Figure 7: Add a Reference

This is X# code, but it should look *very* familiar to you. I threw in a gratuitous Console.WriteLine to show off some of X#'s extra functionality, but other than that this is straight VFP code.

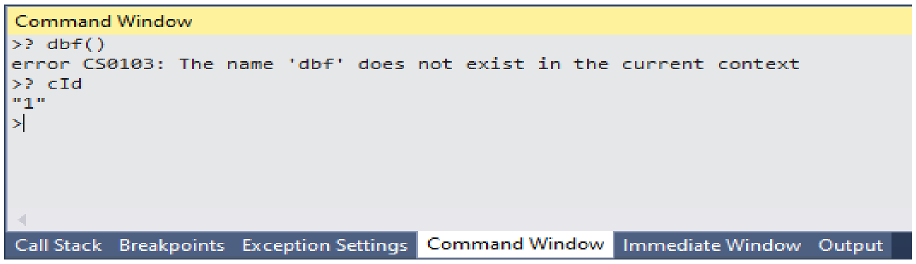
After a few rounds of debugging your code (and as an experienced FoxPro developer you'll have no difficulty grokking Visual Studio's debugger), you should get the expected output in the Output window:



```
C:\Dev\XToDos\ToDoConsoleTests\bin\Debug\ToDoConsoleTests.exe
01: ID: 1, Load up the car, stock up on foodstuff
02: ID: 2, Get in the car and drive until you find wind
03: ID: 3, Arrive on the scene only to find no wind
04: ID: 4, Still waiting!
05: ID: 5, Score by hitting surprise frontal band and hit it big!
06: ID: xraep6Vp, This is a new item
07: ID: ZpjzgSzr, This is a new session
08: ID: 6HWs6Sqt, Think of an idea. Write it up. Easy!

09: ID: EDF53AEF-5C29-4DC4-A, Finish the VFF paper
10: ID: 29CE8A72-44C7-4D57-8, Finish the VFF paper
14: ID: 1CA98324-14A4-4708-87E4-9330117757E7,

EDF53AEF-5C29-4DC4-A: Finish the VFF paper
Press any key to continue...
```

"But wait", you say, "I see a Command Window right there when I'm running this app and you *just said* Visual Studio doesn't have a Command Window! "

Don't get too excited about this. This "Command Window" is only available while you're actively debugging code, and it doesn't even understand all of our FoxPro commands, so it's not particularly useful (yet).



```
>? dbf()
error CS0103: The name 'dbf' does not exist in the current context
>? cId
"1"
>
```

Figure 8: The Command Window

Unit Tests

A better way to test your code is to write unit tests, and in fact true Test-Driven development would have directed us to write those even before we started coding. But this isn't real TDD because a) these aren't "true" unit tests (they interact with the real database), and b) we're just getting around to writing them now. It's important to acknowledge when we know what the right thing to do is, and we're still not doing it.

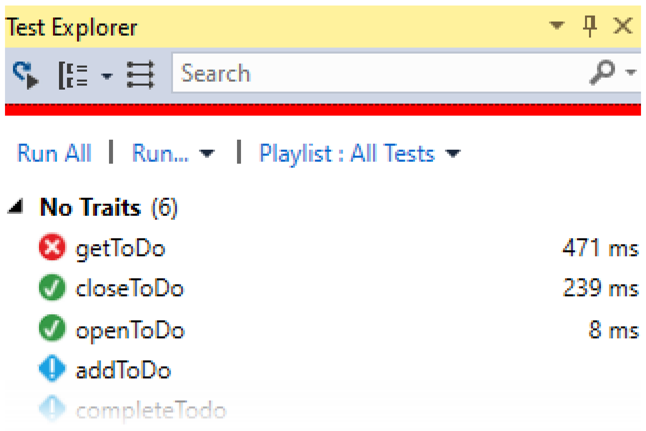
With FoxPro there is one popular unit testing harness, FoxUnit. It's a separate install (via Thor, ideally) and not integrated into the IDE at all.

Visual Studio on the other hand has testing built into the IDE from the ground up, and you have multiple testing frameworks to choose from. I chose Nunit to demo because it's very similar to FoxUnit. In order to create tests, you add a New Project based on the **Class Library with Nunit Testing** to your solution (see **Figure 3**).

Next add a reference to the library we're testing, XSharpToDo, just like we did with our rudimentary console testing app, and write a test:

```
[Test];
   METHOD getToDo AS VOID STRICT
   VAR oToDos := XSharpToDo.XToDos{}
   VAR oTodo := oToDos.getToDo("EDF53AEF-5C29-4DC4-A")
   VAR cExpected := "EDF53AEF-5C29-4DC4-A"
   Assert.AreEqual(cExpected, oTodo.id, "Did not get the right TODO")
   RETURN
```

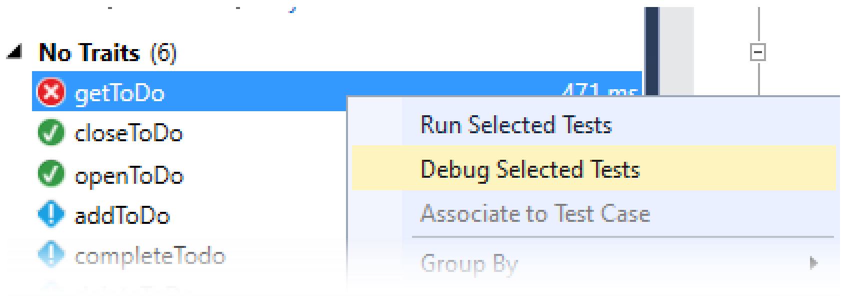
That test now appears in the Test Explorer. When you run the tests, either individually or entirely `(Ctrl+R, A)` there's a clear signal whether the test passed, failed, or hasn't been implemented yet. It also indicates how long the test took to run, which is an early indicator of the performance of your methods.



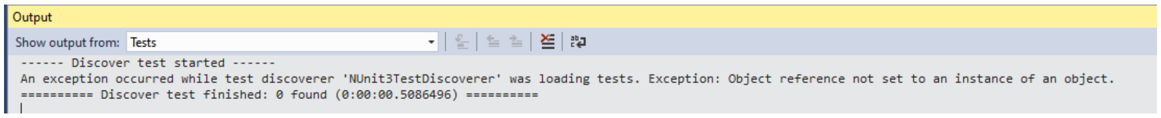
No Traits (6)

getToDo	471 ms
closeToDo	239 ms
openToDo	8 ms
addToDo	
completeTodo	

If you're having difficulty getting the result you expected from a test, select Debug Selected Tests in order to stop at the breakpoints that you've set. That is also when the Command Window will be available to you, as well as Locals and a Watch Window just like Visual FoxPro.



