EasyLanguage Coding Best Practices

C21TradingSystem
Updated: 15 July 2020

Introduction

This document contains some suggestions for best practices when coding in EasyLanguage. By establishing and following a set of coding styles and practices, you will be able to develop code that is more understandable and less likely to contain design flaws and bugs. These best practices are only guidelines. If the case for doing something else is strong you can certainly do that.

When developing a small study of a few dozen lines, it is not that critical. However, once the number of lines starts into the hundreds or thousands, it becomes much more important. If one is developing a trading study using OOEL OrderTickets and DataProviders, a study can quickly end up being 5000-10000+ lines long. **As such, how one writes and structures the code and manages it in the TDE is important.**

Consistency is important. Develop a coding style and coding practices as well as overall design practices that you use in all your code. You will come to depend upon code in that structure and will already have some understanding of old code that you go back to months after last working on it.

Properly designed and coded studies will have fewer bugs and will be easier to debug and modify.

Readability and ease of understanding code is most important. So, having more lines is better than having jam-packed lines. Always keep the KISS principle in mind in your programming.

Code Sections Order

The following list highlights one possible order for the sections in a study:

- **Description** – comments as to type of study, purpose of study, any special design aspects
- **Update History** – comments as a record of changes to the code
- **Using** - Any using statements
- **Inputs** – declarations of inputs to the study – one per line, preferably commented if not obvious
- **Constants** – declarations of any value that is a constant – one per line, preferably commented if not obvious. If a value is to be used multiple times in the code, then a constant should be used to declare the value and the constant be used in the code.
- **Variables** – variable declarations – one per line, preferably commented if not obvious. Although the use of expressions often works in a variable definition, the practice is discouraged as there are situations when it does not work. Note that the expression is only evaluated upon the first execution of the code.
- **Arrays** – array declarations – one per line, preferably commented if not obvious. One should consider using EasyLanguage collections rather than arrays as they are more flexible.
- **Methods** –All methods, commented as to its function
- **Once** - if the AnalysisTechnique_Initialized event method is not used, then a once block is appropriate. There should only be a single once block (without a conditional) in a study. The once block must follow any methods.
- **Body Code** – Code that is executed on each execution of the study. Keep as short as possible, putting code into methods that are called from the body code. The body code must follow any methods.

If you have multiple methods that deal with the same basic issue, keep these together in the same region. For example, the methods that create a data provider, handle the *StateChanged* event, and handle the *Updated* event would be kept together.

Comments

Given that the volume of code will be large, appropriate commenting of the code is important. A well-documented study will have at least 10%-20% of the lines being comments. It is difficult to over-comment code. Most programmers fail to comment at all or add insufficient comments. The comments should be such that:

- Any complex calculation is explained and possibly sourced
- Any complex data structure is explained. For example, a dictionary that contains a dictionary in the item value, and that dictionary contains other collections would be a complex structure.
- You could open a study you have not looked at in 6-months and the comments would allow you to quickly refresh your memory about the study.
- Another developer could look at the comments and relatively quickly understand the study to modify it.
- There are two ways to identify a comment: with curly braces ({xxx}) and with double slashes (//). Both are valid, but with the current TDE the use of curly braces can interfere with the use of the #region-#endregion and the Outline feature of the TDE. **If you do use curly braces, do NOT include curly brace comments within curly brace comments as this significantly slows down processing within the TradeStation Development Environment.**

