FREE eBook

LEARNING visual-foxpro

Free unaffiliated eBook created from
Stack Overflow contributors.

#visual-foxpro

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: visual-foxpro

It is an unofficial and free visual-foxpro ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official visual-foxpro.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with visual-foxpro

Remarks

Foxpro was created in early 80's (originally as FoxBase - 1984?) by Fox software and supported on Mac OS, Unix, DOS, Windows platforms. It was known as the fastest database engine on PCs then. Later on 1992, *unfortunately*, it was acquired by Microsoft. After Microsoft's taking over, in 1994 Foxpro for DOS (FPD) and Foxpro for Windows (FPW) 2.6 released. At the end of 1995, Foxpro got the name "Visual" and the platform support was only limited to windows. It was also the first version of Foxpro where the language turned out to be Object Oriented.

Microsoft's official Visual Foxpro (commonly referred as just VFP) site describes it as:

> Microsoft® Visual FoxPro® database development system is a powerful tool for quickly creating high-performance desktop, rich client, distributed client, client/server, and Web database applications.

Although it is an old language, it is still considered to be the easiest language for creating a data centric application rapidly for the windows desktop. **If** what you need is to create ***a data based application for windows desktop*** then choosing VFP you would really do that easily and fast.

Versions

Version	Released
FPW 2.6a	1994-10-28
Visual Foxpro 3.0	1995-12-16
Visual Foxpro 5.0	1997-01-24
Visual Foxpro 6.0	2000-08-18
Visual Foxpro 7.0	2002-01-04
Visual Foxpro 8.0	2003-10-25
Visual Foxpro 9.0	2004-12-13
Visual Foxpro 9.0 SP2	2007-10-21

Examples

Installation or Setup

Detailed instructions on getting visual-foxpro set up or installed.

Hello World

In all languages traditionally the first example is printing "Hello World". Probably doing that is easiest in VFP:

```
? "Hello World"
```

Add Global Error Handler

A simple way to catch unhandled errors (exceptions) in a VFP application is to use the ON ERROR command near the beginning of your main program.

The following ON ERROR command calls a method in the current program called "errorHandler". The values returned by ERROR (the VFP Error Number), MESSAGE (the VFP Error Message), PROGRAM (name of the currently executing program) and LINENO (the line number of the error) are passed to the errorHandler method.

```
ON ERROR DO errorHandler WITH ERROR(), MESSAGE(), PROGRAM(), LINENO()
```

A simple errorHandler method may look like the following.

```
PROCEDURE errorHandler
    LPARAMETERS tnVFPErrorNumber, tcVFPErrorMessage, tcProcWithError, tnLineNumber

    STORE 'Error message: ' + tcVFPErrorMessage + CHR(13) + ;
        'Error number: ' + TRANSFORM(tnVFPErrorNumber) + CHR(13) + ;
        'Procedure with error: ' + tcProcWithError + CHR(13) + ;
        'Line number of error: ' + TRANSFORM(tnLineNumber) TO lcDetails

    MESSAGEBOX(lcDetails, 16, "Unhandled Exception")

    ON ERROR *
    ON SHUTDOWN
    CLEAR EVENTS

    QUIT
ENDPROC
```

You can also change and restore the error handler in between. For example, at one point you want to open all tables in a folder exclusively, and if you can't you don't want to continue:

```
procedure DoSomethingWithExclusiveLock(tcFolder)
local lcOldError, llInUse, ix && by default these variables have a value of .F.
lcError = on('error') && save current handler
on error llInUse = .T.  && new handler
local array laTables[1]
for ix=1 to adir(laTables, addbs(m.tcFolder) + '*.dbf'))
   use (addbs(m.tcFolder)+laTables[m.ix,1]) in 0 exclusive
endfor
on error &lcError && restore old handler
```

```
if m.llInUse && couldn't get exclusive lock on all tables
   close databases all
   return
endif
* do whatever
endproc
```

Tip: Sometimes, especially during debugging, you would want to restore default error handler which allows you to break and step into the code where the error has occurred, then anywhere before where you got the error, temporarily adding:

```
on error
```

would do this.

Read Getting started with visual-foxpro online: https://riptutorial.com/visual-foxpro/topic/7391/getting-started-with-visual-foxpro

Chapter 2: Operators

Remarks

In VFP, operators are grouped into those:

- Numeric Operators
- Logical Operators
- Character Operators
- Date and Time Operators
- Relational Operators

Also there are operators, implemented as functions (such as bitwise operations, object comparison ...).