Side Note: One *very* frustrating problem I was having, and this is an Nunit problem and not an X# problem, is that I kept getting this message when trying to run my tests:



```
------ Discover test started ------
An exception occurred while test discoverer 'NUnit3TestDiscoverer' was loading tests. Exception: Object reference not set to an instance of an object.
========== Discover test finished: 0 found (0:00:00.5086496) ==========
```

I am not the only person having this issue, and the only reliable solution seemed to be clearing out the Nunit cache (deleting the folder) at

`%appdata%\..\Local\Temp\VisualStudioTestExplorerExtensions\<nUnitVersion>`

and restarting Visual Studio. I may consider XUnit or MSTest in the future!

Now we know how to add and test methods to our XToDo class, and we can continue adding the rest of them. The code is straightforward. See **Appendix B** for the finished X# code for this test class, which of course may be out of date so go to the repository for the current test code.

Adding Another Class to our Namespace

In FoxPro we can, and usually do, stick all the classes for a class library in either one .VCX or one .PRG. The name of our FoxPro project determines the NameSpace of our classes.

In .NET, this is discouraged. Each class in X# should be in its own code file (.PRG) and all of the classes in the library should be in one Project. This organizational structure makes it much easier to determine which class was modified when examining source code check-ins, and reduces file conflicts.

Best Practice

PUT EACH CLASS IN ITS OWN CODEFILE

To add a new class to the existing Project, right-click on the Project and choose **Add, New Item**. You'll get the Add New Item dialog (**Figure 9**). This shows the list of Item Templates, which are similar to Project Templates mentioned before except they only create one file rather entire projects.

Convention says to name your new .PRG with the name of the class you're creating. The Namespace will default to the name of the Project, which is probably what you want, but you may change it if you like. You can have more than one Namespace in a project.

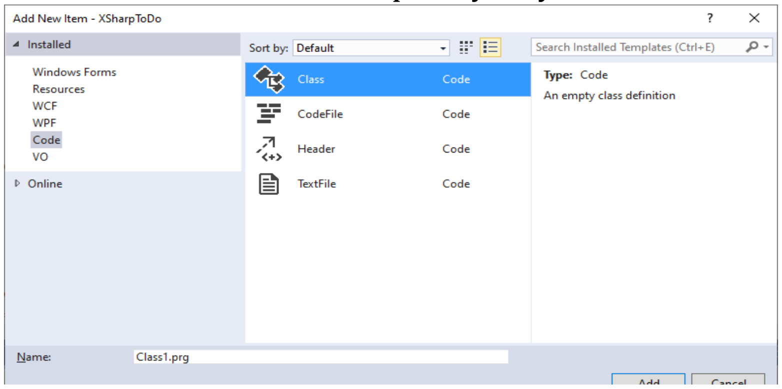


Figure 9: New Item Templates

The Startup Project

Earlier when we created the Console Project to test our software, we set that to be the Startup Project. An application must know what to do when you click Start, so somewhere in your Solution there has to be at least one "Startup Project," and that Startup Project must have a class with a method called `Start()` to get things going for your application. The Start function is responsible for setting up the environment and global variables, opening tables, handling incoming arguments, , etc. To set the Startup Project for your Solution, Right+Click on the Project and choose `Set As Startup`.

It is possible to set up multiple Startup Projects by right+clicking on the Solution and choosing `Set Startup Projects.` This might be useful if you want to, say, fire up an .ASP Net website when you start your Windows Form application.

The exception is if the Solution is a XAML Project, which we'll talk about in the next section.

The User Interface

At this point our business logic is converted and tested, but this application does not yet have a proper user interface (the Console Project notwithstanding). We can choose from any number of user interfaces for our business classes, e.g. an Angular website or a mobile phone app, but a Windows Form is going to emulate our existing VFP application closest.

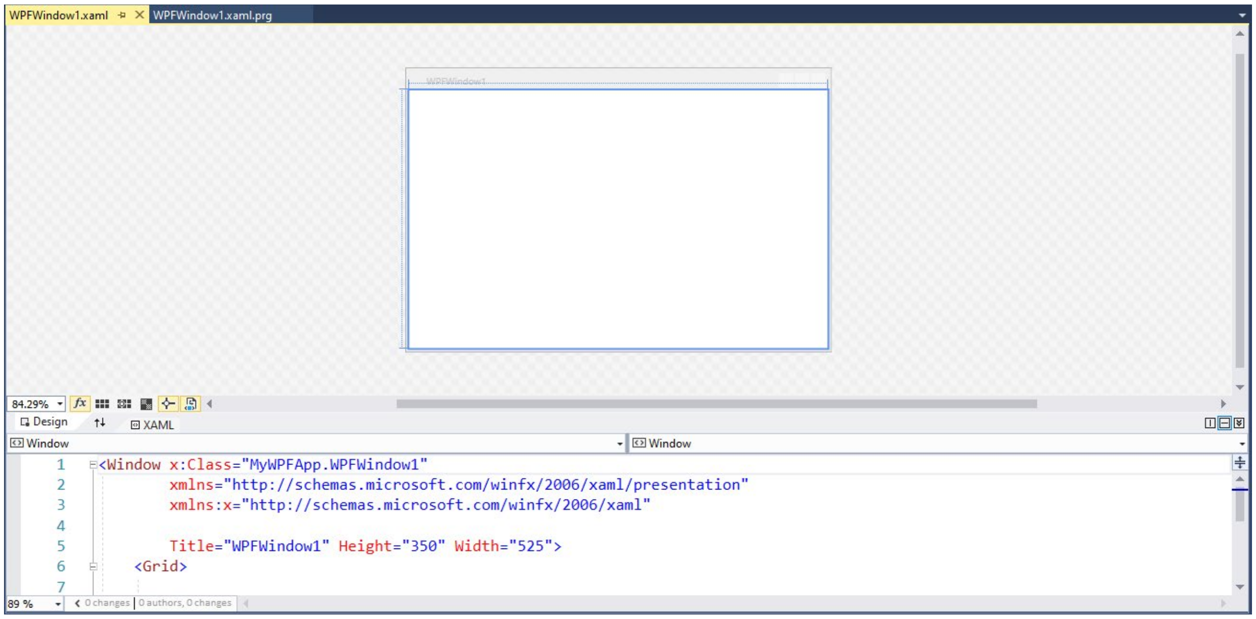
There are two flavors of Windows Forms. The original WinForms at first glance seems to have a lot in common with Visual FoxPro forms. The design surface looks similar, and there is a toolbox with a lot of familiar controls like TextBox and CheckBox. But FoxPro forms were deceptively powerful, allowing you to include controls that were layers upon layers of composition and inheritance, and WinForms simply cannot match that power. Microsoft did say at one point that WinForms was not going to make the leap to .Net Core, the multiplatform version of the .Net framework. They have since walked back on that but the future for WinForms is less certain than the alternative.

Then there's the newer, more complex, and more powerful WPF (Windows Presentation Format) forms. Under the covers it uses an XML dialect called XAML to lay out the interface, but the commands are written using C# or X#. The idea here is that the UI/UX designers on your staff can create the forms, and the coders can work on the actual logic. We FoxPro developers usually do both roles, of course.