Naming Conventions

- **Meaningful Names** - Use meaningful names for inputs, constants, variables, methods, etc. The name should instantly give one an understanding of for what it is actually being used.
- **ValueNN and ConditionNN** - DO NOT use valueNN or conditionNN variables. They can be too easily used for different purposes at the same time and the name does not give you any indication of what value is contained therein. In addition, these are not intrabarpersist nor can their definitions be aliased as there is no defining line for them.
- **Case** – EasyLanguage is case-insensitive so you can decide on your own casing conventions. One common approach is to use Pascal Case, which capitalizes the first letter of each word in the name (WordPerfect). One standard used within TradeStation is that constant names are in all caps.
- **Special Characters** - Generally do not use special characters within a name, although the underscore is acceptable. Starting names with an underscore is not recommended as Intellisense does not handle such names.
- **Inputs and Outputs** - Name all inputs to a study or method or function starting with "i". This help instantly identify them as not being variables. So, an input for length could be "iLength".
- **Event Handlers** - Name the object event handler methods as "objectname_eventname". For example, for an OrdersProvider named MyOP, the StateChanged event handler would be named "MyOP_StateChanged"
- **Study Names** - Name studies so that the names will almost always be unique and not conflict with any built-in studies and reserved words. One approach is to use a two or three character prefix with all the study names. ("XYZ_Trender" or "XYZTrender", "XYZ_BreakOut", "XYZ_OrderTicket", etc.)
- **Function Parameters** - Parameters that are used to return a value or object have an 'o' as the first character, indicating an output parameter.
- **Method Parameters** – Parameters for methods can begin with "p" so that the variable is clearly identified as a method parameter.

Using Statements

"Using" statements should be utilized to simplify the actual class referencing in the code and make the code easier to read. With the appropriate "using" statements, class references do not need to be fully qualified.

The most common ones are:

- using elsystem;
- using elsystem.collections;
- using elsystem.drawing;
- using elsystem.drawingobjects;
- using elsystem.io;
- using elsystem.windows.forms;
- using elsystem.xml;
- using tsdata.common;
- using tsdata.marketdata;
- using tsdata.trading;
- using Strategy;
- using Charting;
- using Platform;

Indentation

Properly indented code instantly gives one an understanding of the grouping of code, such as in a begin-end block. See the code sample below for an example of one approach to indentation. This approach should always result in an 'end' statement occurring at the same indentation as the 'begin' statement that starts the code block.

```
if SignalName.ToUpper() = "BREAKEVEN STOP" then
begin
        switch args.OrderDirection
        begin
                case StrategyOrderDirection.BuyToCover:
                        SignalName = "SXBE";
                case StrategyOrderDirection.Sell:
                        SignalName = "LXBE";
        end;
end;
…
if iPercentOfCapital = 0 then
begin
        NumberOfShares = 100;
end

else
begin
        if iUseInitialCapital <> 0 then
        begin
                NumberOfShares = intportion(InitialCapital / Close);
        end
        else
        begin
                NumberOfShares = intportion(RunningCapital / Close);
        end;
end;
```

Relational Operator Spacing

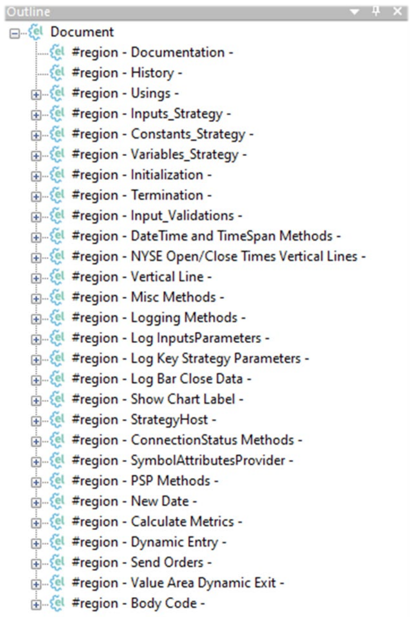
When using the relational operators (=, <>, <, <=, >, >=) always have a blank space both before and after the relational operator. The forum editor is HTML based and the < and > characters have special meaning in HTML. Without the spacing the forum editor may not accept pasted code.

Regions and Outline View

Judicious use of the #region and #endregion compiler directives in the code can result in an outline view that allows quick and easy navigation of the code. Note that the region directives do not reliably work properly in the code, but with careful use they do produce a very usable outline view. In general, all the code should be contained in a region.

Having multiple sets of inputs, constants, variables, or arrays is valid and enclosing them within a region clearly makes finding them easier.

