Important Information and Disclaimer:

TradeStation Securities, Inc. seeks to serve institutional and active traders. Please be advised that active trading is generally not appropriate from someone of limited resources, limited investment or trading experience, or low risk tolerance, or who is not willing to risk at least $50,000 of capital.

This course book discusses how TradeStation EasyLanguage allows you to develop and implement custom indicators and trading strategies. However, neither TradeStation Technologies nor its affiliates provide or suggest any specific indicator or trading strategies. We offer you unique tools to help you design you own strategies and look at how they could have performed in the past. While we believe this is very valuable information, we caution you that simulated past performance of a trading strategy is no guarantee of its future performance or success. We also do not recommend or solicit the purchase or sale of any particular securities or securities derivative products. Any securities symbols referenced in this book are used only for the purposes of the demonstration, as an example; not a recommendation.

This book shall discuss automated electronic order placement and execution. Please note that even though TradeStation has been designed to automate your trading strategies and deliver timely order placement, routing and execution, these things, as well as access to the system itself, may at times be delayed or even fail due to market volatility, quote delays, system and software errors, Internet traffic, outages and other factors.

All proprietary technology in TradeStation is owned by TradeStation Technologies, Inc., an affiliate of TradeStation Securities, Inc. The order execution services accessible from within TradeStation are provided by TradeStation Securities, Inc. pursuant to a technology license from its affiliate and its authority as a registered broker-dealer and futures commission merchant. All other features and functions of TradeStation® and EasyLanguage ® are registered trademarks of TradeStation Technologies, Inc. “TradeStation”, as used in this document, should be understood in the foregoing context.

Option Risk Disclosure
Option trading carries a high degree of risk. Purchasers and sellers of options should familiarize themselves with option trading theory and pricing, and all associated risk factors. Please read the Characteristics and Risks or Standardized Options available from the Options Clearing Corporation website: http://www.optionsclearing.com/publications/risks/riskstoc.pdf or by writing TradeStation Securities, 8050 SW 10 Street, Suite 2000, Plantation, FL 33324.

Published by TradeStation Securities Inc.

While every precaution has been taken in the preparation of this book, the TradeStation Securities assumes no responsibility for error or omission, or for any damages resulting from the use of the information contained herein.

Download Course Materials

www.tradestation.com/education/downloads/ELOBJECTS

1st Printing
October 2011

Table of Contents

About this book

Thank you for purchasing the *EasyLanguage Objects - Home Study Course*.

The goal of this course is to help you become more familiar with EasyLanguage objects and how they can be used to extend the existing EasyLanguage code you are already familiar with. You will accomplish this by creating and examining a number of unique EasyLanguage indicators using objects.

Throughout the book, a simple and consistent format will be followed: a set of EasyLanguage elements and concepts will be introduced and explained, followed by one or more sample **Exercises** related to the concepts.

Create and type each exercise, then apply your work to a Chart or RadarScreen as appropriate.

Prerequisites:
It is strongly recommended that you attend a live in-person or live online 2-day EasyLanguage Boot Camp course, or purchase the EasyLanguage Home Study Course, before attempting to go through the exercises in this course. You may also benefit from reading a little bit about object-oriented programming concepts in a programming manual or on the Internet.

This book serves as an introduction to the EasyLanguage object enhancements that are designed to access market data and place orders within the existing EasyLanguage framework. It is recommended that you have an understanding and familiarity with TradeStation before beginning this course. This includes items such as creating and managing Chart Analysis windows as well as general file, window, workspace, and desktop management skills. An excellent place to start would be with TradeStation's educational resources which reside on the TradeStation Education Center and may be accessed from the *Help* menu inside the TradeStation platform or at: www.tradestation.com/education.

You may have purchased this course to use as a refresher after attending a live EasyLanguage course or perhaps because you have not yet been able to attend in person. Whatever the reason, you now possess a thorough, advanced-level course designed to teach you how to use TradeStation's EasyLanguage to understand and write custom analysis techniques and trading strategies that include objects, properties, and methods.

The TradeStation Team

An Introduction to EasyLanguage Objects

Since its introduction, EasyLanguage has continued to evolve as a programming language that allows traders to analyze trading ideas and implement their own trading strategies more efficiently than with more traditional programming languages. The addition of objects to EasyLanguage is the latest evolutionary step that provides a set of enhanced language elements and editing tools to extend the power and flexibility of EasyLanguage while allowing for easy integration with your existing code.

This book introduces you to new terminology and tools that will help you get started using objects in EasyLanguage. However, it is not intended to be a general introduction to object-oriented programming. Through a series of code examples, you will learn how to use EasyLanguage objects in combination with many of the EasyLanguage statements with which you're already familiar. Whether or not you choose to use the object enhancements to develop new EasyLanguage code, be aware that your existing analysis techniques and strategies will continue to work with no changes.

Learning to Drive

One approach to learning about a subject might be the nuts and bolts approach. For example, we could learn all about cars by studying internal combustion engines, steering linkages, and other equally complex mechanical subjects...or we could simply accept that cars are sophisticated machines and learn how to drive them. In much the same way, we could learn about objects by studying the details of polymorphism, encapsulation, inheritance, and other complex programming concepts...or we could simply accept the fact that objects represent sophisticated programs containing properties and methods and learn how to drive them instead.

The first step in learning to drive is to familiarize ourselves with our vehicle. In this case, that would be the TradeStation Development Environment and several of the object-oriented editing tools that will help us on our journey, including the:

- EasyLanguage Editor
- Toolbox
- Component Tray
- Properties Editor
- EasyLanguage Dictionary

When working with the TradeStation Development Environment and EasyLanguage objects, there are several new terms you should be familiar with. We'll go into more detail about object terminology later, but these will help get you started:

- Components – Drag and drop objects accessed from the Toolbox
- Object – A copy of a component or other object
- Class – A blueprint of the features and behavior of an object
- Properties – Data and information within an object
- Dot Operator – Used to access object properties and methods in your code
- Method – The object equivalent of an EasyLanguage function
- Event – A notification that something has changed
- Collections – Data structures of objects and object data

EasyLanguage Code Editor

The EasyLanguage Development Environment is the editor is where you enter and edit EasyLanguage statements and code comments in an EasyLanguage document.

Whenever you start typing in the editor, the Autocomplete window will pop up to show you possible choices based on what you're typing. In addition to showing you familiar reserved words and functions, the AutoComplete window will also help you select the properties of objects you've created. Many of the new object related properties will be covered in this book.

For example, with an EasyLanguage document open in the code editor, type the characters 'va' and the autocomplete window appears with the word 'value1' highlighted since this is the first word starting with 'va'. Typing one additional letter, so the typed characters read 'var', will move the highlighted word to 'var'. Adding the letter 'i' will move the highlight to the word 'variable'. At any point, pressing the Enter key will place the full highlighted word in your document.

Toolbox

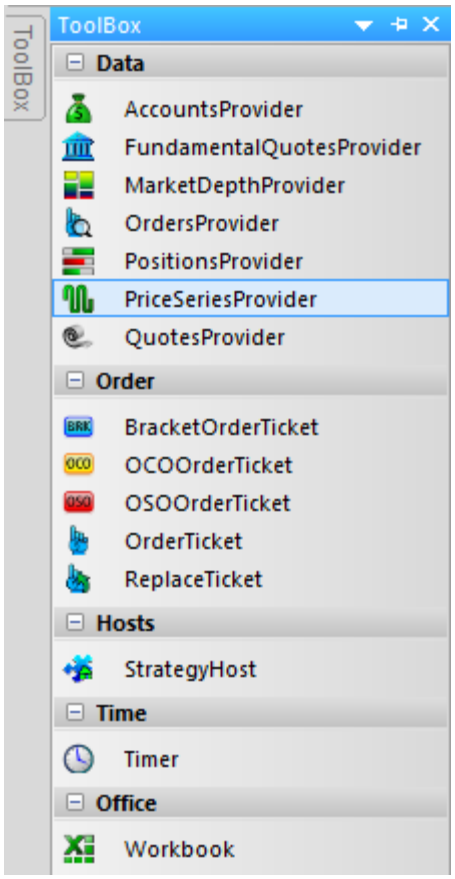
The Toolbox tab, located on the left side of the code editor, displays a list of available components that can be added to your EasyLanguage document as objects. An object combines data and programs into a convenient package that you can access from EasyLanguage code as properties and methods. Toolbox components were designed to make it easy for you to add objects to your analysis techniques, strategies, and functions with a minimum of coding.

Available Components (in the Toolbox)

- Price, Fundamental, Quote, and Market Depth (Level II) Data
- Account Data
- Order and Position Data
- Order Tickets (placing orders)
- Timer
- Workbook (integration with Excel spreadsheets)

Non-Component Objects (not in the Toolbox)

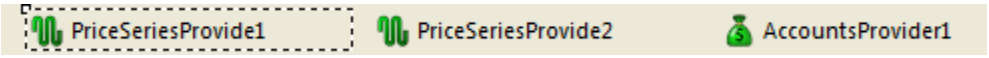
- Global Variable
- Vectors (data collections)
- Custom EL Window Design
- XML Database
- DateTime
- More to come…



You can add a component to an existing EasyLanguage document by highlighting its name and then double-clicking (or dragging-and-dropping) it into the code editor (see Component Tray). In many of the following examples, we'll be using the Toolbox to add component objects to our EasyLanguage documents.

Component Tray

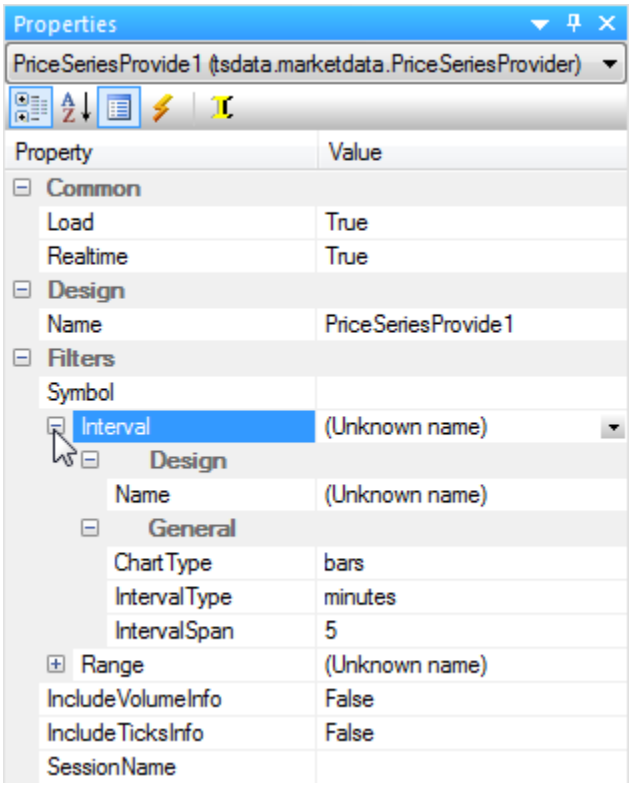
The component tray, at the bottom of the code editor, displays the names of component objects that have been added to your EasyLanguage document from the toolbox. By default, the name is made up of the component name and an instance number.



Different components, or multiple instances of the same component, can be added to a single document. We'll talk more about how this works in later examples.

Properties Editor

The Properties tab, located on the right side of the code editor, is used to access the Properties editor panel (see below) where you can set or review properties for a specific component in your document. To indicate which component's properties you want to edit in the Properties editor, click on a component name in the component tray or select its name from the drop-down list at the top of the Properties panel.



The Properties editor toolbar includes several different icons that let you select between the Property pane and the Event pane along with other editing tools that we'll cover later.

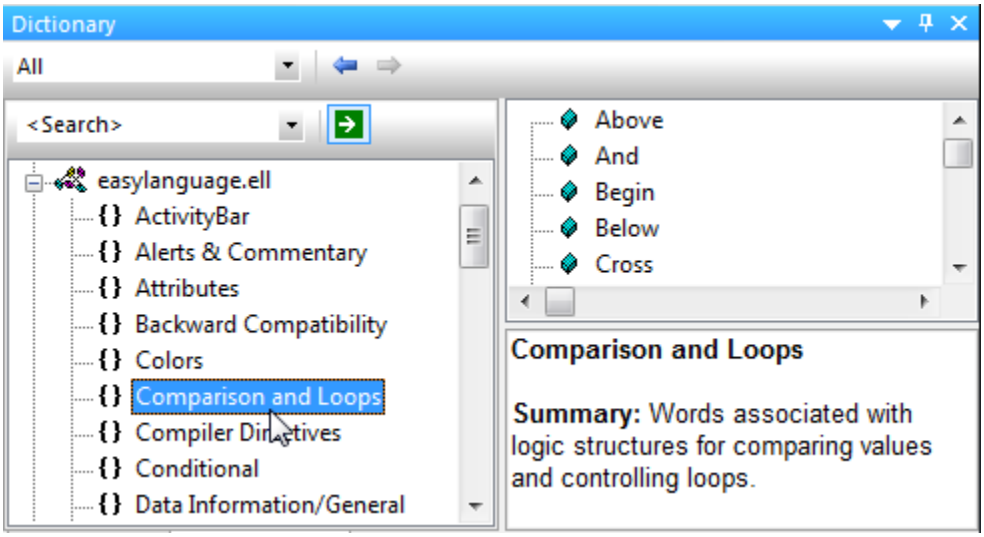
The Property pane is displayed by default. The Property column shows property names grouped by category and the Value column shows their settings. Categories and sub-categories can be expanded or collapsed by clicking the ⊞ and ⊟ symbols. The Event pane includes an Event column and a Value column.

By default, panels will automatically close and appear as a tab again when you click out of the panel. To keep a panel open as you make changes, click the icon at the top of the panel. When you're finished with the panel, click again to close it.

Note: Objects properties should not be confused with indicator properties that are used to set styles, colors, and scaling of indicators.

EasyLanguage Dictionary

The Dictionary tab, also located on the right side of the code editor, is used to access the Dictionary panel where you can search for summaries of EasyLanguage reserved words and functions in addition to information about classes and class members (properties, methods, events, etc.) that will be represented by objects in your EasyLanguage code.



By default, the Objects pane (on the left side of the Dictionary panel) displays a hierarchy of libraries and namespaces containing EasyLanguage classes and reserved words. Clicking the ⊞ and ⊟ symbols let you navigate to other levels in the hierarchy. Click an object in the Objects pane to see related items in the Members pane to the upper right. Click an item in the Members pane to show a summary of that item in the Description pane to the lower right. Icons help identify the type of object or member in both the Objects and Members panes (see TradeStation Development Environment Help for more about icons).

For example (see above), selecting *Comparison and Loops* from the legacy EasyLanguage.ell section of the Objects pane displays a list of reserved words that belong to that category in the Members pane and a summary of the category in the Description pane.

You can also search for an item in the Dictionary by typing one or more words in the Search box at the upper left of the Dictionary panel. The search results will display in the Objects pane where selecting an item will display a summary in the Description pane.

As you become more familiar with objects and classes after going through this course, it may be worth taking the time browse through the various namespaces in the Dictionary to see how classes and their properties relate to one another. The Dictionary contains references to all of the classes, properties, methods, events, and other object characteristics supported by EasyLanguage. Many descriptions also have links to EasyLanguage Help topics that summarize the functionality of the language and its related object elements.

Objects in EasyLanguage

The purpose of this course is to introduce you to the use of objects in EasyLanguage, so the obvious question is, "Why objects?" The simple answer is that objects will let you do some things you couldn't do before in EasyLanguage and, at a minimum, can offer an alternate way of doing things that you're already familiar with doing in EasyLanguage with greater flexibility and structure.

So, what is an object? In the simplest sense, an object represents a particular type of value, or values, that you can access in your EasyLanguage code with a user-specified name. Hold on…that sounds a lot like the definition of a variable, doesn't it? In fact, that's exactly right…an object is used much like a variable, however, instead of it being a traditional variable holding a single numeric, string, or boolean value as we've experienced up to this point using EasyLanguage, an object variable represents an integrated package of values and related elements that act on them.

In objects, values are referred to as *Properties* and can be thought of as object-specific reserved words and variables. Object elements called *Methods*, which are similar to EasyLanguage functions, are used with other elements, such as *Operators* and *Events*, to perform object specific calculations and to manage object tasks.

In keeping with the idea of a package, property and method names are always written along with the object they belong to (more about this in an upcoming section). For example, to refer to a property of an object represented by a variable named *myObject*, you would add a '.' and the desired property name after the object variable name, as in:

```
Value1 = myObject.PropertyName;
```

Because objects are self-contained modules that combine data and programs into a convenient package, you don't have to know anything about the internal coding of an object to use the property values and methods that are supported by the object. This makes objects ideal for accessing sophisticated data structures, such as price data streams, or manipulating otherwise complex program elements, such as windows controls and events. In the case of objects, the type of an object variable is associated with its class and the specific package of values represented by a unique object variable name referred to as an instance.

What Is An Object?

- A copy of a component that knows how to access, return, and manipulate data
- Object data is returned through properties
- Objects have a descriptive name just like a variable

Initial Component Settings

Initially, you'll be adding objects to your EasyLanguage document using components from the Toolbox and setting them up using the Properties editor.

Most components require some initial setup before they can start collecting data or performing the desired action in your analysis technique or strategy. For example, you might need to specify

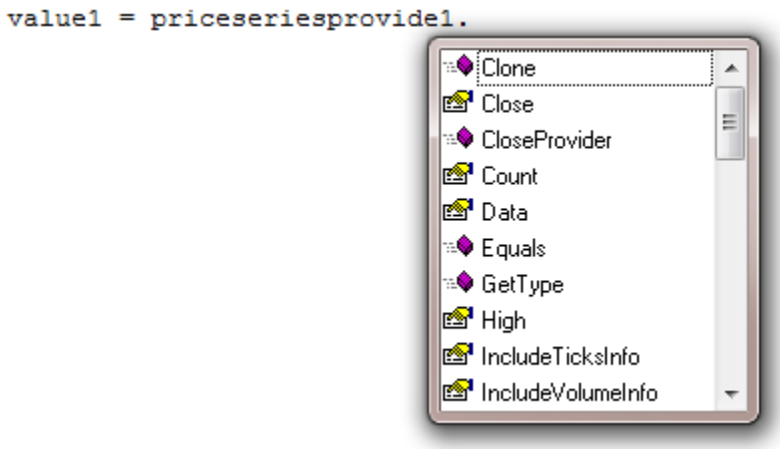
the Symbol for which you want to reference prices using a PriceSeriesProvider object or the AccountID number you'll be using for placing orders with an OrderTicket object.

The course examples in this book will list the settings you'll need to enter for each component used in an example under the heading *Components and Properties Editor Settings*. This should help you get a feel for the properties you'll need to pay attention to when you start using components in your own analysis techniques and strategies.

Accessing Properties in your Code

Once a component object has been added to your document and setup using the properties editor, you will be able to access object values from your EasyLanguage code. This is done using the dot '.' operator between the name of the object and name of the property or method you want to reference. For components, the name of the object is typically the component name followed by an instance number (e.g. PriceSeriesProvide1). When you type the '.' character following an object name, a list of available properties and methods will appear in the AutoComplete window. The list becomes more selective as you type the additional characters of a property name.

As you type the '.' after the name *PriceSeriesProvide1* in the code editor, the autocomplete window appears offering you a list of property choices.



For example, the following statement reads the Count property of a PriceSeriesProvider object and assigns it to a variable. The Count property returns the number of historical intervals (bars) represented by the specific instance of the component object named PriceSeriesProvider1.

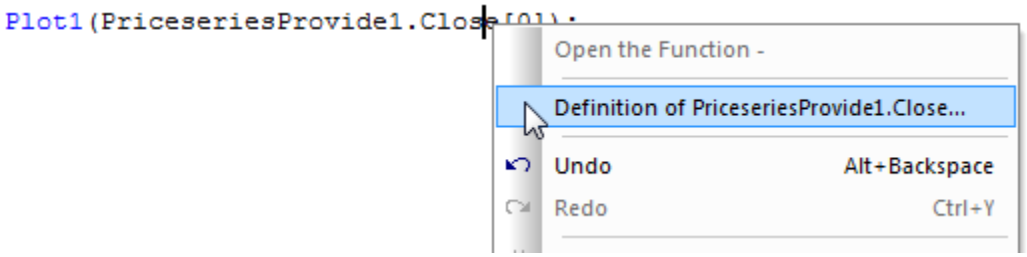
```
Value1 = PriceSeriesProvide1.Count ;
```

Another property of a PriceSeriesProvider is the Close property. However, unlike the Count property that refers to a single value, the Close property actually represents a series of closing prices, starting from the current bar and going back to the beginning of the price series range. To access an individual Close price you'll use the square bracket [ ] index operator and an index value. In this case, the index represents the number of bars ago relative to the interval setting of the object, very much like with the standard Close reserved word in EasyLanguage represents the number of bars ago relative to the bar interval settings in a chart. This type of property, that represents a series of values accessed using a bracketed index value, is called a collections. More about this in the next section.

```
Plot1(PriceSeriesProvide1.Close[0]);
```

In this example, the word Close[0] after the '.' is a property of an object named PriceSeriesProvide1 that represents the closing price of 0 bars ago within the collection of closing prices available to the object. Note that, unlike the similar EasyLanguage reserved word named Close, you must always specify an index [0] as part of the property identifier when referring to an individual closing price of such a collection property.

Once you've added a property reference to your EasyLanguage code, you can right click on the property name and click *Definition of … (PriceseriesProvide1.Close* in this case) from the popup shortcut menu to read the help topic for the selected object and related property.



PriceSeriesProvider

The PriceSeriesProvider component is an object that includes collections of real-time and historical price values for a specified symbol, at some bar interval, over a historical data range. Each PriceSeriesProvider object can be thought of as a virtual chart providing access to price properties having familiar names such as High, Low, Open, Close, Volume, Time, and OpenInterest where individual prices can be referenced using an index that represents the number of 'bars ago'. However, unlike data in a chart, a PriceSeriesProvider object lets you predetermine the symbol and interval to be accessed by your analysis technique independent of the symbol and interval settings in your chart or grid row. The following list some important characteristics of a Price Series Provider:

- Access to the bar data points for any symbol
- Aligned historically with Data1 bars
- Can be used with some standard EL functions
- Symbol, interval, and range are independent of Data1 symbol

Multiple PriceSeriesProvider components can be included in an analysis technique to support multiple data analysis without requiring the component's symbol to be included as a data stream in the chart or grid row. This means that you can use objects to create multi-data indicators for use in RadarScreen and to make it easier to set up multi-data analysis in charts. In addition, you can optionally access Tick and Volume related values, including up/down ticks and up/down volume, independent of the chart or grid interval settings.

The PriceSeriesProvider also supports an Updated event that will trigger in your EasyLanguage code when an underlying price value managed by the provider has changed.

❖ COURSE EXAMPLE #1

Objectives: (PriceSeriesProvider Close Indicator)

- ✓ Drag a PriceSeriesProvider component into the document
- ✓ Use the Properties editor to set component object properties
- ✓ Plot a property value from the component object
- ✓ Understand the difference between Price Series Provider data and chart data

Indicator: $01_DailyBarClose

This example uses a Price Series Provider component to access and plot the daily Close over the bars on a 30-minute chart.

Workspace: $01_DailyBarClose

Building the Chart

Create: 30 min chart
Insert Indicator: $01_DailyBarClose

Components and Properties Editor

PriceSeriesProvide1:

Symbol:	*symbol*	(under category Filters)
IntervalType:	*daily*	(under categories Filters-Interval-General)
IntervalSpan:	*1*	" "
Type	*years*	(under categories Filters-Range-General)
Years	*2*	" "

Indicator Properties

Scaling: Same Axis as Underlying Data

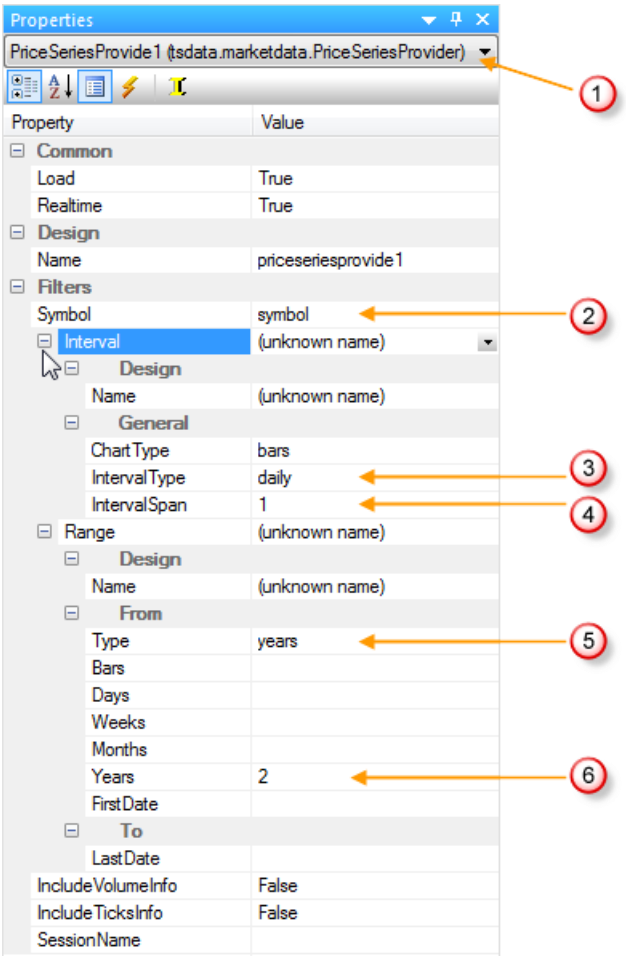
Indicator Exercise #1: '$01_DailyBarClose'

Convention Notes: The EasyLanguage code that you need to type in each exercise will be shown in `courier` font. Code that has been copied from another exercise, or automatically inserted by a component, will have a light `grey` background. Whenever EasyLanguage Reserved Words or Functions are referenced, they will be identified in single quotes (e.g. 'EntryPrice'). Whenever a course exercise input or variable is referenced, it will be identified in italics, e.g., *myInput*.