Entire books have been written about creating WPF forms, so we can only go into the shallowest of details here, just enough to emulate our VFP form. I will say that of all the things I worked on when writing this paper, getting the forms, and particularly the databinding, working correctly was the most difficult. This isn't a knock on X# at all, since it's not their fault. In fact, the X# devs are working on a utility to convert FoxPro's forms to either WPF or WinForms, but it's not available quite yet. It will be invaluable to getting us over the hump once it's released.

We want to put our user interface in a separate Project from our business classes, but it can be part of the same Solution. To create this new Project, right+click on the Solution and choose `Add, New Project` (see **Figure 3**). This time we choose `XSharp, Windows, WPF Application` as the Project Template. Give the project a name, which becomes the folder name under the location where the files are stored on disc.

The Project Template for WPF Apps includes a starter file for a WPF form (WPFWindow1.xaml) which includes a "code-behind" file (WPFWindow1.xaml.prg). It's PRG because we chose the XSharp Project Template, so it uses the X# "Core" dialect syntax. If you want to use VFP syntax you'll need to modify your Project properties (**See Figure 4**).



```
<Window x:Class="MyWPFApp.WPFWindow1"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

        Title="WPFWindow1" Height="350" Width="525">
    <Grid>
```

It also includes a start App.xaml file

```
<Application x:Class="MyWPFApp.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="WPFWindow1.xaml">
    <Application.Resources>
    </Application.Resources>
</Application>
```

and a code-behind App.xaml.prg which has no code in it. If you set this new XAML Project to be your Startup Project, build and run your app, it actually works (**figure 10**)! But how did the application know that it should fire up that form when it began if there isn't a Start() method? In XAML, an alternative way to indicate the starting form is in the StartupURI attribute of the Application tag in the XAML, which you can see in our code listing points to WPFWindows1.xaml.

Figure 10: Our first XAML app

This application isn't particularly powerful though. In fact it does nothing, but notice that we didn't have to do READ EVENTS or DO FORM or anything to get basic functionality, which is something. That's because the application is an instance of the Application class in .Net[1], which is something VFP never had unless you went with a 3rd party framework.

Adding Controls to our Form

Our sample VFP window has two buttons in the upper right hand corner to Print and Add a Task. In WPF we cannot simply add a button to a form though.

We must start by first add some type of layout container to our window. These are similar to FoxPro's container object, but they have the different types of containers have different behaviors, and allow is to forego the absolution positioning and anchoring that we are used to. This makes the layout very flexible, and usable for all kinds of screen resolutions and devices. Here are some popular ones, but there are many more:

- Canvas: This is the most like a FoxPro container, where you have to position the objects manually into the space available.
- Grid: Every control you add goes into a subsequent checkerboard row or column.
- StackPanel: Every control you add goes next to or below the previous object, depending on orientation.
- WrapPanel: Every control you add goes next to the previous one, and wraps around when the horizontal edge is reached.
- DockPanel: Much like VFPs dockable containers, these get positioned along the edges, or in the center, of its parent container.
- ToolBarPanel: A subclass of the StackPanel that puts extra controls into an "Overflow" area if they don't fit.

You can mix and nest these containers as well. Here's the code to add the two buttons to a ToolBarPanel inside of a StackPanel:

```
  <StackPanel Orientation="Vertical">
    <ToolBarPanel Height="60" Orientation="Horizontal" HorizontalAlignment="Right">
      <Button Background="Transparent" BorderThickness="0">
        <Image Name="imgNew"  Source="Images\Ribbon.png" Height="24px" Width="24px"
HorizontalAlignment="Right" Margin="10" ToolTip="Add Task"></Image>
      </Button>
      <Button Background="Transparent" BorderThickness="0">
        <Image Name="imgPrint" Source="Images\Print.png" Height="24px" Width="24px"
HorizontalAlignment="Right" Margin="10" ToolTip="Print To-Do List"></Image>
     </Button>
   </ToolBarPanel>
 </StackPanel>
```

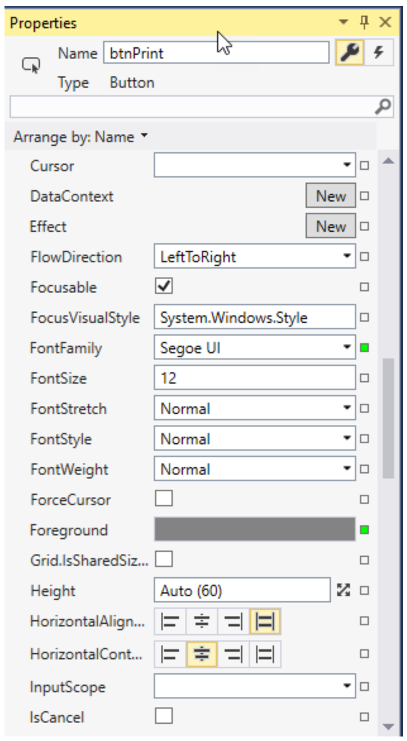
[1] https://docs.microsoft.com/en-us/dotnet/api/system.windows.application?view=netframework-4.7.2

This may seem a little verbose at first, but it's pretty straightforward. Intellisense inside of Visual Studio makes it very easy to add more properties to each control, and you also have a familiar Properties Pane (**figure 11**).

Binding Events to Controls

Right now our Add and Print buttons don't *do* anything, so let's fix that. Because Adding and Printing are events that we may also want to call from the Window's menu (if we had one), and we don't want to duplicate our efforts, we can create a "CommandBinding" in our Window. That gives us a central location to route things through, and also determine if the event is even doable at any given time (e.g. We can't paste unless there's something in the clipboard, or we can't print a task list unless there are tasks to do). Add this code below the <Window> element:

```
<Window.CommandBindings>
  <CommandBinding Command="ApplicationCommands.New"
   Executed="NewCommand_Executed"
   CanExecute="NewCommand_CanExecute" />
  <CommandBinding Command="ApplicationCommands.Print"
    Executed="PrintCommand_Executed"
    CanExecute="PrintCommand_CanExecute" />
</Window.CommandBindings>
```



Properties

Name	btnPrint
Type	Button
Arrange by: Name	
Cursor	
DataContext	New
Effect	New
FlowDirection	LeftToRight
Focusable	✓
FocusVisualStyle	System.Windows.Style
FontFamily	Segoe UI
FontSize	12
FontStretch	Normal
FontStyle	Normal
FontWeight	Normal
ForceCursor	☐
Foreground	▨
Grid.IsSharedSiz...	☐
Height	Auto (60)
HorizontalAlign...	
HorizontalCont...	
InputScope	
IsCancel	☐

Figure 11: Control Properties

This collection of Command Bindings gives a name to each of our commands, tells us what to do when the event gets fired, and whether the event even *can* be fired. In the Code Behind of the Window we add those methods (I went with C# for my WPF project, but I could have chosen X#. There is so little code here that it matters little):

```
private void NewCommand_CanExecute(object sender, CanExecuteRoutedEventArgs e)
       { e.CanExecute = true; }
private void NewCommand_Executed(object sender, ExecutedRoutedEventArgs e)
       { MessageBox.Show("New Task"); }
private void PrintCommand_CanExecute(object sender, CanExecuteRoutedEventArgs e)
        { e.CanExecute = true; }
private void PrintCommand_Executed(object sender, ExecutedRoutedEventArgs e)
         { MessageBox.Show("Print"); }
```

We now just need to add one more attribute to each button to wire them up to the commands:

```
Command="ApplicationCommands.New"
Command="ApplicationCommands.Print"
```

Now when we start the application, we see our window, with its two buttons inside of a ToolBarPanel inside of the StackPanel, we get what we see in **Figure 12.** Sweet!