We will look into each by example.

Examples

Numeric Operators

Numeric operators are the easiest and almost the same as in other languages.

- +, -, * and /. Addition, subtraction, multiplication and division operators (in VFP there is no integer division, you can convert a result to integer with functions INT(), CEILING() and FLOOR()).
- % Modulus operator.
- ^ and **. Power of operator(s). They both do the same thing.
- (). Grouping operators.
- Operators have precedence. The order is:

```
( )
^ (or **)
/ and *
- and +
```

```
? 10 / 5 + 2 && Outputs 4
? 2 + 10 / 5 && Outputs 4 as well. Division has precedence.

* Both multiplication and division have same precedence
* They would be interpreted from left to right.
? 4 * 5 / 2 + 5 && Outputs 15
* Use parentheses whenever you are in doubt or want to be explicit
? ( (4 * 5) / 2 ) + 5 && Outputs 15. Explicit grouping of operations

? 4 * 5^2 && ^ has precedence, this is same as 4 * (5^2) = 100.
? (4 + 5)^2 && Using parentheses we say add 5 to 4 (9) and then square = 81.
```

Logical operators

Logical operators in VFP, in their order of precedence are:

Operator	Description
( )	Parentheses, groups expressions
NOT, !	Logically negate the expression. NOT or ! has no difference.
AND	Logically AND the expressions
OR	Logically OR the expressions
<>, !=, #	Check for inequality. Thus same as logical exlusive OR - XOR

> Historically, NOT, AND, OR are written as .NOT., .AND., .OR. You can still use them if you like to, but AND, OR, NOT is simpler and cleaner.
>
> For false and true, you have to use .F. and .T. literals respectively. You cannot choose to use F and T instead.

```
* Some logical variables
local llOld, llEmail  && any variable declaration implicitly initializes the variable as .F. - false
? m.llOld, m.llEmail && Prints .F. .F.

llOld   = .T.
llEmail = .F.

if ( m.llOld AND m.llEmail )
   ? 'Old AND should be emailed to'
endif
if ( m.llOld OR m.llEmail )
   ? 'Old OR should be emailed to'
endif
if ( m.llOld AND !m.llEmail ) && Same as (m.llOld AND NOT m.llEmail)
   ? 'Old BUT should NOT be emailed to'
endif

* Above code outputs
Old OR should be emailed to
Old BUT should NOT be emailed to
```

> In VFP, logical expresssions are evaluated in a shortcut fashion. That is, if the first part of the check satisfies the whole result, rest of the expression is **not** even interpreted. A sample follows:

```
? 1 = '2' && An obvious error. It would complain operator/operand type mismatch.

* However we could use such an expression in an if and get no error
* because it is not interpreted at all
* (VFP is dynamic and there is no compile time check)
```

```
local llProcess
llProcess = .T.

if (m.llProcess OR (1='2'))
   ? 'Should do processing'
endif

* Would output

Should do processing

* without any error because m.llProcess true means
* the whole expression would be true, thus the expression after OR
* is not interpreted at all.
```

One pitfall that catches newbies is that, sometimes you might need multiple checks, say in an SQL query, which are connected with AND, OR operators. When there are many of them, one might ignore the fact that operators have a precedence (in order (), NOT, AND, OR) and think the interpretation would be done left to right in a chain. Consider a sample:

```
select * from myTable where !isCustomer AND debit > 5000 OR discount > 5
```

what is the intention of this query? If we make it explicit using grouping parentheses it says:

```
((NOT isCustomer) AND debit > 5000) OR discount > 5
```

simplified it looks like "firstExpression" OR (discount > 5). Whatever the intention was, because of this OR it would select:

> all the rows that have (discount > 5) - and also those where it is a customer with over 5000 debit.

Probably the intention was "give me those where it is NOT a customer AND (debit is over 5000 OR discount is over 5)". It would be clear from the start if we used parentheses:

```
select * from myTable where !isCustomer AND (debit > 5000 OR discount >
5)
```

You may use but not worth to have parentheses for the initial NOT operator, when its operand is a single expression its readable enough with its precedence - !isCustomer is clearly read as (NOT isCustomer).

Character operators

There are only 4 character operators, in their order of precedence:

Operator	Description
()	Parentheses for grouping. Note: VFP documentation, that I have, misses this one. Without this, - operator is almost always useless.
+	Concatenates (joins) strings side by side.
-	Concatenates strings by **moving** the trailing spaces from left string to the right string's end.
$	Checks if first string is contained in second.

