

Legal Disclaimer

The currency markets can do ANYTHING at ANY TIME.

No one can guarantee or forecast how these results will perform or behave in future markets.

Anyone who uses this product or this information is responsible for deciding If, Where, When and How this product and information are used.

Anyone who uses this product or this information is responsible and liable for any outcomes that might result from the use of this product or this information.

There is no warranty or guarantee provided or implied for this product or this information for any purpose.

Who am I ?

Biography

Don Baechtel

Cleveland, Ohio USA

dbaechtel@gmail.com

Ph: 216-288-5200



BS degree in Computer Science and Mathematics 1977 from NMIMT, Socorro, New Mexico USA.

45 years Software Development Engineer.

Specialist in Industrial Automation, Robotics, Embedded Control, Digital Signal Processing and Artificial Intelligence.

12 years' investing experience Stocks, Options, and Forex Trading.

2+ years with Apiary Fund

Experienced in Alveo custom Scripts, Indicators and Expert Advisors (Automated trading)

Experienced in Automated Trading platforms in the Cloud using Windows Azure Virtual Machines (VM)

Passionate about writing & coaching the art and science of developing reliable and maintainable code for automating routine Forex trading tasks.

What the Class Is:

- Exchange of Information needed to develop Alveo custom Expert Advisors
- Light coverage of C# programming needed for Alveo Expert Advisors.
- Light coverage of Visual Studio needed to develop Alveo software.
- Hands-on examples to Build Alveo custom Expert Advisors
- Debugging Strategies for Alveo custom Expert Advisors
- Understanding Alveo programming limitations to have a perspective of what can or cannot be done.

What the Class Is Not:

- Extensive detail about Programming, C#, or Visual Studio
- Extensive detail about the Alveo API
- Extensive coverage of existing Alveo Indicators or ExpertAdvisors
- Details and discussions of Automated Trading
- A critique of the Apiary Fund or the Alveo trading platform

What you will receive:

1. Audio/Visual recording of the live Class.
2. A PDF Copy of the Class Notes.
3. A Visual Studio Solution that includes several example custom Alveo Expert Advisors.
4. Limited startup Support from me via email.

Goals:

- Overview of Alveo
- Review of C# programming relevant to the Alveo platform
- Review Visual Studio usage to support program development and debugging for Alveo software.
- Introduction to the Alveo Code Editor
- Create a template Expert Advisor in Alveo
- Create a Visual Studio Project and Solution
- Add the template Expert Advisor source code to the Visual Studio Project
- Add Alveo DLL References to Visual Studio Project
- Build Visual Studio Solution with Expert Advisor
- Transfer Expert Advisor from Visual Studio to Alveo
- Build Expert Advisor with Alveo Code Editor
- Run custom Expert Advisor
- Demonstrate the use of internal classes
- Development of Trading Strategy code.
- Debugging Expert Advisors in Alveo
- Alveo Hints and Secrets
- Additional Discussions (if time permits)

Automated Trading:

Automated Forex trading is the practice of using computers to automate trading tasks on the user's behalf. The goal is to develop an application running reliably, unattended in the Cloud that is generating consistent profitable returns with minimal risk.

Potential Benefits of Automated Trading:

- Can trade 24 hours a day when the Markets are open.
- Never trades tired, angry, frightened, or emotional.
- Helps eliminate subjective decisions and actions , providing consistency in trading results.
- Can analyze more data from multiple sources more quickly than any Human.
- Frees the user from closely monitoring the market action.
- Can enforce Risk and Money Management rules on all trades.
- Can implement Trailing StopLoss limit adjustment to minimize Risk and maximize Profit.
- Can implement sophisticated trading strategies, without requiring close monitoring.
- Can be incrementally optimized for improved results.
- Can be Backtested and Optimized using Historical market data.

Goals of Programming

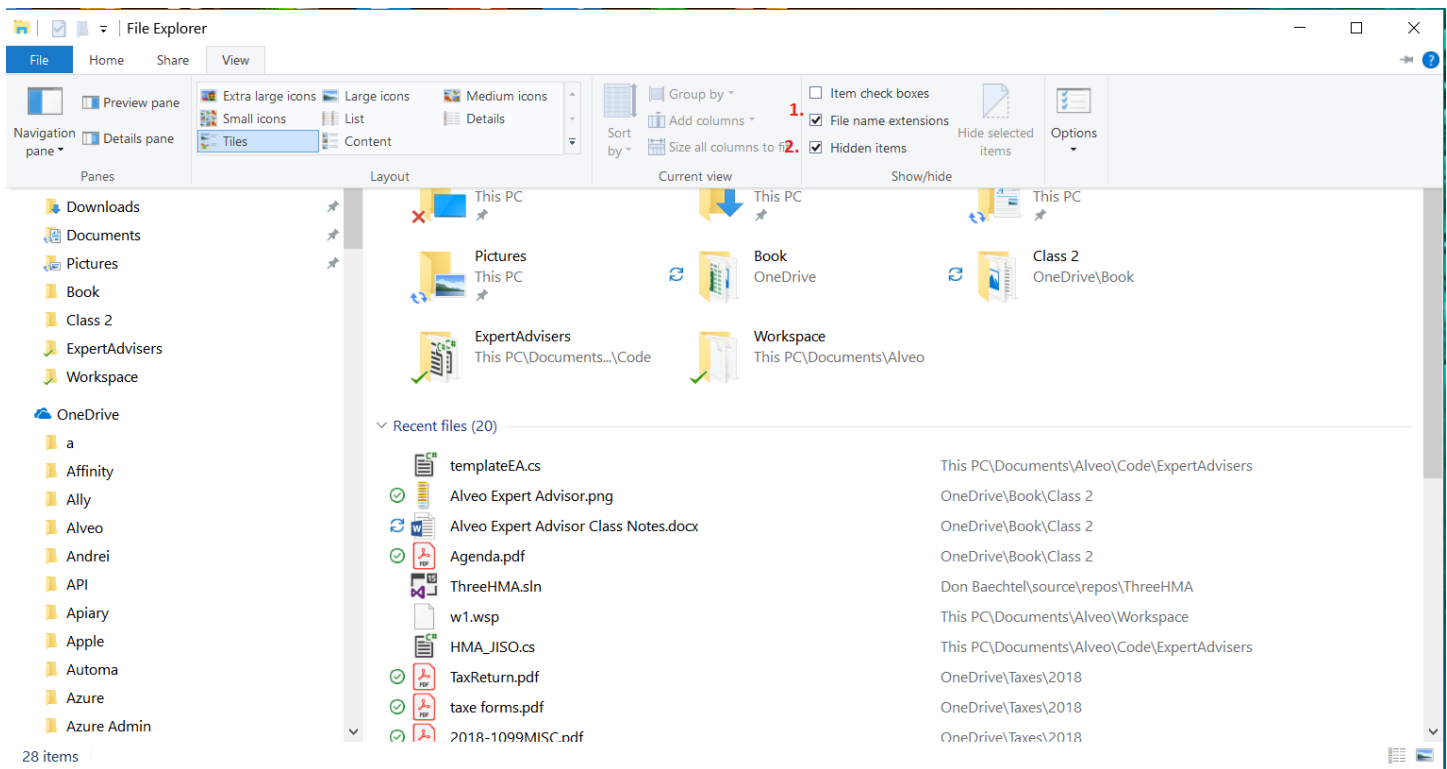
- Reliability
- Readability and Maintainability
- Defensive Programming
- Separation of Concerns and Encapsulation
- Reusability
- Error Handling
- Ease of Debugging
- Design Simplicity
- Efficiency and Performance
- Documentation
- Automating repetitive tasks
- Providing a Profitable Result
- Professional Style

Defensive Programming

- Avoiding errors
- Initialize all variables
- Check validity of Data and Inputs
- Check for null references to objects that have not been instantiated
- Avoid Division by Zero
- Avoid Square Root of negative numbers
- Be wary of comparison for equality of floating-point values
Because of computations, may be not equal by a very small amount. As small as $1 \text{ in } 2^{53} = 1 \text{ in } 9 \cdot 10^{15}$
Instead of **if (double) A == (double)B**
use **if (Math.Abs(A-B) < 1e-10)**
- Add Error and Exception Handling
- Do not add too much code before testing it. Test as you go.
- Test each function
 - * In Forex, check logic for both Long and Short sides
- Read through code with a critical eye.
Keep asking yourself: What might go wrong with this code?
- Simplify Coding and Logic
- Add Print statements for critical code, calculation values or events.
- Dump data files to verify calculations and logic
- Use Visual Studio Unit Testing
- In Alveo:
 - Use Alveo Print function to save information about important events and decisions
 - Use Alveo GetLastError function to check for error returns.
 - Check validity and range of all user inputs
 - Put reasonable limits of StopLoss and TakeProfit limits
 - Put reasonable limits on number of trades per hour and number of trades per day
 - Add Risk and Money management
 - Make sure that Markets are open before attempting to trade

Exercise #1, Add needed features to the File Explorer:

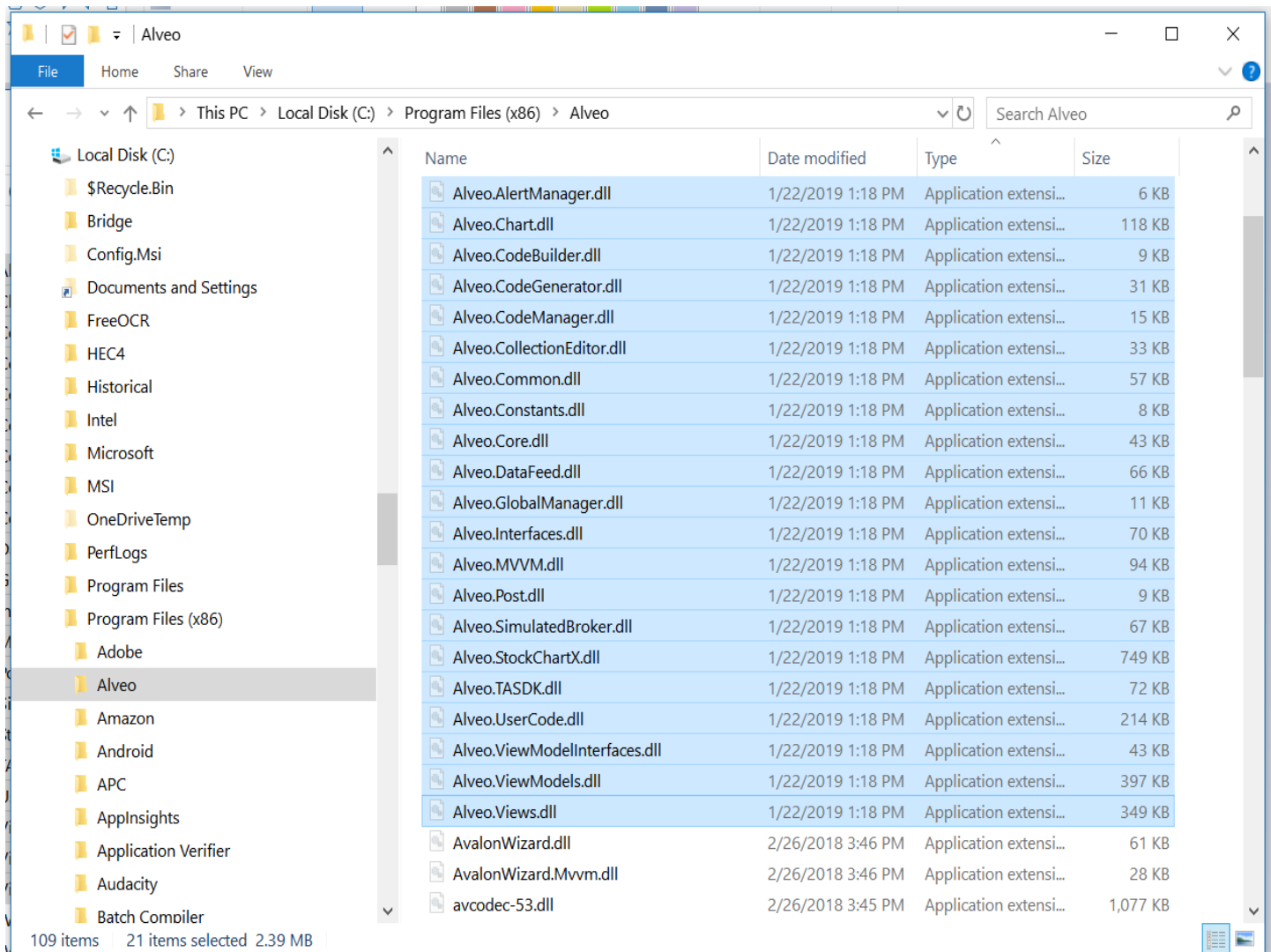
1. In the Windows File Explorer, use the View menu to check the Show File name extensions as shown below.
2. Check the Show Hidden items as well



What are DLLs?

- Not all your code is located in your application, most of the code must be dynamically linked in using various libraries of **compiled** code.
- Microsoft Dynamic Link Libraries
 - See en.wikipedia.org/wiki/Dynamic-link_library
- **Dynamic-link library** (or **DLL**) is Microsoft's implementation of the shared library concept in the Microsoft Windows
- DLLs: Windows, .Net Framework (System), Alveo, and other sources

Alveo DLLs



Overview of the Alveo Trading Platform

Alveo and MT4

- MT4(MQL4) by MetaTrader is a trading platform and was designed using C++
- Alveo is an incomplete port of the MQL4 API using the C# programming language
 - Many MQL4 features are missing (approximately half are missing)
 - Several functions are incomplete in implementation
 - Some functions have limited or partially changed functionality
 - Still, MQL4 API documentation can be used as a guide for Alveo functionality
 - Some MT4 Indicators, ExpertAdvisors can be migrated to Alveo.
Depending on the functionality used, not all MT4 components can be ported to Alveo.
 - Several Alveo functions use Protected access and cannot be used.

Where to find Alveo API Details:

- Alveo DLLs (Visual Studio Object Browser an IntelliSense)
- apiaryfund.com/alveo/codebase/account - incomplete Alveo API doc
- github.com/marlais/Alveo – some sample code
- github.com/dbaechtel/Alveo - some sample code
- github.com/marlais/Alveo/wiki - Alveo Wiki (incomplete)
- docs.mql4.com/ - MQL4 API reference

The screenshot shows the APIARYFUND website interface. At the top, there's a navigation bar with links: Home, Library, Community, Software, Statistics, More, and Help. A search bar is also present. Below the navigation bar, there's a section for 'Documents' and 'Downloads'. The 'Documents' section lists various APIs: Account API, Array API, Basic API, Constants API, File API, FileData API, GlobalVariables API, Helper API, Objects API, Indicator API, TimeSeries API, and Trading API. The 'Downloads' section lists 'Alveo Platform' and 'MQL Converter'. Below this, there's a section for 'Alveo Objects API' with a 'Back to Alveo' link. The main content area displays the MQL4-style code for the 'ObjectCreate' function, including comments and the function signature: `protected bool ObjectCreate(string name, int type, int window, datetime time1 = default(datetime), double price1 = 0, datetime time2 = default(datetime), double price2 = 0, datetime time3 = default(datetime), double price3 = 0)`. Other functions shown include `ObjectDelete`, `ObjectDescription`, and `ObjectFind`.

Exercise #2, API documentation:

1. Review the Alveo and MT4 documentation to see what information that they contain.
2. Pick some function like OrderSend and OrderClose to see what they do and how they do it.

Alveo Bar data type

```
namespace Alveo.Common.Classes
{
    public class Bar
    {
        public Bar();

        public DateTime BarTime { get; set; }
        public decimal Open { get; set; }
        public decimal High { get; set; }
        public decimal Low { get; set; }
        public decimal Close { get; set; }
        public long Volume { get; set; }

        public Bar Clone();
    }
}
```

Example Usage:

```
Bar b = ChartBars[1]; // most recently Closed Chart Bar
Print("Most recently closed Bar, time=" + b.BarTime + " Close Price=" + b.Close);
```


Alveo software components

Scripts

What an Alveo Script does and is used for:

- An Alveo Script is a Short running function to perform some Alveo task
Such as: Deleting all unfilled Pending Orders
- Scripts are terminated if the Chart is Restarted.

Indicators

What an Alveo Indicators does and is used for:

- Long running programs that use Market data to display information on a currency Chart.
Such as: HMA, ATR, MACD, etc.
- Indicators are reinitialized if the Chart is Restarted.

ExpertAdvisors

What an Alveo ExpertAdvisor does and is used for:

- Long running programs that use Market data to perform trade automation on the user's behalf.
Such as: Automated Trading, custom Automated Trailing StopLoss, etc.
- ExpertAdvisors are reinitialized if the Chart is Restarted.
- Alveo Expert Advisors cannot draw on the currency Chart.

Comparison	Alveo Script	Alveo Indicator	Alveo Expert Adviser
Relative Size	short	medium	long
Complexity	simple	simple to complex	complex
Number of Tasks	1 or few	1	few to many
Parameter Settings	yes	yes	yes
Usual Lifetime	few seconds or less	continuous	continuous
Example Usage	simple function or macro	Drawing data on Chart	complex program. automated trading
Selected from	Scripts list	Indicators list	Expert Advisors list
Restarted by Alveo	no	yes	yes

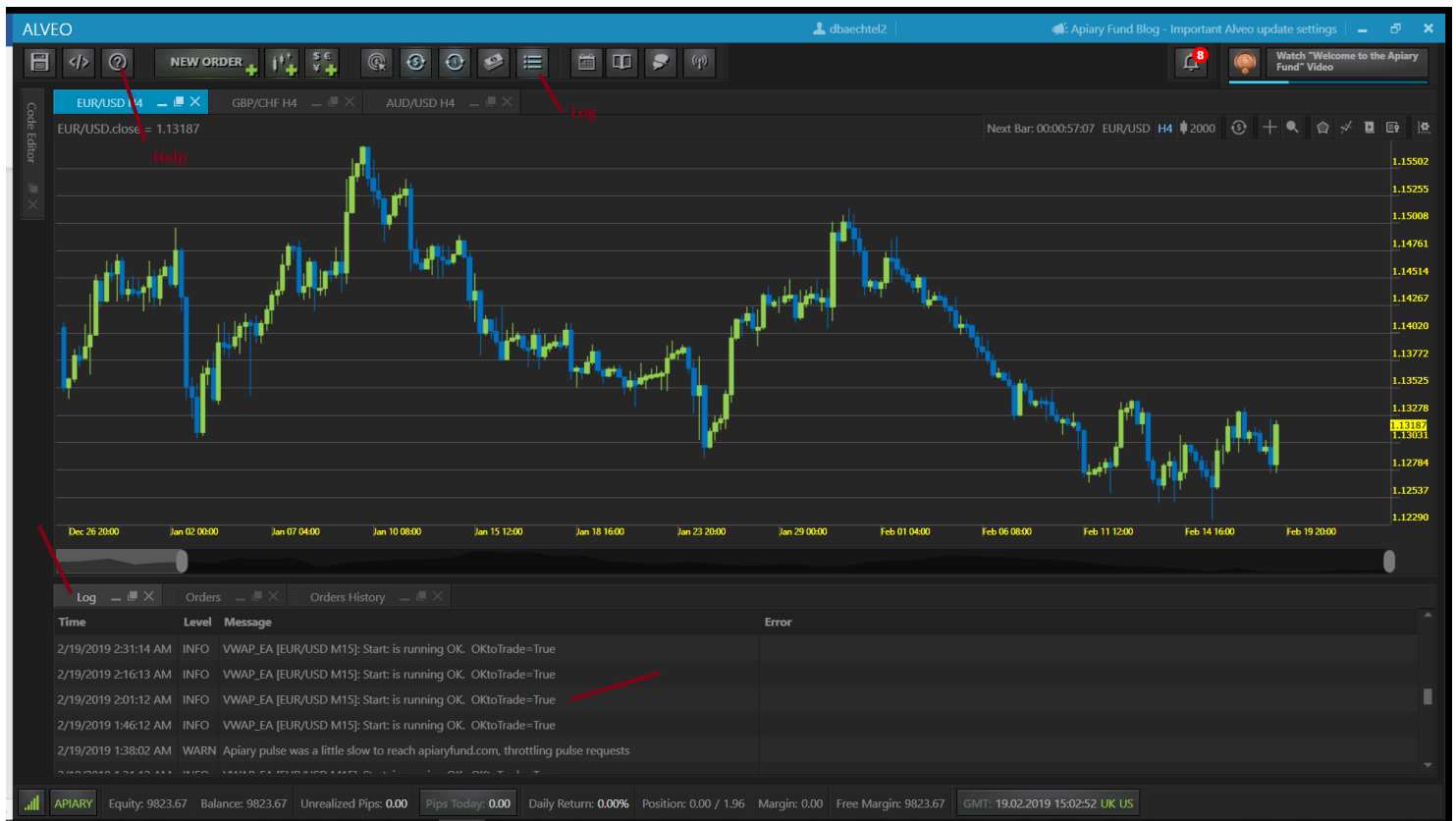
Alveo Code Editor

- Designed to Load, Edit, and Build Alveo software components
- All Alveo software are compiled using the same namespace. ☹️
- When one Alveo component is Built, all the components are rebuilt and restarted. ☹️

The screenshot shows the Alveo Code Editor interface. On the left, there is a chart for EUR/USD with a price of 1.13231. The main area is the Code Editor, showing a file named 'Pricelnd.cs'. The code includes a legal disclaimer and a list of using statements: `using System;`, `using System.IO;`, `using System.ComponentModel;`, `using System.Windows.Media;`, `using Alveo.Interfaces.UserCode;`, `using Alveo.UserCode;`, `using Alveo.Common;`, and `using Alveo.Common.Classes;`. The code is highlighted in yellow. A red arrow points to the 'Code button' in the top toolbar, another red arrow points to the 'Editor' tab, a third red arrow points to the 'Build button' in the top toolbar, and a fourth red arrow points to the 'Source Code' text in the code editor. The bottom status bar shows 'Build Successful' and a log of messages. The bottom right corner displays account information: Equity: 9843.59, Balance: 9843.59, Unrealized Pips: 0.00, Pips Today: 0.00, Daily Return: 0.00%, Position: 0.00 / 1.96, Margin: 0.00, Free Margin: 9843.59, and GMT: 10.02.2019 00:20:13.