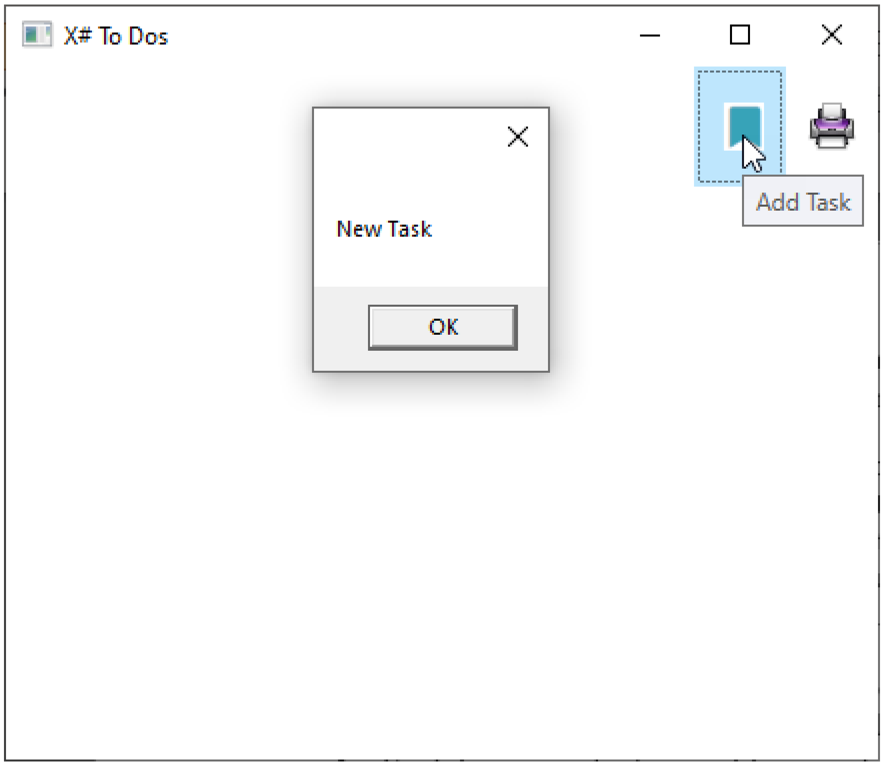


Figure 12: Our application, improved.

Tying our Business Objects to the Form

We spent all that time creating and testing our business objects, and now it's time to put them to use.

1. Add a `using XsharpToDo` so the window can find our classes
2. Add a property to the class :
   ```
   XToDos oTasks = new XToDos();
   ```
3. Load the data into our object when we instantiate the window:
   ```
   this.oTasks.Load();
   ```
4. Change our commands to call our business object methods:
   ```
   this.oTasks.New(""); // in the NewCommand_Executed button
   this.oTasks.Print(); // in the PrintCommand_Executed button
   ```

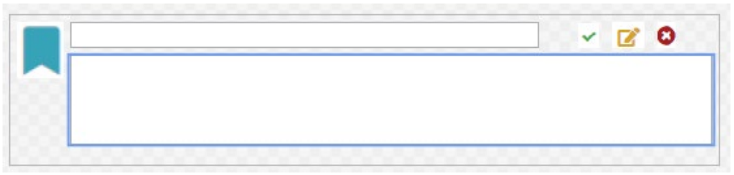
I've highlighted the extra code in **Code Listing 1: The Code Behind for our Window** . Now clicking on the New Task button inserts a blank record into the DBF via the business objects!

```
using XSharpToDo;
using XSharp.VFP;

namespace ToDoInterface
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        XToDos oTasks = new XToDos();

        public MainWindow()
        {
            InitializeComponent();
            this.DataContext = this;
            this.oTasks.Load();
        }

        private void NewCommand_CanExecute(object sender, CanExecuteRoutedEventArgs e)
        {
            e.CanExecute = true;
        }
        private void NewCommand_Executed(object sender, ExecutedRoutedEventArgs e)
        {
            // MessageBox.Show("New Task");
            this.oTasks.New("");
        }
        private void PrintCommand_CanExecute(object sender, CanExecuteRoutedEventArgs e)
        {
            e.CanExecute = true;
        }
        private void PrintCommand_Executed(object sender, ExecutedRoutedEventArgs e)
        {
            // MessageBox.Show("Print");
            oTasks.Print();
        }
    }
}
```

Code Listing 1: The Code Behind for our Window

Creating our Custom Control

In order to emulate our FoxPro form, we'll need to create a "Custom Control" in XAML that we'll put in our grid, much like we did in our original VCX. This sort of control is nearly impossible in straight WinForms, but isn't too bad to create in XAML. Here's what it looks like, and here's the code in **Code Listing 2: Our Custom Control in XAML**:



```
<UserControl x:Class="ToDoInterface2.cntToDo"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             xmlns:local="clr-namespace:ToDoInterface2"
             mc:Ignorable="d"
             d:DesignHeight="114" d:DesignWidth="544">
    <DockPanel>
        <Image Source="Images\Ribbon.png" Height="43px" DockPanel.Dock="Left"
Name="imgRibbon" VerticalAlignment="Top" Margin="5 5 0 0" ></Image>
        <WrapPanel Orientation="Horizontal" DockPanel.Dock="Top">
            <TextBox Height="20px" Width="360px" Margin="5 5 25 5"
Name="txtTitle"></TextBox>
            <Image Source="Images\CheckMark.png" Margin="5" Height="20px"
DockPanel.Dock="Top" Name="imgComplete" ></Image>
            <Image Source="Images\Edit.png" Margin="5" Height="20px" DockPanel.Dock="Top"
Name="imgEdit"></Image>
            <Image Source="Images\Delete.png" Margin="5" Height="20px" DockPanel.Dock="Top"
Name="imgDelete" ></Image>
        </WrapPanel>
        <TextBox Margin="5,0,5,15" FontSize="14" Name="edtDescription"/>
    </DockPanel>
</UserControl>
```

Code Listing 2: Our Custom Control in XAML

Binding Data to Forms

What we need to do is add our custom control into an appropriate grid control and add the bindings. This whole topic is enough fodder for an entire session, and so I recommend you find Robert van der Hulst's session materials on Databinding in X# and take it from there. He gives the deep dive that this topic deserves, and in an X# context.