+ is the simpliest and also used to concatenate strings in many other languages.

```
local firstName, lastName
firstName = "John"
lastName  = "Smith"

? m.firstName + " " + m.lastName
```

Outputs: John Smith

- is a little tricky, and not widely known. It takes trailing spaces from the left string, appends those spaces to the string on the right. Suppose you have a table with first and last names and each of them is 20 characters. We want to concatenate first and last names to create a full name, **and** we also want the resulting size be fixed (in this case 20 + 20 + 1 space = 41). Let's make it you also have a middle name column and we want the full name look like "lastName, firstName middleName_____". It is easiest to do this using - operator but you should note the trick of using parentheses here for grouping so we get exactly what we want to:

```
* Create a cursor for our sample and fill in a few names
Create Cursor Names (firstName c(20), midName c(20), lastName c(20))

Insert Into Names (firstName, midName, lastName) Values ('Cetin','', 'Basoz')
Insert Into Names (firstName, midName, lastName) Values ('John', 'M', 'Smith')
Insert Into Names (firstName, midName, lastName) Values ('John', 'F', 'Kennedy')
Insert Into Names (firstName, midName, lastName) Values ('Tom', '', 'Hanks')

* Select with tricky - operator
Select *, ;
    lastName - (', '+firstName-(' '+midName)) As FullName ;
    from Names ;
    INTO Cursor crsNames ;
    nofilter

Browse
```

And the ouput is like this:

firstName	midName	lastName	fullName
Cetin		Basoz	Basoz, Cetin

firstName	midName	lastName	fullName
John	M	Smith	Smith, John M
John	F	Kennedy	Kennedy, John F
Tom		Hanks	Hanks, Tom

In fullName column all the trailing spaces are pushed to the end nicely. If you check the structure fullName column is 63 characters wide (3 * 20 + 3 characters that we added).

> Note the importance of grouping parentheses (try removing parentheses or arranging in a different way).

Although - operator might be tempting to use in such cases, there is other side of the coin. This operator is VFP specific and thus the SQL is not portable. You could achieve the same result by this ANSI compatible SQL instead:

```
Select *, ;
    CAST(RTRIM(lastName) +', '+ RTRIM(firstName) +' '+ midName as char(63)) As FullName ;
    from Names ;
    INTO Cursor crsNames ;
    nofilter
```

Last operator is $. It simply checks if left string is part of right string.

```
local upcased, digits, hexDigits
upcased = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
digits  = '0123456789'
hexDigits  = m.digits + 'ABCDEF'

? 'A' $ m.upcased && .T.
? 'a' $ m.upcased && .F.
? '1' $ m.digits && .T.
? 'F' $ m.digits && .F.
? 'F' $ m.hexDigits && .T.
```

> **Important:** In VFP, although you can write your code in any case (upper, lower or mixed), strings are always case sensitive. For example: "Smith" and "smith" are two distinct values. Or in your table if there is a country column, you wouldn't find 'USA' if you search it with 'usa'. Same goes with $ operator, "GE" $ "Germany" is false.
>
> Personal note: Although you might like $ for its simplicity and you may find it often used in Microsoft source codes, IMHO it is of very little value. Thinking about many many thousands of lines I have written in my carrier, I think I would find very few occurences of it in my own code. Almost always there is a better alternative (especially when left operand is not a single character and\or case sensitivity matters).

Date and Time operators

There are basically two operators for date, datetime values. + and - are overloaded (probably a C

term) to do date/datetime math:

Operator	Description
+	Adds days (date) or seconds (datetime) to a date/datetime value.
-	Gets the difference of two date/datetime values. Subtracts days (date) or seconds (datetime) from datetime values.

+ is the easier one. It has two operands, one is a date or datetime value and the other is a numeric (although you might use any numeric, it is an integer for all practical purposes).

When one of the operands is a date, then numeric operand is taken as "day":

```
? Date() + 10 && Get the date 10 days later
* VFP is leap year aware when doing date math
? Date(2016, 2, 28) + 1 && Add 1 day to Feb 28, 2016. Returns Feb 29, 2016.
? Date(2017, 2, 28) + 1 && Add 1 day to Feb 28, 2017. Returns Mar 1, 2017.
```

When one of the operands is a datetime, then numeric operand is taken as "second":

> There are 24 * 60 * 60 = 86400 seconds in a day

```
? Datetime() + 86400 && Add 1 day to current datetime.
```

Add 4 hours and 15 minutes to Jan 1, 2016 2:20 PM

```
? Datetime(2016, 1, 1, 14, 20, 0) + (4 * 3600 + 15 * 60)
```

Outputs Friday, January 1, 2016, 6:35:00 PM.