Alveo Expert Advisors Debugging Techniques in Alveo

- Alveo Log and Log files
- Print function to Alveo Log
 public **void Print**(**string** message)
 example: Print("Bartime=" + b.BarTime);
- Writing Data files
- Debugging in Visual Studio with simulation of Alveo and Market functions



Example Routine to Write messages to a Log file

```
internal void WriteLog(string header = null, string msg = null, bool clear = false)
{
    // Delete Log file if clear = true
    // Append header and msg to Log file
    string dataFileDir = "C:\\temp\\"; // file location
    string LogFilename = this.ToString() + ".Log.csv"; // file name
    if (!System.IO.Directory.Exists(dataFileDir)) // create directory if !Exists
        System.IO.Directory.CreateDirectory(dataFileDir);
    var path = dataFileDir + LogFilename;
    if (clear) // Delete log file if clear=true
        System.IO.File.Delete(path);
    if (!string.IsNullOrEmpty(header) // if header or msg has contents
        || !string.IsNullOrEmpty(msg))
    {
        // use StreamWriter to append header and msg lines to file and then Close the file
        System.IO.StreamWriter tradeLogFile = new System.IO.StreamWriter(path, append: true);
        if (!string.IsNullOrEmpty(header))
            tradeLogFile.WriteLine(header);
        if (!string.IsNullOrEmpty(msg))
            tradeLogFile.WriteLine(msg);
        tradeLogFile.Close();
    }
}
```

Example invocations of WriteLog Method

WriteLog(header: "This will be the first line header in the file.", clear: **true**);

WriteLog(msg: "The current Price=" + curPrice);

WriteLog(msg: "The current ATR=" + ATR.value.ToString("F5"));

Alveo Hints

- EA Filenames **must** be the same as EA class name
- Copy the (missing) pdf_js.pak file to C:\Program Files (x86)\Alveo directory.
- Alveo symbol and timeframe arguments not implemented. 😞
 - This means that Alveo Indicators and EAs cannot access data from other symbols or timeframes. 😞
- Alveo Arrays
 - ArrayIsSeries = true means indexing is **Reverse**, Array[0] is most recent.
 - See: <https://docs.mql4.com/array>
- Test snippets of Alveo code before developing too much.
 - Some things don't work the way that you expect.
 - Do not invest too much effort in code without testing it.
- Some Windows functionality is not supported by Alveo
 - Example: Http, XML, Serialization and others
- Not all MQL4 MarketInfo types are implemented in Alveo
- Not all MQL4 Account Information is implemented in Alveo.
- MQL4 Global Variables are not implemented in Alveo
- MQL4 Chart Operations are not implemented in Alveo
- Not all MQL4 Object Functions are implemented in Alveo
- Not all Windows features work in Alveo.
- Not all C# features work in Alveo.
- Alveo did not implement a function like MT4's iCustom routine.
Alveo EAs cannot access technical Indicators that are not part of the built-in set of Technical Indicators. 😞

Unmanaged and Managed Code

- Unmanaged code is the good old C or C++ with no CLR support, therefore unmanaged code does not have a garbage collector and you will have to **keep track of all your memory allocations to avoid memory leaks**. Also, when you build an unmanaged project in Visual Studio, the resulting library or executable is written directly on machine code, therefore it doesn't need the **.NET Framework** to run.
- The managed C# language has CLR support this **allows you to use the Garbage Collector and the .NET Framework classes**.
- Also, when you build a managed project in Visual Studio, the resulting library or executable is written in CLR code (which is then translated to machine code by the CLR), therefore it needs the **.NET Framework** installed to run.
- Writing in managed C# code is **easier** and **has more functionality** than unmanaged code.

Review of C# Programming

Namespace

- A namespace is a declarative region that provides a scope to the identifiers (the names of types, functions, variables, etc.) inside it.
- Namespaces are used to organize code into logical groups and to **prevent name collisions** that can occur especially when your code base includes multiple libraries.
- All identifiers at namespace scope are visible to one another without qualification.
- Identifiers outside the namespace can access the members by using the fully qualified name for each identifier.
- Example: `System.Console.WriteLine("Hello World!");`
- The **using** keyword can be used so that the complete name is not required, as in the following example:

```
#using System;  
...  
Console.WriteLine("Hello");
```

C# preprocessor directives

- See: docs.microsoft.com/en-us/dotnet/csharp/language-reference/preprocessor-directives/#using
- As a directive, when **#using** is used to create an alias for a namespace or to import types defined in other namespaces
- **#region**, **#endregion** used for outlining

Attributes

- An **attribute** is a declarative tag that is used to convey information to runtime about the behaviors of various elements like classes, methods, structures, enumerators, assemblies etc. in your program.

```
[Serializable]  
[Description("description of stuff")]  
[Category("Settings")]  
[DisplayName("name")]
```

About Variables, Global and Local

- Scoping rules
- Type casting
Like: `decimal <-> double`

Example type cast:

```
double closePrice = (double)ChartBars[1].Close;
```

Object creation and garbage collection

- **new** statement to create a new instance of an object
Example: `Queue<double> myQ = new Queue<double>();`
- Garbage collection is executed by .Net to collect and dispose objects that no longer have references to them.

Statements

- Variable declarations
- Assignment statements
- Control Flow
 - If/then
 - For loop
 - Foreach
 - While Loop
 - Switch
- Code Blocks with braces { }
- Class and Class declarations
 - Variables
 - Methods
- Exception Handling
 - try, catch, finally, throw

Function Calls, Arguments and return

- Named arguments
Example: `public double GetThePrice(PriceTypes type, Bar b);`
- Default argument values
Example declaration: `internal void WriteLog(string header=null, string msg=null, bool clear=false)`
Example Invocation: `WriteLog(msg: "Hello there.", clear: true);`

Type and member definition Modifiers

- See: <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/modifiers>

System.Collections.Generic

- See: <https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic?view=netframework-4.7.2>
 - Such as: `List<T>`, `Queue<T>`, `Dictionary<T,T>`, `Stack<T>`, etc.

System.Linq

- The `System.Linq` namespace provides classes and interfaces that support queries that use Language-Integrated Query (LINQ).
 - Such as: `Average`, `Contains`, `Count`, `Max`, `Min`, `Sum`
 - Example:

```
Queue<double> sma = new Queue<double>();
sma.Enqueue(1.23);
sma.Enqueue(2.34);
sma.Enqueue(4.56);
var count = sma.Count; // count = 3
var smaSum = sma.Sum(); // smaSum = 1.23 + 2.34 + 4.56 = 8.13
var smaValue = sma.Average(); // smaValue = (1.23 + 2.34 + 4.56) / 3 = 2.71
var min = sma.Min(); // min = 1.23
```
- The LINQ functions can simplify your code and make it more readable and maintainable.

Naming conventions

In computer programming, a **naming convention** is a set of rules for choosing the character sequence to be used for identifiers which denote variables, types, functions, and other entities in source code and documentation.

See: [en.wikipedia.org/wiki/Naming_convention_\(programming\)](https://en.wikipedia.org/wiki/Naming_convention_(programming))

Choose a naming convention and endeavor to stick with it.

Code Indenting, Edit > Advanced > Format Document

In computer programming, an **indentation** style is a convention governing the **indentation** of blocks of **code** to convey program structure. **Indenting** is not a requirement of most programming languages, where it is used as secondary notation. Rather, **indenting** helps better convey the structure of a program to human readers.

See: https://en.wikipedia.org/wiki/Indentation_style

In Visual Studio, use the Edit > Advanced > **Format Document** feature to adjust the indenting in the document.

Code Comments

Not all code lines need a Comment.

Comments should be used to help readability and aid understanding

.Net Framework

See: https://en.wikipedia.org/wiki/.NET_Framework

.NET Framework (pronounced as "*dot net*") is a software framework developed by Microsoft that runs primarily on Microsoft Windows.

It includes a large class library named Framework Class Library (FCL) and provides features and language interoperability (each language can use code written in other languages) across several programming languages.

The .Net Framework includes a lot of functionality that you will use in your Alveo programs.

Review Visual Studio usage

Solutions

A Solution is a Collection of one or more Projects

Projects, Project Types, Project Properties

Visual Studio Projects are a Collection of one or more elements used to build executable items.

Solution Explorer

- Folders in solution for content
- Add > New > Item – to add new elements (files) to the Solution.
- Add > Reference – to add references to (locations of) the DLL used.

Create new Visual Studio Solution, Project, and C# Class

- Visual Studio Installer, Launch
- File > New > Project > Visual C# > Console App (.Net Framework)
- Project Name, Solution Name
- Solution Explorer > Project > Add > New Folder, then Rename folder
- Folder > Add > New Item > Visual C# > Class > Name (Indicator Name)

Object Browser

- See: https://en.wikipedia.org/wiki/Object_browser
- An **Object Browser** is tool that allows a user to examine the components involved in a software package, such as [Microsoft Word](#) or [software development](#) packages such as Alveo.
- An **object browser** will usually display the hierarchy of components; the properties and events associated with the [objects](#); and other pertinent information; it also provides an interface for interacting with objects.