The last thing we need to do to make this application run like our VFP application is to wire the buttons on the custom control to command events. This is done the same way we wired the Print and Add button to the window itself, but we just need to pass along the ID of the task that's clicked on, and then call the appropriate method in our business objects, sending along that ID to either complete or delete the selected task.

Reports

I'm not aware of any tool that will convert FRX files to a .NET equivalent, though the X# developers do have that on their wish list. There is a paid product called ReportPro 3 that has a report designer that looks very similar to FoxPro's. It has banding, grouping, summarizing, etc. While it does support DBF files, it does not import FRX files yet.

If your data is in a database like SQL Server, MySQL, or really anything except DBF, then there is a plethora of reporting tools available to you, such as Stonefield Reporting, Telerik Reports or the open source FastReport [not an endorsement of either one].

For outputting DBF data, I did see chatter on the X# forums about other attempts to work with FRX reports. One idea was to convert FoxyPreviewer, which is written in FoxPro, to X# itself. Another idea was to invoke FoxPro's native runtime from within X# itself. Either way would require you to retain VFP to design or modify the reports. It turns out FoxPro was (is) powerful and hard to emulate!

Other Development Considerations

Databases

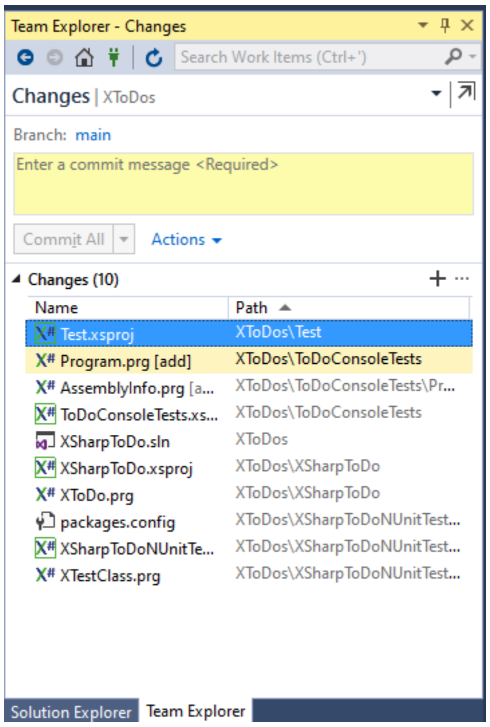
Besides the familiarity of the FoxPro-like syntax, the other compelling reason to choose X# as a development tool is its ability to use your existing DBF files, as shown. But X# can use a myriad of other databases, from SQLite to Oracle. A very popular backend, based on my perusal of the X# forums, is SQLAnywhere from Sybase. Anything that .NET can access through its System.Data library, which is vast and powerful, is usable from X#.

Frameworks?

In FoxPro we had a stable of application frameworks that we could choose from to give us a head start when creating programs from scratch. Products like Mere Mortals, ProMatrix, and CodeBook were often used. There are currently no application frameworks for X# based on the FoxPro dialect, but many C# applications frameworks like Oak Leaf's MM.Net can be combined with X# business objects easily enough to get you further down the road.

Source Control

One of the best features when working with a modern IDE like Visual Studio is its integrated source control. Many developers put in a lot of work to create utilities that would properly serialize/deserialize FoxPro's binaries, and then added those tools to Thor to create as little friction as possible to manage source control, but nothing works as smoothly as the Team Explorer window. From here you can add, stage, commit to a local repository, and push to and pull from a remote repository. [Read Rick Borup's whitepapers on Git for more information on what all that means if you need help.] Because there are no binaries in the code, there's no need to serialize anything.



Name	Path
Test.xsproj	XToDos\Test
Program.prg [add]	XToDos\ToDoConsoleTests
AssemblyInfo.prg [a...	XToDos\ToDoConsoleTests\Pr...
ToDoConsoleTests.xs...	XToDos\ToDoConsoleTests
XSharpToDo.sln	XToDos
XSharpToDo.xsproj	XToDos\XSharpToDo
XToDo.prg	XToDos\XSharpToDo
packages.config	XToDos\XSharpToDoNUnitTest...
XSharpToDoNUnitTe...	XToDos\XSharpToDoNUnitTest...
XTestClass.prg	XToDos\XSharpToDoNUnitTest...

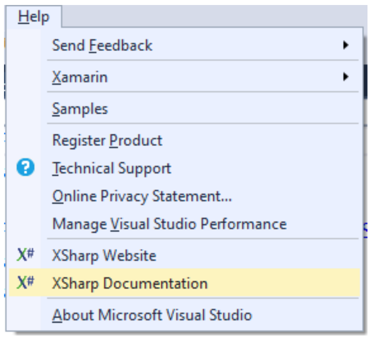
External Libraries and IDE Add-Ons

If you like Thor and (and as a FoxPro dev, you should), you're going to love Visual Studio's equivalents. Thor is a visual manager that extends FoxPro's IDE with utilities like GoFish5, Project Manager, PEMEditor, and FoxUnit. Visual Studio has the Visual Studio Marketplace (https://marketplace.visualstudio.com/) that has hundreds of extensions for Visual Studio.