> With a simple printing using ?, what you see on your screen is dependant on your date settings. For example, if you haven't changed anything your date settings are American style (MDY), 12 hours format (AM/PM) and century is shown with last 2 digits only.
>
> > There is one special symbol ^ for date and datetimes forcing a string to be interpreted 'strictly' as yyyy/MM/dd [HH:mm:ss|hh:mm:ss tt] format. Therefore ^ might be considered as date/datetime operator as well. For example consider some data is incoming from a source in a format like 201610082230 (yyyyMMddHHmm). To get that value as a valid Datetime:

```
Local cSample, tSample
cSample = '201610082230'
tSample = Ctot(Transform(m.cSample, '@R ^9999/99/99 99:99'))
? Transform(m.tSample, '@YL')
```

> Outputs (depending on your system's long date setting):
>
> > Saturday, October 8, 2016, 10:30:00 PM

- is used for subtraction. Its operands are either both are date/datetime values OR one is a date/datetime and the other is a numeric.

Let's start with the simpler date/datetime and numeric operands (like with + operator):

When one of the operands is a date, then numeric operand is taken as "day":

```
? Date() - 10 && What was the date 10 days ago?
? Date(2016, 3, 1) - 1 && Returns Feb 29, 2016.
? Date(2017, 3, 1) - 1 && Returns Feb 28, 2017.
```

When one of the operands is a datetime, then numeric operand is taken as "second":

```
? Datetime() - 86400 && Go back exactly one day
```

Get 1 hour and 30 minutes ago from "now":

```
? Datetime() - (1 * 3600 + 30 * 60)
```

Second form is to get the difference between two date/datetime values. Operands are both date or datetime, you cannot use date and datetime at the same time (do type conversion as needed, VFP doesn't do that for you). Rules are as in + and -, operands are date then the difference is in **days**, operands are datetime then the difference is in **seconds**.

How many days to new year's eve (for year 2016)?

```
? Date(2016, 12, 31) - Date()
```

How many seconds left to midnight?

```
? Dtot(Date()+1) - Datetime()
```

> In the last sample we used a Date/Datetime function, DTOT - DateToTime for getting tomorrow's midnight value. There are many useful date/datetime functions available, we skipped them all as they are technically not considered operators (although they operate on date/datetimes:) Same goes true with other operators as well.
>
> The date/datetime subtraction is **signed**. That is if you use the smaller date/datetime as the first operand then the result would be negative. You might use the abs() function if you need to get a positive result regardless of the order of date/datetimes.

Relational operators

Of all the operators, relational operators are the most complex ones, that is why we left them to the end.

> Relational operators are also known as Comparison operators, they are used to compare things.

Comparison result is boolean false or true.

Interestingly though, if you check it in VFP help you only see a short list operations and a few more lines as if it is all about those operators.

Well, the complexity comes from the fact that, they operate on ***any types*** be it a numeric, date, datetime, logical or a string, and even on objects. Moreover, the behavior might look awkward, you don't get what you expect *unless* you know what effects the results.

Let's start with a list of relational operators:

Operator	Description	MOST Basic sample
>	Greater than	? 1 > 2 && .F.
<	Less than	? 1 < 2 && .T.
>=	Greater than or equal to	? 1 >= 2 && .F.
<=	Less than or equal to	? 1 <= 2 && .T.
=	Equal to	? 1 = 1 && .T.
==	Is exactly equal to (makes sense for strings)	? '1' = '1' && .T.
!=, #, <>	Not equal to (all 3 operators act the same way, choose your favorite)	? 1 != 1 && .F.

Although you can use these with all data types, there should be a type compatibility between the operands. For example, you would get an error if you try to compare a Date to an Integer.

Date and Datetime can be compared, although they are different types, VFP does the conversion implicitly for you.

```
? Date() > DateTime() && .F.
? Date() <= DateTime() && .T.
? Date() < DateTime() && .T. if it is not midnight
```

When the operands are numeric, all these operators are simple and straight forward, they work like they would do in mathematical expression.

With the logical operands, .F. is considered to be less than .T.