Source Code Navigation

- Bookmarks, Find and Replace, Find In Files
- Find All References, Go To Definition, Peek Definition

Debugging in Visual Studio

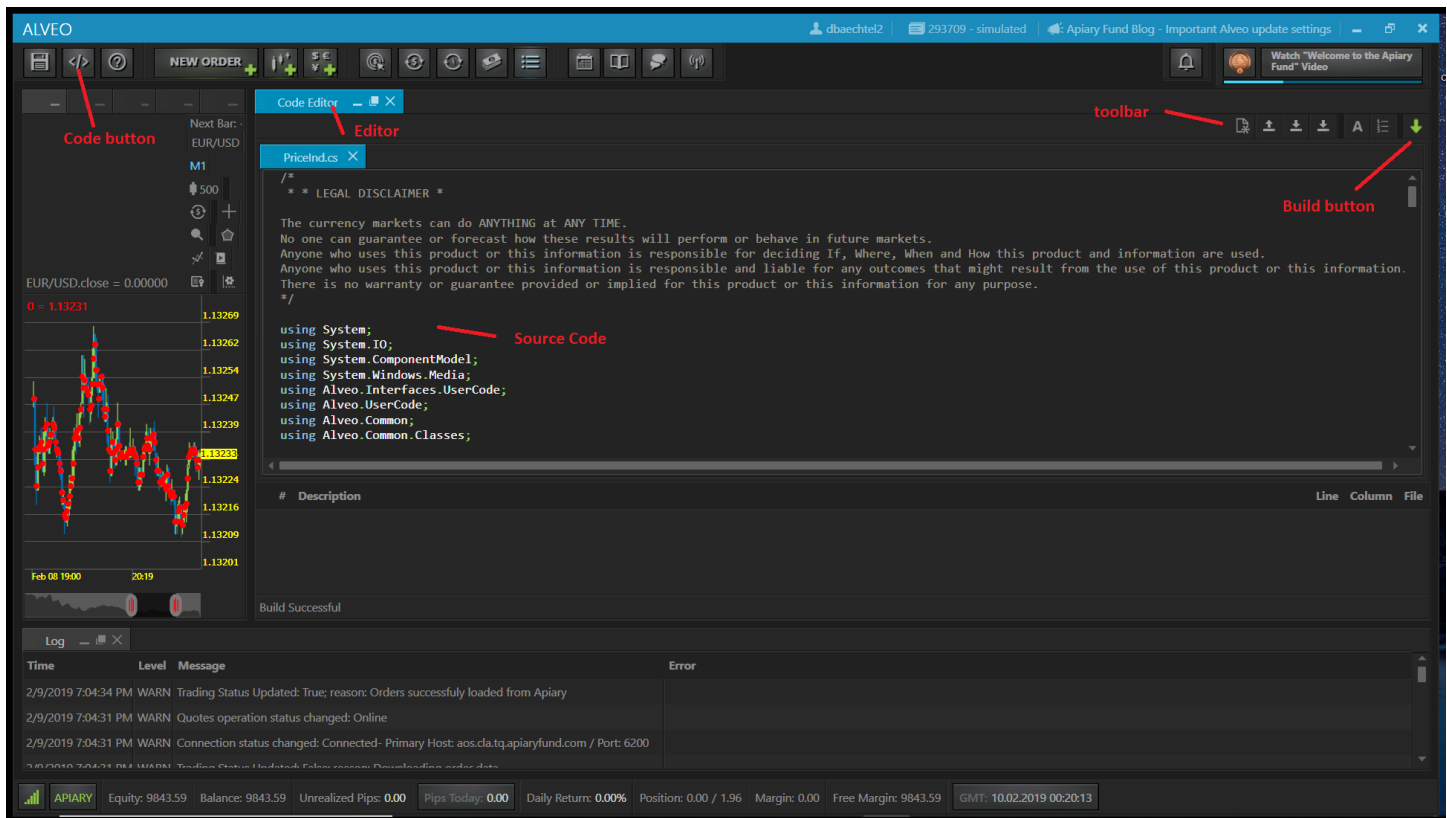
- Run, Pause, End
- Breakpoints
- Continue, Step Into, Step Over, Step Out
- QuickWatch
- Watch Window
- Call Stack
- Writing to Console

Exercise #3, Create Visual Studio Solution, Project, Folder, and C# Class for ExpertAdvisor:

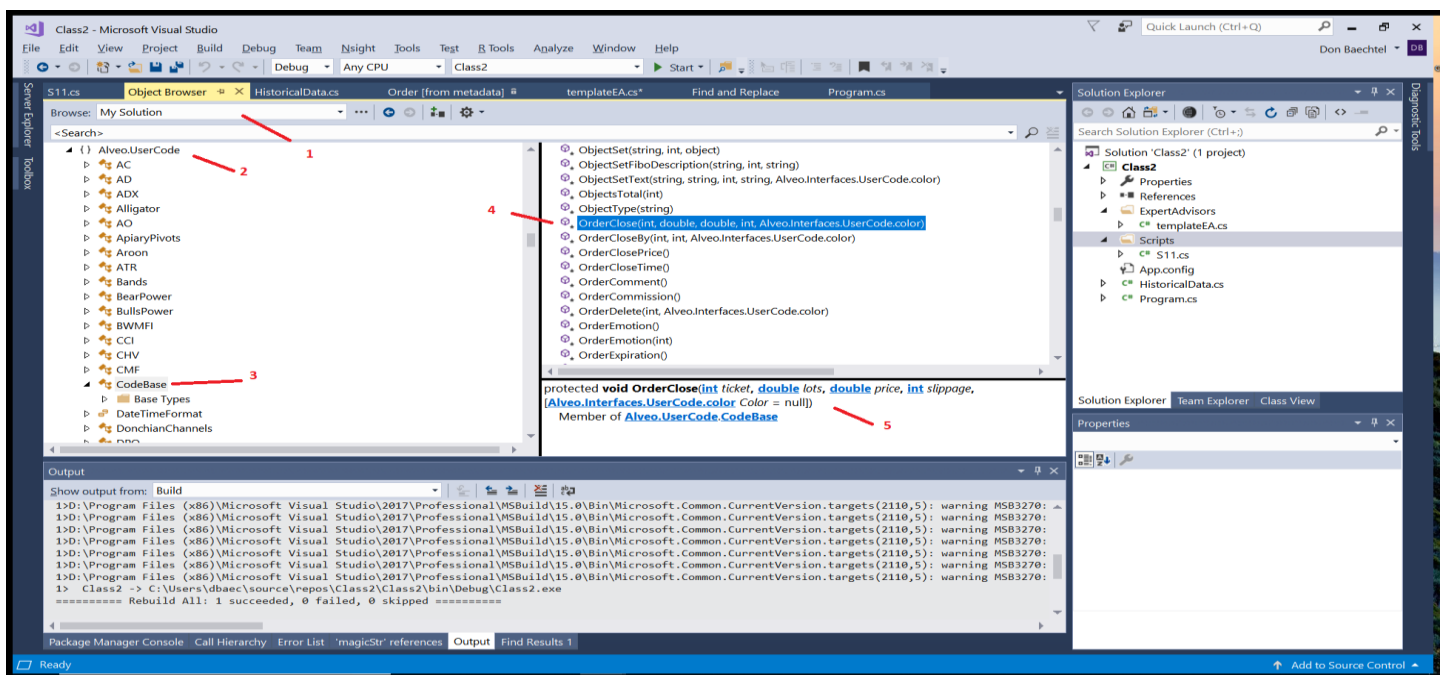
1. Create a new Visual Studio solution with a Console App (.Net Framework) Project.
2. Use the Solution Explorer to Add a new Folder to the Project and Rename it ExpertAdvisors.
3. Choose a name for your Alveo ExpertAdvisor.
4. Add a C# Class in the ExpertAdvisors folder with a Class name the same name as your ExpertAdvisor.
5. Build Visual Studio solution.

Using the Alveo Code Editor

- Alveo Code Editor is used to Edit and Build Alveo software components.
- Alveo Code Directories are usually located at C:\Users\{username}\Documents\Alveo\Code.
- Open, Save, Line Numbers, Build functions are on Code Editor toolbar.
- New and Edit function are located under Alveo's Code button.



Example use of Object Browser from View menu



Anatomy of an Alveo Expert Advisor

#using statements

ExpertAdvisor
class declaration

User Settings
and class variables

Class constructor
(initialize class)

Init()
(initialize variables)

Deinit()
(Termination cleanup)

Start()
(called every Tick or Bar)

Strategy section

Internal Classes

using System;

using System.ComponentModel;
using System.Windows.Media;
using Alveo.Interfaces.UserCode;
using Alveo.UserCode;
using Alveo.Common;
using Alveo.Common.Classes;

namespace Alveo.UserCode

{

[Serializable]

[Description("")]

public class templateEA : ExpertAdvisorBase

{

#region Properties

// User Settings

#endregion

public templateEA()

{

// Basic EA initialization. Don't use constructor to calculate values

copyright = "";

link = "";

}

//+-----+"

//| expert initialization function |"

//+-----+"

protected override int Init()

{

return 0;

}

//+-----+"

//| expert deinitialization function |"

//+-----+"

protected override int Deinit()

{

return 0;

}

//+-----+"

//| expert start function |"

//+-----+"

protected override int Start()

{

return 0;

}

}

Create a template Alveo ExpertAdvisor for Visual Studio

1. In Alveo, Code > Expert Advisor > New
 - a. Code Type should be ExpertAdvisor (Next button)
 - b. Fill out Name
 - c. Define Series (optional, Finish button)
2. Select (click) the Alveo Expert Advisor source and use Cntrl-A Cntrl-C to Copy ALL text to Paste Buffer.
3. In Visual Studio display the Expert Advisor Source file.
4. Select Expert Advisor file and use Cntrl-A to select All text and Cntrl-V to Copy Paste Buffer to source file.
5. Rebuild the VS Solution.
6. Using VS Solution Explorer, in VS Project, Add > Reference to the Alveo DLLs.

To Add the Alveo DLL References to Visual Studio Project:

1. In VS Solution Explorer, right click References line in the Project and select Add Reverence from the list.
2. Click the Browse button.
3. Navigate to the C:\Program Files (x86)\Alveo directory.
4. Sort the files by Type.
5. In the files with Type of Application Extension, select all the DLL files that start with the word Alveo.
6. Click the Add button.
7. Then in the Reference Manager, click the OK button.

Exercise #4, Create Alveo ExpertAdvisor source Template and transfer it to Visual Studio:

1. Using the procedure above, create a template EA and transfer the EA source to the Visual Studio EA file.
2. Add References to all the Alveo DLLs to the Visual Studio Project.
3. ReBuild the Visual Studio solution containing Alveo template ExpertAdvisor without errors. 😊

C# Exception Handling

What is an Exception?

Exceptions are a type of error that occurs during the execution of an application. Errors are typically problems that are not expected. Whereas, **exceptions are expected to happen** within the application's code for various reasons.

Applications use exception handling logic to explicitly handle the exceptions when they happen. Exceptions can occur for a wide variety of reasons, from the infamous `NullReferenceException` to a database query timeout.

Exceptions "percolate" up through the program call chain until it reaches an Exception Handler.

The Anatomy of C# Exceptions

Exceptions allow an application to **transfer control from one part of the code to another**. When an exception is thrown, the current flow of the code is interrupted and handed back to a parent try catch block. C# exception handling is done with the follow keywords: try, catch, finally, and throw

try – A try block is used to encapsulate a region of code. If any code throws an exception within that try block, the exception will be handled by the corresponding catch.

catch – When an exception occurs, the Catch block of code is executed. This is where you can handle the exception, log it, or ignore it.

finally – The finally block allows you to execute certain code if an exception is thrown or not. For example, disposing of an object that must be disposed of.

throw – The throw keyword is used to create a new exception that is the bubbled up to a try catch finally block.