Create a new Indicator and name it: *'#01_DailyBarClose*. We will use the # naming convention to keep the course exercises separate from the course downloadable examples that begin with a $.

The first thing we're going to do is to click on the Toolbox tab at the left edge of the code editor window to open the Toolbox panel. Locate the *PriceSeriesProvider* component, then click and drag the component name into the code editor. The default name *PriceSeriesProvide1* will appear in the component tray at the bottom on the code editor window.

Now, click the Properties tab at the right edge of the code editor to open the Properties panel and make sure that *PriceSeriesProvider* ① is the specified component at the top of the Properties editor.



Under the Filters category, expand open the Interval and Range sections so that you can set initial values for the PriceSeriesProvide1 component to specify what price data to collect. First, type the word '*symbol*' ② next to the Symbol property. Under Interval, change the IntervalType to *daily* ③ and the IntervalSpan value to *1* ④. Then, under Range, change the Type to *years* ⑤ and the Years value to *2* ⑥. *Note:* The Type property is a number of rows above the Years property, so be sure to

enter the value in the right row. The component is now set to get daily price data for *symbol* for the last two years.

You will now be able to use the PriceSeriesProvide1 component as an object in your code to access prices for the current symbol in a chart using the interval and range settings you entered.

In this example, we'll add a `Plot` statement to display the last Close from the PriceSeriesProvide1 object on top of the bars on the chart.

To access the property of an object in your code, type a dot operator '.' after the object name followed by the name of the object property, in this case '**.**`Close`'. Since we want to display the current Close for this property, we'll add '`[0]`' after the property name to specify the closing price of the current bar.

```
plot1(PriceSeriesProvide1.Close[0],"Daily Close");
```

Next, place the edit cursor in a blank area of your EasyLanguage document and right-click to access the shortcut menu, then select 'Properties…' at the bottom of the shortcut menu. On the Scaling tab of the Indicator Properties dialog, change the Axis setting to Scale On: Same Axis as Underlying Data to show the plot on top of the bars.

Verify the Indicator. If not already displayed, you should open the EasyLanguage Output bar from the View-Toolbars-Output menu. This allows you to see any verification errors and helpful information on correcting them.

Objects and Functions

In analyzing market data, it's common to use a calculated value based on a range of prices, such as the moving average of closes over a specified number of bars. With legacy EasyLanguage you might do that using a function and a price:

```
Value1 = Average(Close,10);
```

The 10 bar average of Closes for the current chart is assigned to Value1. You can then plot Value1, or use it in another calculation or expression. This is possible because the reserved word Close doesn't just represent the current Close, but contains a series of bar Closes from the current bar to MaxBarsBack.

In a similar manner, the object property PriceSeriesProvide1.Close, when written without an index, represents a collection of Close prices based on the interval and range specified for the price series component. The following assigns the 10- bar average of Closes from the PriceSeriesProvide1 object to Value2:

```
Value2 = Average(PriceSeriesProvide1.Close,10);
```

Typically, an object or property that contains a collection of numeric price values can be used in combination with a function input that is an EasyLanguage numericseries. You can inspect the input types for any function by right-clicking on the function name in your code and selecting "Open the Function – NAME".

For example, looking at the inputs for the Average function (shown below), the Price input is a numericseries that will accept a reserved word containing historical values (such as High, Low, Open, Close, etc.) in addition to accepting an object with a collection of prices.

```
Average function, first three lines of code:
    Inputs:
        Price( numericseries ),
        Length( numericsimple ) ;
```

Price Collections

A collection is a type of object (or property) that, much like an EasyLanguage array, holds multiple values that are referenced using an index. For example, a price property, such as `.Close` in a PriceSeriesProvider object, holds a series of Closing prices for the range of intervals specified for the component. As you did in the previous example, you can refer to an individual Close using a square bracket [ ] and an index number following the property name, such as PriceSeriesProvide1.Close[2] to get the close of two intervals ago from an component object named PriceSeriesProvide1.

However, when you eliminate the square brackets, the property PriceSeriesProvide1.Close refers to the entire collection of closing prices for the component and can be passed as a parameter to any standard EasyLanguage function that accepts a numeric series. For example, the following calculates and plots the average Close over the last 15 intervals from the PriceSeriesProvide1 objects.

```
Plot1(Average(PriceSeriesProvide1.Close,15);
```

Although we'll go into more detail about collections later, for now all you need to know is that price collections, such as in a PriceSeriesProvider can return individual price values using square brackets containing a 'bars ago' index or can be passed to a function as a price series without the brackets.

PriceSeriesProvider - Count Property

The PriceSeriesProvider component consists of a collection of prices for a symbol over a specified range of intervals. The Count property tells you how many historical prices are available in the PriceSeriesProvider collection relative to the current bar, as the indicator runs through the historical bars on a chart or RadarScreen row.

To be sure that you always have enough price history in the PriceSeriesProvider to perform a calculation on a price collection, you should check the value of the Count property before executing the calculation code. For example, if you want to be sure that you have enough price data to calculate a specified moving average; you could use the Count property as follows:

```
Input: MovAvgLen(10);
If PriceSeriesProvide1.Count > MovAvgLen then
    Plot2(Average(PriceSeriesProvide1.Close,MovAvgLen));
```

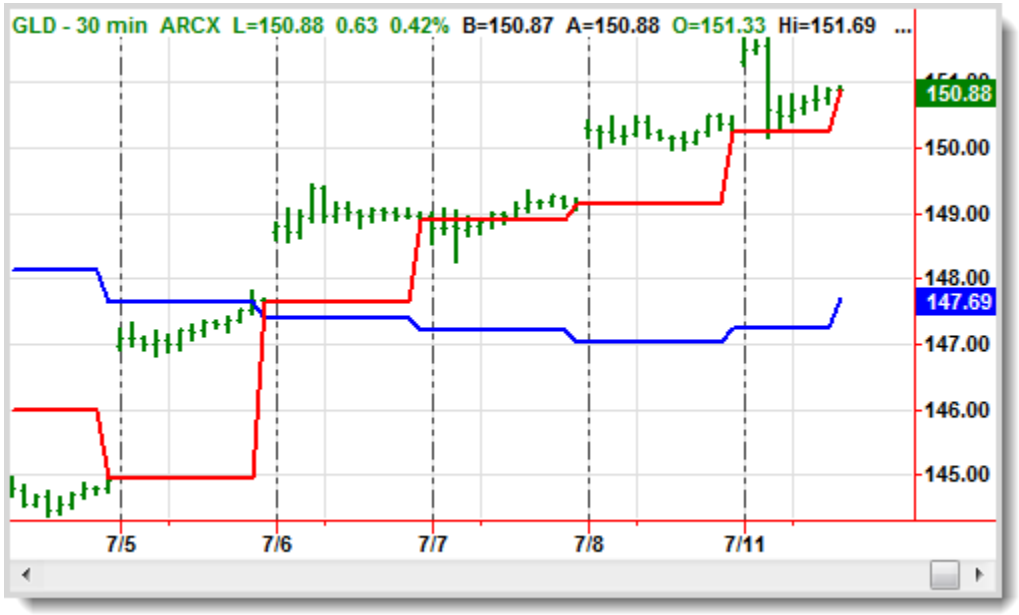
❖ COURSE EXAMPLE #2

Objectives: (MovAvg + Close Indicator)

- ✓ Use a PriceSeriesProvider object with the standard EasyLanguage 'Average' function

Indicator: '$02_DailyAvgClose'

This indicator uses a PriceSeriesProvider component to access and plot a 10-day moving average over the bars.



Workspace: $02_DailyAvgClose

Building the Chart:

Create: 30 min intraday chart
Insert Indicator: $02_DailyAvgClose

Components and Properties Editor Settings:

PriceSeriesProvide1:

Symbol:	*symbol*	(under category Filters)
IntervalType:	*daily*	(under categories Filters-Interval-General)
IntervalSpan:	*1*	" "
Type	*years*	(under categories Filters-Range-General)
Years	*2*	" "

Indicator Exercise #2: '$02_DailyAvgClose'

Convention Notes: The EasyLanguage code that you need to type in each exercise will be shown in `courier` font. Code that has been copied from another exercise, or automatically inserted by a component, will have a light `grey` background. Whenever EasyLanguage Reserved Words or Functions are referenced, they will be identified in single quotes (e.g. 'EntryPrice'). Whenever a course exercise input or variable is referenced, it will be identified in italics, e.g., *myInput*.

For this example, we'll start with the #01_DailyBarClose indicator created in Course Example #1. Save a copy of it as #02_DailyAvgClose. Again, we'll use the # naming convention to keep the course exercises separate from the course downloadable examples.

Since the PriceSeriesProvide1 component was already a part of the indicator you just created, you won't need to add that again. However, if you're starting from scratch, you would drag the PriceSeriesProvider component from the Toolbox into the document and set up the component Symbol, Interval, and Range values using the Property editor as specified under the Components and Properties Editor Settings above.

The plot statement from the previous example displays a line that connects the most recent Close of each PriceSeriesProvide1 bar to the next. Remember, it is virtually identical to plotting the legacy reserved word `Close` on the same bars, except that the object property requires you to specify the current Close using a '`[0]`' after the property name.

Now, let's declare an input for the length of a moving average just before the initial plot statement:

```
input: MovAvgLen(10);

plot1(PriceSeriesProvide1.Close[0],"Daily Close");
```

Next, we're going to add a second plot that shows the moving average of the closing prices in the PriceSeriesProvide1 component collection. We'll check the .Count property of the component to be sure that it contains enough price data to perform the calculation.

The Average function accepts two parameters where the first represents a series of prices and the second the number of bars back on which to calculate the average. Use the Close property (without square brackets) to refer to the entire collection of Close prices based on the PriceSeriesProvide1 interval and range settings. The second parameter refers to the number of intervals back from the current price on which to calculate the average.

```
If PriceSeriesProvide1.Count > MovAvgLen then
   plot2(Average(PriceSeriesProvide1.Close,MovAvgLen),"MovAvgLen");
```

Verify the Indicator.

Inputs and Properties

In addition to setting the initial component properties to a specific value in the Properties editor, you can also set a property to use an EasyLanguage input as its value. This allows you to change the input value when you apply the analysis technique or strategy to a chart or grid and have that value used by the component.

For example, when setting the Symbol property of a component object, such as *PriceSeriesProvide1*, you might have typed "SPY" as the symbol value so that every time your analysis technique runs, the component will access the prices for SPY using the interval settings you specified in the Properties editor. Let's say you wanted to be able to change this symbol using an input, just like you would when writing standard EasyLanguage code. Click on "SPY" to the right of the word **Symbol**, followed by clicking on the  icon at the top of the Properties editor. This inserts the word iSymbol1 as the value for the symbol property and adds the following input statement to your EasyLanguage code, making the entered symbol name the default input.

```
Input: string iSymbol1( "SPY" );
```

Note that if you hadn't previously set the symbol value in the Properties editor, the initial input value for iSymbol1 would be set to "".

Common Provider Properties

A number of toolbox components include the word 'provider' at the end of their name. Using the Properties editor, you'll find that they all include both Load and Realtime properties under the category 'Common'. These properties are used to control the initial state of provider components and both are set to True by default. The Load property, when true, instructs the provider to automatically create a connection to the data that is associated with the provider when your analysis technique or strategy first loads. For example, the AccountsProvider would connect to the data associated with your specified accounts, while the PriceSeriesProvider would connect to the price data stream for the specified symbol, etc. The Realtime property, when true, instructs the provider to continuously request realtime data updates for the associated data. For example, the PositionsProvider would calculate new position values based on the realtime price changes of requested symbol, while the MarketDepthProvider would report realtime changes in bid/ask levels, etc.

In most cases you won't need to modify these settings; however, if you want to set up a provider component that will later be activated when a condition in your code occurs, you could set the Load and Realtime properties to *false* in the properties editor and then assign them to *true* at the appropriate place in your code. Also, if you want to change a an already loaded provider's filter property setting from within your code, you'll need to temporarily set the Load property to *false* to turn off the data connection, assign a new value to the filter property, and then set Load back to *true* to re-establish the data connection based on the new filter setting.

❖ COURSE EXAMPLE #3

Objectives: (RelStrengthRS)

- ✓ Add a Symbol property input
- ✓ Perform multi-data analysis in RadarScreen®

Indicator: '$03_RelStrengthRS'

This indicator uses a pair of PriceSeriesProvider components to calculate the percent change of two symbols over a specified number of days.

	Symbol	Interval	Last	Low	High	Volume Today	$03_RelStrengthRS		
							Rel Perf	Sym Perf	Bench Perf
1	SPY	Daily	132.13	131.84	133.18	110,731,079	0.00	3.55	3.55
2	CSCO	Daily	15.41	15.35	15.62	26,750,312	(1.70)	1.85	3.55
3	IBM	Daily	174.84	174.61	176.15	2,632,484	3.60	7.15	3.54
4	USO	Daily	37.19	36.94	37.63	6,295,587	(8.35)	(4.81)	3.53
5	GLD	Daily	150.88	150.20	151.69	13,284,890	(2.44)	1.10	3.54
6	AAPL	Daily	354.65	353.34	359.77	11,475,256	5.28	8.82	3.54
7	GE	Daily	18.57	18.50	18.78	27,835,150	(2.19)	1.36	3.55

Workspace: $03_RelStrengthRS

Building the RadarScreen:

Create: RadarScreen window with symbols set to Daily interval
Indicator Properties-Grid Style: Set each indicator plot to Number using 2 decimal places
Insert Indicator: $03_RelStrengthRS

Components and Properties Editor Settings:

PriceSeriesProvide1:

Symbol:	***iSymbol1***	(add as an Input)
IntervalType:	daily	(under categories Filters-Interval-General)
IntervalSpan:	1	" "
Type	years	(under categories Filters-Range-General)
Years	2	" "

Indicator Exercise #3: '$03_RelStrengthRS'

In this indicator, we'll be comparing the performance of the current symbol in each row of RadarScreen against a benchmark symbol. This type of multi-data analysis was not possible in RadarScreen before objects where introduced. The performance of current symbol in each row is calculated using the standard Close reserved word, but the benchmark performance is calculated using the collection of Close prices from a PriceSeriesProvider component object for a reference symbol.

Create a new Indicator and name it: *'#03_* RelStrengthRS.

Open the Toolbox, then locate the *PriceSeriesProvider* component and drag the component name into the code editor. The default name *PriceSeriesProvide1* will appear in the component tray at the bottom of the code editor window.

Open the Properties Editor Click on the empty text box to the right of the word **Symbol**, then type the symbol name "SPY" (in quotes) as our reference symbol. Since we may want to change the

reference symbol when we apply the indicator to a chart, we'll convert this symbol value to an input by clicking on the  toolbar icon button at the top of the Properties editor. The word iSymbol1 appears next to Symbol in the Properties editor and you should see the following line inserted at the top of your EasyLanguage code with the default symbol "SPY" as its initial value.

```
Input: string iSymbol1( "SPY" );
```

Still in the Properties Editor, locate the Filters category and expand open the Interval and Range sections so that you can set initial values for the PriceSeriesProvide1 component to specify what price data to collect. Under Interval, change the IntervalType to *daily* and the IntervalSpan value to *1*. Then, under Range, change the Type to *years* and the Years value to *2*. *Note:* The Type property is several rows above the Years property, so be sure to enter the value in the right row. The component is now set to get daily price data for the PriceSeriesProvide1 symbol for the last two years.

In your code, declare an additional input value that sets the default number of bars back to use for the performance look back value.

```
Input: LookBack( 20 );
```

Declare three variables to hold the percent change calculations that are used to measure the performance of the current symbol in each row, the benchmark reference symbol, and the relative difference between them.

```
Vars:  SymbolPerf ( 0 ), BenchPerf( 0 ), RelPerf( 0 );
```

Next we'll calculate the percent change value for the base (current) symbol and for the benchmark symbol.

The performance of the current row symbol is calculated using the reserved word Close which is passed as a parameter to the PercentChange function along with the bar range on which to calculate the percent change.

```
SymbolPerf= PercentChange(Close,LookBack);
```

The benchmark symbol is calculated using the collection of Close prices from the PriceSeriesProvide1object.

```
BenchPerf = PercentChange(PriceSeriesProvide1.Close,LookBack);
```

The relative performance is the difference between the percent change for both symbols and is calculated.

```
Relperf = (SymbolPerf-BenchPerf);
```

Finally, the calculated values are plotted. The reference values are multiplied by 100 to display as a percentage.

```
plot1(RelPerf * 100,"Rel Perf");
plot2(SymbolPerf * 100, "Sym Perf");
plot3(BenchPerf * 100, "Bench Perf");
```

Verify the Indicator.

Insert the indicator to a RadarScreen window with daily symbol intervals.

❖ COURSE EXAMPLE #4

Objectives: (RelStrengthFixed)

- ✓ Use multiple Price Series Provider components

Indicator: '$04_ RelStrengthFixed'

This indicator uses a pair of PriceSeriesProvider components to calculate the percent change of two symbols over a specified number of days.

Workspace: $04_RelStrengthFixed

Building the Chart:

Create: 30 min intraday chart
Insert Indicator: $04_RelStrengthFixed

Components and Properties Editor Settings:

PriceSeriesProvide1:

Symbol:	***iSymbol1***	(under category Filters) (add as an Input)
IntervalType:	*daily*	(under categories Filters-Interval-General)
IntervalSpan:	*1*	" "
Type	*years*	(under categories Filters-Range-General)
Years	*2*	" "

PriceSeriesProvide2:

Symbol:	***symbol***	(symbol in chart or row)
IntervalType:	*daily*	(under categories Filters-Interval-General)
IntervalSpan:	*1*	" "
Type	*years*	(under categories Filters-Range-General)
Years	*2*	" "

Indicator Exercise #4: '$04_ RelStrengthFixed'

This indicator is similar to the previous example, except this is designed to plot the relative performance on a chart using a pair of PriceSeriesProvider components that are both set to daily intervals. This allows you to compare the daily performance of the current symbol against the daily performance of a benchmark symbol, regardless of the interval setting of the chart.

In this example, let's start with the #03_DailyAvgClose indicator created in Course Example #3. Save a copy of it as #04_RelStrengthFixed. Again, we'll use the # naming convention to keep the course exercises separate from the course downloadable examples.

Since the PriceSeriesProvide1 component was already a part of the indicator you just created, you won't need to add it again. However, if you're starting from scratch, drag the PriceSeriesProvider component from the Toolbox into the document and set up the component Symbol, Interval, and Range values for *PriceSeriesProvide1* using the Property editor as specified under the Components and Properties Editor Settings for this exercise.

Next, we're going to add a second component from the Toolbox. Locate the *PriceSeriesProvider* component, then click and drag the component name into the code editor. A second component with the default name *PriceSeriesProvide2* will appear in the component tray at the bottom on the code editor window.

With the *PriceSeriesProvide2* component selected, go to the Properties Editor. Next to the Symbol property, type the word *symbol* to set it to the current symbol for the chart. Under the Filters category, expand open the Interval and Range sections so that you can set initial values for the PriceSeriesProvide1 component to specify the price data to collect. In the Interval section, change the IntervalType to *daily* and the IntervalSpan value to *1*. Then, under Range, change the Type to *years* and the Years value to *2*. The second component is now set to get daily price data for *symbol* for the last two years.

We'll be using the same input names as in Exercise #3; however, let's change the initial LookBack value to '8'.

```
Input: string iSymbol1( "SPY" );
Input: LookBack( 8 );
```

The variable names will also be based on those in Exercise #3, however, let's change the name of the current symbol performance variable to SymbolPerf.

```
Vars:  SymbolPerf( 0 ), BenchPerf( 0 ), RelPerf( 0 );
```

Calculate the percent change values for the base (current) symbol and for the benchmark (reference) symbol. We're going to place the performance calculations inside an "if" statement that will calculate the daily performance values on the first bar of each day. Remember, we're going to get the Close of the daily interval of the reference symbol from the second price series provider component we created so that the performance will always be calculated based on the daily interval of the two providers, even if the chart interval is different.

The current symbol performance is calculated using the collection of Close prices in the PriceSeriesProvide2 component object which is passed as a parameter to the PercentChange function along with the number of bars to look back when calculating the percent change. The benchmark symbol performance is calculated using the collection of Close prices in PriceSeriesProvide1 (based on the input iSymbol1) and the same PercentChange look back length.

```
If Date <> Date[1] then begin
   SymbolPerf = PercentChange(PriceSeriesProvide2.Close, LookBack) ;
   BenchPerf  = PercentChange(PriceSeriesProvide1.Close, LookBack);
end;
```

Calculate the relative performance difference between the base and reference percent change values.

```
RelPerf = SymbolPerf - BenchPerf;
```

Plot the three calculated values, this time removing the `*100` statement since we don't need the values to plot as percentages on a chart. Add a fourth plot for a zero reference line against which to compare the positive or negative performance values.

```
Plot1(SymbolPerf  , "Symbol Perf");
Plot2(BenchPerf   , "Bench Perf");
Plot3(RelPerf  , "Rel Perf");
Plot4(0);
```

Right-click in your Indicator document and select Properties at the bottom of the right-click menu to access the Indicator Properties dialog. Select the Scaling tab and verify that the Axis setting is *Scale On*: *Right Axis*.

Verify the Indicator.

Method

A *method* is a code structure containing EasyLanguage statements that are local to an EasyLanguage document. A method is referenced in your code very much like you would reference a function, but, because it is local, it runs faster and allows you to directly read or write to other variables in your analysis technique or strategy. Although methods, like functions, can accept input parameters and return a value, a void method allows you to execute a set of statements without requiring a return value which allows a method to appear by itself in your code, much like some reserved words, without needing to be assigned to a variable. The following is a summary of method features:

- Methods can return values
- Methods can have one or more parameters
- Methods can modify standard variables
- Methods can declare local variables that are only seen within the method
- Methods are typically faster than standard EasyLanguage functions

Methods are declared using the method reserved word followed by the return type, the name of the method, and any method parameters. Method parameters are surrounded by parentheses and separated by commas. The parentheses are required even if no parameters are specified. The body of the method contains a `begin/end` block containing the EasyLanguage statements preceded by any local variable declaration, if desired.

```
method void myMethod(int param1)
var: double localVar1, int localVar2
begin
    { EasyLanguage statements }
    localVar1 = myObject.Close[0];
    localVar2 = 10;
    if param1<localVar2 then
        plot1(localVar1);
end;
```

A return value of void indicates that the method does not return a value and empty parentheses would indicate that the method requires no parameters.

The code within an EasyLanguage method is executed when the method is called and not necessarily every time the analysis technique or strategy is evaluated. In the following example, the statements in the method above would be executed in your document when Condition1 is true. This is much like would be the case for an external function; however, the method is local to your document. Methods are commonly used when code needs to be called more than once in a document but are also useful for organizing EasyLanguage code into named modules. In addition, methods are used as event handlers and associated with event properties of an object.

```
Value1 = 5;
If Condition1=True then
    myMethod(Value1);
```

Local methods in an EasyLanguage document must always follow the EasyLanguage declarations for Inputs, Vars, and Arrays and come before the main body of code in your analysis technique.

Events

An *event* is a way for an object to provide notifications to the client program (your EasyLanguage code) when something of interest occurs within that object. One of the most familiar uses for events is in programming graphical controls, such as a button, where the program is notified when a user clicks a button. Another example is a timer control that notifies the calling program when a specified time counts down to zero. Here are some features of events:

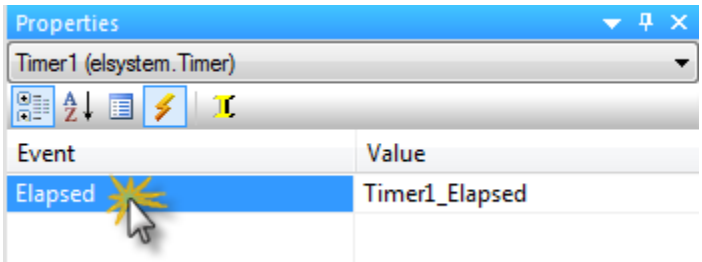
- Events listen for something to happen or change
- Events call an event handler method when triggered
- Code within an event handler method is executed to process an event
- Events are connected to event handler methods in the properties editor

Events, however, need not be used only for graphical interfaces or timers. Many data provider components in EasyLanguage also support events to notify you when data related to an object changes, such as updated prices, positions, or account status. In the past, you might have had to write a polling loop to look for changes in the incoming data, but with object events this can happen automatically by adding an event handler method to your analysis technique or strategy.

Event Handlers

In EasyLanguage, *event handlers* are methods in your EasyLanguage code that are called when a corresponding event occurs within an object. The event handler method is where you create your own EasyLanguage statements to program how your analysis technique, function, or strategy responds to the event.

For an event handler method to become useable, it must be associated with an event. With component objects this is done by associating the method with a named event using the Properties editor. For example, with a Timer component in your EasyLanguage document, the Event section of the Properties editor shows a blank Elapsed property.