Note that you may put regions inside regions. Just ensure that every #region has a match #endregion.



```
Outline
Document
    #region - Documentation -
    #region - History -
    #region - Usings -
    #region - Inputs_Strategy -
    #region - Constants_Strategy -
    #region - Variables_Strategy -
    #region - Initialization -
    #region - Termination -
    #region - Input_Validations -
    #region - DateTime and TimeSpan Methods -
    #region - NYSE Open/Close Times Vertical Lines -
    #region - Vertical Line -
    #region - Misc Methods -
    #region - Logging Methods -
    #region - Log InputsParameters -
    #region - Log Key Strategy Parameters -
    #region - Log Bar Close Data -
    #region - Show Chart Label -
    #region - StrategyHost -
    #region - ConnectionStatus Methods -
    #region - SymbolAttributesProvider -
    #region - PSP Methods -
    #region - New Date -
    #region - Calculate Metrics -
    #region - Dynamic Entry -
    #region - Send Orders -
    #region - Value Area Dynamic Exit -
    #region - Body Code -
```

Cautions:

- Avoid the use curly braces ("{" and "}") to comment out sections of code as this use may result in the outline view becoming incorrect. The only spot where this does not seem to cause a problem is at the start of the study in the Documentation region.
- Certain reserved words do not appear in the region description. These include input, variables, constants, arrays. Combining these words with an underscore works well (i.e. #region - Variables_Study -)

Begin-End Blocks

Using begin-end blocks, even where they may not initially be needed, often saves trouble later on and also visually helps show code structure. Although it may seem like overkill, a begin-end block after an "if" statement, an "else" statement, a "once", a "while" and a "for" statement clearly shows that all the statements within it execute as a block.

Switch-Case Structure

A switch-case structure rather than a long string of "if-then-else…" often makes code clearer. The use of a "default" case at the end is often useful in catching conditions missed when identifying and coding for all the expected conditions in the "case" statements. You may believe you know all the cases and do not need a "default" case but having one may uncover cases you did not think about or think possible. The default case can raise an error or otherwise draw your attention to its occurrence.

The break reserved word can be used to end the code block for each case statement but is not needed and it is recommended that it not be used unless the case block has no actual code.

Methods

Extensive use of methods can help keep code manageable. The general rule is that a method should be relatively short (even one line is OK) and have one primary task. Putting code in a method and calling it from another method or the body code keeps the calling code shorted and simpler.

Every attempt to put as much code into methods will ultimately result in the most flexible and manageable code. Keep body code to a minimum, preferably calling methods.

Good programming practice normally requires that a method only reference variables passed in through its parameter list or locally declared variables. In some cases this is just too much trouble or not practical, so referencing variables declared at the study level can and should be done.

When developing studies with a large number of lines, creating numerous methods to contain code that handles a single specific function is the best way to go. Properly contained in a #region with a useful description will make navigation in a large study a lot easier using the outline view.

The importance of utilizing numerous short methods cannot be overstated. Failure to learn this early in the development will ultimately be a painful lesson.

One should target to have as few lines as body code as possible.

Once Block, AnalysisTechnique_Initialized and Initialize Component

There two approaches to code that needs to run when a study starts are essentially equivalent. The AnalysisTechnique_Initialized event will run when the study is loaded by the platform. Any Once block code without any condition will run once the study executes. One advantage of the AnalysisTechnique_Initialized is that it is a method and therefore have local variables defined.

Note that when a study is loaded, the AnalysisTechnique_Initialized event is raised in the study and all functions called. As such in functions one should be careful about what code is put inside an AnalysisTechnique_Initialized event handler versus what code is put in a “once” block. Unless the code is very time consuming **and does not depend upon any input values**, one should avoid the use of AnalysisTechnique_Initialized inside a function and use the Once block instead.

If code contains the AnalysisTechnique_Initialized event then one needs to publish the code as an ELD to ensure the Designer code is included.

InitializeComponent should NEVER be used in a study's body code. The event in your body code can have negative impact should the study be re-initialized.

Functions

Code that can be placed in a function and perform properly can reduce the number of lines of the primary study. Just be aware that each unique call to a function is calling a separate instance of that function. Understanding how functions work and the distinction between series and simple functions is important.