With objects what we are comparing is the reference of the object in memory. Thus the most used comparison is to determine if two object variables are pointing to the same object. ie:

```
local o1, o2
o1 = createobject('Label')
o2 = createobject('Label')
? m.o1 = m.o2 && is o1 and o2 the same object?
? m.o1 > m.o2 && this would work too but likely you would never use

* remember we are comparing their references in memory
*
* They are different objects, but do they have any difference in their properties?
? CompObj(m.o1, m.o2) && .T. They are identical properties wise
```

> Comparison of character data type, aka comparison of strings is the most confusing one in VFP. It doesn't work as in other languages and/or databases and unique to VFP (and maybe to some other xBase language).
>
> > Many years back, I have even seen some really advanced members in the community who weren't yet aware how these operators work in VFP. So it is quite understandable slight nuances might confuse the newbies easily.

Comparison is basically about being equal or not. If they are not equal, then we might think about the operators >, <, >=, <=, right? With strings it is confusing when *two strings are considered equal* .

> **Important:** VFP strings are case sensitive. 'A' and 'a' are two distinct strings. This is not the case with many databases where the default is to use a case insensitive collation. For example in postgreSQL, or MS SQL Server on a table created with case insensitive (CI) collation:

```
select * from myTable where Country = 'TURKEY'

select * from myTable where Country = 'Turkey'
```

> would yield the same result. In VFP though you only get those where casing matches. However VFP to has some collation support and makes case insensitive comparison. (Do not trust, see below)

- If two strings are not equal, so far so good, **provided that you didn't change any defaults** then they are compared *based on their ASCII values*.

  ```
  ? 'Basoz' < 'Cetin' && is true.
  ? 'basoz' < 'Cetin' && is false.
  ? 'Cetin' < 'David' && is true.
  ? 'Çetin' < 'David' && is false.
  ```

The default for collation is 'machine' and this is what you get then. When you change the collation to something else then you get the comparison based on that collation's sort order. With collation setting other than default **machine** you are also implying a case insensitivity on comparison (do NOT trust this for equality):

```
  set collate to 'GENERAL'
  ? 'Basoz' < 'Cetin'
```

```
? 'basoz' < 'Cetin'
? 'Cetin' < 'David'
? 'Çetin' < 'David'
```

Now all of these expressions are TRUE.

> **Personal** advice: Collations in VFP has never been reliable enough. I suggest you not to use collations and stick with default 'MACHINE'. If you would use collations, then keep in mind to check it first when you experience something that is very unexpected regarding character data. I have seen and demonstrated that it fails in many cases, but then I stopped trying to use it much before VFP9 version, it might be consistent now, I really don't know.

Considering we covered inequality cases with strings, the tricky one is the equality case. In VFP basically two settings effect the comparison:

1. SET EXACT (Default is OFF and effects regular comparison's - those except SQL)
2. SET ANSI (Default is OFF and effects comparisons in SQL only. SET EXACT **has no effect** on comparisons made within SQL queries.

With SET EXACT OFF, read the comparison as "does string at right start with the string at left"? They are compared up to the right string's length.

```
? "Bobby" = "B" && Bobby starts with B, so TRUE
? "Bobby" = "Bob" && Bobby starts with Bob, so TRUE
? "Bobby" = "Bob " && Bobby starts with Bob but there is a trailing space there, FALSE
? "Bobby" = "bob" && would be true with collation set to GENERAL
```

> Note that with regular comparison, "Bobby" = "B" is TRUE, but "B" = "Bobby" is FALSE. In other words, the place of operands are important.

With SET EXACT ON the strings must match fully but their trailing spaces are ignored (we are ignoring set collate here which would also do case insensitivity):

```
? "BOBBY" = "BOB" && FALSE
? "BOBBY" = "BOBBY" && TRUE
? "BOBBY" = "BOBBY     " && TRUE
? "BOBBY     " = "BOBBY" && TRUE
```

Now, with SQL commands SET EXACT has no effect and it would behave like SET EXACT OFF does.

```
Select * from Customers where Country = 'U'
```

Would select the customers from USA, UK any country beginning with 'U'.

In SQL, however, by definition changing the order of operands should yield the same result. Thus:

```
Select * from Customers where 'U' = Country
```

would also work the same way (note the difference from non-SQL commands).

When you want to imply exact matches, one option is to turn on ANSI:

```
SET ANSI ON
Select * from Customers where Country = 'USA'
```

returns all those customers from USA. Note that, the trailing spaces in country field OR on the right expression is ignored. It wouldn't matter how many trailing on either side you have. You get the comparison as if it were done like: RTRIM(Country) = RTRIM('USA').