Double-clicking on Elapsed will automatically create an event handler method in your EasyLanguage code and associate that method (Timer1_Elapsed in this case) with the Elapsed event. Now, every time the Elapsed event of Timer1 fires, the EasyLanguage code you add to the associated method will be executed. Event handler methods typically have a Sender and an Args parameter as shown. These parameters are used to receive information about the event and should not be edited by you.

```
method void Timer1_Elapsed( elsystem.Object sender,
elsystem.TimerElapsedEventArgs args )
begin
    { Insert your EasyLanguage statements below }
end;
```

For components, the event handler method name is associated with the event using the Properties Editor.

For non-component objects, you assign the name of the event handler method to the event property of an object in your code using the following syntax:

```
Object.EventName += Method_Name;
```

Designer Generated Code

In addition to the code that you write using the EasyLanguage editor, your analysis document also includes information about each component that is automatically programmed for you when a component is created and its properties edited. This information resides in a special *read-only* page referred to as Designer Generated Code that you access from the **View** menu of the code editor.

Designer Generated Code is created by the system and is updated through the Properties editor whenever you enter or change a property value for a component. It *cannot be directly modified* by you using the code editor.

When you create an Event using the Properties editor, such as the Elapsed event for a Timer object, an EasyLanguage statement is added to the Designer Generated Code block that assigns the name of the event handler method (that is located in the main section of your EasyLanguage code) to the component's event.

```
timer1.elapsed += timer1_elapsed;
```

In the above example, whenever a Timer1 component object triggers an Elapsed event, the method named 'timer1_elapsed' in your EasyLanguage code is called and the statements within the method are executed.

For now, it's not necessary that you understand the Designer Generated Code created by components, but it's an interesting resource to begin to develop an understanding of how EasyLanguage objects are created and manipulated.

Timer

The Timer component creates an object that lets you set a timer that counts down a specified number of milliseconds and calls an event handler method when the time expires. The timer can be set to automatically restart every time it expires so that the event handler can be called over and over again.

❖ COURSE EXAMPLE #5

Objectives: (Bar Countdown Indicator)

- ✓ Use a toolbox component to create a timer object
- ✓ Use the Properties editor to create an event
- ✓ Run some EasyLanguage code on the event notification

Indicator: '$05_BarPcntLeft'

This indicator uses a Timer component to set a timer that repeatedly updates the time of day every 1000 milliseconds. 1000 milliseconds = 1 second.

Workspace: $05_BarPcntLeft

Building the Chart:

Create: 1- minute chart
Insert Indicator: $05_BarPcntLeft

Components and Properties Editor Settings:

Timer1:

- Interval : *1000* (under category General)
- AutoReset: *True* " "
- Enable: *True* " "
- Elapsed: Timer1_Elapsed

Indicator Exercise #5: '$05_BarPcntLeft'

Create a new Indicator and name it: *'#05*_BarPcntLeft'. We will use the # naming convention to keep the course exercises separate from the course downloadable examples that begin with $.

Click on Toolbox tab at the left edge of the code editor window and drag the Timer component into the code editor. The default name *Timer1* will appear in the component tray at the bottom on the code editor window.

Now, click the Properties tab at the right edge of the code editor to open the Properties panel and make sure that *Timer1* is the specified component at the top of the Properties editor.

Under the General category of the Property  pane, set the Interval value to *1000*. This value is in milliseconds, so you're setting the timer to count down to zero in one second. For this example, we'll set the AutoReset property to *True* so that the timer will automatically reset and begin counting down again once a second. Also, set Enable to *True* so that the timer will begin running as soon as the analysis technique is applied to the analysis window.

Next, switch to the Event pane of the property editor by clicking the  icon. You will see Elapsed listed in the Event column with a blank value. The Elapsed event will be triggered by the Timer1 object whenever the timer counts down to zero where we'll want to call an event handler method in our EasyLanguage document. Simply double-click on the event name Elapsed to create a method in the code editor and have the new method name added to the Value column.

A method named Timer1_Elapsed has been added to your document. Remember, the parameters within the parentheses of the event handler are auto generated and should not be removed or changed by you. Your EasyLanguage code should be inserted between the begin and end statements.

```
method void Timer1_Elapsed( elsystem.Object sender,
   elsystem.TimerElapsedEventArgs args )
begin
   { Insert your EasyLanguage statements below }
end;
```

On the line above the Timer1_Elapsed method statement, type three variable declarations. The first of these will be an `intrabarpersist` variable the counts the number of elapsed timer events for the current bar. This is an intrabarpersist variable because it needs to retain each new value after every tick update instead of being reset the way a normal variable would be. The second variable holds the maximum counter value. The third variable holds the bar number of the last closed bar.

```
var: intrabarpersist iCounter(60),iMax(60),barnumb(0);
```

Note: Remember that the variable declaration statement always needs to be above any methods.

Between the `begin` and `end` statements in the Timer1_Elapsed method you are going to type one `if` statement that decrements the iCounter value for every timer event and another `if` statement that plots the bar counter histogram.

```
method void Timer1_Elapsed( elsystem.Object sender,
    elsystem.TimerElapsedEventArgs args )
begin
   If iCounter > 0 then
      iCounter = iCounter-1;
   If currentbar > barnumb then
      PlotValues();
end;
```

Add another method named PlotValues() that will be called when a timer event occurs. The first plot shows a histogram bar of the percentage of bar time remaining. The second and third plots show reference lines to keep the indicator the same size.

```
Method void PlotValues()
begin
   plot1((iCounter/iMax)*100, "PctLeft");
end;
```

The last section of code checks to see if the last bar on the chart has closed and resets the counter and maximum counter values based on the number of minutes in the bar interval. It also saves the closing bar number so that the next plot will occur only after the next bar tick.

```
Once If Bartype = 1 then
            iMax = Barinterval*60;

if LastBarOnChart then begin
      if BarStatus( 1 ) = 2 then begin
         plot1(0, "PctLeft");
         iCounter = iMax;
         barnumb = currentbar;
      end;
end;

   plot2(100, "100");
   plot3(0, "zero");
```

Verify the Indicator.

AccountsProvider

The AccountsProvider component creates an object that lets you reference values for your real or simulated TradeStation accounts.

The Accounts filter property, listed in the Properties editor for an AccountsProvider object, is used to specify which account, or accounts, you want to access data from. Using the filter criteria you specify, the AccountsProvider object builds a collection of accounts and sets the Count property to the number of elements in the collection. By default, a blank Accounts filter will collect account information from all active brokerage accounts.

The Updated event can be used with the AccountsProvider to allow your code to be notified when a value associated with any of the referenced accounts changes. This is especially important when an account changes for a symbol that is not in your chart or grid but that you want to know about in your analysis technique or strategy.

In your code, you'll typically use the Count property to determine if any accounts were found by the AccountsProvider before trying to access an indexed element from the Account collection.

The Account property of AccountsProvider1 represents a collection of one or more accounts (actually Account objects), along with a set of properties that can be read from each account. For example, you would reference the first account in the collection by appending the square bracket index identifier, in this case [0], after the Account property of the provider component.

```
AccountsProvider1.Account[0]
```

Once the specific Account is referenced, you add the property of the account you're interested in by including a '.' and property name after the indexed Account[0] identifier, such as .BDAccountNetWorth for the beginning-day net worth of the first account in the collection.

```
Value1 = AccountsProvider1.Account[0].BDAccountNetWorth;
```

If you specify just a single account number for the Accounts property in the Properties editor, your collection will include data for Account[0]. However, if you requested data for three accounts (such as "SIM12345,SIM67890, SIM1357X"), the second account would be referenced as Account[1] and the third would be Account[2]. The following represents a statement that reads the value of the *Unsettled* funds property of Account[2] (third account zero based) from a provider containing as least three accounts.

```
Value1 = AccountsProvider1.Account[2].UnsettledFund;
```

Collections

Component objects often manipulate collections of items. For example, the Accounts Provider object manages a collection of all your accounts that you can access (specified by the Accounts filter).

The following statement displays the account number of the first account (0^{th} index) in the collection:

```
Plot1(AccountsProvider1.Account[0].AccountID);
```

The Count property tells you how many items are in the object collection list in order to navigate through the collection list. If count is 0, then no items are in the collection list.

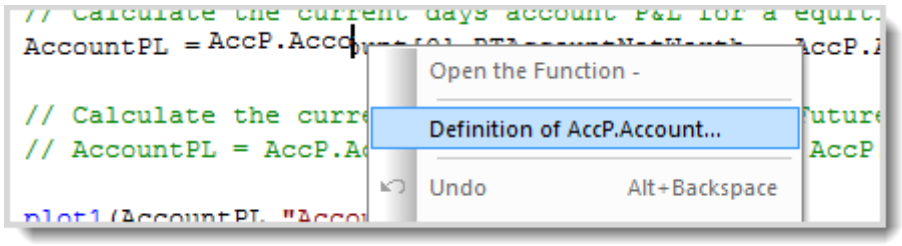
```
Value1 = AccountsProvider1.Count;
```

The Accounts filter property, listed in the Properties editor of an Accounts Provider object, is used to specify which account, or accounts, you want to access data from. Using the filter criteria you specify, the Accounts Provider object builds a collection of accounts and sets the Count property to the number of elements in the collection. By default, a blank Accounts filter will collect account information from all your active brokerage accounts.

Looking Up Definitions and Help

There are several ways of finding out more about the properties and methods for a component. For example in the Properties Editors, when you select a component property, the lower panel of the editor displays a brief description of the property from the dictionary. Just above the description pane is a link that reads Help about *Property* that will take you to the Help topic for the class the component is based on. Help topics typically include a list of properties, methods, and events for a given class in addition to links to related classes and properties.

When in the code editor, you can access Help for an object by right-clicking on the object reference in the code and then selecting **Definition of *object*…** from the right-click menu. This will take you directly to the Help topic for the object class.



By the way, if you right-click on a property name, the **Definition of *property*…** menu item will take you to the class containing that property. However, be aware than a property may be in a different class from the component since many properties are inherited from other classes and their definitions will be contained in the parent class. An example of this would be the .RTAccountNetWorth property that is accessed from the AccountsProvider component but is really a member of the Account class that is associated with the Account[0] property of the component.

❖ COURSE EXAMPLE #6

Objectives: (Account Profit & Loss Indicator)

- ✓ Create an object that can access your equities account information
- ✓ Use the Count property to see if you have an Account
- ✓ Rename provider to a shorter name

Indicator: '$06_AccountPL'

This indicator uses an AccountsProvider component to access account values for a specified equities account. The difference between the account's Beginning-Day Net Worth and Real-Time Net Worth property values is calculated and plotted as a histogram. The color of the plot is green if the account P&L is greater than 0 and is red if the net worth difference is less than 0.

Workspace: $06_AccountPL

Building the Chart:

Create: 5- minute chart with 5 days of price data
Insert Indicator: $06_AccountPL

Components and Properties Editor Settings:

AccP:

Name:	Acc*P*	(changes component name)
Accounts:	***iAccounts1***	(add as an Input)
Updated:	AccP_Updated	

Indicator Exercise #6: '$06_AccountPL'

Create a new Indicator and name it: *'#06*_AccountPL. We will use the # naming convention to keep the course exercises separate from the course downloadable examples that begin with $.

Click on Toolbox tab at the left edge of the code editor window and drag the AccountsProvider component into the code editor. The default name *AccountsProvider1* will appear in the component tray at the bottom on the code editor window.

Now, click the Properties tab at the right edge of the code editor to open the Properties panel and make sure that *AccountsProvider1* is the specified component at the top of the Properties editor.

At the top of the Properties editor, select *AccountsProvider1*, then locate the **Name** property and change its value to *AccP*. This shortens the name of the component object so that it requires less typing when writing code. However, the characteristics of the component object and its properties remain unchanged. Notice that the changed name now appears in the component tray at the bottom of the code editor.

Next, click on the empty text box to the right of the property name **Accounts**, followed by clicking on the  icon at the top of the Properties editor. The word iAccounts1 should appear next to Accounts in the Properties editor. In addition, you will see the following line was inserted at the top of your EasyLanguage code.

```
Input: string iAccounts1( "" );
```

Between the quotes, type the number of one of your simulated accounts that will be used for the *AccP* component object. When you request an account property for *AccP* this is the account number that it will reference.

```
Input: string iAccounts1( "SIM00000" );
```

Let's declare another input to specify the account P&L alert plot lines.

```
Input: PLAlert( 500 );
```

We'll also declare a variable to hold the calculated account profit prior to plotting.

```
var: AccountPL( 0 );
```

Next, switch to the Event pane of the Properties editor by clicking the  icon. You will see Updated listed in the Event column with a blank Value. The Updated event will be triggered by the AccountsProvider1 object whenever any value in your account has changed and will call an event handler method in our EasyLanguage document. Simply double-click on the event name Updated to create the new event handler method name.

An event handler method named AccP_Updated is automatically added to your document.

```
method void AccP_Updated( elsystem.Object sender,
elsystem.TimerElapsedEventArgs args )
begin
   { Insert your EasyLanguage statements below }
end;
```

Replace the auto generated comment after the `begin` statement with a call to a method named `PlotValues()`.

```
method void AccP_Updated( elsystem.Object sender,
   tsdata.trading.AccountUpdatedEventArgs args )
begin
   PlotValues();
end;
```

Now, create a method that will calculate and plot the account P&L every time an account updated event occurs. It will only plot on the last bar on a chart and when the provider has collected data for the specified account.

```
Method void PlotValues()
begin
   If LastBarOnChart and AccP.Count > 0 then begin
      AccountPL = AccP.Account[0].RTAccountNetWorth -
        AccP.Account[0].BDAccountNetWorth;
      plot1(AccountPL,"AccountPL");
```

The next set of statements sets the color of the net worth difference plot to light grey if it is zero, or to green or red based on a positive or negative account profit value.

```
      setplotcolor(1,LightGray);
      If AccountPL > 0 then
         setplotcolor(1,green);
      If AccountPL < 0 then
         setplotcolor(1,red);
   end;
end;

plot2(0,"Zero");
plot3(PLAlert);
plot4(-PLAlert);
```

Change the AccountPL plot to *Histogram* on the Chart Style tab of the document properties.

Verify the Indicator.

Note: The code for calculating the account P&L of a Futures or Forex Account are:

```
{ AccountPL = AccP.Account[0].RTAccountEquity -
AccP.Account[0].BDAccountEquity; }
```

When you apply the indictor to a chart, you should review the input settings and change the account number and profit alert value as appropriate for the symbol on your chart. For example, if you are charting an equities symbol, you would change the iAccounts1 input to your simulated equities account and change the profit alert based on the current net worth of your account.

Filter Properties (TokenList)

In many cases, a property included in the Filters section of the Properties editor expects a string containing one or more comma separated values, also known as a Token List.

For example, the PositionsProvider component includes three filter properties that are used to narrow the selection of possible positions for the provider. Let's say you wanted to gather position information for a specific group of symbols. Set the Symbol property to gather any positions in your account for MSFT, AAPL, and CSCO using string indicated:

```
Accounts
Symbols        "msft,aapl,csco"
Types
```

The PositionsProvider object will return your positions for the three symbols specified but not for any others in your account. In addition, the Types and Accounts properties were left blank to indicate that any type of position in any account for the three specified symbols will return results. However, you could just as easily add a list of specific accounts or position types and your results would be limited to include just those items that match all filter conditions. For example, the following filter settings would find just the long positions for the specified symbols in either of the two accounts.

```
Accounts       "SIM12345,SIM67890"
Symbols        "msft,aapl,csco"
Types          "long"
```

Instead of entering the name of a specific symbol, you could also type the reserved word `symbol` (without quotes) for the Symbols filter to return a position for just the current symbol in a chart or grid row, as below:

```
Symbols        symbol
```

+= Addition Assignment Operator

Along with objects, some new assignment operators have been added to EasyLanguage that are common to other object-oriented languages. One of these is an operator that adds the value of an expression to a variable without needing to repeat the variable name on both sides of the assignment statement. You can also say that that the expression 'adds to' the result.

For example, the following expression:

```
result += 1
```

is the same as writing

```
result = result + 1
```

except that *result* is only specified and evaluated once. Later, you will also see this operator used when an event handler name is assigned to, or added to, an event property.

PositionsProvider

The PositionsProvider component creates an object that lets you reference positions values for a specified symbol or list of symbols. You can also filter the results to include positions only for certain accounts or position types. Using the filter criteria you specify for Symbols, Accounts, and Types, the PositionsProvider builds a collection of positions that match your criteria and sets the Count property to the number of elements in the collection. If you don't specify any filter criteria, the PositionsProvider will build a collection that includes positions for all symbols, accounts, and types.

An Updated event associated with the PositionsProvider allows your code to be notified when a value associated with any of the referenced positions changes. This is especially important when a position changes for a symbol that is not in your chart or grid but that you want to know about in your analysis technique or strategy.

In your code, you'll typically use the Count property to determine if any positions were found by the PositionsProvider before trying to access an indexed element from the position collection.

For example, the following code displays some of the various position properties for the first position element in the Positions Provider collection:

```
Plot1(PosProvider1.Position[0].OpenPL, "P/L");
Plot2(PosProvider1.Position[0].Quantity, "PosSize");
Plot3(PosProvider1.Position[0].AveragePrice, "AvgPx");
Plot4(PosProvider1.Position[0].InitialMargin, "I Margin");
Plot5(PosProvider1.Position[0].RequiredMargin, "R Margin");
```

❖ COURSE EXAMPLE #7

Objectives: (Position Value Indicator)

- ✓ Use a toolbox component to create an object that reads your position information.
- ✓ Use the Properties editor to initialize component properties.
- ✓ Plot the value of a position held for a plotted symbol.

Indicator: '$07_PosValue'

This indicator uses a PositionsProvider component to access information about your position for a symbol in either RadarScreen or a chart. If you do not have a position, it will display '0'.

	Symbol	Interval	Last	Low	High	Volume Today	$07_PosValue		
							Mkt Value	Qty	P/L
1	SPY	5 Min	132.55	132.42	133.15	38,818,835	13,255	100	$57.00
2	CSCO	5 Min	15.68	15.58	15.77	12,060,641	1,568	100	$31.00
3	IBM	5 Min	183.25	183.01	184.42	1,984,059	91,625	500	$4,160.00
4	USO	5 Min	38.40	38.30	38.52	1,213,427			
5	GLD	5 Min	154.80	154.12	154.81	5,450,070	15,480	100	$393.99
6	AAPL	5 Min	387.18	386.47	396.27	16,204,528	38,718	100	$3,291.60
7	GE	5 Min	18.62	18.60	18.82	13,450,374			

Workspace: $07_PosValue

Building the RadarScreen window:

Create: any interval
Insert Indicator: $07_PosValue

Components and Properties Editor Settings:

PosP:

- Name: *PosP* (changes component name)
- Symbol: *symbol* (under category Filters)
- Updated: PosP_Updated

Indicator Exercise #7: '$07_PosValue'

Create a new Indicator and name it: *'#07_PosValue*.

Click on Toolbox tab at the left edge of the code editor window and drag the PositionsProvider component into the code editor. The default name *PositionsProvider1* will appear in the component tray at the bottom on the code editor window.

Now, click the Properties tab at the right edge of the code editor to open the Properties panel and make sure that *PositionsProvider1* is the specified component at the top of the Properties editor.

Change the Name property of the component to the shorter name *PosP*.

Under the Filters category, set type Symbol property to the reserved word *symbol* so that the provider only returns positions for the current symbol. Leave the Accounts and Types properties blank so that the provider looks for any position for the current *symbol* in all related accounts that you might have, such as different margin or cash accounts.

You will now be able to use the *PosP* component as an object in your code to access position values for the current symbol in a grid or chart.

Next, switch to the Event pane of the Properties editor by clicking the  icon. You will see Updated listed in the Event column with a blank Value. Double-click on the event name Updated to create an event handler method in your EasyLanguage document and have the new method name associated with the property.

A method named PosP_Updated has been added to your document. *Remember, the parameters within the parentheses of the event handler are auto generated and should not be removed or changed by you.*

```
method void PosP_Updated( elsystem.Object sender,
   tsdata.trading.PositionUpdatedEventArgs args )
begin
   PlotValues();
end;
```

Insert a call to PlotValues() between the `begin` and `end` statements of the PosP_Updated method.

This indicator requires no inputs or variables.

Next, create the PlotValues() method.

```
Method void PlotValues()
Begin
```

The first statement after `begin` is an if condition that reads the `Count` property of the component to see if any positions were found for the current symbol. If no position exists for the current symbol, the values for that symbol in a grid window will start out blank. If any positions do exist for the current symbol, then the indicator will plot three position values.

```
if (PosP.Count > 0) then
      begin
         Plot1(PosP.Position[0].MarketValue, "Mkt Value");
         Plot2(PosP.Position[0].Quantity, "Qty");
         Plot3(PosP.Position[0].OpenPL, "P/L");
```

The color of the Open P/L plot is changed to Green for a positive value and Red for a negative value.

```
            if PosP.Position[0].OpenPL >= 0 then
               SetPlotColor(3,Green)
            else
               SetPlotColor(3,Red);
         end
      else
         begin
```

If there is no position for the current symbol, the following code will display ‘0’ for the totals.

```
            Plot1(0, "Mkt Value");
            Plot2(0, "Qty");
            Plot3(0, "P/L");
         end;
   end;
```

Verify the Indicator.

This indicator is designed for a grid analysis window, such as RadarScreen, and will show any positions held for the symbols. However, it will also plot position values on a subgraph of a chart where the indictor status line is most useful for reading the Quantity, Market Value, and P/L numbers in that order.

Method Variables

In much the same way that you can declare variables in a function that can only be seen/used in that function, you can also declare variables within a method that are local to the method and are not accessible from the rest of the analysis technique. These local variables are created each time the method runs and go away after the code in the method has been executed. Local variables are useful when performing temporary calculations on values updated by event handlers or for values such as counters and indexes that are only needed within a method. You can mix local and analysis technique variables in a method and both can be included in calculations, however, only the value of the standard analysis technique variable will be saved when the method is exited.

A local variable declaration statement in a method is very similar to the standard variable declaration statement you already know how to use at the beginning of an analysis technique or strategy. However, there are two important differences: 1) a local variable statement must always specify the variable type preceding the variable name (int, double, bool, or string), and 2) you cannot specify an initial value for the local variable following its name. Also, because you can't rely on a local variable to have an initial value, it's a good idea to assign the local variable a value in the method code between the, begin and end, prior to using it in a calculation.

```
method void myMethod()
var: int myIndex, double priceTotal;
begin
    {EasyLanguage statements and calculations}
end;
```

Variable Types Review:

Int = Integer – A positive or negative number without decimal values
Double = Double Float – A positive or negative number with decimal values
Bool = Boolean – A TRUE or FALSE condition
String = Text – An alphanumeric series of characters

NumToStr(num,dec)

You can use the reserved word NumToStr to convert numeric property values to a string for plotting. This is most useful when you want to control the number of decimal places in the text version of the number.

For example, if you want to plot the LimitPrice property of the first order in an Order collection and only display two decimal places you use the NumToStr reserved word, referencing the numeric property value as the first argument and specifying the number of decimal places as the second argument.

```
Plot1(numtostr(ordersprovider1.Order[0].LimitPrice,2),"Limit");
```

Be aware that converting a numeric value to a string in a grid application means that the resulting value in that column will not sort numerically and that the decimal places settings will not apply.

❖ COURSE EXAMPLE #8

Objectives: (Position and Account Value Indicator)

- ✓ Combine different components
- ✓ Use toolbox components to create objects that read position and account information.
- ✓ Use the Properties editor to initialize component properties.

Indicator: '$08_PosValueAcct'

This indicator uses a PositionsProvider and Account Provider to calculate and displays the market value of an equities position as a percent of the overall account. If you do not have a position it will display '0'.

	Symbol	Interval	Low	High	Volume Today	$08_PosValueAcct			
						Mkt Value	Qty	P/L	% of Acct
1	SPY	5 Min	132.42	133.15	39,078,751	13,262	100	63.90	4.336 %
2	CSCO	5 Min	15.58	15.77	12,172,557	1,568	100	31.15	0.513 %
3	IBM	5 Min	183.01	184.42	1,992,024	91,630	500	4,165.00	29.960 %
4	USO	5 Min	38.30	38.52	1,223,117				
5	GLD	5 Min	154.12	154.81	5,474,550	15,480	100	394.00	5.061 %
6	AAPL	5 Min	386.47	396.27	16,261,089	38,722	100	3,295.60	12.660 %
7	GE	5 Min	18.60	18.82	13,581,523				

Workspace: $08_PosValueAcct

Building the RadarScreen window:

Create: any interval
Note: List symbols for which you want to monitor positions
Insert Indicator: $08_PosValueAcct

Components and Properties Editor Settings:

PosP:

- Name: *PosP* (changes component name)
- Symbol: *symbol* (under category Filters)
- Updated: PosP_Updated

AccP:

- Name: *AccP* (changes component name)

Indicator Exercise #8: '$08_PosValueAcct'

For this example, we'll start with the #07_PosValue indicator created in Course Example #7. Save a copy of it as #08_PosValueAcct.

Since the *PosP* component was already a part of the previous indicator, you won't need to add that again. However, if you're starting from scratch, you would drag the PositionsProvider component from the Toolbox into the document and set the value for *Symbol* and for the *Updated* event using the Property editor as specified under the Components and Properties Editor Settings above.

Click on Toolbox tab at the left edge of the code editor window and drag the AccountsProvider component into the code editor. The default name *AccountsProvider1* will appear in the component tray at the bottom on the code editor window.

Now, click the Properties tab at the right edge of the code editor to open the Properties panel and make sure that *AccountsProvider1* is the specified component at the top of the Properties editor.

Change the Name property of the component to the shorter name *AccP*.