Exercise #5, Add Exception Handling:

1. Instrument the Init, Deinit, and Start routines in the ExpertAdvisor source with **try/catch blocks** as shown below.

```
protected override int Init()
{
    try
    {
        // place Init code here
    }
    catch (Exception e)
    {
        Print("Init: Exception: " + e.Message);
        Print(" Exception: " + e.StackTrace);
    }
    return 0;
}

protected override int Start()
{
    try
    {
        // place Start code here
    }
    catch (Exception e)
    {
        Print("Start: Exception: " + e.Message);
        Print(" Exception: " + e.StackTrace);
    }
    return 0;
}
```

Add User Settings to ExpertAdvisor Class

Alveo ExpertAdvisors User Settings are values that an EA user can change in Alveo to affect the EA's behavior. Add the User Settings as variables in the EA class with the Attributes **[Category]** and **[Description]** as shown below. Make sure that each of the User Setting variables is marked **public** and that the variable **type** is correct. You probably have some variables that you want to set when the EA is started. Add and initialize them here.

The image displays a code editor on the left and the Alveo 'templateEA Settings' dialog on the right. Red arrows indicate the mapping between the code and the UI:

- An arrow points from the `templateEA` class name in the code to the dialog title.
- An arrow points from the `[Category("Settings")]` attribute to the 'Settings' section header in the dialog.
- An arrow points from the `[Description("HEMA Period in Bars [ex: 7]")]` attribute to the description text 'HEMA Period in Bars [ex: 7]' below the input field.
- An arrow points from the `public int HMA_period` variable declaration to the 'HMA_period' input field, which contains the value '12'.
- An arrow points from the `HMA_period = 12;` initialization in the constructor to the same input field.

```
public class templateEA : ExpertAdvisorBase
{
    #region classVariables

    [Category("Settings")]
    [Description("HEMA Period in Bars [ex: 7]")]
    public int HMA_period { get; set; }

    #endregion

    public templateEA() // class constructor
    {
        HMA_period = 12;
    }
}
```

Exercise #6, Add and Test Alveo Expert Advisor User Settings:

1. Add one or more User Settings to the EA class variables and initialize them in the class constructor as shown.
2. Rebuild the Solution using the VS Build menu to make sure there are no errors.
3. Select the EA source code file and use Cntrl-A, Cntrl-C to Copy ALL the code into the Paste Buffer.
4. In Alveo, make sure that the EA source is displayed in the Code Editor.
5. Select the EA source code in Alveo and use Cntrl-A, Cntrl-V to paste the EA source code from VS into Alveo.
6. In the Alveo Code Editor, Build the new EA source code with Build Successful.
7. Close or Minimize the Code Editor to see only the Alveo currency Chart.
8. Select the EA from Alveo Expert Advisors list to display the EA Settings panel.
9. Make sure that the initial values of the Settings are correct.
10. Select the Settings on the Settings panel and make sure the description is correctly displayed as shown above.
11. Change any of the User Settings as you desire.
12. Select OK on Settings panel to add EA to the Chart with the Settings or Cancel to not add the EA to the chart.

Now you have EA variables that you (the User) can set as desired when the EA is started or refreshed on the Chart.



The ExpertAdvisor's Init, Deinit and Start routines

The EA Init routine is called by Alveo every time the EA is started (only once) or each time the Chart is refreshed. Chart refresh happens whenever you either force refresh by Refresh command or you change either chart time frame or symbol. The Init routine should be used to do whatever is needed to be done one time when the EA is started. Typically, much of the initialization of the EA class variables is done here.

The Deinit routine is called by Alveo just before the EA execution is terminated when the EA is closed. Typically, not much is done in the Deinit routine and it is often nearly empty. For example, you could use Deinit to close any open files or write the final status to files just before the ExpertAdvisor is terminated.

The EA Start routine is called by Alveo every Price Tick or when a new Bar on the Chart is closed. Typically, the EA Start routine is used for the bulk of the EA tasks most if not all of the trading related tasks. Be cautious in designing your logic to execute what code runs on every tick and what code runs on creation of every new bar. This can significantly impact usage of the computer resources.

Much of the time, code must be added to the EA Start routine to detect when new Chart Bar is Closed. There are several methods to do this. One method is to detect when the number of Chart Bars has increased. Another approach is to look at the BarTime of ChartBar[1] to see if the time is newer than the previous call.

Exercise #7, Add detection for newly Closed Chart Bar and Test it:

1. To your VS EA source code, add class variables `int nBars`, `int prevBars`, and `Bar curBar`.
2. Give these 3 variables initial values in the EA's Init routine.
3. In the EA's Start routine, add code to return to Alveo if no new closed Bar.
4. In the EA's Start routine, add code when Bar closes to Print out the DateTime of the newly closed Bar.
5. Rebuild VS solution to check for syntax errors.
6. Transfer the EA source to Alveo and Build it successfully.
7. Run the EA on a Alveo M1 chart and check Alveo Log pane to see if the message print out as expected.

The above exercise demonstrates a powerful method of testing functionality of your code as you are developing it.

Your EA source code should now look like the example code on the following page.

Read through the example source code to see if you understand all that is happening.

Do you realize how far you have come?

- You know the elements of an Alveo custom ExpertAdvisor,
- You know what each part of the Alveo EA is used for,
- You know the entry points where Alveo will call you EA and when,
- You know how to get recent Bar data from Alveo
- You know how to transfer EA source code easily to/from Alveo and Visual Studio,
- And you have executed your own Expert Advisor developed in Visual Studio and run it in Alveo. 😊

BTW: If you see warnings about an "unknown class" in the Alveo Log messages, ignore them. They are bogus and will not affect the operation of your EA or Alveo.

```

using System;
using System.ComponentModel;
using Alveo.Interfaces.UserCode;
using Alveo.UserCode;
using Alveo.Common;
using Alveo.Common.Classes;
namespace Alveo.UserCode
{
    [Serializable]
    [Description("")]
    public class templateEA : ExpertAdvisorBase
    {
        // classVariables
        [Category("Settings")]
        [Description("HEMA Period in Bars [ex: 7]")]
        public int HMA_period { get; set; }

        int nBars;    // number of Bars on Chart
        int prevBars;
        Bar curBar;    // most recent Closed Bar

        public templateEA() // class constructor
        {
            HMA_period = 12;
        }

        protected override int Init() // called by Alveo when EA is started, or Chart refreshed
        {
            try
            {
                nBars = 0;
                prevBars = nBars;
                curBar = null;
            }
            catch (Exception e)
            {
                Print("Init: Exception: " + e.Message);
                Print(" Exception: " + e.StackTrace);
            }
            return 0;
        }

        protected override int Start() // called by Alveo every Tick or closed Bar
        {
            try
            {
                // following code runs every Tick or closed Bar
                nBars = Bars; // get number of Bars on the Chart from Alveo
                if (nBars <= prevBars || nBars < 2)
                    return 0; // no new bars
                // following code runs every newly closed Bar
                prevBars = nBars; // update prevBars
                curBar = ChartBars[1]; // most recent closed Bar on Alveo Chart
                Print("Start: newly Closed Bar. time=" + curBar.BarTime);
            }
            catch (Exception e)
            {
                Print("Start: Exception: " + e.Message);
                Print(" Exception: " + e.StackTrace);
            }
            return 0;
        }
    }
}

```


C# Classes

Class as basic concept of Object Oriented Programming which revolve around the real-life entities, e.g. autos. A class is a user-defined blueprint or prototype from which objects are created. Basically, a class combines the fields and methods(member function which defines actions) into a single unit. In C#, classes support the polymorphism, inheritance, and provide the concept of derived classes and base classes.

See: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/classes>

Declaration of class

Generally, a class declaration contains only keyword **class**, followed by an **identifier(name)** of the class. But there are some optional attributes which can be used with class declaration according to the application requirement. In general, class declarations can include these components, in order:

- **Modifiers:** A class can be public or internal etc. By default, modifier of class is *internal*.
- **Keyword class:** A *class* keyword is used to declare the type class.
- **Class Identifier:** The variable of type class is provided. The identifier (or name of class) should begin with a initial letter which should be capitalized by convention.
- **Base class or Super class:** The name of the class's parent (superclass), if any, preceded by the : (*colon*). This is optional.
- **Interfaces:** A comma-separated list of interfaces implemented by the class, if any, preceded by the : (*colon*). A class can implement more than one interface. This is optional.
- **Body:** The class body is surrounded by { } (curly braces).

Constructors in class are used for initializing new objects. Fields are variables that provide the state of the class and its objects, and methods are used to implement the behavior of the class and its objects.

Example:

```
// declaring public class Geeks

public class Geeks : People // Geeks is a subclass of the People Class
{
    // field variables
    public int a, b;

    // class constructor
    public Geeks()
    {
        // initialize class variables
        a = 0;
        b = 1;
    }

    // member functions or methods
    public void display() // display message on Console
    {
        Console.WriteLine("Geek Class in C#.");
    }

    public int Increment() // increment a by b
    {
        return a + b;
    }
}
```

C# Classes are useful because they encapsulate the related variables, methods and events into a single class object that can be created (instantiated) one or more times. Encapsulated code in the class can concentrate on representing and processing a single issue. This aids with simplifying the code by Separation of Concerns.

See: https://en.wikipedia.org/wiki/Separation_of_concerns

Use of Encapsulated Classes for Indicator calculations

Any C# Class, like your ExpertAdvisor class, can contain one or more internal C# classes for various purposes.

One good example of the use of an internal Class inside and Alveo ExpertAdvisor is the inclusion of a class that will execute the same data calculations as an Alveo Indicator. The ExpertAdvisor often needs filtered data for the currency Chart and by using the same Indicator data calculations as an Alveo Indicator, you can be sure that the EA is “seeing” the same results as an Alveo Indicator by using the same data calculation class.