> Although it is not mentioned in Operators in VFP, an SQL *operator* is LIKE. When you use LIKE, you get an exact match comparison regardless of SET ANSI setting (using LIKE forces and implicit ANSI ON case - it is an ANSI operator after all). However, beware there is a slight difference in behavior. It wouldn't ignore trailing spaces, **unless** the total size with trailers is equal to or less than field size. For example if Country field is C(10), then Country = 'USA' or Country = 'USA__' would work, but Country = 'USA_____' would fail (*underscores* denote a space and last one has more then 7 trailing spaces).

At last we are up to the last operator, ==. That means exactly equal and makes to use with strings. One advantage is that, using == you always mean that you want exact match **regardless of** SET EXACT or SET ANSI settings. However beware again, its behavior is different when it is an SQL command or nonSQL regular command.

With SQL:

```
Select * from Customers where Country == 'USA'
```

whatever the ANSI and EXACT settings are, we want all the customers from USA only. Trailing spaces on either side is ignored.

With Non-SQL:

```
? m.lcString1 == m.lcString2
```

would be true ***only if*** they are exactly same, regarding their casing and length (trailing spaces are NOT ignored). It is not effected from SET ANSI, EXACT or COLLATE settings.

Read Operators online: https://riptutorial.com/visual-foxpro/topic/7625/operators

Chapter 3: VFP Interop with .NET

Introduction

This topic will cover interop between VFP and .NET.

Examples

Using wwDotNetBridge to Run .NET Code

With the help of West Wind's wwDotNetBridge, you can easily have access .NET code within a VFP program.

The white paper has all the details, but this concise example will help illustrate the basic steps to running a method in a .NET assembly.

Note that wwDotNetBridge can directly access simple properties like strings, ints, etc. In order to access more complicated structures like lists, you first need to use the wwDotNetBridge function CreateArray to convert the .NET structure to a VFP COM array (as shown at the bottom of this example).

```
*!* Load WestWind .NET wrapper library (wwdotnetbridge.prg assumed to be in the search path)
IF (!wwDotNetBridge())
    RETURN .F.
ENDIF

lowwDotNetBridge = CREATEOBJECT("wwDotNetBridge","V4")

*!* Load .NET Assembly (include full or relative path if necessary)
IF !lowwDotNetBridge.LoadAssembly("SomeDotNetAssembly.dll")
    lcAssemblyLoadError = "LoadAssembly error: " + lowwDotNetBridge.cErrorMsg
    =MESSAGEBOX(lcAssemblyLoadError, MB_ICONSTOP, "Error")
    RETURN .F.
ENDIF

*!* Parameters to pass to class constructor
*!* You can pass up to 5 paramenters to the constructor
lcParameter1 = "StringParameter1"
lcParameter2 = "StringParameter2"
lnParameter3 = 3
lcParameter4 = .NULL.

*!* Get an instance of the assembly class
loAssemblyReference = lowwDotNetBridge.CreateInstance("MyDotNetProject.MyDotNetClass", ;
    lcParameter1, lcParameter2, lnParameter3, lcParameter4)
IF lowwDotNetBridge.lError
    lcAssemblyLoadError = "An error occurred loading the class: " + lowwDotNetBridge.cErrorMsg
    RETURN .F.
ENDIF

*!* Usage Example
```

```
*!* This example runs a method that return a boolean
*!* and populates a List<string> (SomeStringList).
*!*
*!* The assembly has a public property named "LastErrorMessage"
*!* with details about any handled exceptions/problems.

IF (!loAssemblyReference.SomePublicMethod())
    msg = "There was a problem executing the method:" + CRLF + ;
        loAssemblyReference.LastErrorMessage
    =MESSAGEBOX(msg, MB_ICONSTOP, "Error")
    RETURN .F.
ENDIF

*!* At this point the string list (SomeStringList) should be populated
*!* wwDotNetBridge can convert that list to a VFP COM array (0-based)

laVFPArrayOfStrings = lowwDotNetBridge.CreateArray()
laVFPArrayOfStrings.FromEnumerable(loAssemblyReference.SomeStringList)

FOR x = 0 TO laVFPArrayOfStrings.Count-1
    ? laVFPArrayOfStrings.Item(x)
ENDFOR
```

Read VFP Interop with .NET online: https://riptutorial.com/visual-foxpro/topic/9390/vfp-interop-with--net

Credits

S. No	Chapters	Contributors
1	Getting started with visual-foxpro	Cetin Basoz, Community, Steve
2	Operators	Cetin Basoz
3	VFP Interop with .NET	Steve