As a reminder, the lines of code in grey are those copied from exercise #7.

```
method void PosP_Updated( elsystem.Object sender,
tsdata.trading.PositionUpdatedEventArgs args )
begin
   PlotValues();
end;
```

The first line of EasyLanguage code you'll add will be variable declaration in-between the method header and begin. You'll declare two local variable declarations that are only seen within the PlotValues method. Remember, local method variable declarations require that you specify a type and do not accept a default value.

The first variable is used to calculate the percent of the Account net worth that the current symbol position represents and the second will be used to hold the AccountID of the position.

```
Method void PlotValues()

var: double PcntOfAccount, string PosAccountID;

begin
   if (PosP.Count > 0) then
      begin
         Plot1(PosP.Position[0].MarketValue, "Mkt Value");
         Plot2(PosP.Position[0].Quantity, "Qty");
         Plot3(PosP.Position[0].OpenPL, "P/L");

         if PosP.Position[0].OpenPL>=0 then
            SetPlotColor(3,Green)
         else
            SetPlotColor(3,Red);
```

The following code first checks to see if any accounts are available by checking the Count property of the AccP component and then calculates and displays the market value of the position as a percent of the overall account. Notice that we get the account number (ID) for the current position from the Position provider and then use that to reference the RT net worth value.

```
            PosAccountID = PosP.Position[0].AccountID;
            If AccP.Count>0 then begin
               Value1 = PosP.Position[0].MarketValue;
               Value2 = AccP.Account[PosAccountID].RTAccountNetWorth;
               PcntOfAccount = Value1/Value2;
               Plot4(numtostr(PcntOfAccount*100,3)+" %","% of Acct");
            end;
      end
else
      begin
            Plot1(0, "Mkt Value");
            Plot2(0, "Qty");
            Plot3(0, "P/L");
            Plot4("0","% of Acct");
      end;
end;
```

Verify the Indicator.

This indicator is designed for a grid analysis window, such as RadarScreen, and will show any positions held for symbols along with the percentage of the specified account that each position represents based on the market value.

ToString()

The toString() method is a handy way to convert a value of an object property to a string. This is often used so that you can display the numeric or date value of a property using a plot or print statement without needing to worry about the property type.

For example, the following plot statement will generate a run-time error because Plot1 doesn't know how to display the referenced CurrentTime property that is DateTime object type:

```
Plot1(elsystem.datetime.CurrentTime);
```

However, adding `.toString()` to the end of the property reference will automatically convert the CurrentTime property to a displayable string that can be plotted with a plot or print statement.

```
Plot1(elsystem.datetime.CurrentTime.toString());
```

OrdersProvider

The OrdersProvider allows access to both real-time and historical order information based on user-specified filters from a list of specified TradeStation brokerage accounts. This information is similar to that found on the Orders tab of the TradeManager. The Order class describes the properties available for a specific order (see below).

You can also filter the results to include orders only for certain accounts, date ranges, order ID numbers, and order states (e.g. received, filled, rejected, etc.). Using the filter criteria you specify for Symbols, Accounts, From/To dates, and Order States, the OrdersProvider builds a collection of Order objects that match your criteria and sets the Count property to the number of elements in the collection. If you don't specify any filter criteria, the OrdersProvider will build a collection that includes orders for all symbols, accounts, ranges, and states.

An Updated event associated with the OrdersProvider allows your code to be notified when a value associated with any of the referenced order changes. For example, any change in the status of an order managed by a specific instance of your OrdersProvider will fire a notification event so that you can plot the changed status. This is also useful when an order changes for a symbol that is not in your chart or grid but that you want to know about in your analysis technique or strategy.

It's always a good idea to use the Count property in your code to determine if any orders were found by the OrdersProvider before trying to access an indexed element from the Order collection.

For example, the following displays some of the various Orders properties for the first order element in the Orders Provider collection:

```
Plot1(OrdersProvider1.Order[0].AvgFilledPrice, "AvgFillPx");
Plot2(OrdersProvider1.Order[0].FilledQuantity, "FillQTY");
Plot3(OrdersProvider1.Order[0].State.ToString(), "State");
Plot4(OrdersProvider1.Order[0].LimitPrice, "LimitPx");
Plot5(OrdersProvider1.Order[0].Duration, "Duration");
```

Note: The orders collection is sorted chronologically, from 0 (newest) to oldest.

❖ COURSE EXAMPLE #9

Objectives: (Order Status Indicator)

- ✓ Use a toolbox component to create an object that reads your order status information.
- ✓ Use the Properties editor to initialize component properties.
- ✓ Plot the quantity, order type, and status (state) of an order for a plotted symbol.

Indicator: '$09_OrderStatus'

This indicator uses an OrdersProvider component to access information about the status of the last order submitted for a symbol in either a RadarScreen grid or a chart. If you do not have any order activity for a symbol, the RadarScreen row is blank.

	Symbol	Interval	Last	Low	High	Volume Today	$09_OrderStatus			
							Quantity	Type	Limit Price	State
1	MSFT	5 Min	26.03	0.00	0.00	33,537				
2	CSCO	5 Min	18.38	0.00	0.00	128,941	100	limit	18.45	received
3	IBM	5 Min	161.51	0.00	0.00	1,700				
4	USO	5 Min	42.69	0.00	0.00	887,937	500	limit	43.07	filled
5	GLD	5 Min	140.16	0.00	0.00	880,030				
6	EURUSD	5 Min	1.39961	1.39549	1.40356	0				
7	@QM(D)	5 Min	105.275	104.250	106.950	8,051				

Workspace: $09_OrderStatus

Building the RadarScreen window:

Create: any interval
Insert Indicator: $09_OrderStatus

Components and Properties Editor Settings:

OrdersProvider1:

- Symbols: *symbol* (under category Filters)
- Updated: OrdersProvider1_Updated

Indicator Exercise #9: '$09_OrderStatus'

Create a new Indicator and name it: *'#OrderStatus*.

Click on Toolbox tab at the left edge of the code editor window and drag the OrdersProvider component into the code editor. The default name *OrdersProvider1* will appear in the component tray at the bottom on the code editor window. Now, click the Properties tab at the right edge of the code editor to open the Properties panel and make sure that *OrdersProvider1* is the specified component at the top of the Properties editor. Change the Name property of the component to the shorter name *OrdP*

Under the Filters category, set type Symbol property to the reserved word *symbol* so that the provider only returns order status information for the current symbol. Leave the Accounts, Status, and Orders properties blank so that the provider looks for any order for the current *symbol* in all of your accounts. You will now be able to use the OrdP component as an object in your code to access position values for the current symbol in a grid or chart.

Next, switch to the Event pane of the property editor by clicking the ⚡ icon. You will see Updated listed in the Event column with a blank Value. Double-click on the event name Updated to create an event handler method in your EasyLanguage document and have the new method name associated with the property.

A method named OrdP_Updated has been added to your document. As previously mentioned, the parameters within the parentheses of the event handler are auto-generated and should not be removed or changed by you.

Insert a method call to `PlotValues()`.

```
method void OrdP_Updated( elsystem.Object sender,
   tsdata.trading.OrderUpdatedEventArgs args )
begin
   PlotValues();
end;
```

Now, create the following that includes a method that will plot selected order status properties from `Order[0]` that represents the first, and most recent, order in the collection. The `.Count` property indicates the actual number of order status items for the current symbol, but, in this case, we're only interested in the first one. Also, notice the use of the `.tostring()` method after the property name of several plots which causes those non-text values to display as a string on the chart status line if inserted into charting.

```
Method void PlotValues() begin
     If OrdP.Count>0 then begin
        plot1(OrdP.Order[0].EnteredQuantity,"Quantity");
        plot2(OrdP.Order[0].type.tostring(),"Type");
        plot3(OrdP.Order[0].LimitPrice,"Limit Price");
        plot4(OrdP.Order[0].state.tostring(),"State");
     end;
end;
PlotValues();
```

Verify the Indicator.

This indicator is designed for a grid analysis window, such as RadarScreen, and will show any orders placed for symbols.

LastBarOnChart

The LastBarOnChart function is used to determine if the current bar being evaluated is the last bar on the chart. This is useful when writing conditions that need to create and send orders so that they are placed only on real-time price conditions and not based on historical price conditions.

```
If LastBarOnChart and OrderCondition=true then
    Orderticket1.send();
```

In this example, a true OrderCondition will only send an order on the current bar.

Analysis Technique – Initialized and Uninitialized Events

When an analysis technique or strategy is applied to your chart or gird application, it can automatically trigger an Initialized event that will call a specified event handler method in your EasyLanguage document when it first runs. In a similar manner, an Uninitialized event can be triggered when an analysis technique ends, such as when it is removed from a chart or when platform is shut down. This is very useful when you want to set up some conditions in your analysis technique that will be executed one time when the technique first runs, just before the rest of the analysis technique code runs on the bars of a chart or grid window, or as the last thing when an analysis technique shuts down. You can add an Initialized or Uninitialized event to any analysis technique by selecting *Analysis Technique* s from the drop-down list at the top of the Properties Editor and then switching to the events tab by clicking on the ⚡ icon. In the Event column you will find a pair of events, Initialized and Uninitialized. Double-clicking on either name will automatically create an event handler method in your EasyLanguage document named *AnalysisTechnique_Initialized* or *AnalysisTechnique_UnInitialized* and will insert a reference to that method in the Value column in the Properties Editor. Simply add your own EasyLanguage statements to the new event method and they will be executed once every time the analysis technique or strategy is loaded or unloaded as appropriate.

IntrabarPersist

IntrabarPersist is a variable declaration keyword that is used to create an EasyLanguage variable or array that can store and update values tick by tick. This is especially useful when you want to count intrabar ticks or to save the status of a condition that may change within a bar. By default, the value of a variable is saved at the close of each bar.

```
Var: IntrabarPersist tickcount(0);
tickcount = tickcount + 1;
```

In this example, the variable tickcount is able to count the total number of ticks on each bar.

OrderTicket

The OrderTicket component generates an order ticket for a specified symbol and sends the order directly to the market directly from your EasyLanguage analysis technique or strategy. OrderTicket objects support most of the order parameters that are available when manually placing an order from the TradeStation Order Bar. Just as with manual orders, the status of all orders created from an OrderTicket will appear on the Orders tab of the TradeManager along with other regular orders.

Setting up an OrderTicket component typically follows the same steps as with any of the previous components. You first use the properties editor to fill in your order values. Some properties, such as the Order type and Duration are set to default values. Others, such as the Symbol, Symbol Type (asset type), Account, Quantity, and market Action (such as buy, sell, etc.) must be entered by you to create a valid order. When all of the properties are set, you add a statement in your EasyLanguage code to place the order using the component object's Send() method, as follows:

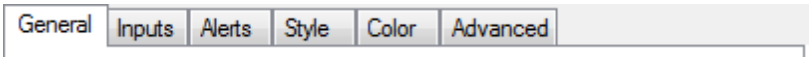
```
OrderTicket1.Send() ;
```

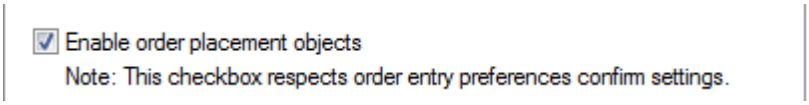
In addition to using the properties editor, you can also set the OrderTicket properties directly in your EasyLanguage code for those cases where you want to programmatically determine the symbol, quantity, order action, etc. prior to sending the order. In either case, calling the Send() method for the specified component submits the order.

```
OrderTicket1.Symbol = "XYZ"
OrderTicket1.Quantity = 200;
OrderTicket1.Action = tsdata.trading.orderaction.Buy;
OrderTicket1.Send() ;
```

Enable Order Placement Objects

To avoid accidental placement of orders from an analysis technique or strategy that contains an OrderTicket, or other order component, you must specifically enable order placement for each analysis technique or strategy that will be sending orders. This is done by going to the Format – General tab of each technique or strategy and checking the *Enable order placement objects* setting (midway down the tab) before orders can be generated from EasyLanguage.





Note: A run-time error will occur if this is not checked and the indicator attempts to send an order.

❖ COURSE EXAMPLE #10

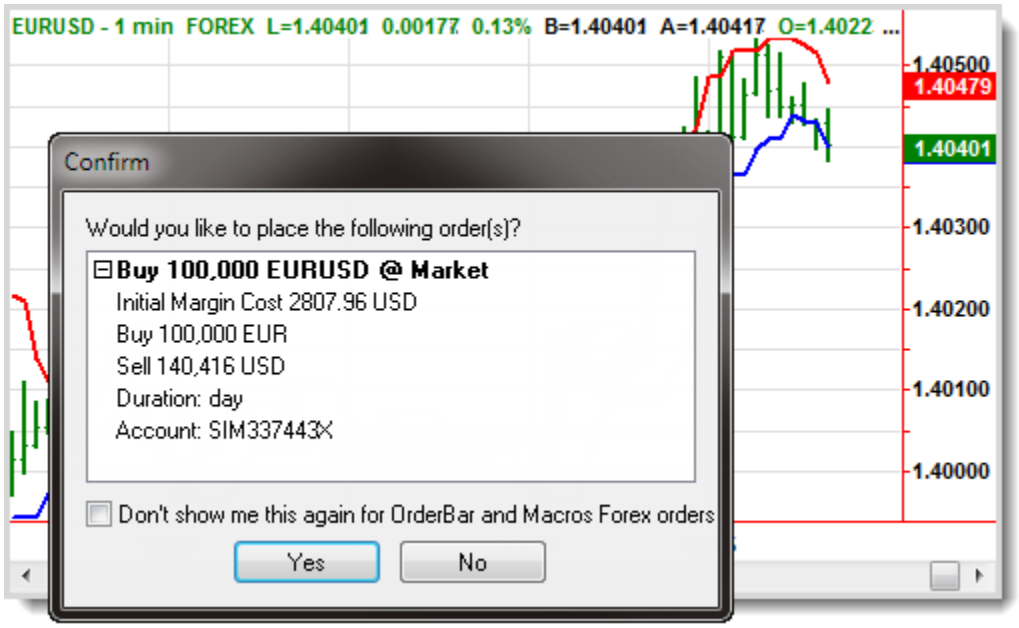
Objectives: (Market Order Indicator)

- ✓ Create an order ticket
- ✓ Send the order only once on the last bar
- ✓ Enable the order placement objects setting for the indicator

Indicator: '$10_MarketOrder'

This example uses an OrderTicket component to place a Buy market order based on a simple low or high price target order rule. Once an order is sent, the order active status flag is set to false. Reloading the indicator resets the Indicator to place another order.

WARNING: This indicator will generate a market order – make sure you are NOT logged onto your real-money account. This should only be applied when logged into TradeStation simulator.



Workspace: $10_MarketOrder

Building the Chart:

Create: 1-minute chart
Scaling: Same axis as underlying data
Insert Indicator: $10_MarketOrder

Components and Properties Editor Settings:

OrderTicket1:

Symbol:	*symbol*	(type reserved word)
Account:	***iAccounts1***	(add default Account as an Input)
Quantity*:*	***iQuantity1***	(add desired Quantity as an Input)
Action*:*	*Buy*	
Type*:*	Market	

Analysis Technique:

Initialized:	AnalysisTechnique_Initialized

Indicator Exercise #10: '$10_MarketOrder'

Create a new Indicator and name it: *'#10_MarketOrder*.

Click on Toolbox tab at the left edge of the code editor window and drag the OrderTicket component into the code editor. The default name *OrderTicket1* will appear in the component tray at the bottom on the code editor window.

Now, click the Properties tab at the right edge of the code editor to open the Properties panel and make sure that *OrderTicket 1* is the specified component at the top of the Properties editor.

Under the Filters category, set the Symbol property to the reserved word *symbol* so that the order will be placed for the current symbol.

As we've done in a previous example, we're going to make the account number an input. Next to the property name **Account**, enter your simulated Forex account number. Then, click on the  icon at the top of the Properties editor to make it an input. The word iAccount1 should appear next to Account in the Properties editor and you will see the following line inserted at the top of your EasyLanguage code. This input will be the default account for the *OrderTicket1* component. For the next several examples, double check that you are using simulated account numbers when generating orders from these examples.

```
Input: string iAccount1( "SIM00000" );
```

Still in the properties editor, enter the default **Quantity** of 100000 for this simulated Forex order. As before, click the  icon at the top of the Properties editor to make an input named iQuantity1 which will also be added to your code.

```
Input: int iQuantity1( 100000 );
```

While in the Properties editor, use the drop-down selector at the top of the editor to select *Analysis Technique*. Switch to the Event pane of the property editor by clicking the  icon. You will see Initialized listed in the Event column with a blank Value. Double-click on the event name Initialized to create an event handler method in your EasyLanguage document and have the new method name associated with the property. We'll be adding code to this method shortly.

```
method void AnalysisTechnique_Initialized( elsystem.Object sender,
  elsystem.InitializedEventArgs args )
begin
   { Insert your EasyLanguage statements below }
end;
```

You will now be able to use the OrderTicket1 component as an object in your code to send a market order for the current symbol in a grid or chart.

In your EasyLanguage code, let's add one more input that will represent whether order placement is active when the indicator first runs.

```
Input: OrderActive(True);
```

Next we'll declare variables for the high price target and the low price target that will be used in the order rules that generate an order. Also, we'll create an intrabarpersist variable that will allow us to control when the order rules are active and allow an order to be sent only once.

```
Vars: double HiTarget( 0 ), double LoTarget ( 0 ),
      intrabarpersist AllowTradeFlag( True );
```

In the AnalysisTechnique_Initialized method, we'll first set the asset type of symbol for which we're placing the order to the asset type (Stock, Futures, Forex) of the current symbol using the Category reserved word. Then we'll set the initial value of the AllowTradeFlag variable to the value of input OrderActive and later reset AllowTradeFlag so that an order only gets placed once.

```
method void AnalysisTechnique_Initialized( elsystem.Object sender,
     elsystem.InitializedEventArgs args )
begin
   OrderTicket1.SymbolType = Category;
   AllowTradeFlag = OrderActive;
end;
```

Create a method that will plot values for the indicator. Note that the plot statements in this method are executed only when the PlotValues() method is called later in your code.

```
Method void PlotValues() begin
   Plot1(HiTarget,"High Target");
   Plot2(LoTarget,"Low Target");
   Plot3(AllowTradeFlag.tostring(),"Active");
end;
```

In this example, the high and low price targets are calculated based on the highest or lowest price over the past three bar intervals. This is so that our example gets order condition soon after applying the indicator.

```
HiTarget = HighestFC(High, 3)[1];
LoTarget = LowestFC(Low, 3)[1];
```

Add the following two statements to plot values on each tick of a chart.

```
PlotValues();
```

With the next piece of code, the order condition will be executed when the current price hits the high or low price target and when the AllowTradeFlag variable is True on the last bar of the chart. The statement after `begin` uses the OrderTicket1.send() method to place the order. The next statement resets the AllowTradeFlag variable to False so that no other orders are sent until the chart is refreshed.

```
If ( ( Close <= LoTarget OR Close >= HiTarget ) AND
     ( LastBarOnChart AND AllowTradeFlag ) ) then begin
   OrderTicket1.send();
   AllowTradeFlag = False;
end;
```

Verify the Indicator.

Go to a Chart and insert the indicator.

From the chart, go to the **Format #10_MarketOrder** dialog and select the **General** tab. The check box labeled **Enable order placement objects** needs to be checked before orders can be generated from your EasyLanguage code. This setting only retains its setting as long as the indicator is applied to the Chart. If you remove and reapply the indicator you will have to enable the check box again.

Declaring an Object Variable

Up to this point, the objects we've been using were automatically created for us when we dragged a component into your document. As we saw earlier, the EasyLanguage code used to create and setup component objects has been attached to your analysis technique in a hidden code block called Designer Generated Code that you can access from the **View** menu in code editor.

For example, let's look the Designer Generate Code for the $10_MarketOrder example that uses an OrderTicket component. Here are the first couple of lines:

```
{ Components declaration (Designer generated code) }
    var: tsdata.trading.OrderTicket OrderTicket1( NULL );
```

This contains a declaration for a variable of type tsdata.trading.OrderTicket named OrderTicket1 with a default value of 'NULL'. This name should look familiar since it's the name of the component object we setup in the properties editor and called to send the order. Each object variable, whether created for you in the Designer Generated Code, or added by you directly in your analysis technique code appear much the same with the object type followed by the object name and an initial value of 'NULL':

In the next example, we'll be tracking the status of an order using an Order object. The following is a variable declaration of an object variable that will let you track an Order:

```
    Var: tsdata.trading.Order myOrder(null);
```

Later in this course, you will declare other object variables that will allow you to create and reference your own non-component objects.

Tracking the Order Status of an Order Ticket

To track the status of the order, you save the sent order instance to an Order object variable and create an event handler method to automatically notify you of changes in the status of that specific Order. The following describes the steps needed to track the status of a specific order that you send with an OrderTicket:

```
{ declare an Object variable named myOrder to hold an instance of an Order object }
    Var: tsdata.trading.Order myOrder(null);

{ method to handle Order update events }
    Method void OrderStatusUpdate(elsystem.Object sender,
      tsdata.trading.OrderUpdatedEventArgs args)
    begin
        Plot2(myOrder.State);
    end;

{ copy the sent order to the object variable myOrder as an object instance }
    myOrder = OrderTicket1.Send();

{ associate the event handler method with the myOrder object  }
    myOrder.Updated += OrderStatusUpdate;
```

BracketOrderTicket

The BracketOrderTicket component generates a bracket OCO order that consists of two of the same order actions (buy, sell) for a specified symbol, and sends the combined OCO order to the market directly from your EasyLanguage analysis technique. BracketOrderTicket objects support most of the order parameters that are available when manually placing an order using the OCO button of the TradeStation Order Bar.

Setting up a BracketOrderTicket component typically follows the same steps as with any of the previous components. You first use the properties editor to fill in the basic order values, such as the Symbol, Symbol Type (asset type), Account, Quantity, and Order Action (such as buy, sell, etc.), that are required to create a valid order. In addition, you will need to specify the TargetType and ProtectionType limit and stop prices.

When all of the properties are set, you add a reference to the BracketOrderTicket component in your EasyLanguage code to place the order using the Send() method.

Just as with manual orders, the status of all orders created from a BracketOrderTicket will appear on the Orders tab of the TradeManager along with other regular orders.

```
BracketOrderTicket1.Send() ;
```

Instead of using the properties editor, you can also set the BracketOrderTicket properties directly in your EasyLanguage code for those cases where you want to programmatically determine the symbol, quantity, order action, stop/limit prices, etc. prior to sending the order. In either case, calling the Send() method for the specified component submits the order.

```
BracketOrderTicket 1.Symbol = "XYZ"
BracketOrderTicket 1.Quantity = 100;
BracketOrderTicket1.LimitPrice = myOrder.AvgFilledPrice + .05);
BracketOrderTicket1.StopPrice = myOrder.AvgFilledPrice - .05;
BracketOrderTicket 1.Action = tsdata.trading.orderaction.Sell;
BracketOrderTicket 1.Send() ;
```

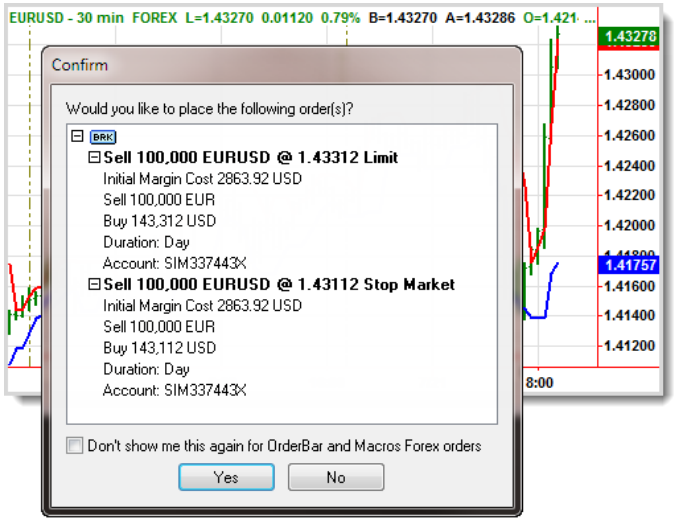
❖ COURSE EXAMPLE #11

Objectives: (OSO Bracket Order Indicator)

- ✓ Create a bracket order ticket
- ✓ Associate and track an order ticket with an Order object
- ✓ Create an event handler method to track the Order status
- ✓ Use the print log to track the status of the market order until filled.

Indicator: '$11_OSOBracketOrder'

This indicator uses an OrderTicket and a BracketOrderTicket component to send a buy at market order based on a specified price target and, when the order is filled, sends a Sell bracket OCO order to set a profit target and protective stop target. The order status is displayed in the print log.



Workspace: $11_OSOBracketOrder

Building the Chart window:

Create: 1-minute interval
Scaling: Same axis as underlying data
Insert Indicator: $11_OSOBracketOrder

Components and Properties Editor Settings:

OrderTicket1:

- Symbol: *symbol* (under category Filters)
- Accounts: ***iAccounts1*** (add as an Input)
- Quantity: ***iQuantity1*** (base quantity on Input)
- Action: *Buy*

BracketOrderTicket1:

- Symbol: *symbol* (under category Filters)
- Accounts: ***iAccounts1*** (add as an Input)
- Quantity: ***iQuantity1*** (base quantity on Input)
- Action: *Sell*
- TargetType: *Limit*
- ProtectionType: *StopMarket*

Analysis Technique:

- Initialized: AnalysisTechnique_Initialized

Indicator Exercise #11: '$11_OSOBracketOrder'

For this example, we'll start with the #10_MarketOrder indicator created in Course Example #10. Save a copy of it as #11_OSOBracketOrder.

