

Wealth-Lab Pro®

Version 6.9 (.NET)

WealthScript Programming Guide

© 2015 FMR LLC All rights reserved.



Wealth-Lab Pro® WealthScript Programming Guide

Revised: Wednesday, December 02, 2015

Wealth-Lab Pro WealthScript Programming Guide

© 2015 FMR LLC All rights reserved.

No parts of this work may be reproduced in any form or by any means - graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems - without the written permission of the publisher.

Any screenshots, charts or company trading symbols mentioned, are provided for illustrative purposes only and should not be used or construed as an offer to sell, a solicitation of an offer to buy, or a recommendation for the security.

Third party trademarks and service marks are the property of their respective owners.

While every precaution has been taken in the preparation of this document, the publisher and the author assume no responsibility for errors or omissions, or for damages resulting from the use or misuse of information contained in this document or from the use or misuse of programs and source code that may accompany it. In no event shall the publisher and the author be liable for any loss of profit or any other commercial damage caused or alleged to have been caused directly or indirectly by this document.

Printed: Wednesday, December 02, 2015

Special thanks to:

Wealth-Lab's great on-line community whose comments have helped make this manual more useful for veteran and new users alike.

EC Software, whose product HELP & MANUAL printed this document.

Table of Contents

Foreword	0
Part I Introduction	1
Part II How to Run Example Code	3
Part III The Bars Object	4
1 OHLC/V Series	6
2 Scale, Interval, and Date	8
3 Symbol Info	10
Part IV DataSeries	11
1 Series Operators	12
2 Accessing a Single Value from a DataSeries	15
3 Filling a Custom DataSeries	16
4 Accessing Secondary Symbols	18
Secondary Series Synchronization	19
Part V Painting the Chart	22
1 Chart Panes	23
2 Charting Multiple Symbols	24
3 Plotting DataSeries	26
Controlling the y-scale	27
4 Writing Text	28
5 Drawing Objects	30
6 Plotting Fundamental Items	33
Part VI Indicators	34
1 Series Method	36
2 Static Value Method	37
3 Custom Indicators	38
4 Indicators of Secondary Symbols	42
5 Stability of Indicators	43
Part VII Programming Trading Strategies	45
1 Trading Signals	48
2 Trading Loop	50
3 Single-Position Strategies	51
AtLimit, AtStop Orders	52
Time-Based Exits	54

4 Peeking	57
Valid Peeking	59
5 Strategy Parameters	62
6 Multi-Position Strategies	65
7 Multi-Symbol Strategies	67
8 Alerts	68
9 Options Strategies	71
Part VIII Fundamental Analysis	74
1 Corporate Fundamental Data	76
2 Economic Indicator Data	78
3 Market Sentiment Data	79
4 GICS Data	80
5 Equity Summary Score	82
Part IX Multi-Time Frame Analysis	84
1 Intraday/Intraday	85
2 Intraday/Daily	87
3 Intraday/Weekly,Monthly	90
4 Daily/Weekly	91
5 Daily/Monthly	92
Part X Backtesting with ChartStyles.Trending	94
1 Column Class (Reference)	95
Kagi	96
TKagi	96
Renko	97
TRenko	97
Line Break	98
TLineBreak.....	98
Point and Figure	98
TPnF Class.....	99
TrendLine Class.....	100
Part XI WealthLab.Rules Classes	102
1 ArmsIndex	103
2 Candlesticks	104
Bearish	105
Bullish	107
Neutral and Reversals	108
3 DateRules	110
4 DataSeriesOp	111
5 FundamentalsDateRules	112
6 FundamentalsRatio	113

7	OptionExpiryDate	119
8	NewHighLow	120

Part XII Techniques **123**

1	Creating a Screener	124
2	Sizing with 100% of Equity	126
3	Symbol Rotation	127
4	Most Probable AtStop	128
5	Access ad-hoc Data from the Internet	129
6	Utility Scripts	130
	Data Checking	130
	DataSet Symbol Details	130

Part XIII APIs **132**

Index	133
--------------	------------

1 Introduction

Welcome to the WealthScript Programming Guide

The purpose of the WealthScript Programming Guide is to provide you with the basic (and some not-so-basic) concepts to express your trading strategies using Wealth-Lab Pro's WealthScript language. WealthScript is a class library of charting and trading functions in the { }WealthLab .NET namespace. You'll be amazed with what you can accomplish by coding trading systems with WealthScript!