Example:

```
public class myEA : ExpertAdvisorBase // the Alveo EA Class, created and called by Alveo when added to Chart
{
    // myEA class variables
    myEA parent;           // reference to this Indicator to pass to children
    Ind_clsObj ind;         // reference to indicator class object
    ...
    protected override int Init() // initialize myEA instance, this function called by Alveo
    {
        ...
        parent = this;           // set reference to this Class
        ind = new Ind_clsObj(parent); // create new Ind_clsObj instance and pass a reference to this parent class
        ind.Init();              // call the Init function of ind instance
        var indValue = ind.value; // fetch value calculated by ind
        Print(this.Name + ".Init: indValue=" + ind.value );
        ...
    }
    ...
    internal class Ind_clsObj // internal class object
    {
        ...
        myEA Parent;          // for reference to Parent class
        Bar curBar;           // most recently closed bar
        internal double value; // the value of the Ind_clsObj, accessible to callers
        ...
        Ind_clsObj (myEA parent) // constructor for Ind_clsObj class
        {
            ...
            Parent = parent; // save Parent reference
            value = 0;
            ...
        }
        ...
        internal void Init( ) // Init function for Ind_clsObj
        {
            ...
            // All calls to Alveo functions or variables in this Class obj require
            // a reference to the Parent IndicatorBase class
            Parent.Print("Ind_clsObj.Init: started.");
            curBar = Parent.ChartBars[1]; // get most recent closed Bar from Alveo
            value = (double)curBar.Close; // convert decimal value to double...
            // reload Ind_clsObj using Parent.ChartBars, if desired
            var nBars = Parent.Bars; // get number of Bars on Chart from Alveo
            for(int i = nBars-1; i>0; i--) // get closed chart bars from oldest to newest, accessed in reverse order
            {
                var bar = Parent.ChartBars[i];
                ... // do whatever is needed
            }
        }
    }
}
```

Exercise #8, Internal Class

1. Add an internal class to your ExpertAdvisor, similar to the code above.
2. Make sure that you address **all** of the **yellow highlighted** areas shown above.
3. Add Print statements that will help understand what is happening in the code.
4. Rebuild the Solution to be sure there are no syntax errors.
5. Transfer the new EA code to Alveo, Build it, and test the result.

Develop a Trading Strategy

Most Alveo Expert Advisors implement some trading Strategy. It is helpful, before you enter your C# code for the strategy, is to write the Logic for the Strategy in a familiar Structured English format, from which the C# code can be modeled. See: https://en.wikipedia.org/wiki/Structured_English

It is useful to encapsulate the Strategy logic in its own replaceable code routine separate from the details of the remainder of the ExpertAdvisor code. The Strategy routine is called by the EA's Start routine on every newly Closed Bar.

Example Structured English and C# code:

Chart: GBP/USD M15

Indicator: HMA3(period=130, threshold=75), ATR(21)

User Settings: Quantity, AddPips, MaxSpread

Trade Entry:

If no Open trades and (Ask – Bid) < MaxSpread

If HMA3.IsRising (Long)

StopLoss = Min(SwingLow, HMA3 value) – AddPips

TakeProfit = curPrice + 20 Pips

Open Market order, Side:Buy, Qty:Quantity,

SL: StopLoss, TP: TakeProfit

If HMA3.IsFalling (Short)

StopLoss = Max(SwingHigh, HMA3 value) + AddPips

TakeProfit = curPrice - 20 Pips

Open Market order, Side:Sell, Qty:Quantity,

SL: StopLoss, TP: TakeProfit

Trade Exit:

For all Open trades

If order Side=Buy and curPrice <= HMA3 value)
then Close order

If order Side=Sell and curPrice >= HMA3 value)
then Close order

```
void Strategy(Bar curBar)
{
    double curPrice = (double)curBar.Close;
    // Trade Entry
    var total = GetTotalTrades(); // get number of Open trades
    if (total == 0 && (Ask-Bid) < MaxSpread)
    {
        if (HMA3.IsRising) // Long
        {
            var StopLoss = Math.Min(SwingLow,
                HMA3.value - (double)AddPips * point);
            var TakeProfit = curPrice + 20 * 10 * point;
            OpenOrder(tradeType: TradeType.Market, Qty: Quantity,
                SL: StopLoss, TP: TakeProfit);
        }
        if (HMA3.IsFalling) // Long
        {
            var StopLoss = Math.Min(SwingHigh,
                HMA3.value + (double)AddPips * point);
            var TakeProfit = curPrice - 20 * 10 * point;
            OpenOrder(tradeType: TradeType.Market, Qty: Quantity,
                SL: StopLoss, TP: TakeProfit);
        }
    }

    // Trade Exit
    foreach (var order in Orders)
    {
        if (order.Side == TradeSide.Buy && curPrice <= HMA3.value)
            CloseOrder(ticket: order.Id);
        if (order.Side == TradeSide.Sell && curPrice >= HMA3.value)
            CloseOrder(ticket: order.Id);
    }
}
```

Exercise #9, Strategy Logic and C# Code

1. Write your EA Trading Strategy logic in Structured English or use the logic example above.
2. Add a routine called Strategy to your EA class and place a call to the Strategy routine in the EA Start routine.
3. Convert your Structured English logic into C# code in the Strategy routine inside the EA.
4. Add the User Settings from your Strategy to your EA and initialize their values in the EA constructor.
5. Complete the details for the internal class and routines like GetTotalTrades, OpenOrder, CloseOrder, etc.
6. Add more details until the VS Solution rebuilds without errors. (Tie up any loose ends.)

Now you know how to design Strategy logic, convert it to code and add it to the EA to be called by the Start routine. ☺

Debugging and Backtesting your ExpertAdvisor using Visual Studio

Debugging and Backtesting an ExpertAdvisor using Alveo is very difficult and can take a very long time.

With some code changes, your ExpertAdvisor can be made to run in Visual Studio so that it can be debugged, backtested and even optimized with the aid of the Visual Studio features.

Some of the advantages of testing in Visual Studio:

- Better syntax checking and error messages.
- Console messages from code provide information detail on execution.
- Breakpoints to pause execution at some key points.
- VS Debug Start, Stop, Continue and Restart features.
- Step Into and Step Over to single step code execution to verify proper execution.
- Locals and Watch window display values of important variables.
- Quickwatch feature can display details on any variable or expression.
- Call Stack window displays and navigate the execution call stack.
- In VS Debug mode, hover the mouse pointer over any variable and display its current value.
- Using Historical Forex data, an entire year of M1 bars can be backtested in a few minutes.

Visual Studio provides a wealth of Debugging features that will prove to be invaluable in the development and testing of your ExpertAdvisor. With these features you can prove to yourself that your ExpertAdvisor code is working well and producing the desired results before you run it in Alveo. 😊

Exercise #10, Exploring VS code Debug features.

Using your test Program and ExpertAdvisor code in Visual Studio, try the following:

1. Make sure that your test Program and EA rebuild in Visual Studio without any errors. Fix any errors.
2. Set some VS Breakpoints at a few key points in the Program and EA code.
3. Set a VS Breakpoint on the last line of the test Program Main routine.
4. Add one or more Console.WriteLine("message"): statements at various points in the code.
5. Use the VS Debug Start Debugging, F5 key, or Start button to VS toolbar to start test Program execution.
6. When code execution pauses on a Breakpoint, use the mouse to hover over a variable to see its value.
7. See the values of the Local variables in the Locals Window.
8. Select a fully qualified variable, like `ea.simulate`, right click on the highlighted variable and select Add Watch from the list to add the variable to the Watch window.
9. Select the VS Call Stack pane to see the execution call stack up to the point of execution.
10. Experiment with Step Into and Step Over buttons to single step execution one line at a time.
11. Study more of the VS Debugging features from tutorials like:
docs.microsoft.com/en-us/visualstudio/debugger/debugger-feature-tour

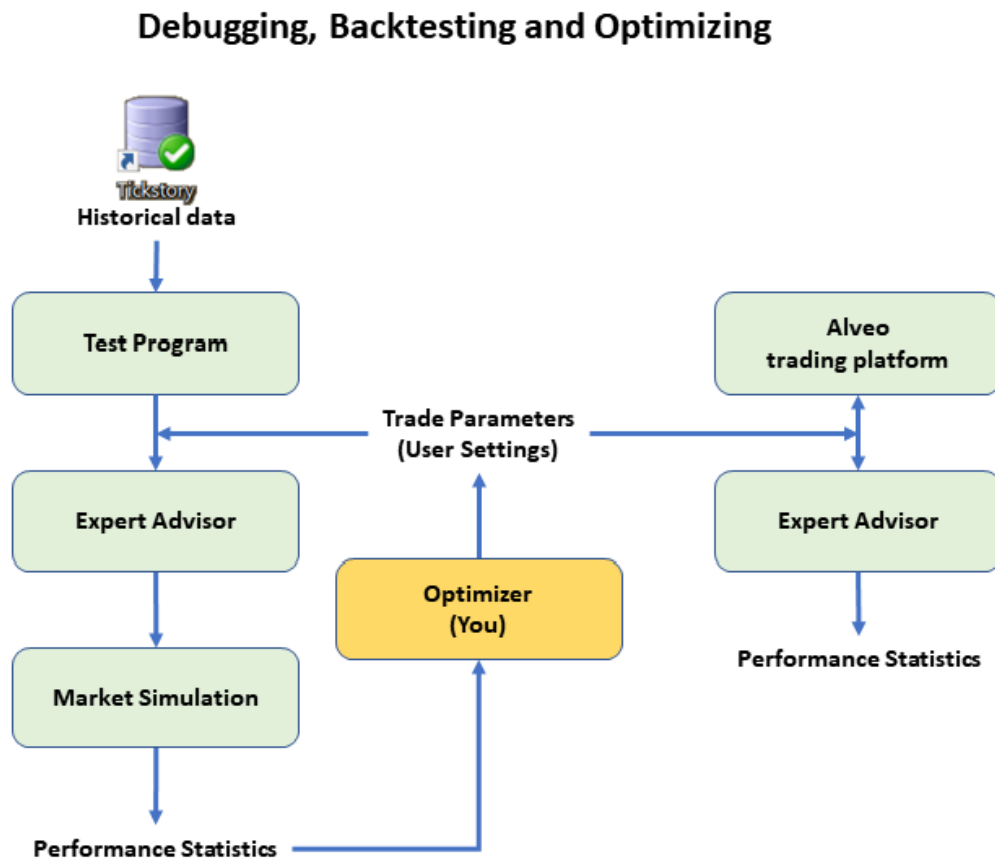
Debugging, Backtesting and Optimizing

Normally the Alveo Expert Advisor, given a set of trading parameters (user settings), interacts with the Alveo platform, getting Price data, issuing trading commands to Alveo, and generating Performance Statistics so that the results of the trading action can be assessed.

In order to Debug, Backtest and Optimize the Expert Advisor, in Visual Studio we replace the Alveo trading platform with a Test Program that feeds the ExpertAdvisor Historical Data, one Bar at a time, just like Alveo does, and Market Simulation code that simulates the trading action of Alveo and the Markets, such as order Filling, order Spreads, StopLoss and TakeProfit limit trips and Order Closing.

To Optimize the trading parameters for a specific currency pair and timeframe, the Performance Statistics, provided by the ExpertAdvisor, can be used to modify the Trading Parameters and run the Backtest again, until a desired result is achieved. There are automatic ways to do this optimization, but these methods are complicated. So, for now you are must serve as the Optimizer, using the Performance Statistics to guide your changes to the trading parameters.