Since the OrderTicket1 component was already a part of the indicator you previously created, you won't need to add that again. However, if you're starting from scratch, you would drag the OrderTicket component from the Toolbox into the document and set up the component Symbol, Symbol Type, Accounts, Quantity, Action, and Type values using the Property editor as specified under the Components and Properties Editor Settings above.

Click on Toolbox tab at the left edge of the code editor window and drag the BracketOrderTicket component into the code editor. The default name *BracketOrderTicket1* will appear in the component tray at the bottom on the code editor window.

Now, click the Properties tab at the right edge of the code editor to open the Properties panel and make sure that *BracketOrderTicket1* is the specified component at the top of the Properties editor.

Under the Filters category, set the Symbol property to the reserved word *symbol* so that the bracket order will be placed for the current symbol.

Click the empty text box next to the **Account** property, then click the down arrow to the right of the box and select input iAccount1 from the drop-down list. This is the same simulated Forex account number we are using for the market order ticket. Also, do the same with the **Quantity** property by clicking the arrow to the right of the empty text box and selecting iQuantity.

Still in the Properties editor, set the **Action** property to *Sell*. Then set **TargetType** to *Limit* and **ProtectionType** to *StopMarket* to indicate the type of orders we'll placing above and below the market when we place the bracket Sell order.

Back in our EasyLanguage code, after the previously declared inputs add a new input for the bracket amount that will used to set the prices for the upper and lower bracket orders.

```
Input: int iQuantity1( 100000 );
Input: string iAccount1( "SIM00000" );
Input: OrderActive(TRUE);
Input: BracketAmt(.001);
```

Next, declare a new intrabarpersist variable that will tell us when a bracket order has been sent so that the bracket order is only generated once.

```
Vars:    double HiTarget( 0 ), double LoTarget ( 0 ),
         intrabarpersist AllowTradeFlag( True ) ,
         intrabarpersist BracketSent( FALSE );
```

Also, declare an order object variable that will be used to reference the market order object that is created when the market order is sent.

```
Var:     tsdata.trading.Order MyOrder(null);
```

In the `AnalysisTechnique_Initialized` method, add a statement that sets the symbol type of the bracket order to the asset type of the current symbol just like we did in the previous example for the market order. Remember, the *AnalysisTechnique_Initialized* method is automatically created when you double-click on the event name Initialized in the Properties editor for the Analysis Technique (refer to example 10) .

```
method void AnalysisTechnique_Initialized( elsystem.Object sender,
    elsystem.InitializedEventArgs args )
begin
   OrderTicket1.SymbolType = Category;
   BracketOrderTicket1.SymbolType = Category;
   AllowTradeFlag = OrderActive;
end;
```

Now we're going to create a new method that will allow us to track the status of the market order and the bracket order. We'll associate this method with an order status event a little later in this example. Declare a local method variable named `myStatus` before the `begin` statement to hold the status message string.

```
Method void OrderStatusUpdate(elsystem.Object sender,
    tsdata.trading.OrderUpdatedEventArgs args)
var: string myStatus;
begin
```

The next set of statements in the method will build the order status message string and display it in the print log.

```
   myStatus = myOrder.State.tostring();
   myStatus = myStatus + "  " + myOrder.OrderID;
   myStatus = myStatus + " - " + myOrder.Action.tostring();
   myStatus = myStatus + "  " + myOrder.FilledQuantity.tostring();
   myStatus = myStatus + "  " + symbol;
   print(myStatus);
```

The following statements are executed only once after the market order is filled. The bracket prices are set above and below the filled price of the market order, and then the bracket order is sent. The BracketSent status variable is set to True so that the bracket order is only placed this one time. Finally, the bracket order status message is displayed in the print log.

```
   If BracketSent = False AND
    myOrder.State = tsdata.trading.orderstate.filled then begin
      BracketOrderTicket1.LimitPrice =
      myOrder.AvgFilledPrice + BracketAmt;
      BracketOrderTicket1.StopPrice =
      myOrder.AvgFilledPrice - BracketAmt;
      BracketOrderTicket1.Quantity = iQuantity1;
      BracketOrderTicket1.Send();
      BracketSent = True;
      print("Bracket order sent - ",
         BracketOrderTicket1.LimitPrice.tostring(),
         "  ", BracketOrderTicket1.StopPrice.tostring() );
   end;
```

Complete the method with an `end` statement.

```
end;
```

Here is the code from the previous example to plot the indicator values and send the initial market order.

```
Method void PlotValues() begin
   Plot1(HiTarget, "High Target");
   plot2(LoTarget,"Low Target");
   plot3(AllowTradeFlag.tostring(),"Active");
end;

HiTarget = HighestFC(High, 3)[1];
LoTarget = LowestFC(Low, 3)[1];

PlotValues();

If ((Close <= LoTarget OR Close >= HiTarget ) AND
  (LastBarOnChart AND AllowTradeFlag)) then begin
```

However, in this example we're going to create an order object that will allow us to track the order as it is sent, received, and filled. We'll do this by assigning the order object returned by the `OrderTicket1.Send()` method to the `MyOrder` object variable we declared earlier. Then, we'll associate the previously created `OrderStatusUpdate` event handler method with the `Updated` event of the `MyOrder` object so that we can print the order status each time it changes and send the bracket order when the market order fills.

```
   MyOrder = OrderTicket1.Send();
   MyOrder.Updated += OrderStatusUpdate;
   AllowTradeFlag = False;
end;
```

Verify the Indicator.

Go to a Chart and insert the indicator.

Go to the **Format #11_OSOBracketOrder** dialog and select the **General** tab. The check box labeled **Enable order placement objects** needs to be checked before orders can be generated from your EasyLanguage code. You can turn this on for all RadarScreen rows at once, or individually for each selected row. This setting only retains its setting as long as the indicator is applied to the RadarScreen window. If you remove and reapply the indicator you will have to enable the check box again.

MarketDepthProvider

The MarketDepthProvider component creates an object that lets you reference updated collections of market depth data including bid/ask quotes and bid/ask levels for a specified symbol. This information is similar to that found on the TradeStation MarketDepth window.

Market Depth Collections:

- Bids & Asks.
- Bid Levels & Ask Levels.
- Participants (ECNs, Market Makers).

The Bids and Asks collection properties allow you to access the number of shares listed by Participants (ECNs) at each price level. The BidLevels and AskLevels collection properties allow you to access the combined data of all Participants at a particular price. And the Participants properties collection allows you to access total for a unique Participant.

General properties let you control the number of levels on which to collect bid/ask quotes and whether to include detailed ECN book data and Level II data in the provided data.

The Updated event can be used with the MarketDepthProvider to allow your code to be notified when a value associated with any of the referenced market depth data changes.

In your code, you'll typically use the Count property to determine the number of bid/ask quote items or the depth of the bid/ask levels before trying to access an indexed element from these data collections.

The Market Depth Provider stores information in five data collections:

Bids & Asks – Indexed by each row:

```
(marketdepthprovide1.Bids[row].Price)
```

Bid Levels & Ask Levels – Indexed by each Price Level:

```
(marketdepthprovide1.Asklevels[lvl].TotalSize)
```

Participants – Indexed by each Participant:

```
(marketdepthprovide1.Participants[par].Bids.Count)
```

❖ COURSE EXAMPLE #12

Objectives: (Market Depth Indicator)

- ✓ Use a toolbox component to create an object that calculates and displays the total number of shares or contracts over a number of Bid and Ask levels.
- ✓ Access Market Depth data by price level
- ✓ Loop through levels to total Bid size and Ask size

Indicator: '$12_TotalBidAskSize'

This RadarScreen indicator uses a MarketDepth component to display the total Bid size and total Ask size for a specified number of Market Depth price levels.

	Symbol	Interval	Last	Low	High	Volume Today	$12_TotalBidAskSize	
							Bid Size	Ask Size
1	MSFT	5 Min	26.64	26.55	26.80	25,864,185	250,912	192,942
2	CSCO	5 Min	15.41	15.35	15.62	26,337,619	603,464	480,145
3	IBM	5 Min	174.77	174.61	176.15	2,561,994	2,000	1,619
4	USO	5 Min	37.18	36.94	37.63	6,258,507	194,089	225,658
5	GLD	5 Min	150.91	150.20	151.69	13,170,402	11,480	32,850
6	AAPL	5 Min	354.73	353.34	359.77	11,274,930	500	900
7	GE	5 Min	18.54	18.50	18.78	27,506,556	396,357	281,914

Workspace: $12_MarketDepth

Building the RadarScreen window:

Create: 5- minute interval
Insert Indicator: $12_TotalBidAskSize

Components and Properties Editor Settings:

MDP:

Name:	*MDP*	(changes component name)
Symbol:	*symbol*	(set to symbol)
MaximumLevelCount:	***iMaximumLevelCount1***	(add as an int Input)
Updated:	MDP_Updated	

Indicator Exercise #12: '$12_TotalBidAskSize'

Create a new Indicator and name it: *'#12_TotalBidAskSize*.

Click on Toolbox tab at the left edge of the code editor window and drag the OrdersProvider component into the code editor. The default name *MarketDepthProvide1* will appear in the component tray at the bottom on the code editor window.

Now, click the Properties tab at the right edge of the code editor to open the Properties panel and make sure that *MarketDepthProvide1* is the specified component at the top of the Properties editor.

Change the Name property of the component to the shorter name *MDP*.

Under the Filters category, set the Symbol property to *symbol* so that the provider only returns market depth information for the current symbol. Set the MaximumLevelCount property to an input named *iMaximumLevelCount1*. The matching input declaration is added to your EasyLanguage document.

Click the Events icon and double click on the Updated event to create an event method in your document.

In your EasyLanguage document you will now have an input declaration and an event handler method. Change the default input parameter to '3'. Add a call to `PlotValues()` in the handler method.

```
Input: int iMaximumLevelCount1( 3 );

method void MDP_Updated( elsystem.Object sender,
  tsdata.marketdata.MarketDepthUpdatedEventArgs args )
begin
   PlotValues();
end;
```

Now, let's create the PlotValues method which will include three local variables before the `begin` statement. The first be used as a loop counter when totaling the size of the bid and ask levels. The two local variables will accumulate the total market depth (size) of the ask and bid levels that are being requested from the MarketDepthProvider component.

```
Method void PlotValues()
var: int level, int AskDepthTot, int BidDepthTot;
begin
```

The next set of statements will total the size of both the ask and bid levels from the component object. First, set the ask and bid total variables to 0. Then, test to see if the provider contains enough bid levels to calculate a total for the maximum number of levels you specified in your iMaximumLevelCount1 input. If so, add the size of each bid level to the total in a loop. Create an equivalent `if` and `for` loop for the ask levels.

```
      AskDepthTot = 0;
      BidDepthTot = 0;
      If MDP.bidlevels.count >= iMaximumLevelCount1 then
         For level = 0 to iMaximumLevelCount1 - 1 begin
            BidDepthTot += MDP.bidlevels[level].totalsize;
         End;
      If MDP.asklevels.count >= iMaximumLevelCount1 then
         For level = 0 to iMaximumLevelCount1 - 1 begin
            AskDepthTot += MDP.asklevels[level].totalsize;
         end;
```

Plot the totals and set the background color of the bid size plot to green if the bids are greater than the asks, or set the background color of the ask size plot to red if the asks are greater than the bids.

```
      Plot1(BidDepthTot,"Bid Size");
      Plot2(AskDepthTot,"Ask Size");

      If (BidDepthTot > AskDepthTot) then begin
         setplotbgcolor(1,darkgreen);
         setplotbgcolor(2,black);
         end
      Else begin
         setplotbgcolor(2,darkred);
         setplotbgcolor(1,black);
      end;
   end;
```

Verify the Indicator.

Go to RadarScreen and insert the indicator.

QuotesProvider

The QuotesProvider component creates an object that lets you reference quote field (snapshot) values for a specified symbol. Quote field values may update in realtime but no historical values are available.

In the Properties editor, the Symbol property must specify the symbol for which you want quotes and the Fields property must include a comma delimited list of field names to query. Both are required filter properties. Only the requested quotes fields (e.g. "AskSize, DailyLimit") will be available from a given copy of the QuotesProvider object.

You access a quote field value using the Quote["*Name*"] property of the QuotesProvider along with the data value type (doublevalue, stringvalue, datevalue, etc) of the field requested, such as:.

```
plot1(QuotesProvider1.Quote["Ask"].doublevalue,"Ask");
```

Quote field prices are typically referenced as a DoubleValue, volume/trade size values as an IntegerValue, Date and Time as a DateValue, and names/descriptions as a StringValue. The QuoteFields class help topic lists the available quote field names along with their related data value type. The value type for a quote field is also shown in the Example section of the EL Dictionary Description pane for a specific field name.

An Updated event associated with the QuotesProvider allows your code to be notified when a value associated with any of the referenced quote fields changes. This is especially important when you are asking for quotes for a symbol that is not in your chart or grid but that you want to know about in your analysis technique or strategy.

As with other providers, use the Count property to determine if any quotes were found by the QuotesProvider before trying to access an indexed element from the collection.

❖ COURSE EXAMPLE #13

Objectives: (Quotes Indicator)

- ✓ Access Quote (snapshot) data using a Quotes Provider object
- ✓ Build a token list of Quote Fields to access
- ✓ Use a 'value' property to read and plot the appropriate value type for a quote field

Indicator: '$13_Quotes'

This indicator uses the Quotes Provider component to access Quote field data.

	Symbol	Interval	Last	Low	High	Volume Today	$13_Quotes		
							Last	Last Vol	Time
1	SPY	5 Min	132.57	132.42	133.15	39,397,680	132.57	100	10:25:12 AM
2	CSCO	5 Min	15.68	15.58	15.77	12,290,415	15.68	800	10:25:10 AM
3	IBM	5 Min	183.17	183.01	184.42	1,999,558	183.17	314	10:25:10 AM
4	USO	5 Min	38.42	38.30	38.52	1,225,917	38.42	100	10:25:10 AM
5	GLD	5 Min	154.83	154.12	154.84	5,517,164	154.83	900	10:25:12 AM
6	AAPL	5 Min	387.09	386.47	396.27	16,314,922	387.09	100	10:25:12 AM
7	GE	5 Min	18.61	18.60	18.82	13,648,549	18.61	300	10:25:12 AM

Workspace: $13_Quotes

Building the RadarScreen window:

Create: 5-minute interval
Insert Indicator: $13_Quotes

Components and Properties Editor Settings:

QP:

Name:	*QP*	(changes component name)
Symbol:	*symbol*	(under category Filters)
Fields:	*"last,tradevolume,tradetime"*	
Updated:	QP_Updated	

Indicator Exercise #13: '$13_Quotes'

Create a new Indicator and name it: *'#13_Quotes*.

Click on Toolbox tab at the left edge of the code editor window and drag the QuotesProvider component into the code editor. The default name *QuotesProvider1* will appear in the component tray at the bottom on the code editor window. Now, click the Properties tab at the right edge of the code editor to open the Properties panel and make sure that *QuotesProvider1* is the specified component at the top of the Properties editor.

Change the Name property of the component to the shorter name *QP*.

Under the Filters category, set the Symbol property to the reserved word *symbol* so that the provider only returns quote information for the current symbol in the chart or grid row. Set the Fields property to the text string "*last,tradevolume,tradetime"* that is used to specify the Quote fields that will be requested for the symbol.

A list of available Quote field names and data types is found under the QuoteFields class in the dictionary and QuoteFields help topic in the EasyLanguage Object Reference. You will now be able to use the QP component as an object in your code to access values for the current symbol in a grid or chart.

Next, switch to the Event pane of the property editor by clicking the  icon. You will see Updated listed in the Event column with a blank Value. Double-click on the event name Updated to create an event handler method in your EasyLanguage document and have the new method name associated with the property.

A method named QP_Updated has been added to your document. The parameters within the parentheses of the event handler are auto generated and should not be removed or changed by you.

In the event handler method, replace the auto generated comment after the `begin` statement with a call to a method named `PlotValues()`.

```
method void QA_Updated( elsystem.Object sender,
  tsdata.marketdata.QuoteUpdatedEventArgs args ) begin
  PlotValues();
end;
```

Now, add the following statement in a method that will plot the specified quote fields from the Quote collection. To read a specific quote value you will also need to add the appropriate property type (i.e. doublevalue, integervalue, etc.) after the Quote["*fieldname*"] as shown for each of the three plots. Refer to the QuoteFields class in the dictionary or EasyLanguage Object Reference help for a list of field names and data types. For example, when looking at the dictionary description for the *Last* field name in QuoteFields class, refer to the Example at the bottom of the description item to see a code snippet that shows the data type as `.doublevalue`.

```
Method void PlotValues() begin
  plot1(QP.Quote["Last"].doublevalue,"Last");
  plot2(QP.Quote["TradeVolume"].integervalue,"Volume");
  plot3(QP.Quote["TradeTime"].datevalue.tostring(),"Time");
end;
PlotValues();
```

Verify the Indicator.
Go to RadarScreen and insert the indicator.

FundamentalQuotesProvider

The FundamentalQuotesProvider component creates an object that lets you reference fundamental quote field values for a specified symbol. Fundamental quote field values are typically not updated in realtime and some may allow you to refer to values from previous reporting periods.

In the Properties editor, the Symbol property must specify the symbol for which you want to retrieve fundamental quotes and the Fields property must include a comma delimited list of fundamental field names to query. Both are required filter properties. Only the requested fields (e.g. "sbbf,snpm,sgrp") will be available from a given copy of the FundamentalQuotesProvider object.

You access a fundamental quote field value using the Quote["**Name**"] property of the FundamentalQuotesProvider along with the data value type (doublevalue, stringvalue, datevalue, etc.) of the field requested and number of reporting periods ago (use [0] for the most recent period), such as:.

```
plot1(FundamentalQuotesProvider1.Quote["SBBF"].doublevalue[0],"EPS");
```

Note that the square-bracketed periods ago index is required after the value type for a fundamental field, even if the specific field does not report historical values.

Prices are typically referenced as a DoubleValue, size values (including Volume and Trades) as an IntegerValue, Date and Time as a DateValue, and names/descriptions as a StringValue. The Fundamental QuoteFields class help topic lists the available fundamental quote field names along with their related data value type. The value type for a fundamental quote field is also shown in the Example section of the EL Dictionary Description pane for a specific field name.

An Updated event associated with the Fundamental QuotesProvider allows your code to be notified when a value associated with any of the referenced quote fields changes. This is especially important when you are asking for quotes for a symbol that is not in your chart or grid but that you want to know about in your analysis technique or strategy.

You can also access Commitments of Traders for most Futures contracts using the Fundamental QuotesProvider.

As with other providers, use the Count property to determine if any quotes were found by the FundamentalQuotesProvider before trying to access an indexed element from the collection.

❖ COURSE EXAMPLE #14

Objectives: (Fundamental Quotes Indicator)

- ✓ Use a toolbox component to create an object that displays fundamental quote values for the specified symbol.
- ✓ Access Historical Fundamental Data
- ✓ Use the Properties editor to specify fundamental fields to query.

Indicator: '$14_FundQuotes'

This indicator uses a FundamentalQuotesProvider component to access data from several Fundamental Quote fields. It also calculates and plots a real-time P/E ratio for each symbol.

	Symbol	Interval	Last	$14_FundQuotes						Net Chg	Net %Chg
				EPS	Prev EPS	RT P/E	DivYield %	Px/Bk	UpDate		
1	CSCO	30 Min	14.10	0.33	0.28	42.86	1.71	1.64	6/8/2011	0.04	0.32%
2	INTC	30 Min	20.38	0.56	0.58	36.92	4.08	2.24	7/20/2011	-0.22	-1.07%
3	T	30 Min	28.60	0.61	0.58	47.67	5.96	1.50	7/21/2011	-0.25	-0.87%
4	TRV	30 Min	50.60	(0.87)	1.94		3.17	0.87	7/22/2011	-1.09	-2.11%
5	UTX	30 Min	70.38	1.48	1.13	48.49	2.68	2.86	7/26/2011	-1.19	-1.66%
6	DD	30 Min	46.30	1.31	1.54	35.72	3.51	3.69	7/29/2011	-0.41	-0.88%
7	HD	30 Min	29.54	0.51	0.36	59.37	3.32	2.65	8/2/2011	-0.61	-2.02%
8	JNJ	30 Min	61.50	1.00	1.27	62.32	3.67		8/2/2011	-0.70	-1.13%

Workspace: $14_FundQuotes

Building the RadarScreen window:

Create: 30-minute interval
Insert Indicator: $14_FundQuotes

Components and Properties Editor Settings:

FQP:

- Name: *FQP* (changes component name)
- Symbol: *symbol* (under category Filters)
- Fields: *"sbbf,yield,price2bk,f_lstupdat* "
- Updated: FQP_Updated

Indicator Exercise #14: '$14_FundQuotes'

Create a new Indicator and name it: *'#14_FundQuotes*.

Click on Toolbox tab at the left edge of the code editor window and drag the FundamentalQuotesProvider component into the code editor. The default name *FundamentalQuotesP1* will appear in the component tray at the bottom on the code editor window.

Now, click the Properties tab at the right edge of the code editor to open the Properties panel and make sure that *FundamentalQuotesP1* is the specified component at the top of the Properties editor.

Change the Name property of the component to the shorter name *FQP*.

Under the Filters category, set the Symbol property to the reserved word *symbol* so that the provider only returns fundamental quotes for the current symbol in the chart or grid row. Set the Fields property to the text string *" sbbf,yield,price2bk,f_lstupdat"* that specifies the Fundamental Quote fields that will be requested for the symbol. A list of available fundamental quote field names and data types is found under the FundamentalQuoteFields class in the dictionary and FundamentalQuoteFields help topic in the EasyLanguage Object Reference.

You will now be able to use the FQP component as an object in your code to access values for the current symbol in a grid or chart.

Next, switch to the Event pane of the property editor by clicking the  icon. You will see Updated listed in the Event column with a blank Value. Double-click on the event name Updated to create an event handler method in your EasyLanguage document and have the new method name associated with the property.

A method named FQP_Updated has been added to your document. The parameters within the parentheses of the event handler are auto generated and you should not remove or change them.

In the event handler method, replace the auto generated comment after the `begin` statement with a call to a method named `PlotValues()`.

```
method void FQP_Updated( elsystem.Object sender,
  tsdata.marketdata.FundamentalQuoteUpdatedEventArgs args )
begin
  PlotValues();
end;
```

Now, add the following statements in a method that will plot the specified fundamental quote fields from the component's Quote collection. This time we'll be referencing each element of the Quote collection using a numeric index that matches the order they were listed in the Field filter property set earlier, where '0' is the first element, etc. As in the previous example, you need to add the appropriate property value type for each Quote[n] item as shown for each of the five plots. Because fundamental quotes can contain historical values for previous reporting periods, you also need to specify which period to reference using square bracket 'periods ago' notation for each value. Refer to the FundamentalQuoteFields class in the dictionary or EasyLanguage Object Reference help for a list of field names and data types. For example, looking up "SBBF" (Basic EPS) in the help topic shows a date type of **double** which means that you need to add the property `DoubleValue[0]` to the end of the quote to be able to read the value for the most recent "SBBF" quote. If you fail to specify the data type of the Quote or the 'periods ago' index you will likely get a run-time error.

```
Method void PlotValues()
begin
   If FQP.Count > 0 then begin
      if FQP.HasQuoteData(0) then
         Plot1(FQP.Quote[0].DoubleValue[0],"EPS");
```

Next we will access and plot the EPS of one period ago.

```
      if FQP.HasQuoteData(0) then
         Plot2(FQP.Quote[0].DoubleValue[1],"Prev EPS");
```

Then, we'll plot the remaining fields for the current period.

```
      if FQP.HasQuoteData(1) then
         Plot4(FQP.Quote[1].DoubleValue[0],"DivYield %");
      if FQP.HasQuoteData(2) then
         Plot5(FQP.Quote[2].DoubleValue[0],"Px/Bk");
      if FQP.HasQuoteData(3) then
         Plot6(FQP.Quote[3].DateValue[0].tostring(),"UpDate");

      If Plot1 > Plot2 then
         SetPlotColor(1, Cyan)
      else
         SetPlotColor(1, Magenta);
   end;
end;
```

Finally, we will calculate and plot a real-time price to earnings ratio based on the current price. Then, add another PlotValue call so that values also plot on a tick update.

```
if FQP.Count > 0 AND FQP.Quote[0].DoubleValue[0] > 0 then
   Plot3(Last / FQP.Quote[0].DoubleValue[0], "RT P/E" );

PlotValues():
```

Verify the Indicator.

Go to RadarScreen and insert the indicator.

Workbook (Excel)

The Workbook component creates an object that lets you establish a connection between an analysis technique and an external workbook in a Microsoft Excel® spreadsheet.

In the Properties editor, the FileName filter property must specify the full path and file name of an existing Excel spreadsheet file on your computer. Other filter properties are used to determine additional settings for connection, such as whether the spreadsheet can be shared between other analysis techniques or whether to save the data to the spreadsheet when done.

The Workbook component lets you refer to a specific tabbed sheet within your spreadsheet file and then uses a Cells property of the referenced sheet to read or write data from a specified cell column and row.

For example, the following statement writes the number 123.45 to a cell in the second column and 4th row of a tab (sheet) name named "Prices" in the spreadsheet referred to in the FileName property:

```
Workbook1["Prices"].Cells[2,4] = 123.45;
```

Likewise, this next statement reads a numeric value in the third column and fourth row of the same "Prices" tab (sheet) and assigns it to Value1:

```
Value1 = Workbook1["Prices"].CellsAsDouble[3,4];
```

We used the CellsAsDouble property to read from the cell to ensure that any numeric value in the cell (or a blank cell) will read. However, if the cell contains a text string the statement will generate an error because the data type of the cell doesn't match.