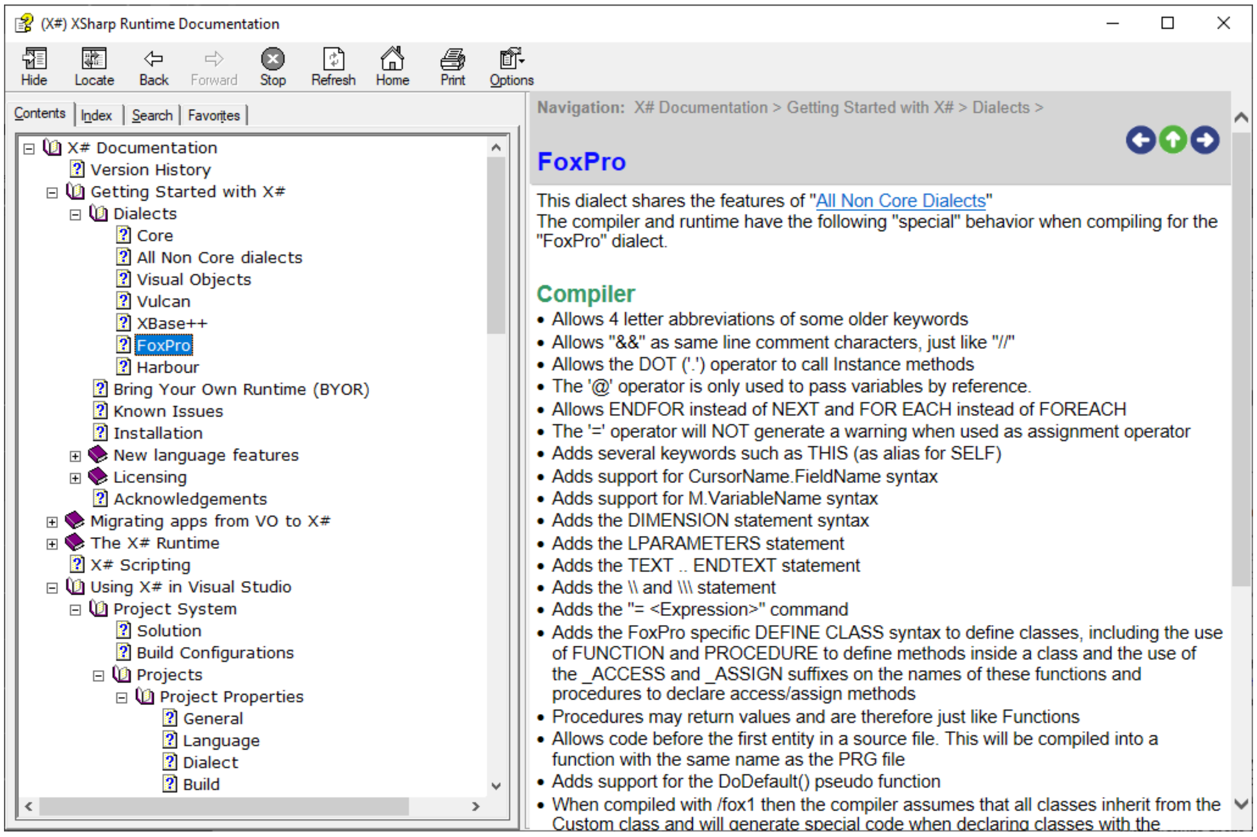
VFPX is the central source code repository for many of FoxPro's IDE utilities and additional class libraries, like FoxCharts and Log4VFP that you can use to embellish your applications. Nuget is the package manager for Visual Studio. It has thousands of class libraries you can take advantage of, and most of them are free. Everything from Payment Gateways to Entity Frameworks to entire Application Frameworks.

Getting Help

If you need assistance while you're learning X#, the best place to start is www.xsharp.info. You can easily get there by choosing Help, Xsharp Website from the Visual Studio menu. There are active user forums on there, and a specific one just for FoxPro migrators. The core group of developers jump in quickly to answer questions.



The Help File that comes with the X# download is also quite good. It will remind you of Visual FoxPro's help file. You can find it either on your Start Menu or choose Help, XSharp Documentation in Visual Studio.



X# Documentation
Version History
Getting Started with X#
Dialects
Core
All Non Core dialects
Visual Objects
Vulcan
XBase++
FoxPro
Harbour
Bring Your Own Runtime (BYOR)
Known Issues
Installation
New language features
Licensing
Acknowledgements
Migrating apps from VO to X#
The X# Runtime
X# Scripting
Using X# in Visual Studio
Project System
Solution
Build Configurations
Projects
Project Properties
General
Language
Dialect
Build

Navigation: X# Documentation > Getting Started with X# > Dialects >

FoxPro

This dialect shares the features of "All Non Core Dialects"
The compiler and runtime have the following "special" behavior when compiling for the "FoxPro" dialect.

Compiler

- Allows 4 letter abbreviations of some older keywords
- Allows "&&" as same line comment characters, just like "//"
- Allows the DOT ('.') operator to call Instance methods
- The '@' operator is only used to pass variables by reference.
- Allows ENDFOR instead of NEXT and FOR EACH instead of FOREACH
- The '=' operator will NOT generate a warning when used as assignment operator
- Adds several keywords such as THIS (as alias for SELF)
- Adds support for CursorName.FieldName syntax
- Adds support for M.VariableName syntax
- Adds the DIMENSION statement syntax
- Adds the LPARAMETERS statement
- Adds the TEXT .. ENDTEXT statement
- Adds the \\ and \\\ statement
- Adds the "= <Expression>" command
- Adds the FoxPro specific DEFINE CLASS syntax to define classes, including the use of FUNCTION and PROCEDURE to define methods inside a class and the use of the _ACCESS and _ASSIGN suffixes on the names of these functions and procedures to declare access/assign methods
- Procedures may return values and are therefore just like Functions
- Allows code before the first entity in a source file. This will be compiled into a function with the same name as the PRG file
- Adds support for the DoDefault() pseudo function
- When compiled with /fox1 then the compiler assumes that all classes inherit from the Custom class and will generate special code when declaring classes with the

Conclusion

This session barely covers all of the power and awesomeness of Visual Studio and coding in X# and the .NET Framework. I hope this walkthrough of converting a simple FoxPro app to X# has been enlightening. In my limited experience I think X# provides a very nice entré into understanding .NET development. I also think it's important to get that, though X# gives you access to FoxPro-like syntax and concepts that give you an anchor into your development history, you're not constrained to that – the entire .NET framework is available to you.

X# is never going to be able to take your existing FoxPro code and just compile it – it will take effort on your part. But it's really not that difficult and it will give you the opportunity to revisit and refactor your code, as well as add robustness through unit tests and integrated version control.

The FoxPro compatibility has made impressive strides since I wrote the 2019 session. X# is open source, and while other developers in our community are contributing, the focus of the core of developers will be the wish list of the people who pay to be members of Friends of X#. Hey, they have to pay the bills! If you'd like to see development focused on the FoxPro compatibility features, you will want to support them with a membership.

Credits and Bibliography

Twitter icon from Icons made by Smashicons from www.flaticon.com

https://docs.microsoft.com/en-us/windows/apps/desktop/visual-studio-templates
https://fox.wikis.com/wc.dll?Wiki~GUIDGenerationCode~VB
https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/fields
https://www.youtube.com/watch?v=CniIPEFZ1Oo (Data bindings)
https://www.xsharp.info/itm-help/foxpro-compatibility-list
https://www.wpf-tutorial.com

For an interesting philosophical discussion on the future of Fox-based development in general: https://support.west-wind.com/Thread5U70W2EQW.wwt

Appendix A: The Original FoxPro Classes

```
* Collection of ToDos
DEFINE CLASS Todos AS Custom
  DIMENSION aToDos[1]    && Array of ToDo
Objects
    nTodos = 0
    cTableName = "data\ToDos"


    PROCEDURE Init
    SET EXCLUSIVE OFF

    PROCEDURE OpenTodos
    IF NOT USED(This.cTableName)
      USE (This.cTableName)
    ENDIF
    RETURN USED(This.cTableName)

    PROCEDURE Load
    LOCAL n
    This.OpenTodos()
    SET DELETED ON
    COUNT TO This.nToDos
    DIMENSION This.aToDos[This.nToDos]
    n = 1
    SCAN
      This.aToDos[n]=CREATEOBJECT("Todo",
id)
      n = n + 1
    ENDSCAN
    This.CloseTodos()
    RETURN This.nToDos

    PROCEDURE New
    This.nTodos = This.nTodos + 1
    DIMENSION This.aToDos[This.nToDos]
    This.aTodos[This.nTodos] =
CREATEOBJECT("Todo")
    This.aTodos[This.nTodos].Save()
    RETURN This.nToDos

    PROCEDURE CloseToDos
    LPARAMETERS lLeaveOpen
    IF NOT lLeaveOpen
       USE IN SELECT("ToDos")
    ENDIF

    PROCEDURE Complete
    oToDo =CREATEOBJECT("ToDo", ToDos.id)
    oToDo.oData.Completed=.t.
    RETURN oToDo.Save()

    PROCEDURE Delete
    oToDo =CREATEOBJECT("ToDo", ToDos.id)
    RETURN oToDo.Delete()
ENDDEFINE
```