If the function is declared as “simple” and everything it needs to process is passed in with input parameters and no values are needed to be retained between calls, then this function can be called from the primary study from different points.

If calls from multiple spots is required and the function is retaining values or is a “series” function then have a single call made from inside a method in the primary study.

When designing functions for use with a trading study, every attempt should be made to make them general purpose. If you later decide to extend the function with additional parameters, you will need

to go back and correct all the studies that use it.
For example a function to build an OrderTicket may want to consider being able to accept a number of parameters, such as:

- Account
- Action
- Duration
- OrderType (and associated properties)
- Quantity
- Symbol and SymbolType

Since OOEL trading will make use of complex orders (such as OSO and OCO) which are typically constructed from multiple simple orders, how one designs the simple OrderTIcket function may be important to being able to use it from functions to build complex orders.

Methods versus Functions

In deciding whether to implement a section of code in a method versus a function there are a number of factors to consider:

- The difference in call overhead between functions and methods is typically insignificant
- A function can be called from numerous studies whereas a method must have it code copied into each calling study. If the code is generic and potentially useful in a number of studies, then a function may be a better choice from a code management perspective in that there is only one set of code to maintain.
- Generic functions should probably be simple and should not depend upon values being preserved between calls.
- Series type functions should probably remain as functions since implementing them in methods is more complex and error prone.
- Except for some exceptional situations, one should not create data providers and other objects that have events in functions as the events will be raised in the function and the calling study will not be aware of these events.
- It is acceptable to pass an object into a function and have the function utilize data available from that object, and if the object permits it allow the function to modify the object (such as an XML document).

Legacy and OOEL

Object-Oriented EasyLanguage (OOEL) refers to EasyLanguage with version 9.0 and beyond. OOEL contains numerous classes that significantly expand the functionality of EasyLanguage. You should embrace OOEL. It allows easier ways of coding, expanded functionality and the ability to do things not possible without the use of classes.

For example:

- **DateTime** - combines the date and time into a single object, with resolution to 1 second.
- **TimeSpan** - more flexible approach to doing date and time arithmetic
- **Parse and TryParse** - methods for converting strings into double, integer, bool, datetime values
- **DrawingObjects** - includes TrendLines and TextLabels, plus Rectangles, HorizontalLines, VerticalLines, Circles. The code can detect the addition, moving and deletion of these drawing objects.
- **String Methods** - expanded string manipulations, such as ToUpper, ToLower, Trim, Substring, Format, etc.
- **StreamWriter and StreamReader** - facilitates writing and reading text files
- **FileSystem Access** – Using the File and Directory classes
- **WinForms** - Add one or more forms to a study

- **TS Charts** - Detect mouse-clicks and keystrokes (hotkeys) on a chart.
- **Strategy** – Obtain order events via the StrategyHost.
- **Charts** - Excel-like charts.
- **Collections** - Effective use of collections of data using vectors, dictionaries, AppStorage.
- **GlobalDictionary** - Inter-study communications.

Division by Zero

Any time there is a division by an input, variable or an expression, the divisor should be checked for being zero and the division not done if it is.

Back References

Using a back reference on a variable or function is a legitimate approach. However, doing back references that do not have a limit on the number of bars that may be back referenced often leads to runtime errors. Numerous TradeStation functions have hard-coded limits to avoid this.

If a study needs to know if a certain condition occurred in the past and the number of bars back could be unlimited, then a better design approach is to detect the condition as bars are processed in the forward direction from bar #1 to the current bar. When the condition is detected, the relevant information can be saved in a set of intrabarpersist variables (if only the last occurrence is ever needed) or in a collection such as a dictionary or set of vectors (if multiple occurrences are needed.)

Lookup Operator

The look-up operator causes the string that is provided as its parameter to be translated into a supported foreign language when the string is displayed to the user. The translation is based on the contents of the EasyLanguage study's string table. For example,

```
Print( !( "stock" ) ) ;
```

Unless you are developing for a foreign market with one of the supported languages, it is not necessary to use the Lookup operator.

Print Statement

The Print statement is the most commonly used approach to debugging code. As such you should learn to use it effectively. The use of the **string.Format** function along with **Composite Formatting** can make the print statements a powerful tool for debugging and logging.