Be aware that to use the Workbook component, you must have Microsoft Excel installed on your computer and must have already created the spreadsheet file in the folder location referenced in the FileName property.

Writing a Value to Excel

```
wkbk["sheet name"].Cells[Col, Row] = 25.50;
wkbk["AcctPL"].Cells[5,10] = 25.50;
```

Reading a Value to Excel

```
Value1 = wkbk["sheet name"]. CellsAsDouble[Col, Row] ;
Value1 = wkbk["AcctPL"]. CellsAsDouble[5,10] ;
```

Saving Spread Sheet Data:

If you modify an Excel Workbook with EasyLanguage and want to then save those changes when you shut down your analysis technique, you can set an object Property to 'SaveOnClose=TRUE'.

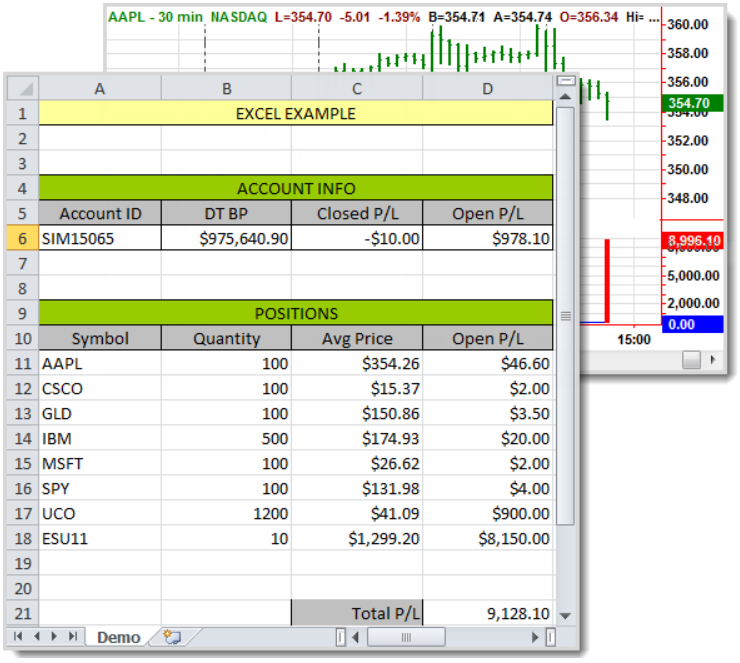
❖ COURSE EXAMPLE #15

Objectives: (Excel Indicator)

- ✓ Reference a pre-built Excel workbook
- ✓ Write updated values to an Excel spreadsheet
- ✓ Read calculated values from a spreadsheet

Indicator: '$15_Excel'

This indicator uses a Workbook component to communicate with a specified Excel spreadsheet in either a RadarScreen or a chart window. NOTE: This indicator will only work if you have a version of Microsoft Excel installed on your computer.



	A	B	C	D
1	EXCEL EXAMPLE			
2				
3				
4	ACCOUNT INFO			
5	Account ID	DT BP	Closed P/L	Open P/L
6	SIM15065	$975,640.90	-$10.00	$978.10
7				
8				
9	POSITIONS			
10	Symbol	Quantity	Avg Price	Open P/L
11	AAPL	100	$354.26	$46.60
12	CSCO	100	$15.37	$2.00
13	GLD	100	$150.86	$3.50
14	IBM	500	$174.93	$20.00
15	MSFT	100	$26.62	$2.00
16	SPY	100	$131.98	$4.00
17	UCO	1200	$41.09	$900.00
18	ESU11	10	$1,299.20	$8,150.00
19				
20				
21			Total P/L	9,128.10

Demo

Workspace: $15_Excel

Building the Chart:

Create: 30 minute interval
Insert Indicator: $15_Excel

Components and Properties Editor Settings:

WkBk:

Name:	*WkBk*	(changes component name)
FileName:	*"c:\exceldemo.xls"*	(under category Filters)

AccP:

Name:	*AccP*	(changes component name)
Accounts:	***iAccount1***	
Updated:	AccP_Updated	

PosP:

Name:	*PosP*	(changes component name)
Symbols:	*symbol*	
Updated:	PosP_Updated	

Indicator Exercise #15: '$15_Excel'

Create a new Indicator and name it: *'#15_Excel*.

Click on Toolbox tab at the left edge of the code editor window and drag the following three components into the code editor: Workbook, AccountsProvider, and PositionsProvider. The default names *Workbook1*, *AccountsProvider1*, and *PositionsProvider1* will appear in the component tray at the bottom on the code editor window.

Now, click the Properties tab at the right edge of the code editor to open the Properties panel and make sure that *Workbook1* is the specified component at the top of the Properties editor. Change the Name property of the component to the shorter name *WkBk*. Under the Filters category, set the FileName property to *c:\exceldemo.xls* so that workbook knows what Excel file to communicate with. You will now be able to use the *WkBk* component as an object in your code to access values for the current symbol in a grid or chart.

Select *AccountsProvider1* at the top of the Properties editor. Change the Name property of the component to the shorter name *AccP*. Switch to the Event icon of the property editor and double-click on Updated to create an AccP event handler method in your document. This method will be called every time a change occurs in an Account value.

Select *PositionsProvider1* at the top of the Properties editor. Change the Name property of the component to the shorter name *PosP*. Switch to the Event icon of the property editor and double-click on Updated to create a PosP event handler method in your document. This method will be called every time a change occurs in one of your positions.

At the top of your document, create two variables. The first specifies the name of the tab you will be referencing in the spreadsheet. The second will hold the value of the total position P/L as calculated in the spreadsheet and read by your EasyLanguage code.

```
Var: WBTab( "Demo" ), TotalPL( 0 );
```

In the AccP event handler method we'll add a set of statements that write several properties from the first Account in the provider collection to the first four columns of row six in the spreadsheet. As before, we first test to see if the AccP component contains any account data. Notice that each WkBk reference includes the name of the spreadsheet tab (from the WBTab variable) that contains the cells you want to write.

```
method void AccP_Updated( elsystem.Object sender,
    tsdata.trading.AccountUpdatedEventArgs args )
begin
    if (AccP.Count > 0) then
    begin
      WkBk[WBTab].Cells[1, 6] = AccP[0].AccountID;
      WkBk[WBTab].Cells[2, 6] = AccP[0].RTDayTradingBuyingPower;
      WkBk[WBTab].Cells[3, 6] = AccP[0].RTRealizedPL;
      WkBk[WBTab].Cells[4, 6] = AccP[0].RTUnrealizedPL;
    end;
end;
```

In the PosP event handler method we'll add a set of statements that write several properties from the each Position in the provider collection to the first four columns of rows 11 through 20 in the spreadsheet.

```
method void PosP_Updated( elsystem.Object sender,
    tsdata.trading.PositionUpdatedEventArgs args )
var: int it;
begin
    for it = 0 to PosP.Count - 1
    begin
      WkBk[WBTab].Cells[1, 11 + it] = PosP[it].Symbol;
      WkBk[WBTab].Cells[2, 11 + it] = PosP[it].Quantity;
      WkBk[WBTab].Cells[3, 11 + it] = PosP[it].AveragePrice;
      WkBk[WBTab].Cells[4, 11 + it] = PosP[it].OpenPL;
    end;
```

This example assumes that we have no more than 10 positions. After writing out the existing position data, a row of blank cells is added to make sure that no previous row displays when a position is removed from the collection

```
    WkBk[WBTab].Cells[1, 11 + it ] = "";
    WkBk[WBTab].Cells[2, 11 + it ] = "";
    WkBk[WBTab].Cells[3, 11 + it ] = "";
    WkBk[WBTab].Cells[4, 11 + it ] = "";
```

The spreadsheet cell in column 4, row 21 contains a formula that sums up rows 11 – 20 in column four. This calculated value is read from the spreadsheet and saved to the variable TotalPL and plotted in realtime.

```
   TotalPL = WkBk[WBTab].CellsAsDouble[4, 21];

   If LastBarOnChart then
      plot1(TotalPL, "TotalP&L");
end;
```

Finally, a zero line is also displayed serve as a reference for the positive or negative P/L value.

```
Plot2(0);
```

This extra code at the end allows Plot1 to display when the market is closed.

```
If LastBarOnChart then begin
   TotalPL = WkBk[WBTab].CellsAsDouble[4, 21];
   plot1(TotalPL, "TotalP&L");
end;
```

Verify the Indicator.

Go to a chart and insert the indicator.

Creating Non-Component Objects

Not all new objects are built using Toolbox components. You can add objects to your code by declaring an object variable and assigning a new instance of the object to the variable. The type (class) of the object variable and of the new object instance must be the same. The process of creating a new instance of an object and assigning it to an object variable is also known as *instantiation*.

Although not required, it is common to create new object instances when your EasyLanguage code first runs by using an AnalysisTechnique_Initialized method that is triggered by the Initialize event of the analysis technique (set in the Properties editor).

Examples of non-component objects include: Collections, Win Forms, and XML Databases.

New

The **new** reserved word is used in an assignment statement to create an instance (or copy) of an object of a specified class (type) and to invoke the constructor of that class. The new word appears to the right of the equals sign (=) and in front of the class type. The newly created object is assigned to a previously declared object variable of the same class type.

For example, the following declares a variable for a vector object named *vectorname* and assigns a new instance of an object of type Vector to the variable:

```
var: elsystem.collections.Vector vectorname(null);
vectorname = New elsystem.collections.Vector;
```

Note that the Vector class type used in the variable declaration and the creation of the new object instance includes the full namespace reference to where the class is located in the dictionary.

Create()

The **Create()** method can also be used to create an instance of an object of a specified class (type) and to invoke the constructor of that class. The Create() method is an alternative to the new reserved word and is often used when a new object requires initialization parameters to be passed as part of the Create() method.

For example, the following declares a variable for a global dictionary object named *gdname* and assigns a new instance of an object of type GlobalDictionary to the variable. Note that this Create() method accepts additional parameters that, in this case, specify that the GlobalDictionary can be shared across analysis window types and has a unique shared name.

```
var: elsystem.collections.GlobalDictionary gdname(null);
gdname = elsystem.collections.GlobalDictionary.Create(true,"IDstring")
```

Vector Collection

A vector is used to save a set of values as indexed elements of a named collection. This is similar to an EasyLanguage array; however, vectors have additional capabilities such as a variety of ways for inserting values into the collection and the ability to store elements such as objects within the collection.

Most of the objects we've been working with up to this point were created from components that you dragged into your document from the Toolbox. To create a vector, or any non-component object, you first need to declare a variable of the proper type and then assign a new instance of the same type of object to the variable.

For example, here is a declaration statement for a variable named myValues that has been created as an elsystem.collection.Vector type with an initial value of null. Note that object variables are always declared with an initial value of null.

```
var: elsystem.collections.Vector myValues(null);
```

The next step is to assign a new instance (copy) of a Vector object to the variable. This can be done anywhere in your code, but it most commonly done when your code first runs, as follows:

```
method void AnalysisTechnique_Initialized( elsystem.Object sender,
   elsystem.InitializedEventArgs args )
begin
    myValues = New elsystem.collections.Vector;
end;
```

Once created, the length of the Vector can grow and shrink as you add or remove data at any location. Vectors can be used with some core EL functions to average or sum up numeric values in the collection. The first element index in a Vector collection is 0.

Adding a data element to a Vector:

Push_Back – Adds an element to the end of the Vector collection.

```
vectorname.Push_Back( 25.50);
```

Insert – Adds an element at a specified location in the Vector collection, pushes the other elements down one index.

```
vectorname.Insert(Index, 25.50);
```

cont.

Removing a data element from a Vector:

Pop_Back – Removes the element at the end of the Vector collection.

```
vectorname.Pop_Back();
```

Erase – Removes an element at a specified location in the Vector collection and moves the other elements up one index.

```
vectorname.Erase(Index);
vectorname.Erase(iStart, iEnd);
```

Clear - Deletes all data elements from a Vector:

```
vectorname.Clear();
```

Accessing data elements from a Vector:

```
Value1 = vectorname[index];
```

Using Vectors in standard functions:

```
Value1 = Highest(vectorname, vectorname.count);
```

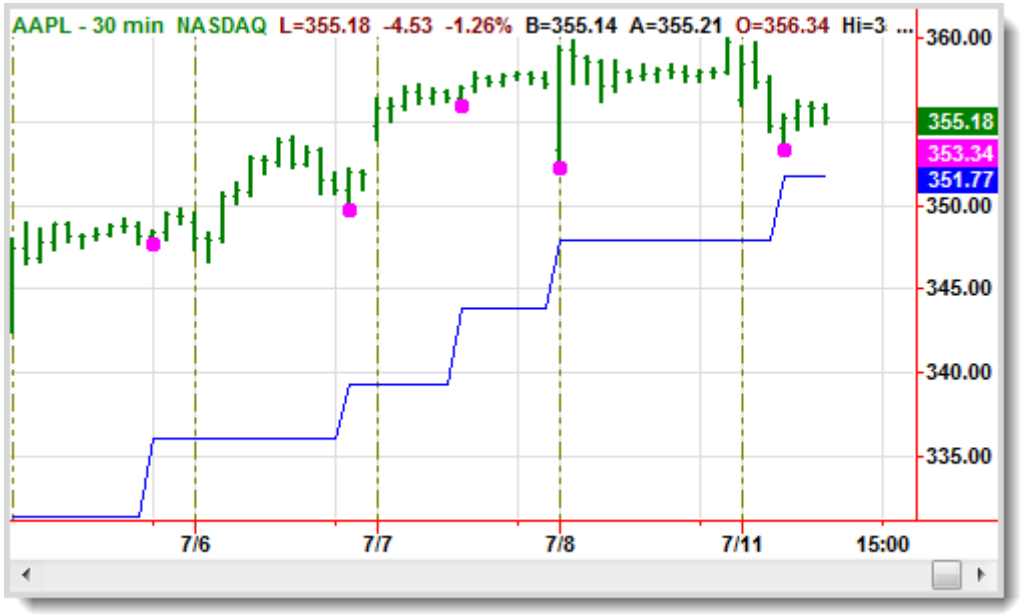
❖ COURSE EXAMPLE #16

Objectives: (Vector Indicator)

- ✓ Create a new vector object instance directly in your code
- ✓ Add values to a vector
- ✓ Access values in a vector
- ✓ Use the vector collection with a standard EL function.

Indicator: '$16_Vector'

This indicator uses a Vector collection to store the low bar price for a specified number of key reversal patterns in the chart and then plots average of the lows as the results on a chart.



Workspace: $16_Vector

Building the Chart window:

Create: 30- minute interval
Insert Indicator: $16_Vector

Components and Properties Editor Settings:

No components

Analysis Technique:

- Initialized: AnalysisTechnique_Initialized

Indicator Properties

Scaling: Same Axis as Underlying Data
Chart Style: Set first plot to type *Point*

Indicator Exercise #16: '$16_Vector'

Create a new Indicator and name it: *'#16_Vector* .

Up to this point, we've used components to create objects in the examples. For this example, we're going to use a non-component object called a Vector that is similar to an EasyLanguage array, however, unlike an array the vector lets us add data to any element in the collection and more easily manage its length.

First, let's declare an input that will define the maximum number of elements that will be used in our vector.

```
input: VectorMax(5);
```

Next we're going to create an object variable that will be used to hold the Vector object. Object variables are declared using the type of class the object is based on, in this case the object type is specified using the full class name elsystem.collections.Vector. The name of the object variable is AvgLow and all object variables are declared will an initial value of `null`.

```
var: elsystem.collections.Vector AvgLow(null);
```

A second variable will be a true-false value used to identify when a key reversal condition occurs.

```
var: KR(False);
```

Next, go the Properties editor, click the Events icon, and double-click Initialized to create an event handler method that is executed only once when the indicator first runs.

Inside the AnalysisTechnique_Initialized method you'll add a statement that creates a New instance of a Vector object (again using the full class type name) and assigns it to the object variable AvgLow. If you go back and look at the designer code in the previous examples that included component created objects you'll see a similar assignment statement for each component object you were using.

```
method void AnalysisTechnique_Initialized( elsystem.Object sender,
  elsystem.InitializedEventArgs args )
begin
   AvgLow = New elsystem.collections.Vector ;
end;
```

The following statement sets KR to True whenever a key reversal pattern is found where the Low of the current bar is lower than the three previous bars and the Close of the current bar is greater than the Close of 1 bar ago. If the pattern was not found, KR is set to False.

```
KR = ( Low < Lowest( Low, 3 )[1] and Close > Close[1] );
```

If a key reversal pattern was found, a new value is added to the vector object AvgLow using the Insert() method. In this case, it adds the Low price of the current bar as the 0^{th} element of the vector. The important thing to note here is that the most recent value is always added as the 0^{th} element of the vector and the previous values in the vector are automatically moved down one element position. Once the number of items in the vector exceeds the specified maximum, the Pop_Back() method is used to remove the last item so that it always has VectorMax or less items in the collection. Once the vector has been updated the current bar Low is plotted. You'll need to right-click in the document, go the Chart Style tab, and change the Type value for "KR BAR" to a Point with increased Weight so that a thick dot appears on the chart wherever a key reversal is found.

```
if KR then begin
   AvgLow.insert(0,Low);

   If AvgLow.Count > VectorMax then
      AvgLow.pop_back();

   Plot1(Low, "KR BAR");
end;
```

Finally, if any items exist in the vector, we'll plot the average of the previous Lows in the vector as a line that represents the recommended stop exit price for an order placed based on the current key reversal.

```
If AvgLow.Count = VectorMax then
   plot2(Average(AvgLow,VectorMax),"KR Avg Low");
```

Right-click in the document to change the plot type for KR BAR to be a *point*.

Verify the Indicator.

Go to a chart and insert the indicator.

Global Dictionary Collection

A global dictionary is used to save values (or objects) that can be shared between analysis techniques across windows. Values are added to the GlobalDictionary using a *key/value* pair where the *key* is the name of the item in the dictionary (such as "mySymbol") and the *value* is any number, string, boolean, or object that can be saved and retrieved using the *key* name.

Most of the objects we've been working with up to this point were created from components that you dragged into your document from the Toolbox. To create a non-component object, you first need to declare a variable of the proper type and then assign a new instance of the same type of object to the variable.

For example, here is a declaration statement for a variable named myGD that has been created as an elsystem.collection.GlobalDictionary type with an initial value of null.

```
var: elsystem.collections.GlobalDictionary myGD(null);
```

Next, a new instance of a GlobalDictionary object is created and assigned to the *myGD* variable.

```
myGD = elsystem.collections.GlobalDictionary.Create();
```

Using the Create() method with no parameters creates a default (unnamed) GlobalDictionary that is shared by analysis techniques running in the same window type (Chart, RS, etc.). Each analysis technique that reads or write to this default global dictionary needs an object variable that is assigned an instance of the GlobalDictionary using the same blank Create() method.

As an alternative, a two parameter Create(shareType,shareName) method can be used to create a GlobalDictionary object that can be shared across window types (between a chart and RadarScreen for instance) and uses a specific name to avoid conflicts that might occur using the default dictionary. Each analysis technique that reads or write to this named global dictionary needs an object instance of the GlobalDictionary made with a Create (shared,name) method having the same true/false and share name parameters. The following statement creates an instance of a GlobalDictionary that works across window types using a shared name.

```
myGD = elsystem.collections.GlobalDictionary.Create(TRUE, "GD_sharedname");
```

Add a value to the Global Dictionary if the key doesn't already exist.

```
If myGD.Contains("keyname") = false then
    myGD.Add("keyname", initialvalue);
```

Once a key has been added to the dictionary, the value associated with the key can be changed or read back using the Items["keyname"] property. Please note that when reading a dictionary value the Items[] value needs to be assigned to a variable using the proper data type.

Change a value associated with an existing Global Dictionary key

```
If myGD.Contains("keyname") then
    myGD.Items["keyname"] = newvalue;
```

Read a value associated with an existing Global Dictionary key

```
If myGD.Contains("keyname") then
    value1 = myGD.Items["keyname"] astype type;
```

❖ COURSE EXAMPLE #17

Objectives: (GDWrite)

- ✓ Create a Global Dictionary object to share values between any type of analysis window
- ✓ Give the Global Dictionary a specific shared name
- ✓ Write values to the global dictionary

Indicator: '$17_GDWrite'

This example uses a Global Dictionary object to write information into a global dictionary that can be accessed from another analysis technique also using the Global Dictionary.

Workspace: $17_GDWrite

Building the Chart window:

Create: Daily interval
Insert Indicator: $17_GDWrite

Components and Properties Editor Settings:

No components

Analysis Technique:

- Initialized: AnalysisTechnique_Initialized

Indicator Exercise #17: '$17_GDWrite'

Create a new Indicator and name it: *'#17_GDWrite.*

In this example, we're going to use a non-component object called a Global Dictionary collection and write a pair of values to the global dictionary that can be read by another analysis technique that reads the values from the same global dictionary.

First, let's declare an input that defines the look back period that will be used when calculating the percent change of the benchmark symbol's closing price.

```
Input: int LookBack( 20 );
```

Then, declare an object variable that will be used to reference an instance of the global dictionary and another variable for the calculated benchmark performance value.

```
Var: elsystem.collections.GlobalDictionary GD(null),
     double BenchPerf( 0 );
```

Next, go the Properties editor, select Analysis Technique, click the Events icon, and double-click Initialized to create an event handler method that is executed only once when the analysis technique first runs.

Inside the AnalysisTechnique_Initialized method you'll add a statement that includes a two parameter GlobalDictionary Create method and assigns the new "myAll" instance to the object variable GD.

```
method void AnalysisTechnique_Initialized( elsystem.Object sender,
   elsystem.InitializedEventArgs args )
begin
   GD = elsystem.collections.GlobalDictionary.Create(true,"myAll");
```

The next set of statements looks to see if the dictionary already contains an item name "TopSymbol" (which it may if another GDWrite indicator is running elsewhere). If that item doesn't exist, we'll add it and the related "TopValue" item to the dictionary. Remember, because these EasyLanguage statements are in the AnalysisTechnique_Initialized method, they are executed only once when the analysis technique loads.

```
   If GD.Contains("TopSymbol") = false then begin
      GD.Add("TopSymbol","");
      GD.Add("TopValue",BenchPerf);
   end;
end;
```

Once the chart has loaded and reached the last bar, the following statements are executed. They calculate the benchmark symbol's performance based on the percent change of the Close over the look back period, assign the bench performance value and symbol name to the previously created Global Dictionary items, and plot the just saved benchmark value from the dictionary. A second plot displays a zero reference line. Because this code is part of the main body of the indicator, the performance value is calculated and written to the global dictionary on every tick.

```
If LastBarOnChart then begin
   BenchPerf  = PercentChange(Close, LookBack) * 100;
   GD.Items["TopValue"] = BenchPerf;
   GD.Items["TopSymbol"] = symbol;
   plot1(GD.Items["TopValue"] astype double,"Bench Perf");
end;

Plot2(0);
```

Verify the Indicator.

Go to a 30-minute chart and insert the indicator.

❖ COURSE EXAMPLE #18

Objectives: (GDRead Indicator)

- ✓ Reference a Global Dictionary object from another analysis technique
- ✓ Read values from a specific global dictionary as they are updated

Indicator: '$18_GDRead'

This indicator uses a Global Dictionary object to read information from a global dictionary into an analysis technique whenever the global dictionary information changes.

	Symbol	Interval	Last	Net Chg	Net %Chg	$18_GDRead			Bid
						Bench Perf	SymRow Perf	Rel Perf	
1	SPY	Daily	132.06	-2.34	-1.74%	3.50	3.50	-0.01	132.06
2	AAPL	Daily	354.74	-4.97	-1.38%	3.50	8.85	5.35	354.71
3	IBM	Daily	174.90	-1.59	-0.90%	3.50	7.18	3.68	174.88
4	CSCO	Daily	15.39	-0.35	-2.22%	3.50	1.79	-1.72	15.38
5	MSFT	Daily	26.62	-0.30	-1.11%	3.50	12.32	8.82	26.62
6	EURUSD	Daily	1.40305	-0.02285	-1.60%	3.50314	-2.64507	-6.14821	1.40305
7	@QM(D)	Daily	94.625	-1.575	-1.64%	3.495	-5.090	-8.586	94.600

Workspace: $18_GDRead

Building the RadarScreen window:

Create: Daily interval for each symbol
Insert Indicator: $18_GDRead

Components and Properties Editor Settings:

No components

Analysis Technique:

- Initialized: AnalysisTechnique_Initialized

Indicator Exercise #18: '$18_GDRead'

Create a new Indicator and name it: *'#18_GDRead*.

In this example, we're going to create a reference to the same GlobalDictionary object created in the previous exercise, only this time we're going to read the benchmark performance value every time it is changed by the other analysis technique.

To save a little typing, you may want to copy the input and variable declarations from the previous example and then add variables to save two other performance values along with the benchmark symbol name from the dictionary.

```
Input: int LookBack( 20 );

Var: elsystem.collections.GlobalDictionary GD(null),
     double BenchPerf(0), SymRowPerf(0), RelPerf(0),
     string BenchSymbol("");
```

As before, go the properties editor, select Analysis Technique, click the Events icon, and double-click Initialized to create an event handler method that is executed only once when the analysis technique first runs.