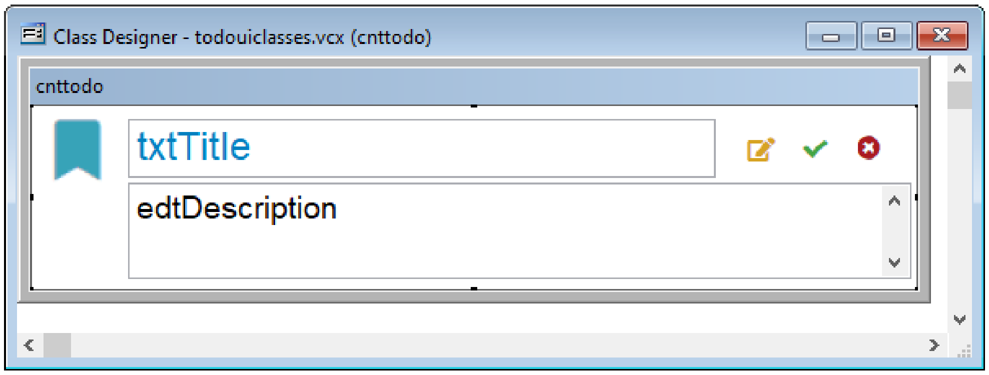
```
* Individual ToDo
DEFINE CLASS ToDo AS Custom
    Name = "ToDo"
    cId = ""
    oData = .null.
    lNew = .f.
    lSaved = .f.
    lLoaded = .f.
    oException = .null.


    PROCEDURE Init
    LPARAMETERS cId
    This.cId = cId
    IF EMPTY(cId)
       This.New()
    ELSE
       This.Load(This.cId)
    ENDIF
    ENDPROC


    PROCEDURE New
    lUsed = This.OpenToDos()
    SCATTER BLANK NAME This.oData MEMO
    This.lNew = .t.
    This.CloseTodos(lUsed)
    RETURN This.oData


    PROCEDURE Load
    LPARAMETERS cId
    LOCAL lUsed
    cId = EVL(cId,This.cId)
    IF NOT EMPTY(cId)
       TRY
          lUsed = This.OpenToDos()
          LOCATE FOR id = cId
          IF FOUND()
             SCATTER NAME This.oData
MEMO
             This.cId = cId
             This.lLoaded = .t.
             This.lNew = .f.
          ENDIF
       CATCH TO oEx
          This.oException = oEx
       FINALLY
          This.CloseTodos(lUsed)
       ENDTRY
    ENDIF
    RETURN This.lLoaded


    PROCEDURE Save
    LOCAL lUsed
    This.lSaved = .F.
    IF This.lLoaded OR This.lNew
      lUsed = This.OpenToDos()
      TRY
       IF This.lNew
* There are many ways to create a GUID,
including calls to CoCreateGUID in
Ole32.dll, but this is easy. From
https://fox.wikis.com/wc.dll?Wiki~GUIDGen
erationCode~VB
            LOCAL oGUID
            oGUID =
CreateObject("scriptlet.typelib")
            This.oData.Id =
Strextract(oGUID.GUID, "{", "}" )
            This.oData.Entered = DATETIME()
            INSERT INTO ToDos FROM NAME
This.oData
            This.cId = This.oData.Id
        ELSE
            LOCATE FOR id = This.cId
            GATHER NAME This.oData MEMO
        ENDIF
        This.lSaved = .t.
        This.lNew = .f.
    CATCH TO oEx
        This.oException = oEx
    FINALLY
        This.CloseTodos(lUsed)
    ENDTRY
    ENDIF
```

```
    RETURN This.lSaved

    PROCEDURE Delete
    LOCAL lUsed, lReturn
    IF NOT EMPTY(This.cId)
        lUsed = This.OpenToDos()
        LOCATE FOR id = This.cId
        lReturn = FOUND()
        IF lReturn
            DELETE
        ENDIF
        This.CloseTodos(lUsed)
    ENDIF
    RETURN lReturn

    PROCEDURE OpenTodos
    LOCAL lUsed
    lUsed = USED("ToDos")
    IF NOT lUsed
        USE data\ToDos IN 0
    ENDIF
    SELECT ToDos
    RETURN lUsed

    PROCEDURE CloseToDos
    LPARAMETERS lLeaveOpen
    IF NOT lLeaveOpen
        USE IN SELECT("ToDos")
    ENDIF

ENDDEFINE && ToDo
```

Go to https://github.com/eselje/FoxToDos for complete and up-to-date source code

cntToDo Visual Class – in Code

```
***********************************
*-- Class:            cntToDo
*-- ParentClass:      container
*-- BaseClass:        container
*
DEFINE CLASS cntToDo AS container


   Width = 544
   Height = 114
   BackColor = RGB(255,255,255)
   Name = "cnttodo"


ADD OBJECT imgTask AS image WITH ;
      Picture =
"..\images\ribbon.png", ;
      Height = 43, ;
      Left = 10, ;
      Top = 6, ;
      Width = 37, ;
      Name = "imgTask"


ADD OBJECT txtTitle AS textbox WITH
;
      FontSize = 18, ;
      Height = 36, ;
      Left = 60, ;
      Top = 9, ;
      Width = 360, ;
      ForeColor = RGB(0,128,192), ;
      Name = "txtTitle"


ADD OBJECT imgCompleted AS image
WITH ;
      Picture =
"..\images\checkmark.png", ;
      Height = 28, ;
      Left = 469, ;
      Top = 12, ;
      Width = 23, ;
      Name = "imgCompleted"


ADD OBJECT imgDelete AS image WITH;
      Picture =
"..\images\delete.png",;
      Height = 20, ;
      Left = 507, ;
      Top = 16, ;
      Width = 14, ;
      Name = "imgDelete"


ADD OBJECT imgEdit AS image WITH ;
      Picture =
"..\images\edit.png", ;
      Height = 20, ;
      Left = 435, ;
      Top = 16, ;
      Visible = .F., ;
      Width = 25, ;
      Name = "imgEdit"


ADD OBJECT edtDescription AS
editbox WITH ;
      FontSize = 14, ;
      Height = 59, ;
      Left = 60, ;
      Top = 48, ;
      Width = 480, ;
      ControlSource = "descript", ;
      NullDisplay = "", ;
      Name = "edtDescription"


ENDDEFINE
*
*-- EndDefine: cnttodo
***********************************
```