Try-Catch-Finally

The Try-Catch structure can be very useful for catching exceptions that occur that can actually be handled by the code. For example, trying to open a StreamReader for a file that does not exist based upon user input. The user can be notified with an error message and given another try at entering a correct file path.

The Finally block is used when there is code that should be executed, regardless of whether the catch block was entered or not.

Intrabarperist

Scalar variables declared at the study level may be declared as intrabarpersist. Variables that are classes do not need intrabarpersist.

The help for intrabarpersist states:

"A reserved word used when declaring a variable (or array) that indicates that the value of the variable (or array element) can be updated on every tick. By default, the value of a variable or an array element is only updated at the close of each bar. This word declares that the value of a variable or array element is to be updated intrabar."

In this description "updated" really relates to whether the value to which the variable was updated in code will be available the next time the code executes. Without intrabarpersist in the variable declaration the code can update the variable during execution, but once the code finishes executing the variable is reset back to the value it had when the execution started. Only when the code executes on bar close would the value to which the variable is set be retained.

Declaring a variable as intrabarpersist causes the value to be retained on all executions.

This behavior is different than most other programming languages but has some very good uses, especially when creating functions that are used to perform intrabar calculations for indicator display.

In most studies except functions declaring the scalar variables as intrabarpersist will result in the behavior one would expect.

Note that setting a study-scoped variable from an event handler would require that the variable be intrabarpersist. Otherwise the variable would be reset at the end of the event handler code's execution.

Note that in a TradingApp all scalar variables should be declared as intrabarpersist.

Data Typing

It is a good practice to type inputs, constants and variables.

For items in Vectors and Dictionaries, it is necessary to cast the values retrieved using the astype reserved word. It is also a good practice to cast numeric values placed into a vector or a dictionary item value to ensure that the data type is the desired type. For example, is a 0 being inserted into a Vector a double zero or an integer zero. One should use the astype.

```
MyVector.push_back(0 astype double);
```

Enumerations

Enumerations effectively allow using a name to represent an integer value. It is best practice to NOT use the integer values but rather to use the enumeration as it is more meaningful. For example to set the SymbolType property of an OrderTicket object one should use:

```
MyOrderTicket.SymbolType = SecurityType.Stock;
```

and not use

```
MyOrderTicket.SymbolType = 2;
```

Referencing Prices

Although prices such as High, Low, Open and Close can be referenced by the first letter of the reserved word, it is better to use the entire word. This avoids any issue with accidentally typing 0 (zero) instead of O (letter), and similarly for the 1 (number) and l (letter).

InitializeComponent Event

Do not use this event in normal code. It is designed to be used in Designer Code only.

Boolean Expressions

A Boolean expression is an expression that uses the 'AND' and 'OR' reserved words. You should **always** use parentheses to force the order of evaluation. Do not assume that there is an operator precedence of 'AND's being evaluated before 'OR's.

For example:
'A and B or C and D' will not be evaluated the same as '(A and B) or (C and D)' when A and B are False and C and D are true.

Capitalization

EasyLanguage code is not case sensitive. In general capitalization conventions can make code easier to read. Pascal capitalization rules are following most of the time. Names of constants are to be in all uppercase. The reserved words listed below are typically done in lower case.

a	crosses	numericarrayref	than
above	day	numericref	the
ago	days	numericseries	then
all	default	numericsimple	this
an	does	objectref	to
and	downto	objectsimple	total
array	else	of	true
arrays	end	on	truefalse
at	entry	once	truefalsearray
at$	false	or	truefalsearrayref
bar	finally	out	truefalseref
bars	for	over	truefalseseries
begin	from	point	truefalsesimple
below	higher	points	try
break	if	repeat	under
by	input	return	until
case	inputs	share	var
catch	is	shares	variable
const	limit	stop	variables
constant	lower	string	vars
constants	market	stringarray	void
consts	method	stringarrayref	was
continue	next	stringref	while
contract	null	stringseries	
contracts	numeric	stringsimple	
cross	numericarray	switch	