Inside the AnalysisTechnique_Initialized method we'll add the same two parameters to the GlobalDictionary, and then create a method assignment that refers to the "myAll" shared dictionary. Then, associate two of the global dictionary events with a handler method that will be called whenever the global dictionary item is added or changes in value.

```
method void AnalysisTechnique_Initialized( elsystem.Object sender,
   elsystem.InitializedEventArgs args )
begin
   GD = elsystem.collections.GlobalDictionary.Create(true,"myAll");
   GD.ItemAdded += Global_ItemChanged;
   GD.ItemChanged += Global_ItemChanged;
end;
```

The following method must be manually typed since there is no automatic mechanism for creating handler methods for non-component objects. It will call the PlotValues method to display for the global dictionary items whenever they change.

```
method void Global_ItemChanged( elsystem.Object sender,
   elsystem.collections.itemprocessedEventArgs args )
begin
   PlotValues();
end;
```

In the PlotValues() method you'll add statements that get the benchmark symbol name and performance value from the dictionary if they exist (if they don't exist, that means that the GD_Write indicator hasn't run.)

```
Method void PlotValues()
begin
   If GD.Contains("TopValue")=true then begin
      BenchPerf = GD.Items["TopValue"] astype double;
      BenchSymbol = GD.Items["TopSymbol"] astype string;
   end;
```

The next statement calculates the performance of the symbol in the current row of RadarScreen based on the Closing price over the look back period. It is followed by a second statement that calculates the difference in performance between the current symbol and benchmark symbol.

```
      SymRowPerf = PercentChange(Close, LookBack) * 100;

      RelPerf  = SymRowPerf - BenchPerf;
```

The benchmark symbol name and related performance values are plotted.

```
      Plot1(BenchSymbol, "Bench Sym");
      Plot2(BenchPerf, "Bench Perf");
      Plot3(SymRowPerf, "SymRow Perf");
      Plot4(RelPerf, "Rel Perf");

   end;
```

Since performance values are based on either a change in the global dictionary benchmark value or an update from the current row symbol value, an additional call to the PlotValues() method appears in the main body of your code.

```
   PlotValues();
```

Verify the Indicator.

Go to RadarScreen and insert the indicator. Add a group of symbols to RadarScreen and change the symbol intervals to 30 minutes to match the benchmark interval from the chart. Set the Symbol Link of the chart and the RadarScreen window to the same color so that clicking on a RadarScreen symbol changes the benchmark symbol in the chart using TradeStation's symbol linking feature.

Using – (Reserved Word)

The **using** reserved word allows you to reference object class names in your EasyLanguage code without needing to type the full namespace portion of the name every time.

For example, when creating forms you need to declare an object type for each variable for each control, such as:

```
vars:   elsystem.windows.forms.Form form1(Null),
        elsystem.windows.forms.Button button1( Null ),
        elsystem.windows.forms.TextBox textbox1( Null );
```

The ‘using’ reserved word lets you indicate that the namespace portion of the object type can be assumed so that only the object type itself needs to be specified in each declaration, as follows:

```
using   elsystem.windows.forms;

vars:   Form form1(Null),
        Button button1( Null ),
        TextBox textbox1( Null );
```

Form Controls

The EasyLanguage Forms classes allow you to create objects that produce freestanding windows as part of an analysis technique or strategy. Form objects consist of *container* Controls (such as forms, groups, or panels) that are used to group and display *basic* Controls (such as buttons, text areas, up/count numeric spinners, combo boxes, and more).

- Form windows are created and activated within an analysis technique or strategy.
- Form controls allow you to create interactions with TradeStation analysis windows.
- The properties for each container and control object are accessed through EasyLanguage.

Each of the various form controls can be set up in the initialized method by setting object properties for size, location, labels, and certain functionality.

Form Containers

- Form - A window in which to place the form's related controls and containers.
- Panel - An area within a form to visually organize a set of controls.
- Group Box - An area used to visually group a set of form controls.

Form Controls

- Button - A push-button that generates to a user click event and calls a handler method.
- Checkbox - A small box that can be checked or unchecked by the user. A user click event calls a method used to review the checked or unchecked state.
- Combo Box - Presents a list of items to a user from a drop-down list and triggers an event when a selection is made that can call a handler method.
- Label - Displays non-editable text on form. Labels can be used to provide descriptions or identify surrounding controls.
- List View - Displays a collection of text items in a multi-column format. An event is triggered when a list view item selection is made that can call a handler method.
- Numeric Up Down - Displays a numeric value that users can quickly increment or decrement at a predefined step value using up-down arrows. Each value change event calls an event method.
- Radio Button - Typically placed in groups of two or more, allows users to select from a group of mutually exclusive options. Radio buttons generate a user click event that can call a method used to review the button state.
- Text Box - Displays text that can be edited by a user.

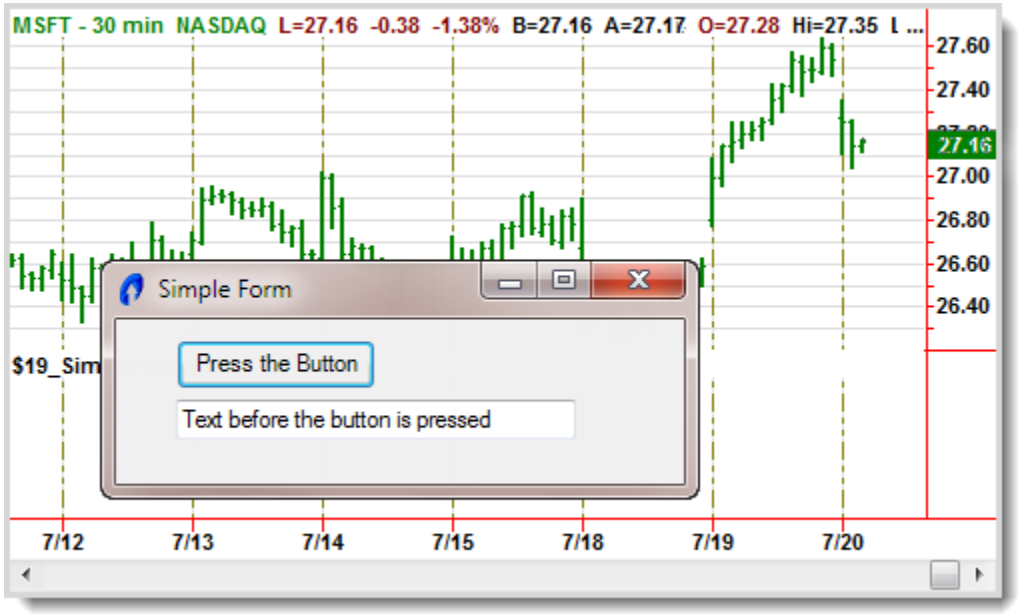
❖ COURSE EXAMPLE #19

Objectives: (Forms Indicator)

- ✓ Create a set of form control objects that display in a pop-up window
- ✓ Assign events to certain controls that are handled by your code.

Indicator: '$19_Simple_Form

This example shows how to use a button click to change some text in a form.



Workspace: $19_Simple Form

Building the Chart window:

Create: 30 minute interval
Insert Indicator: $19_Simple Form

Components and Properties Editor Settings:

No components

Analysis Technique:

- Initialized: AnalysisTechnique_Initialized

Indicator Exercise #19: '$19_Simple Form

Create a new Indicator and name it: *'#19_SimpleForm*.

To eliminate the need to enter the full namespace path for each object type reference, we'll include a *using* statement for the assumed namespace. In this case, since we'll be working with Forms objects, all of the objects we expect to reference will be in the `elsystem.window.forms` namespace portion of the EasyLanguage object hierarchy.

```
using elsystem.windows.forms;
```

Next, we'll declare three object variables to hold each of the three form controls we'll be using in this example. These include a form container that will contain a button control and a text control.

```
vars: Form form1( Null ),
      Button button1( Null ),
      TextBox textbox1( Null );
```

An analysis technique initialized method is created that will contain statements that are executed when the analysis technique first runs. Use the Properties editor to access the events for the Analysis Technique and double-click the Initialized event to create the method.

```
method void AnalysisTechnique_Initialized( elsystem.Object sender,
   elsystem.InitializedEventArgs args )
begin
```

After the `begin` statement, we will create an instance of each control and assign it to the respective object variable. For most form controls, the create method is used to specify the initial text displayed by the control along with the width and height of the control in screen pixels. For example, the `form1` control (the window in which the controls will appear) will be 300 pixels wide and 120 pixels high with the words "Simple Form' displayed on the title bar of the window.

```
    form1 = form.create("Simple Form", 300, 120);
    button1 = button.create("Press the Button", 100, 25 );
    textbox1 = textbox.create("Text before pressing button",200,25);
```

After specifying the size and initial text of each control, we'll need to specify the location of the control within the form container. In this case, the upper left corner of the button will be indented 30 pixels from the left edge of the form and 10 pixels from the top of the form. Likewise, the textbox control will be indented 30 pixels and placed 40 pixels down from the top of the form.

```
    button1.Location( 30,10 );
    textbox1.Location( 30, 40 );
```

Many controls support events that will notify you when certain things happen to a control, such as clicking on the control or an element of a control. Here, we'll set an event handler method named `OnButtonClick` to respond to the `button1` Click event. The important thing is that every time the button is pressed, the code in the associated method will be executed.

```
    button1.Click += OnButtonClick;
```

Once the controls are created, positioned, and set to respond to events, we'll add them to the form container and then show the form window with the controls appearing in the form at the locations specified above.

```
      form1.AddControl(button1);
      form1.AddControl(textbox1);
      form1.Show();
   end;
```

Now, we'll create the event method that will be called whenever the button in the form is clicked. The method parameters are typically the same for any form control event handler so you can use this as a model for other form control event handlers.

```
   method void OnButtonClick(elsystem.Object sender,
      elsystem.EventArgs args )
   begin
```

The next statements will change the text in the textbox and button as shown when the button is pressed. Note how we are using the button text to determine the state, and then setting the new button text for the next button click event.

```
      If button1.Text = "Press the Button" then begin
         textbox1.Text = "Text after the button has been Pressed";
         button1.Text = "Button Pressed";
      End else begin
         textbox1.Text = "You Pressed it Again";
         button1.Text = "Press the Button";
      end;
   end;
```

Verify the Indicator, then go to a chart and insert the indicator.

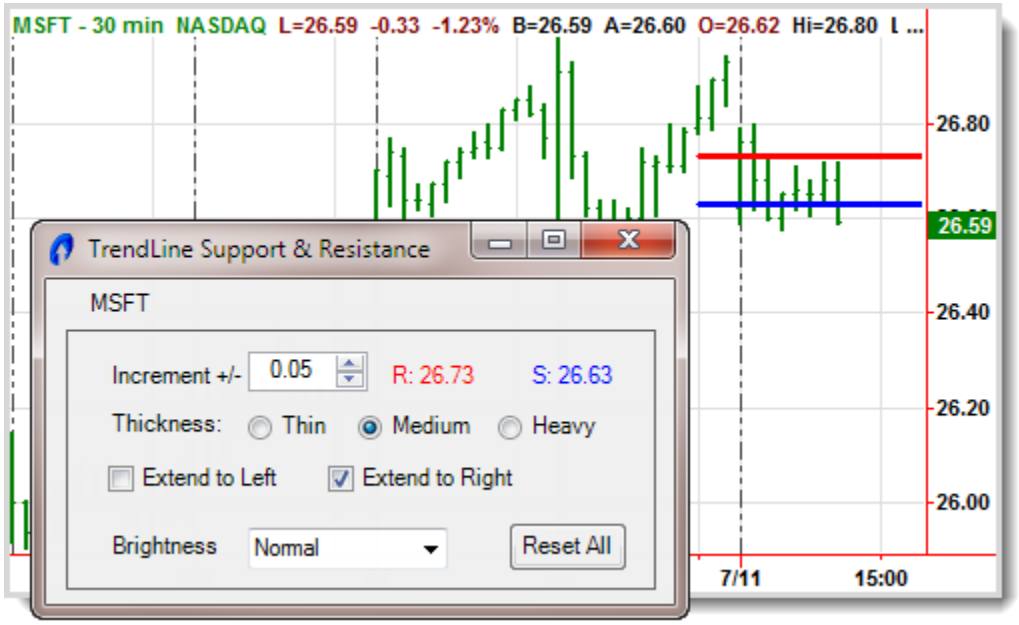
❖ COURSE EXAMPLE #20

Objectives: (Forms Indicator)

- ✓ Create a set of form control objects that display as a pop-up window
- ✓ Assign events to certain controls that manipulate a pair of trendlines

Indicator: '$20_TrendLine S&R'

This indicator uses form controls in a custom pop-up window to change the location and style of a pair of Trendlines on a chart.



Workspace: $20_TrendLine S&R

Building the Chart window:

Create: 30 minute interval
Insert Indicator: $20_TrendLine S&R

Components and Properties Editor Settings:

No components
No properties to set

Analysis Technique:
Initialized: AnalysisTechnique_Initialized

Indicator Exercise #20: '$20_TrendLine S&R'

Create a new Indicator and name it: *'#20_TrendLineS&R.*

This example is much longer and appears to be more complex than the simple button and textbox exercise we just did, but the main difference is the number of form objects created and the various event handler methods that respond to form control events.

Regardless of the number of controls, there are seven basic steps involved in creating a form window (container) and using controls (buttons, labels, combo boxes, etc.) within it. These steps consist of 1) declaring an object variable for each control, 2) creating a new instance of each control (including the initial text and its size, 3) setting the location of controls within a form container, 4) setting other parameters for the controls, 5) associating event methods to the controls, 6) adding controls to their container, and 7) displaying the form window.

As before, we'll include a *using* statement for each assumed namespace to minimize retyping the full namespace portion of each object type when declaring object variables or creating object instances.

```
Using elsystem.windows.forms;
Using elsystem.drawing;
```

First, we'll declare a set of object variables that will be used to reference the form control objects and their properties. This is *Step 1* of the seven basic steps for creating a form.

```
vars: Form form1(Null),
      Panel panel1( Null ),
      Button button1( Null ),
      NumericUpDown spinner1( Null ),
      Label label1( Null ),
      Label label2( Null ),
      Label label3( Null ),
      Label label4( Null ),
      Label label5( Null ),
      Label label6( Null ),
      RadioButton radio1 ( Null ),
      RadioButton radio2 ( Null ),
      RadioButton radio3 ( Null ),
      CheckBox checkbox1( Null ),
      CheckBox checkbox2( Null ),
      ComboBox combobox1( Null );
```

Next, we'll declare two numeric variables to keep track of the high and low position of the trendline that we'll be adding to the chart.

```
vars: TL_High(0.0),TL_Low(0.0);
```

An analysis technique initialized method is created that will contain statements that are executed the when the analysis technique first runs. Since we only need to execute the form creation and initialization statements once, this is an ideal place for this code. Use the Properties editor to access the events for the Analysis Technique and double-click the Initialized event to create the method.

```
method void AnalysisTechnique_Initialized( elsystem.Object sender,
   elsystem.InitializedEventArgs args )
begin
```

In the first set of statements in the method, we'll create and assign an instance of each form control object, including any initial text along with the width and height of the control, to the respective object variable for the control. This is *Step 2* of the seven basic steps for creating a form.

```
form1 = Form.create("TrendLine Support & Resistance",330,200);
panel1 = Panel.create(295,130);

button1 = Button.create("Reset All", 60, 25 );
spinner1 = NumericUpDown.create(60,25);
label1 = Label.create(symbol,80,18);
label2 = Label.create("Increment +/-",80,18);
label3 = Label.create("Thickness:",60,18);
label4 = Label.create("R: 123.45",60,18);
label5 = Label.create("S: 123.45",60,18);
label6 = Label.create("Brightness",60,18);
radio1 = RadioButton.create("Thin",45,25);
radio2 = RadioButton.create("Medium",65,25);
radio3 = RadioButton.create("Heavy",65,25);
checkbox1 = CheckBox.create("Extend to Left",100,18);
checkbox2 = CheckBox.create("Extend to Right",100,18);
combobox1 = ComboBox.create("",100,22);
```

Then, we'll set the relative location of that each control by specifying the x,y offset of the upper left corner of the control within its container. This is *Step 3* of the seven basic steps for creating a form.

```
panel1.Location(10,25);
button1.Location(220,95);
spinner1.Location(90,10);
label1.Location(20,5);
label2.Location(20,15);
label3.Location(20,39);
label4.Location(160,15);
label5.Location(230,15);
label6.Location(20,100);
radio1.Location(90,35);
radio2.Location(145,35);
radio3.Location(215,35);
checkbox1.Location(20,65);
checkbox2.Location(130,65);
combobox1.Location(90,98);
```

The following statements set additional properties for controls, such as their initial displayed values or appearance. This is *Step 4* of the seven basic steps for creating a form. Refer to the EasyLanguage Dictionary or Help topics for more information about the properties that are available for a specific control.

```
label4.ForeColor = Color.Red;
label5.ForeColor = Color.Blue;

panel1.BorderStyle = 1;

spinner1.DecimalPlaces = 2;
spinner1.TextAlign = 2;
spinner1.Increment = .05;
spinner1.Value = .05;

radio1.Checked = true;
checkbox1.Checked = false;
checkbox2.Checked = false;

combobox1.AddItem("Lighter");
combobox1.AddItem("Normal");
combobox1.AddItem("Darker");
combobox1.SelectedIndex = 1;
```

Next, we'll associate event handlers with various control events, such as when they are clicked or when a control value changes. This is *Step 5* of the seven basic steps for creating a form. More about what how each event is used when we review the event handler method code later in this exercise.

```
button1.Click += OnButtonClick;

spinner1.ValueChanged += SpinnerClick;

radio1.Click += Resize_TrendLine;
radio2.Click += Resize_TrendLine;
radio3.Click += Resize_TrendLine;

CheckBox1.Click += Extend_TrendLine;
CheckBox2.Click += Extend_TrendLine;

ComboBox1.SelectedIndexChanged += ChangeColor;
```

Once the properties (size, location, initial values, and events) have been set for each control, they are ready to be added to a form container. In this example, we'll be using an interim container, called a panel, to group and hold most of the controls, and then we'll add the panel to the underlying form window container. The advantage of grouping controls in a secondary container, such as a panel, is that we can easily move the entire group of controls around in the window, or copy the entire panel and its related controls to another form, without needing to reposition the individual controls. Once the controls have been added to the panel, a single label is added to the top of the form followed by the entire panel container. This completes *Step 6* of the seven basic steps for creating a form.

```
    panel1.AddControl(button1);
    panel1.AddControl(spinner1);
    panel1.AddControl(label2);
    panel1.AddControl(label3);
    panel1.AddControl(label4);
    panel1.AddControl(label5);
    panel1.AddControl(label6);
    panel1.AddControl(radio1);
    panel1.AddControl(radio2);
    panel1.AddControl(radio3);
    panel1.AddControl(checkbox1);
    panel1.AddControl(checkbox2);
    panel1.AddControl(combobox1);

    form1.AddControl(label1);
    form1.AddControl(panel1);
```

Before showing the form window, its location is set to 100 pixels from the left and 100 pixels from the top of the screen so that it will always appear at the same screen location. The form property TopMost is set to true so that the form window floats on top of other windows instead of jumping behind other windows when an area outside the form is clicked. This completes *Step 7* of the seven basic steps for creating and displaying a custom form window.

```
    form1.Location(100,200);
    form1.TopMost = true;
    form1.show();
```

Finally, when the chart begins calculating, a pair of trendlines is created at the far left of the chart and the ID for each is saved. The initial color of the upper trendline (resistance) is set to red and the lower trendline (support) is set to blue. By the way, since the trendlines will be repositioned on the last bar of the chart you may see them at this initial location.

```
    TL_High = TL_New(Date[10], Time[10], High, Date, Time, High);
    TL_Low = TL_New(Date[10], Time[10], Low, Date, Time, Low);

    TL_SetColor(TL_High,RGB(255,0,0));
    TL_SetColor(TL_Low,RGB(0,0,255));
end;
```

Next, we'll start creating the methods for handling each of the form control events.

The first of these is a method that responds to clicking the button labeled "Reset All". In this method is a single statement that throws a special type of exception 'message' that causes the chart to reload and the form controls to reset.

```
Method void OnButtonClick(elsystem.Object sender,
   elsystem.EventArgs args )
begin
   throw elsystem.RecalculateException.Create("");
end;
```

Create a method named `Resize_Trendline` using the default `sender` and `args` parameters for a form control event handler (the same as in the previous method). Declare two local variables: one is an object variable of type RadioButton and the other an integer.

The method uses the `sender` parameter to get the radio button that was selected and then reads the text of the button to determine the thickness style of the selected button and passes the new thickness value to the TL_SetSize function that changes the style of the upper and lower trendlines in the chart. Each of the Radio button controls determines the thickness style of the trendlines, set to either "Thin", "Medium", or "Heavy".

```
Method void Resize_TrendLine ( elsystem.Object sender,
   elsystem.EventArgs args )
var: RadioButton rButton, int TLSize;
begin
   rButton = sender astype RadioButton;
   TLSize = 0;

   Switch (rButton.Text)
   Begin
      Case "Thin":
         TLSize = 0;
      Case "Medium":
         TLSize = 2;
      Case "Heavy":
         TLSize = 4;
   End;

   TL_SetSize(TL_High,TLSize) ;
   TL_SetSize(TL_Low,TLSize) ;
end;
```

Create the `ChangeColor` method to handle the selection of a color item from the ComboBox1 drop-down control. The case statement reads the index value of the selected combobox item and sets the color of the upper and lower trendlines to the appropriate RGB (red-blue-green) value.

```
Method void ChangeColor( elsystem.Object sender,
   elsystem.EventArgs args )
begin
   Switch (combobox1.SelectedIndex)
   Begin
      Case 0:
         TL_SetColor(TL_High,RGB(255,127,127));
         TL_SetColor(TL_Low,RGB(127,127,255));
      Case 1:
         TL_SetColor(TL_High,RGB(255,0,0));
         TL_SetColor(TL_Low,RGB(0,0,255));
      Case 2:
         TL_SetColor(TL_High,RGB(127,0,0));
         TL_SetColor(TL_Low,RGB(0,0,127));
   End;
end;
```

Create the `Extend_TrendLine` method to handle the selection of either check box control and to set the left and right extension value for the both trendlines based on the checked state of the respective check box.

```
method void Extend_TrendLine( elsystem.Object sender,
elsystem.EventArgs args )
begin
   TL_SetExtLeft(TL_High,checkbox1.checked);
   TL_SetExtLeft(TL_Low,checkbox1.checked);
   TL_SetExtRight(TL_High,checkbox2.checked);
   TL_SetExtRight(TL_Low,checkbox2.checked);
end;
```

Create the `SpinnerClick` method to handle a change in the value of the numeric up/down (spinner1) control. This numeric value is changed by clicking the up/down spinner arrows or typing a new value into the number box of the control. The new value is passed to another method that updates the trendlines.

```
method void SpinnerClick( elsystem.Object sender,
   elsystem.EventArgs args )
begin
   Show_TrendLine(spinner1.value);
end;
```

The `Show_TreadLine` method is used to display the support and resistance values on the form and to redraw the trendlines above and below the previous bars mid-price based on the value of the *myNum* method input parameter. The upper (resistance) trendline is plotted above and the lower (support) trendline is plotted below the mid-price by the offset amount of *myNum*. Each trendline is 10 bars wide when not extended.

```
Method void Show_TrendLine(double myNum)
var: double midprice;
begin
   MidPrice = (High[1]+Low[1]) * .5;
   Label4.Text = "R: " + NumToStr(MidPrice+MyNum,2);
   Label5.Text = "S: " + NumToStr(MidPrice-MyNum,2);
   TL_SetEnd(TL_High, Date, Time, MidPrice+myNum);
   TL_SetBegin(TL_High, Date[10], Time[10], MidPrice+myNum);
   TL_SetEnd(TL_Low, Date, Time, MidPrice-myNum);
   TL_SetBegin(TL_Low, Date[10], Time[10], MidPrice-myNum);
end;
```

We also want to update the position of the trendlines based on real-time price changes, so we'll add a statement to call the trendline plotting method on every price tick of the last bar.

```
If LastBarOnChart then Show_TrendLine(spinner1.value);
```

Verify the Indicator.

Go to a chart and insert the indicator.

BONUS EXERCISE SECTION

The following section includes bonus examples that cover additional material not included in the online course. These examples combine components and objects that were introduced earlier with some new items that you may find useful as you continue to explore the capabilities of EasyLanguage objects.

Cancelling an Order

Pending orders can be canceled using the Cancel() method of an Order object. A common way to access pending orders for a specific symbol is to use the OrdersProvider component to create a collection of '*received*' orders for the current *symbol*. The first order in the resulting collection is the most recent open order. Adding the Cancel() method to the end of this order identifier will cancel that order.

```
OrdersProvider.Order[0].Cancel();
```

To cancel all open orders for the same symbol, you would simply loop through the 'received' orders for the symbol, using the Cancel() method on each Order identifier.

To Cancel-Replace, you simply cancel the desired order as described above, then send another OrderTicket at the new price.

Limit Order

Any of the order ticket components allow you to place a limit order. For example, with an OrderTicket component you could use the Properties editor to set up the order parameters with an order Type of Limit and a specified LimitPrice. Then, use the Send() method of the OrderTicket to submit the Limit order you described.

```
OrderTicket1.Send()
```

Or, you could modify the order parameters for a previously set up OrderTicket component directly in your code just before you send the order, such as:

```
OrderTicket1.Type= tsdata.trading.OrderType.Limit;
OrderTicket1.LimitPrice = Close - .10;
OrderTicket1.Send();
```

❖ BONUS EXAMPLE #21

Objectives: (Limit Order - Cancel)

- ✓ Create a simple order form that can place and cancel a Limit order
- ✓ Cancel a Limit order

Indicator: '$21_LimitCancel'

This indicator uses an OrderTicket and OrdersProvider to place a Buy Limit order from a form and allows a pending order to be canceled.