Appendix B: The X# Classes

```
// XToDos.prg – the collection of Tasks

USING System.Collections.Generic
USING System.Collections.ObjectModel
USING System.Linq
USING System.Text

BEGIN NAMESPACE XSharpToDo

    DEFINE CLASS XToDos AS Custom
    cTableName AS String
    cAlias AS String
    cLastId AS String
    nTodos AS Int
    aToDos AS ObservableCollection<XToDo>

    PROCEDURE Init()
        SET EXCLUSIVE OFF
        This.cTableName = "C:\DEV\XToDos\ToDos.dbf"
        This.cAlias = "ToDos"
        This.nToDos = 0
        This.aToDos = ObservableCollection<XToDo>{}
        return
    end function

    public FUNCTION openToDos() AS Boolean
        LOCAL cTableName
        cTableName = This.cTableName
        IF NOT used("TODOS")
            SELECT 0
            USE (cTableName) SHARED
        ENDIF
        return used("TODOS")
    END FUNCTION

    public FUNCTION closeToDos() AS Boolean
        USE IN (CoreDb.SymSelect("ToDos"))
        return not used("Todos")
    END FUNCTION

    PUBLIC FUNCTION getToDo(cId as string) as XToDo
        VAR oToDo = createObject("XToDo")
        oToDo.Load(cId)
    return oToDo

    public FUNCTION Load() as Int
        LOCAL nToDos
        LOCAL oToDo AS XToDo
        This.OpenTodos()
        SET DELETED ON
        COUNT TO nToDos
        This.aToDos.Clear()
        SCAN
            oTodo = CreateObject("XToDo", ToDos.id)
            This.aToDos.Add(oToDo)
        ENDSCAN
        This.CloseTodos()
        This.nToDos = nToDos
    RETURN This.nToDos

    public Function New(cTitle AS String)  AS Int
        This.nTodos = This.nTodos + 1
        VAR  oTodo = CreateObject("XToDo")   // No ID
        oToDo.New(cTitle)
        oToDo.Save()
        This.cLastId = oToDo.Id
        aToDos.Add(oToDo)
        RETURN This.nToDos

    public function toggleCompleted(oTask as XToDo) as boolean
        oTask.completed = ! oTask.completed
        oTask.SAVE()
        This.Load()
        return oTask.completed

     public function deleteTask(oTask as XToDo) as boolean
        LOCAL lDeleted AS boolean
        lDeleted = oTask.Delete()
        This.Load()
        return lDeleted

    public Function GetLast() AS XToDo
        VAR  oTodo = CreateObject("XToDo")   // No ID
        LOCAL cId AS String
        IF EMPTY(This.cLastId)
            * Go to the bottom of the ToDos table and get that ID
            This.openToDos()
            SET ORDER TO
            SET DELETED ON
            GO BOTTOM
            cId = ToDos.id
        ELSE
            cId = This.cLastId
        ENDIF
        oTodo.Load(cId)
        RETURN oToDo
    END DEFINE


END NAMESPACE

// XToDo.prg


USING System
USING System.Collections.Generic
USING System.Text

BEGIN NAMESPACE XSharpToDo

    DEFINE CLASS XToDo as Custom
    id = ""
    title = ""
    descript = ""
```

```
entered = DateTime.Now
completed = .f.
isEditing = .f.
isNew = .f.

public FUNCTION Constructor() // No Parameter. New Task.
This.new()

public FUNCTION Constructor(cId AS String) // Parameter. Existing Task
    This.id = cId
This.load(cId)

public PROCEDURE clear
    This.id = ""
    This.title = ""
    This.descript = ""
    This.completed = False
end procedure

public function reload() AS Boolean
RETURN This.load(This.id)

public FUNCTION load(cId AS String) AS Boolean
    var lReturn = False
    This.openToDos()
    SET EXACT OFF
    GO TOP
    LOCATE FOR ToDos.id = cId
    lReturn = FOUND()

    IF lReturn
            This.id = ToDos.id
            This.title = ToDos.title
            This.descript = ToDos.descript
            This.entered = Todos.entered
        This.completed = ToDos.completed
    else
        This.clear()
    ENDIF
    This.CloseToDos()
    return lReturn
END FUNCTION

PUBLIC FUNCTION SAVE() AS Boolean
    LOCAL lReturn AS Boolean
    LOCAL lSaved AS Boolean
    LOCAL cId AS String
    LOCAL cDescript AS String
    LOCAL cTitle AS String
    LOCAL lCompleted AS Boolean
    cId = This.id
    TRY
            This.openToDos()
            LOCATE FOR ToDos.ID = cId
            lReturn = FOUND()
            IF NOT lReturn
                // INSERT INTO TODOS (ID) VALUES (CNEWID)
                APPEND BLANK
```

```
                    REPLACE id WITH cId, isEditing WITH .t., Entered WITH DateTime.NOW
                ENDIF
                cTitle = This.title
                cDescript = This.descript
                lCompleted = This.completed
                IF RLOCK()
                    REPLACE Title WITH cTitle, ;
                    Descript WITH cDescript, ;
                    Completed WITH lCompleted, ;
                    isEditing WITH .f.
                    UNLOCK
                ENDIF
                This.isNew = .f.
            lSaved = .t.
            CATCH
        FINALLY
            This.closeToDos()
        END TRY
        RETURN lSaved
    END FUNCTION

    PUBLIC FUNCTION New()
        This.New("")
    RETURN This.id

    PUBLIC FUNCTION New(cTitle AS STRING)
        LOCAL lUsed
        This.Clear()
        VAR g = GUID.NEWGUID()
        This.id = g.TOSTRING()
        this.entered = DateTime.Now
        this.isEditing = true
        This.isNew = .t.
        This.title = cTitle
    RETURN This.id

    PUBLIC FUNCTION MarkCompleted(lComplete as Boolean) AS Boolean
        LOCAL lReturn as boolean
        This.completed = lComplete
        lReturn = This.SAVE()
    RETURN lReturn

    PUBLIC FUNCTION Delete AS Boolean
        LOCAL lReturn as boolean
        LOCAL cId as string
        cId = This.id
        This.openToDos()
        LOCATE FOR ToDos.ID = cId
        IF FOUND() AND RLOCK()
            DELETE
            UNLOCK
            lReturn = deleted()
        ENDIF
    RETURN lReturn

    public FUNCTION openToDos() AS Boolean
        //         XToDos:openToDos()
    IF NOT USED("TODOS")
```

```
        SET DEFA TO "C:\DEV\XTODOS"
            SELECT 0
            USE "TODOS" ALIAS "ToDos" SHARED
        ENDIF
        return used("TODOS")
    END FUNCTION

    public FUNCTION closeToDos() AS Boolean
        USE IN (CoreDb.SymSelect("ToDos"))
        return not used("Todos")
        END FUNCTION
    end define

END NAMESPACE // XSharpToDo
```