While this guide assumes that you're familiar with .NET programming, even if you're not, the *explanation-by-example* method should be easy to follow. We encourage you to run these ready-made solutions for yourself by following the instructions: [How to Run Example Code](#)³. In many cases, you'll be able to find precisely the solution that you're searching for.

Though many of the most-essential WealthScript functions are used in this guide to demonstrate programming and trading system development concepts, it is not within the scope of the WealthScript Programming Guide to highlight every single WealthScript function. A complete list of functions available for use in Trading Strategies with syntax, descriptions, and examples can be found in the WealthScript QuickRef ([Tools](#) menu, or *F1*).



Functions that exist in the { }WealthLab namespace but that are not documented in the QuickRef have no guarantee or warranty for use in Strategy code.

Programming in .NET

Wealth-Lab Pro Version 6 is a technical analysis and trading platform in which you can express and test trading strategies in *any .NET language*. That's right. In the .NET Framework, all .NET languages compile to *Intermediate Language (IL)*, so you can choose from more than 20 languages to work with.

Wealth-Lab's Strategy Editor, however, comes with a C# compiler, and consequently the Strategy Builder generates code with C# syntax, so naturally we provide coding examples for C#. Our preference, however, does not preclude the possibility of testing and employing Strategies *compiled* in other .NET languages.

Deciding on a .NET language?

We recommend C# for newbies that want to take a course in programming.

C# is a truly modern, developed-for-.NET language that borrows some of the best features from other languages. If you've programmed before, the learning curve for C# is a snap.

But I'm not a programmer!

Wealth-Lab Pro Version 6's Strategy Builder has been enhanced from its predecessor to automatically include all Wealth-Lab-aware indicators, giving you the ability to combine them in any number of conditions. The combinations of strategies that you can generate without knowing anything about programming is just about endless. Nonetheless, it may not be possible to create more-complex strategies that involve multiple time frames,

symbol rotation strategies, and the like. Once you come across a strategy that the Wizard isn't able to fully program, you'll need to dig in to the world of programming using WealthScript.

How to use this Guide

While Wealth-Lab Pro has a number of drag-and-drop Wizards for charting and even programming Strategies, the true power belongs to those who can program. To make the WealthScript Programming Guide as useful a reference as possible, we concentrate on explaining by example. It's useful to browse through each chapter in order since the basics are covered first.

Tip:

When search for a specific solution, try browsing the "How to" references in the Index.

2 How to Run Example Code


In general, use the following procedure to run example code.

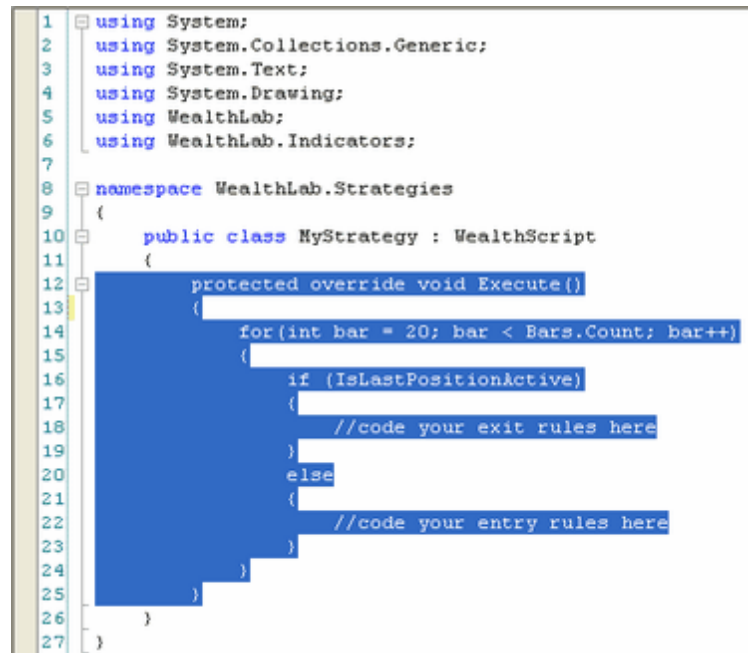
1. Launch a new Strategy Window (*Ctrl+Shift+S*).
2. Click on the Editor tab.