WARNING: This indicator will generate a limit order – make sure you are NOT logged onto your real-money account. This should only be applied when logged into TradeStation simulator.

Workspace: $21_LimitCancel

Building the Chart window:

Create: 5-minute interval using a Forex symbol
Insert Indicator: $21_LimitCancel

Components and Properties Editor Settings:

OrderTicket1:

Symbol:	***symbol***	(reserved word for current symbol)
Accounts:	***iAccounts1***	(add default Account as an Input)
Quantity:	***iQuantity1***	(add order Quantity as an Input)
Action:	*buy*	
Type:	*limit*	

OrdersProvider1:

Load :	***false***	(**IMPORTANT** - start with component turned off)
Symbols:	*symbol*	(reserved word for current symbol)
Accounts:	***iAccounts1***	(use the same Account Input)
Updated:	OrdersProvider1_Updated	

Analysis Technique:

Initialized:	AnalysisTechnique_Initialized

Indicator Exercise #21: '$21_LimitCancel'

Create a new Indicator and name it: *'#21_LimitCancel'*

This example combines several components and objects that we've used previously.

Add an OrderTicket component named *OrderTicket1* object.

Add an OrdersProvider component named *OrdersProvider1* object.

Use the Properties editor to set the properties for *OrderTicket1* as indicated above, including an input for the initial order Quantity of 10000 and your simulated Forex Account number. The following two lines will be added to your code when you create the inputs.

```
Input: int iQuantity1( 100000 );
Input: string iAccount1( "SIM00000" );
```

Use the Properties editor to set the properties for *OrdersProvider1* as indicated above. Create a handler method for the Updated event. **Note**: Make sure to set the Load property to *false* to prevent the order status event from accessing the form elements before they are displayed.

While in the Properties editor, select *Analysis Technique* and create a handler method for the Initialized event.

Add a *using* statement for the forms and trading namespaces to minimize having to type full control names.

```
using elsystem.windows.forms;
using tsdata.trading;
```

Declare object variables for the main form and three controls, including a textbox, a label, and a button.

```
vars: Form form1( Null ),
      TextBox OrderPx( Null ),
      Label label1 ( Null ),
      Button button1( Null );
```

In the AnalysisTechnique_Initialized method, create the objects for a simple form consisting of controls for setting the limit price and buttons for sending or canceling the order. The Send button will be re-labeled to the Cancel/Replace button once the Limit order is generated.

```
method void AnalysisTechnique_Initialized( elsystem.Object sender,
    elsystem.InitializedEventArgs args )
begin

   form1 = form.create("Limit Order", 250, 120);
   form1.location(200,200);
   form1.topmost = true;

   OrderPx = TextBox.create("",50,20);
   label1 = label.create("Set Limit Price:",90,20);
   button1 = button.create("Place Buy Limit Order", 140, 25 );

   OrderPx.Location( 100,10 );
   label1.Location(10,12);
   button1.Location( 40,40 );
```

Associate the button with the event handler method, add the controls to the form, set the initial text box text value to the current inside bid, and display the form with the chart.

```
button1.Click += OnButtonClick;

OrderPx.Text = NumtoStr(InsideBid,5);

form1.AddControl(OrderPx);
form1.AddControl(label1);
form1.AddControl(button1);

form1.show();
```

Set the Load property of the provider to `true` so that order status events can update the form, now that it has been created.

```
OrdersProvider1.Load = true;
```

A call to the `ReplotButton` method is added to the `AnalysisTechnique_Initialized` method and to the OrdersProvider1_Updated event handler method below.

```
   ReplotButton();
end;
```

Add the OrdersProvider1_Updated method.

```
method void OrdersProvider1_Updated( elsystem.Object sender,
tsdata.trading.OrderUpdatedEventArgs args )
begin
   ReplotButton();
end;
```

The `ReplotButtons` method is used to set the text of the button based on the whether a limit order is active or not. The `Count` property greater than '0' indicates that an order is pending and we can then check to see if it is 'received' and if it is a 'limit' order. If so, then the button text is set to "Cancel Order". If a Limit order is active, then also set the textbox text to the Limit order Price. If there is no pending limit order, then the button text is set to; "Place Buy Limit Order".

```
Method void ReplotButton()
begin
   if OrdersProvider1.Count > 0 AND
      OrdersProvider1.Order[0].State = OrderState.received AND
         OrdersProvider1.Order[0].Type = OrderType.limit
   then begin
      button1.Text = "Cancel Order";
      OrderPx.Text= NumtoStr(OrdersProvider1.Order[0].LimitPrice,5);
   end
   else begin
      button1.Text = "Place Buy Limit Order";
   end;
end;
```

Finally, create an `OnButtonClick` method to handle buttons clicks. Check the `button1` text to determine which order to send, and make sure the text matches exactly. If the most recent order is an active limit order, on a button click, a cancel order will be sent. If there is no active limit order, a Buy Limit order at the price specified in the textbox will be sent to the market.

```
method void OnButtonClick( elsystem.Object sender,
elsystem.EventArgs args )
begin
   If LastBarOnChart then begin
      if button1.Text = "Cancel Order" AND
      OrdersProvider1.Count > 0 AND
         OrdersProvider1.Order[0].State = OrderState.received AND
            OrdersProvider1.Order[0].Type = OrderType.limit
      then
         OrdersProvider1.Order[0].Cancel();

      If button1.Text = "Place Buy Limit Order" then begin
         orderticket1.SymbolType = Category;
         orderticket1.LimitPrice = StrtoNum(OrderPx.Text);
         orderticket1.Quantity = iQuantity1;
         orderticket1.send();
      end;
   end;
end;
```

Verify the Indicator.

Go to a Chart and insert the indicator.

DateTime

A DateTime object represents an instant in time, expressed as a date and time of day. Values range from 12:00:00 midnight, January 1, 0001 A.D. through 11:59:59 P.M., December 31, 9999 A.D.

In addition to being able to read the current day and time, a variety of properties and methods are provided so that you can easily set and display date and time values in any manner you choose.

```
Var: DateTime myDateTime(null);

// sets date and time based on a string value and displays in an alternate format
myDateTime.Value = “12/10/2011 9:35pm”          // sets user specified date/time
plot1(myDateTime.Format(("%m-%d-%y  %H:%M"));  // display as 12-10-11 21:35

myDateTime = elsystem.datetime.Now          // get the computer date and time
myDateTime = elsysetm.datetime.CurrentTime  // gets the current time only
myDateTime = elsystem.datetime.Today        // get the current date without time
```

TimeSpan

A TimeSpan object represents a time interval that is used to measure the positive or negative number of days, hour, minutes, and seconds between date/time values. You can use a time span to save the difference between two dates and times or you can add (or subtract) a time span to an existing date/time to specify a new date/time.

For example,

```
Var: TimeSpan myTimeSpan(null), DateTime myDateTime(null);

// adds 5 days to the current date
myTimeSpan.TotalDays = 5
myDateTime = elsystem.datetime.Today + myTimeSpan;

// calculates time difference between current time and 5 mins before end of market
myDateTime.Value = “3:55pm”
myTimeSpan = myDateTime - elsystem.DateTime.CurrentTime;
```

TokenList

A TokenList object represents a collection of values that can be created from a string containing a comma delimited list of words or numbers or by adding values using the Add property.

For example, you can create a token list containing four symbols and then access any value from the resulting collection using the indexed Item property.

```
Var: TokenList myList(null);
myList = TokenList.Create(“msft,csco,dell,ibm”);     // loads values into token list
print( myList.Item[1] );                             // prints the second item “csco”
```

❖ BONUS EXAMPLE #22

Objectives: (AlarmClock)

- ✓ Create a pair of alarms from a user supplied time (or date & time)

Indicator: '$22_AlarmClock'

This indicator uses DateTime and TimeSpan objects to get the time difference between the current time and a target time (with optional date). It sets a countdown timer that calls an event method when the target time is reached.

Workspace: $22_AlarmClock

Building the window:

Create: Chart or RadarScreen
Insert Indicator: $22_AlarmClock

Components and Properties Editor Settings:

Timer1:

- Interval : *1000* (default value)
- AutoReset: *True* " "
- Enable: *False* (start with timer off)
- Elapsed: Timer1_Elapsed

Timer2:

- Interval : *1000* (set to count once a second)
- AutoReset: *True* " "
- Enable: *True* (start timer with indicator)
- Elapsed: Timer2_Elapsed

Analysis Technique:

- Initialized: AnalysisTechnique_Initialized

Indicator Exercise #21: '$22_AlarmClock'

Create a new Indicator and name it: *'#22_AlarmClock'*

Add a Timer component named *Timer1*. Add another Timer component named *Timer2*.

Use the Properties editor to set the properties for *Timer1* and *Timer2* as indicated above, including adding a handler method for each Elapsed event. Note that *Timer1* is not enabled initially.

Using the Properties editor, select *Analysis Technique* and create a handler method for the Initialized event.

Create inputs for two target times. The target times can include an optional date, such as "8/22/2012 9:25am". The first target should be the earlier time (or date), since the second target will only be evaluated after the first target expires. By default, the target times are set to 5 minutes before the regular start session and end session times.

```
input: string TargetTime1("9:25am"),string TargetTime2("3:55pm");
```

Declare object variables to hold the target time, the initial time difference to the target, and the list of alarm times. Although this example only uses two alarms, you can easily add other times to the list object.

```
var: elsystem.DateTime myAlarmTime(null),
    elsystem.TimeSpan myTimeSpan(null),
    tsdata.common.TokenList myAlarmList(null),
```

Declare two additional variables for the name of the sound file to be played when the alarm elapses and the index number of the active alarm in the alarm list.

```
    string AlarmSound("alarmclockbeep.wav"),
    intrabarpersist int myAlarmIndex(0);
```

In the Initialized method, create the alarm list as an instance of a TokenList object, using the first target as the initial value in the list. Use the 'add to' operator to append the second target time to the alarm list. Note: This is where you would append additional times to the list, if you choose.

```
method void AnalysisTechnique_Initialized( elsystem.Object sender,
   elsystem.InitializedEventArgs args )
begin
   myAlarmList = tsdata.common.TokenList.create(TargetTime1);
   myAlarmList += TargetTime2;
```

Use `GetAlarmTime` to get the next un-elapsed target time from the list. Use `GetAlarmSpan` to find the time difference between the current time and the target time. Then, use SetAlarm to set and enable the alarm clock count down.

```
   myAlarmTime = GetAlarmTime(myAlarmList);
   myTimeSpan = GetAlarmSpan(myAlarmTime);
   SetAlarm(myTimeSpan);
end;
```

Then, add the following statements to the event handler method associated with Timer1. When the timer elapses, it prints the alarm time in the print log, plays the sound file specified by the input, and gets the next alarm time span to set.

```
method void Timer1_Elapsed( elsystem.Object sender,
  elsystem.TimerElapsedEventArgs args )
begin
   print("ALARM sounded at - ",
         elsystem.datetime.currenttime.tostring());
   condition1 = playsound(AlarmSound);

   myAlarmTime = GetAlarmTime(myAlarmList);
   myTimeSpan = GetAlarmSpan(myAlarmTime);
   SetAlarm(myTimeSpan);
end;
```

The Timer2_Elapsed method is executed once a second. It gets the time span of the next alarm target and builds a string with the days, hour, minutes, and seconds remaining, or a message that all alarms have expired.

```
method void Timer2_Elapsed(elsystem.Object sender,
elsystem.TimerElapsedEventArgs args)
var: string TimeLeft, elsystem.TimeSpan TS;
begin
   TS = GetAlarmSpan(myAlarmTime);

   If myTimeSpan.TotalSeconds > 0 then
      TimeLeft = TS.Days.tostring() + "d "
                + TS.Hours.tostring() + "h "
                + TS.Minutes.tostring() + "m "
                + TS.Seconds.tostring() + "s "
   Else
      TimeLeft = "No alarms active";
```

The first plot displays the time remaining.

```
   plot1(TimeLeft,"Count Down");
```

A second plot displays the actual target time and ‘T’arget number.

```
   If myAlarmTime.ELDateTimeEx>=1 then
      plot2(myAlarmTime.format("%m/%d/%y %H:%M  T"
            + myAlarmIndex.tostring()),"Target")
   else
      plot2(myAlarmTime.format("%H:%M  T"
            + myAlarmIndex.tostring()),"Target");
end;
```

Create a method that converts a time span (time remaining) to milliseconds, assigns it to the Timer1 counter `Interval` property, and turns the timer on. If the time remaining is less than one second (<1000 ms) the timer is turned off.

```
method void SetAlarm(elsystem.TimeSpan myTimeSpan)
begin
   Timer1.Interval = myTimeSpan.TotalMilliseconds;
   If Timer1.Interval < 1000 then
      Timer1.Enable = false
   Else
      Timer1.Enable = true;
end;
```

The GetAlarmTime method returns an object representing the next alarm target. The first statement creates a DateTime object named myAlarmTime. The For loop iterates through the list of alarm times (strings in the token list) and uses the DateTime.Parse method to convert an alarm string to a date time object.

```
method elsystem.DateTime GetAlarmTime(tsdata.common.tokenlist
myAlarmList)
begin
   myAlarmTime = elsystem.DateTime.Create();

   For myAlarmIndex = 0 to myAlarmList.Count-1  begin

      myAlarmTime = elsystem.datetime.Parse(myAlarmList[myAlarmIndex]);
```

The call to GetAlarmSpan gets the time remaining and returns the alarm time if the span is a positive number. If no positive time span is found, a blank alarm time is returned.

```
      myTimeSpan = GetAlarmSpan(myAlarmTime);

      If myTimeSpan.TotalSeconds>0 then begin
         myAlarmIndex = myAlarmIndex + 1;
         Return myAlarmTime;
      end;
   end;
   return myAlarmTime;
end;
```

The GetAlarmSpan method returns an object representing the time remaining between now and the myAlarmTime. The first statement creates a TimeSpan object named myTimeSpan. If the number of days to the next alarm is one or more, the time span is calculated using a full date and time. If just a time is found, the time span is calculated from the current time.

```
method elsystem.TimeSpan GetAlarmSpan(elsystem.DateTime myAlarmTime)
begin
   myTimeSpan = elsystem.TimeSpan.Create();

   If myAlarmTime.ELDateTimeEx >= 1 then
      myTimeSpan = myAlarmTime - elsystem.datetime.now

   Else if myAlarmTime.ELTime > 0 then
      myTimeSpan = myAlarmTime - elsystem.datetime.currenttime;

Return myTimeSpan;
end;
```

Verify the Indicator.

Go to a Chart or RadarScreen and insert the indicator. Use the inputs to change the alarm times.

Analysis Technique - Uninitialized Event

The Uninitialized event of an Analysis Technique is used to call an event handler method that contains code that is executed prior to the calculation of the analysis technique finishing.

An *AnalysisTechnique_Uninitialized* event handler method might be used to save data, or execute any cleanup code that might be required to run, before an analysis technique shuts down. This may happen when the analysis technique is refreshed; when it is removed from an analysis window; or when a window, workspace, or desktop is closed. However, in the case of an exception error, there are times that the uninitialized event handler may not get called.

Try-Catch

Try-Catch blocks are used to test the execution of a set of code statements in the Try block so that you may deal with any failure, error, or exception in the Catch block.

For example, you might want to test for an anticipated error condition when accessing data from a component. If a specific condition occurs, you could set a flag than can be used later to prevent further calculations or print a custom message to the user.

```
try
    Value1 = FQ1.Quote[FundamentalField].DoubleValue[0] ;
    OkToCalculate = true ;
catch ( NoFundamentalDataAvailableException NFDEx )
    OkToCalculate = false ;
    Print( "NMF - No fundamental value for primary symbol." ) ;
end ;
```

Or, you may want to determine the existence of a file by testing for the success or failure of a file read statement and creating the file if it doesn't exist.

```
try
    doc.Load("c:filename");        // try reading from a file
catch(elsystem.io.filenotfoundexception ex)
    doc.Save("c:filename");        // create file if not found
end;
```

XML Objects

EasyLanguage supports a set of objects that allow you to create an XML data structure, save it to a file, and read it back at a later time. The XML objects in EasyLanguage are based on the commonly used XML Document Object Model that includes nodes, elements, attributes, and other items associated with XML data hierarchies.

The basic building block in XML is the *element*, which consists of data placed between a pair of start and end tags. The start tag is made up of angled brackets enclosing the element name, such as <symbol>, and the end tag is the same except for the addition of a 'slash' in front of the element name, such as </symbol>.

An XML data structure consists of multiple elements, where each element represents data ranging from a single piece of information, such as a word or number, to other child elements with additional data. In EasyLanguage, there are XML objects to create elements, add and remove elements in a data structure, and navigate the structure to read and write data within the elements.

For example, the following XML data structure consists of a 'root' element that contains a 'symbol' element containing three related data elements: a symbol name, date, and price.

```
<root>
    <symbol>
        <name>MSFT</name>
        <date>9/10/2011</date>
        <price>34.56</price>
    </symbol>
<root>
```

In addition, a start tag can include additional information inside the angle brackets called *attributes*. For example, the following consists of two 'step' elements where the first has an attribute number of '1' and the second a number of '2'.

```
<root>
    <step number="1">First Item</step>
    <step number="2">Second Item</step>
</root>
```

❖ BONUS EXAMPLE #23

Objectives: (XMLPersist Indicator)

- ✓ Create an XML file to save and restore data when an indicator recalculates

Indicator: '$23_XMLPersist'

This indicator uses XML objects to save information about the current symbol to an XML file when an indicator is exited or refreshed (uninitialized) that can be read back when the indicator is re-started (initialized). For example, this allows you to persist symbol values from the current indicator instance that you want to remember the next time the indicator runs on the same symbol.

Workspace: $23_XMLPersist

Building the window:

Create: 30-minute interval
Insert Indicator: $23_XMLPersist

Components and Properties Editor Settings:

No components

Analysis Technique:

- Initialized: AnalysisTechnique_Initialized
- Uninitialized: AnalysisTechnique_UnInitialized

Usage Note:

When you first apply the indicator to a new chart or symbol, the status line of the indicator will display only the symbol name and no additional information. When you exit the chart, or press Control-R to refresh the chart, the Close price is saved to the XML file with the current date and time. The next time the symbol is charted, the saved price and time from the XML file is displayed on the status line of the sub-graph following the symbol name.

Indicator Exercise #23: '$23_XMLPersist'

Create a new Indicator and name it: *'#23_XMLPersist'*

Add a *using* statement for the XML namespace to eliminate the need to type the full names of XML objects.

```
using elsystem.xml;
```

Declare object variables for a set of XML objects that will be used to create and manage data in an external XML file.

```
var:  XmlDocument doc(null),
      XmlElement root(null),
      XmlElement eLevel1(null),
      XmlElement eLevel2(null),
      XmlNode nNode(null);
```

Declare another variable that will be used to test for the existence of the XML data file. If false, the file needs to be created.

```
Var:  bool filefound(false);
```

Declare an input for the XML file name.

```
input:  string xmlFileName("C:\RefSym.xml");
```

Using the Properties editor, select *Analysis Technique* and create a handler method for the *Initialized* event. Then, create another handler method for the *Uninitialized* event that will be called whenever we exit or recalculate the analysis technique.

In the Initialized method, create an instance of a xml document object and initialize the file found flag to `true`.

```
method void AnalysisTechnique_Initialized( elsystem.Object sender,
  elsystem.InitializedEventArgs args )
begin
   doc = new xmldocument;
   filefound = true;
```

Next, we're going to test to see if the XML document already exists using a Try-Catch statement. If the `doc.Load` call in the `Try` clause fails, the file found flag is set to `false` under the `Catch` clause.

```
   try
      doc.Load(xmlFileName);
   catch(elsystem.io.filenotfoundexception ex)
      filefound = false;
   end;
```

If the file is not found, we'll create it by defining a `root` object named 'data' and saving it to the new XML file.

```
   If filefound = false then
   Begin
      root = doc.CreateElement("data");
      doc.AppendChild(root);
      doc.Save(xmlFileName);
   End
```

If the file already exists, read the file's XML structure to the `root` object.

```
   Else
   Begin
      root = doc.DocumentElement;
   end;
```

The `FindSymbol` method returns the node containing data for the current symbol. The data is blank if the symbol hasn't previously been used or if the file was just created.

```
   nNode = FindSymbol();
end;
```

The AnalysisTechnique_UnInitialized method is executed when the indicator is shut down, such as when exiting TradeStation, or whenever the indicator is recalculated, such as with a chart refresh. In this method, add a call to `SaveSymbol`.

```
method void AnalysisTechnique_UnInitialized( elsystem.Object sender,
elsystem.UnInitializedEventArgs args )
begin
   SaveSymbol();
end;
```

In the `SaveSymbol` method we'll update the data we want to save and write the data out to the XML file, in this case the current date/time and the close price.

```
Method void SaveSymbol()
begin
   nNode.Item["date"].InnerText = elsystem.datetime.now.Format("%m-
%d-%Y  %H:%M");
   nNode.Item["price"].InnerText = Close.tostring();
   doc.Save(xmlFileName);
end;
```

The `FindSymbol` method is used to find data for a saved symbol, or to create blank data for a new symbol, and returns the XML node of the requested data. The first two local variables are used to find and save the index number of the element that matches the current symbol. The next local variable declares an XML object that will hold the list of 'symbol' elements to search. The final variable will hold the node containing symbol data that is returned by the method.

```
Method xmlNode FindSymbol()
var: int iIndex, int sNumb, XmlNodeList tList, XmlNode tNode;
begin
```

The first several statements get a list of first level 'symbol' elements and loop through them to see if any 'name' sub element matches the current symbol. Variable `sNumb` is set to the index number of the symbol element, if found.

```
   tList = root.GetElementsByTagName("symbol");
   sNumb = -1;

   for iIndex = 0 to tList.count-1 begin
      If tList.ItemOf[iIndex].Item["name"].innertext=symbol then
         sNumb = iIndex;
   end;
```

If the symbol was not in the list, a new symbol element is created with the name, date, and price appended as second level child elements. The new symbol element structure is appended to the root and saved to the file.

```
      If sNumb = -1 then begin
         eLevel1 = doc.CreateElement("symbol");

         eLevel2 = doc.CreateElement("name");
         eLevel2.innertext = symbol;
         eLevel1.AppendChild(eLevel2);

         eLevel2 = doc.CreateElement("date");
         eLevel1.AppendChild(eLevel2);

         eLevel2 = doc.CreateElement("price");
         eLevel1.AppendChild(eLevel2);

         root.AppendChild(eLevel1);

         doc.Save(xmlFileName);
```

The newly created symbol element structure is assigned to the local method object `tnode`, or if the symbol previously existed, `tnode` is set to the indexed symbol element. The return value of the method is then set to the node.

```
         tNode = eLevel1;
      end

      Else begin
         tNode = tList.ItemOf[sNumb];
      End;

      Return tNode;
   end;
```

Finally, create a method to plot the values for the current node. For existing or refreshed symbols, the date and price will display the saved values. Notice that the call to PlotValues() is the only statement in the main body of your EasyLanguage code.

```
Method void PlotValues()
begin
   plot1(nNode.Item["name"].Innertext,"Symbol");
   plot2(nNode.Item["date"].InnerText,"Saved Date");
   plot3(nNode.Item["price"].InnerText,"Saved Price");
end;

PlotValues();
```

Verify the Indicator.

Go to a Chart and insert the indicator. You will find that the newly created XML file has been added to the specified path and will contain initial XML data for the current symbol.

Appendix A

Commonly Used Fundamental Fields

Snap Shot Fields (Non Historical)

Fundamental Quote String	Description
ATA	Assets Total (FY)
QTA	Assets Total (MRQ)
BETA_DOWN	Beta (S&P500) Down Market
BETA_UP	Beta (S&P500) Up Market
ABVPS	Book Value per Share (FY)
QBVPS	Book Value per Share (MRQ)
ACURRATIO	Current Ratio (FY)
QCURRATIO	Current Ratio (MRQ)
ADIVSHR	Dividend per Share (FY)
YIELD	Dividend Yield
AEPSINCLXO	EPS Including Extra (FY)
QEPSINCLXO	EPS Including Extra (MRQ)
IPCTHLD	Institutional Percent Held
ACURLIAB	Liabilities Current (FY)
QCURLIAB	Liabilities Current (MRQ)
APRICE2BK	Price to Book (FY)
PRICE2BK	Price to Book (MRQ)
APR2REV	Price to Sales (FY)
QPR2REV	Price to Sales (MRQ)
AQUICKRATI	Quick Ratio (FY)
QQUICKRATI	Quick Ratio (MRQ)
ATOTD2EQ	Total Debt to Total Equity (FY)
QTOTD2EQ	Total Debt to Total Equity (MRQ)
F_LSTUPDAT	Last Financial Update

Historical Fields

ATOT	Total Assets
STLD	Total Debt
LTLL	Total Liabilities
SBBF	Earnings Per Share
SNCC	Net Change in Cash
RTLR	Total Revenue
ETOE	Total Operating Expense
NINC	Net Income
DDPS1	Dividends per Share - Common Stock Primary Issue
QTCO	Total Common Shares Outstanding

FY = Fiscal Year, MRQ = Most Recent Quarter

Appendix B

Downloading the EasyLanguage code examples for this course

All the EasyLanguage examples and workspaces used in this course may be downloaded from the web. The name of the file is ELOBJECTS.ZIP. It may be downloaded from the following link:

www.tradestation.com/education/downloads/ELOBJECTS