Our goal is to build a system in Visual Studio where the **same** ExpertAdvisor can be developed, debugged, backtested and optimized in Visual Studio, and also run on the Alveo trading platform without modification. 😊



Preparing to Debug or Backtest the ExpertAdvisor in Visual Studio

1. Modify the Project Program.cs file to Call the EA like Alveo does.
2. Add public DoInit, DoDeinit and DoStart routines in EA to allow calls to protected Init, Deinit and Start routines.
 - a. Add calls in the test Program Main function to call the EA's DoInit, DoStart and DoDeinit routines.
 - b. Add breakpoints inside (first code line) of the EA's Init, Start, and Deinit routines.
 - c. Run the test Program with VS Start button to see if it hits each of the EA's Init, Start, and Deinit routines.

Now you have a VS test Program that can execute your EA and feed it data!! 🤖🤖

3. Add a "simulate" Boolean variable to your EA to indicate running with Alveo or Visual Studio.
 - a. Set bool simulate = false; in the EA code.
 - b. Change ea.simulate = true; after instantiating the EA in the test Program.

In this way, the EA uses Alveo functions and variables, or simulated values in Visual Studio.

4. Wrap **all** calls to Alveo functions or variables within simulation capable routines in your EA.
 - a. Example function with simulation capability:

```
int GetBars()
{
    if (!simulate)           // Alveo running
        return Bars;        // return number of Bars on Alveo Chart
    else                     // VS Program running
        return simBars;     // return some simulated value
}
```

5. Replace **all** calls to Alveo functions and usage of Alveo variables with calls to simulation functions.
6. Add Simulation variables to EA and set them in the test Program.
7. Gather Historical market data.
 - a. Use Tickstory Lite or some other application to generate CSV file of Historical data.
See: <https://tickstory.com/download-tickstory>
8. Add the HistoricalData.cs file from <https://github.com/dbaechtel/Alveo.Class2> to your VS Project.
9. Download GBPUUSD.M15.Bar.UTC.csv file from <https://github.com/dbaechtel/Alveo.Class2> to C://temp.
10. Add the BarData class into your ExpertAdvisor class
11. Add interfaces needed to test Program to use LoadDataFile to load in Historical Bar data from file into RAM.
12. Add call to Monitoring routine in EA Start to simulate Alveo and Market actions on trades.
13. Feed all Historical data from the test Program to the EA one Bar at a time, just like Alveo does.
14. Debug the ExpertAdvisor with historical data using VS Breakpoints and Step Into, Step Over.
15. Gather Statistics on EA performance.
 - a. EA trading parameters (User Settings) values used
 - b. Profit/Loss
 - c. Number of Wins
 - d. Number of Losses
 - e. Total number of Trades
 - f. Overall WinRate = Number of Wins / Total number of Trades
 - g. Expectancy = Profit/Loss / Total number of Trades
 - h. Daily Account Balance
 - i. Maximum Daily Drawdown = Account Balance at the End of the Day – starting Daily Account Balance.
16. Test. Test. Test to verify proper execution of your EA.

Now you know how to simulate Alveo functions and variables in your EA and how to use your test Program for debugging and backtesting your EA using Historical data!!! 🤖🤖🤖

Example Test Program in Visual Studio solution to call and pass data to the EA:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Alveo.UserCode;           // allows reference to your EA code using Alveo.Usercode namespace

namespace Class2
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                var ea = new templateEA();           // instantiates your EA and calls EA's constructor
                ea.DoInit();                         // calls EA's Init routine
                ea.DoStart();                        // calls EA's Start routine
                ea.DoDeinit();                       // calls EA's Deinit routine before terminating
                Console.WriteLine("Test Program completed.");
            }
            catch (Exception e)
            {
                Console.WriteLine("Test Program returned Exception: " + e.Message);
                Console.WriteLine("Test Program returned Exception: " + e.StackTrace);
            }
        }
    }
}
```

Example Alveo custom ExpertAdvisor code:

```
using System;
using System.ComponentModel;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Alveo.Interfaces.UserCode;
using Alveo.UserCode;
using Alveo.Common;
using Alveo.Common.Classes;
using Alveo.Common.Enums;

namespace Alveo.UserCode
{
    [Serializable]
    [Description("")]
    public class templateEA : ExpertAdvisorBase    // EA class
    {
        #region classVariables

        [Category("Settings")]
        [Description("HMA Period in Bars [ex: 21]")]
        public int HMA_Period { get; set; }

        // should be User Settings, TBD
        double MaxSpread;
        int AddPips;
        double Quantity;

        double point;           // used to convert integer Points to double price (e.g. 1e-5)
        int nBars;              // number of Bars on Chart
        int prevBars;
        Bar curBar;             // most recent Closed Bar
        HMA3 hma;
        Dictionary<int, Order> Orders;
        double SwingLow;
        #endregion

        public templateEA() // class constructor
        {
            point = 1e-5;      // TBD
            // initialize User Settings to default values
            HMA_Period = 21;
            MaxSpread = 24 * point;
            AddPips = 5;
            Quantity = 0.01;
        }

        internal void DoInit() // external interface to protected Init routine for testing
        {
            Init();
        }

        protected override int Init() // initialize the state of the EA on startup
        {
            try
            {
                nBars = 0;
                prevBars = nBars;
                curBar = null;
                hma = new HMA3(HMA_Period); // instantiate hma object
                Orders = new Dictionary<int, Order>(); // create Dictionary of Orders, key = ticket ID
            }
            catch (Exception e) // Exception handling
            {
            }
        }
    }
}
```



```

        Print("Init: Exception: " + e.Message);
        Print(" Exception: " + e.StackTrace);
    }
    return 0;
}

internal void DoDeinit()    // external interface to protected Deinit routine
{
    Deinit();
}

protected override int Deinit()
{
    try
    {
        // place Deinit code here, if any
        ;
    }
    catch (Exception e)
    {
        Print("Deinit: Exception: " + e.Message);
        Print(" Exception: " + e.StackTrace);
    }
    return 0;
}

internal void DoStart() // external interface to protected Start routine
{
    Start();
}

protected override int Start() // called every Tick or closed Bar
{
    try
    {
        // following code runs every Tick or closed Bar
        nBars = Bars;    // get number of Bars on the Chart
        if (nBars <= prevBars || nBars < 2)
            return 0;    // no new bars
        // following code runs every newly closed Bar
        prevBars = nBars;    // update prevBars
        curBar = ChartBars[1]; // most recent closed Bar
        Print("Start: newly Closed Bar. time=" + curBar.BarTime);
        Strategy(curBar);    // execute trading Strategy
    }
    catch (Exception e)
    {
        Print("Start: Exception: " + e.Message);
        Print(" Exception: " + e.StackTrace);
    }
    return 0;
}

void Strategy(Bar curBar)    // Trading Strategy, called every closed Bar
{
    if (curBar == null) throw new Exception("Strategy: curBar = null."); // check validity
    double curPrice = (double)curBar.Close;
    double SwingLow = 0;    // should be most recent SwingLow, TBD
    double SwingHigh = 0;    // should be most recent SwingHigh, TBD

    // Trade Entry
    var total = GetTotalTrades();    // get total number of Open trades
    if (total == 0 && (Ask - Bid) < MaxSpread) // condition to Open new trade
    {
        if (hma.isRising) // Long
        {
            var StopLoss = Math.Min(SwingLow, hma.value - (double)AddPips * point);
            var TakeProfit = curPrice + 20 * 10 * point;

```

```

        OpenOrder(tradeType: TradeType.Market, Qty: Quantity,
                  SL: StopLoss, TP: TakeProfit);
    }
    if (hma.isFalling) // Short
    {
        var StopLoss = Math.Min(SwingHigh, hma.value + (double)AddPips * point);
        var TakeProfit = curPrice - 20 * 10 * point;
        OpenOrder(tradeType: TradeType.Market, Qty: Quantity,
                  SL: StopLoss, TP: TakeProfit);
    }
}

// Trade Exit
foreach (var order in Orders.Values)
{
    if (order.Side == TradeSide.Buy && curPrice <= hma.value)
        CloseOrder(ticket: (int)order.Id);
    if (order.Side == TradeSide.Sell && curPrice >= hma.value)
        CloseOrder(ticket: (int)order.Id);
}
}

int GetTotalTrades() // return number of Open orders
{
    Orders.Clear(); // clear Orders dictionary contents
    int ntrades = 0;
    // calculate number of open orders with same Symbol, TBD
    return ntrades;
}

int OpenOrder(TradeType tradeType, double Qty, double SL, double TP)
{
    Print("OpenOrder");
    // SendOrder( TBD );
    return 0;
}

void CloseOrder(int ticket)
{
    // OrderClose(ticket); // TBD
    return;
}
}

internal class HMA3 // partial HMA3 internal Indicator class, TBD
{
    int Period;
    internal double value;
    internal bool isRising;
    internal bool isFalling;

    HMA3() // default constructor
    {
        Period = 0;
        value = 0;
        isRising = false;
        isFalling = false;
    }

    internal HMA3(int period) : this() // constructor with parameter(s), calls default constructor
    {
        Period = period; // store indicator Period
    }
}
}
}

```

BarData Class and methods needed in the EA for Historical data

```
public class BarData
{
    internal string symbol;
    internal Bar bar;
    internal DateTime BarTime;
    internal double open, high, low, close;
    internal long volume;
    internal double bid, ask;
    internal bool startOfSession;
    internal bool buySide;
    internal double typical;
    internal double change, pctChange;
    internal double spread;
    internal DayOfWeek wd;
    internal TimeSpan tod;

    public BarData() // BarData object constructor
    {
        BarData_Init();
    }

    public BarData(Bar ibar, double iBid = 0, double iAsk = 0) // create a Bar from BarData
    {
        BarData_Init();
        bar = ibar;
        BarTime = ibar.BarTime;
        open = (double)ibar.Open;
        high = (double)ibar.High;
        low = (double)ibar.Low;
        close = (double)ibar.Close;
        volume = ibar.Volume;
        wd = BarTime.DayOfWeek;
        tod = BarTime.TimeOfDay;
        bid = iBid;
        ask = iAsk;
        if (bid * ask > 0) // if bid and ask are both > 0
            spread = ask - bid;
    }

    internal void BarData_Init() // initialize a BarData object
    {
        symbol = "";
        BarTime = DateTime.MinValue;
        high = 0;
        low = 0;
        close = 0;
        volume = -1;
        bid = 0;
        ask = 0;
        startOfSession = true;
        buySide = true;
        typical = 0;
        change = 0;
        pctChange = 0;
        spread = 0;
        wd = BarTime.DayOfWeek;
        tod = BarTime.TimeOfDay;
    }
}
```

The Order class, MT4 magic integer, and the Alveo Order Comment field

Alveo's Order class is used to store and retrieve various information about trades placed in the Market. This data is shared by Alveo, the Market and your ExpertAdvisor. See `Alveo.Common.Classes.Order` and the `OrderSelect` function.

It is useful to uniquely identify trade Orders to specify their source. In MT4, the order integer magic is often used to hold a user defined integer value to uniquely identify orders as coming from a specified source. The magic number can be specified in the `OrderSend` function.

Alveo maintains the magic argument in the `OrderSend` function but Alveo's `OrderMagicNumber` function is not properly implemented and always returns a value of zero, making it unusable. 😞

This leaves only the `Order.Comment` string available in the Order data for the EA to store unique information about the order that can be retrieved later.

It is often useful to store the unique EA name, Chart Symbol and Chart timeframe and other information in a "magicStr" that can be used in the `Order.Comment` field and used by functions that scan all Alveo orders for ones that are only of interest to (created by) this specific EA:

```
magicStr = ea.Name + "," + symbol.Replace("/", "") + "," + timeframe.ToString();
```

By placing a magicStr in the Order Comment field, it is easy to uniquely tag EA generated Orders and to identify those Orders when retrieved later for specific processing.

Exercise #11, Using Order.Comment field.

1. In your EA, declare and construct a sting `magicStr` to uniquely identify Orders placed by your EA.
2. Test and make sure that the `magicStr` is properly constructed.
3. Add the `magicStr` as the Comment argument in Alveo's `OrderSend` function.
4. In your EA functions that retrieve order data using the `OrderSelect` function, exclude orders whose Comment field does not contain the `magicStr` string in it (that were not created by this EA).

Introduction to Price Type P7

$$P7 = \text{MidBar} + 2 * \text{MidBody} + \text{Close} = (\text{High} + \text{Low} + 2 * \text{Open} + 3 * \text{Close}) / 7$$

Goals of the P7 Price

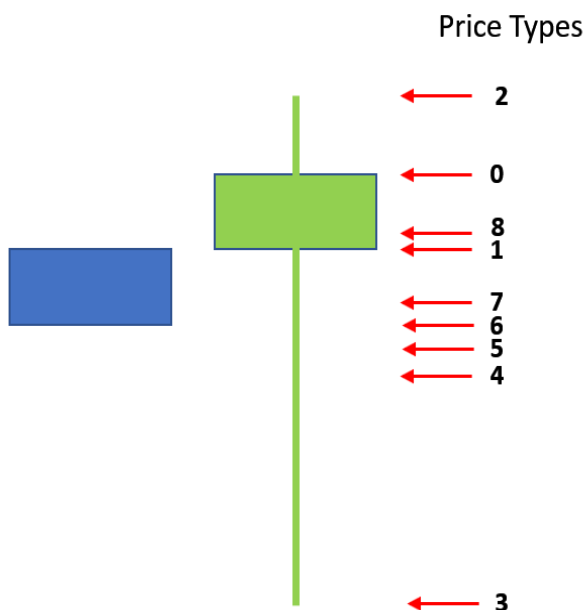
- Better represents average price movement
- Smoother than Close, Median (MidBar), Typical, OHLC, or Weighted Price Types
- Reduces influence of wicks High or Low
- Biased towards the most recent Close Price.
- Better Price value for Indicators and ExpertAdvisors.

Alveo PriceTypes enumeration

```
public enum PriceTypes
{
    PRICE_CLOSE = 0,
    PRICE_OPEN = 1,
    PRICE_HIGH = 2,
    PRICE_LOW = 3,
    PRICE_MEDIAN = 4,    // (b.high + b.low) / 2 (i.e. Mid Bar)
    PRICE_TYPICAL = 5,   // (b.high + b.low + b.close) / 3
    PRICE_WEIGHTED = 6,  // (b.high + b.low + 2 * b.close) / 4
    PRICE_OHLC = 7,      // (b.open + b.high + b.low + b.close) / 4
    PRICE_P7 = 8         // (b.low + b.high + 2 * b.open + 3 * b.close) / 7
}
```

A new PriceType P7

$$P7 = \text{MidBar} + 2 * \text{MidBody} + \text{Close} = (\text{High} + \text{Low} + 2 * \text{Open} + 3 * \text{Close}) / 7$$



Which Price Type do you think better represents the movement of Price between the Blue and Green bars?

Comma Separated Variables (CSV) formatted file for storing data

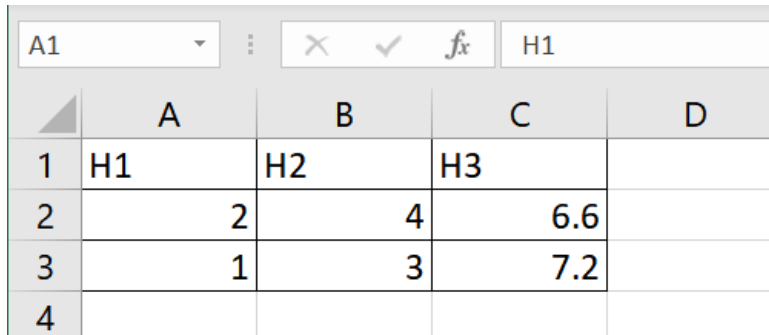
- Simple Format: variables separated by commas

Example file: H1,H2,H3

 2,4,6.6

 1,3,7.2

CSV file loaded directly into Excel



	A	B	C	D
1	H1	H2	H3	
2	2	4	6.6	
3	1	3	7.2	
4				

- Human and Computer readable file
- Can Load directly into Excel for analysis
- Can contain Numbers, Strings, Dates, Time, etc.
- Easy to Generate/Retrieve data in Code
- Can be used to dump data and results of calculations and logic

Using Embedded C# Class Objects

Advantages:

- Separation of Concerns
Each Class has a dedicated purpose.
- Encapsulate code necessary for Indicator calculations separate from Alveo Indicator infrastructure.
- Embedded Class can contain variables, methods, and events needed for the embedded Class object.
- Embedded Class can provide access to more than one result data item.

Examples:

```
* internal double value           // result
* internal double prevValue       // previous value
* internal bool isRising          // value Slope is Rising
* internal bool isFalling         // value Slope is Falling
* internal DateTime LastUpdate    // date and time of last update
```

- Embedded Class can contain additional embedded classes.
Example: Alveo Indicator > ATR Class > SMA Class
- Embedded Classes can be reused for Indicators, Expert Advisors, Scripts

Example Coding Embedded C# Class Object

```
#using ... // libraries needed
namespace Alveo.UserCode // namespace
public class VWAP3 : IndicatorBase // Alveo Indicator declaration
{
    VWAPObj vwap; // Internal VWAPObj
    Bar theBar;
    ...
    public VWAP3() // parameterless (default) constructor for VWAP3 called by Alveo
    {

    }

    protected override int Init() // Alveo EA Init entry point called when Chart is refreshed
    {
        vwap = new VWAPObj(this, ...); // Instantiate VWAPObj
        ...
        return 0;
    }

    protected override int Deinit() // Alveo Deinit entry point for termination cleanup
    {
        return 0;
    }

    protected override int Start() // Alveo Start entry point called every Tick & closed Bar
    {
        ...
        theBar = ChartBars[1]; // most recently closed Bar
        double VWAPvalue = vwap.Calc(theBar); // VWAPvalue to be displayed on the Chart
        return 0;
    }
}
```

```
internal class VWAPObj // embedded VWAPObj Class object
{
    ... // variables
    VWAP3 Parent;
    internal double value;

    internal VWAPObj() // parameterless (default) constructor for VWAPObj
    {
        // initialize VWAPObj variables
        Parent = null;
        value = 0;
    }

    internal VWAPObj(VWAP3 parent, ...) : this() // constructor for VWAPObj w/ parameters
    {
        Parent = parent;
        ...
    }

    internal void Init(ref Bar b) // Initialize VWAPObj using data from Bar
    {
        ...
        value = (double)b.Close;
        Parent.Print("VWAPObj initialized.");
    }

    internal double Calc(ref Bar b) // Calculate VWAPObj value using Bar b and return double
    {
        if (value == 0) Init(ref b);
        value = ... // VWAPObj Calculations
        return value;
    }
}
```

Next Steps for you to take:

1. Use Alveo and Alveo Indicators to visualize a Trading Strategy.
2. Write down your Trading Strategy in detail using Structured English logic. Include:
 - a. Indicators to use.
 - b. User Setting parameters for the EA.
 - c. Trade Entry conditions.
 - d. Trade Exit conditions.
3. Take one of the sample Visual Studio solutions provided and Copy the directory and all files in it.
4. Rename the Solution, directory and some Files in it, as needed for your EA.
5. Rename the EA *.cs source file.
6. Rename the EA class name and its default constructor.
7. Rebuild the VS Solution and make any needed changes in the Test Program and HistoricalData class.
8. Rebuild the VS Solution and modify the code until no errors are reported.
9. Add the User Settings for the EA.
 - a. Initialize the User Settings in the EA Constructor.
10. Replace and Add any Indicator calculation class objects that are needed.
11. Instantiate and initialize the Indicator objects in the EA Init routine.
12. Add calls in the EA Start routine to update the Indicator values whenever a new Chart Bar arrives.
13. Add a Strategy routine to the EA with a call to it in the EA Start routine.
14. Convert your Strategy Structured English to C# code in the Strategy routine.
15. Add Print statements to your EA at key calculation and decision points that will provide details running in Alveo.
16. Rebuild the Solution and clean up any loose ends in the code.
17. Put Breakpoints at:
 - a. all the Exception catch routines,
 - b. the beginning of the EA Init and Start routines,
 - c. when a new Order is entered,
 - d. where Open Orders are closed,
 - e. where the Indicator classes are initialized and when they are called to compute new values,
 - f. the end of the Test Program Main routine to catch it before program termination.
18. Run the Test Program and make sure that:
 - a. the Init and Start routines are called by the Test Program,
 - b. the Test Program passes new Bar data with every call to the EA's Start routine.
 - c. the Indicator objects are properly instantiated and successfully calculate new values,
 - d. step through the Strategy routine one line at a time to ensure that it works as expected,
 - e. make sure that the conditions are correct for Order entries and exits,
 - f. make sure that both the Long order and Short order logic work properly,
 - g. review the EA Performance Statistics to ensure that they are acceptable,
 - h. use Visual Studio to Backtest the EA and optimize the trading parameters
19. Transfer your EA source code to Alveo and Build it successfully.
20. Run your EA in an Alveo Practice account and review the Alveo Log messages that are printed.
21. Run your EA in an Alveo Live account with the order Quantity set low initially.

Now, you have your own Alveo ExpertAdvisor that works the way that you want, can be debugged in Visual Studio, can be backtested with historical data, and can be incrementally improved. Also, You also have an Alveo ExpertAdvisor template solution that can be readily copied and modified to execute various Trading Strategies. 🤖 🤖 Whew!!!