Example code comes in two forms: *complete* and *partial*.

3. Complete: The code is complete if you see with "using" statements at the beginning (as in the image below). In this case, copy the example and *completely replace* all of the code, if any, showing in the Editor. After pasting the code, skip to step 5.

Partial: To focus on the essence of the example, most often only the `Execute()` method is given. In this case, completely replace the `Execute()` method, which is highlighted in the image below.

4. Pay attention to green comment lines in the sample code, which may include special initialization or setup instructions.
5. Click  Compile in the Editor's toolbar and ensure "Strategy compiled successfully!" in the lower message frame.
6. Click *F5* on the keyboard, or click a symbol of your choice in the Data Panel to execute the example.



```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4 using System.Drawing;
5 using WealthLab;
6 using WealthLab.Indicators;
7
8 namespace WealthLab.Strategies
9 {
10     public class MyStrategy : WealthScript
11     {
12         protected override void Execute()
13         {
14             for(int bar = 20; bar < Bars.Count; bar++)
15             {
16                 if (IsLastPositionActive)
17                 {
18                     //code your exit rules here
19                 }
20                 else
21                 {
22                     //code your entry rules here
23                 }
24             }
25         }
26     }
27 }
```

Replacing the Execute method in the Strategy Editor.

3 The Bars Object

A *bar* is any interval of time that has an open, high, low, and closing (OHLC) price, whose values may be different or equal. These can consist of any number of minutes (e.g., 1, 2, 5, 20, 30-minute bars), or one day, week, month, quarter, or year. Wealth-Lab also includes support for bars that are multiples of ticks or seconds.

The **Bars** object represents a historical series of OHLC/V data and has the capability of supporting additional, named data series such as open interest for futures. Conceptually, you can think of **Bars** as a summary of a company or trading instrument. While in fact **Bars** does hold lists of values for prices and volume with corresponding dates (a company's price history), **Bars** also has the properties to access the Symbol under test, its Security Name, and other information.

How to: Count the number of bars in a chart

Use the Count property of **Bars**. Note that Count gives you the total number of bars, however, since bars are store internally as zero-indexed Lists, the first bar number is actually 0, and the number of the last bar in a chart is **Bars.Count** - 1. Knowing this, you won't be surprised that Wealth-Lab's chart status bar indicates the last bar number is 999 for a 1000-bar chart!

Example ([How to run Example code?](#)^{3h})

```
C#
PrintDebug("The chart has " + Bars.Count + " bars.");
PrintDebug("The index number of the first bar is 0,
    and the number of the last bar is " + (Bars.Count - 1));
```

How to: Find the first intraday bar number of the most-recent day

While bar numbers in the chart are always from 0 to **Bars.Count** - 1, it's often useful to know the number of an intraday bar, where 0 is the first intraday bar of the day. To find the intraday bar number of a particular bar, use the **Bars.IntradayBarNumber** method. The example uses **IntradayBarNumber** in reverse to find the first bar number of a specified day.

Example ([How to run Example code?](#)^{3h})

```
C#
using System;
using System.Collections.Generic;
using System.Text;
using System.Drawing;
using WealthLab;
using WealthLab.Indicators;

namespace WealthLab.Strategies
{
    public class MyStrategy : WealthScript
    {
        //Returns the first bar of DaysAgo for an intraday chart
        public int FirstBarXDaysAgo(int DaysAgo)
        {
            int startBar = Bars.Count;
            for (int day = 1; day <= DaysAgo; day++)
                startBar = startBar - 1 - Bars.IntradayBarNumber(startBar - 1);
        }
    }
}
```

```

        return startBar;
    }
    protected override void Execute()
    {
        ClearDebug();
        PrintDebug("First bar number of current day = " + FirstBarXDaysAgo(1));
        PrintDebug("First bar number of yesterday = " + FirstBarXDaysAgo(2));
    }
}

```

How to: Find the bar number that corresponds to a specific date

Use the **ConvertDateToBar** method of **Bars**. Unless you're certain that the date will be found in the chart, it's convenient to specify the *exactMatch* parameter as *false*. For example, to find the first intraday bar with a particular date, with *exactMatch* = *false* you need only to specify the date, not the time.

Example ([How to run Example code?](#)³⁷)

```

C#
protected override void Execute()
{
    ClearDebug();
    int year = 2007;
    int month = 11;
    int day = 16;
    DateTime dt = new DateTime(year, month, day);
    PrintDebug("The bar number for 11/16/2007 is "
+ Bars.ConvertDateToBar(dt, false));
}

```

➔ Alternatively, see **DateTimeToBar** in the QuickRef (F11).

3.1 OHLC/V Series

The Open, High, Low, Close, and Volume (OHLC/V) series that make up the primary chart are accessed by their same-name property in **Bars**. It's difficult to talk about the OHLC/V series in **Bars** without prematurely introducing the *DataSet* type, which is covered in the [next topic](#)^[11]. For now, suffice to say that values in *DataSet* are double-precision floating point numbers, which means that they're accurate to at least 15 significant digits.

➔ Significant digits of a floating point number include the numbers *before and after* the decimal.

How to: Access a OHLC/V value

As with any **DataSet**, use the square bracket [] indexers to access a value at a specified bar. The example code shows an explicit method to find the average closing value of the last 5 bars in the chart. (Later, we'll discover a way to perform the same task in one statement using [Indicators](#)^[34].)

Example ([How to run Example code?](#)^[37])

```
C#
protected override void Execute()
{
    const int period = 5;
    double sum = 0d;
    for(int bar = Bars.Count - 1; bar >= Bars.Count - period; bar--)
        sum = sum + Bars.Close[bar];
    double avg = sum / period;
    PrintDebug("The " + period + "-Period average is: " + avg);
}
```

➔ Since the *DataSet* Open, High, Low, Close, and Volume also exist in the WealthScript base class, When referencing any of the five OHLC/V *DataSet*s, you don't have to qualify them with **Bars**. In other words, **Close** is equivalent to **Bars.Close**, **High** is **Bars.High**, etc.

How to: Plot an OHLC/V Series

It's usually never required to explicitly plot an OHLC/V *DataSet* because the Chart Style selection does this automatically for you. However, say you wanted show high and low "bands" around a Line chart, which automatically plots **Close**. Here's an example of a WealthScript function that takes a *DataSet* parameter, which we use to pass the High, Low, and Volume *DataSet*s.

Example ([How to run Example code?](#)^[37])

```
C#
protected override void Execute()
{
    PlotSeries(PricePane, High, Color.Gray, WealthLab.LineStyle.Solid, 2);
    PlotSeries(PricePane, Low, Color.Gray, WealthLab.LineStyle.Solid, 2);
    // Envelope the Volume histogram
    PlotSeries(VolumePane, Volume, Color.Fuchsia, WealthLab.LineStyle.Solid, 2);
}
```

See [Painting the Chart](#)^[22] for more on plotting.

How to: Change OHLC/V Series Values

In some cases, you may wish to apply data corrections to the primary OHLC/V on the fly

without actually saving the changes. One such application is to make futures continuous-contract data non-negative. The example checks the value of the lowest-low, and if it's negative adds an integer value that will make all OHLC/V DataSeries non-negative.

[Example \(How to run Example code?\)](#)³⁾

■ C#

```
protected override void Execute()
{
    // If the lowest value is negative, round it up, and add to the OHLC DataSeries
    double low = Lowest.Value(Bars.Count - 1, Low, Bars.Count);
    if (low < 0)
    {
        low = Math.Ceiling(Math.Abs(low));
        for (int bar = 0; bar < Bars.Count; bar++)
        {
            Open[bar] = Open[bar] + low;
            High[bar] = High[bar] + low;
            Low[bar] = Low[bar] + low;
            Close[bar] = Close[bar] + low;
        }
    }
}
```

3.2 Scale, Interval, and Date

Bars.Scale returns a **BarScale** enumeration with the current scale of **Bars**, which you can change temporarily by one of the "SetScale" WealthScript functions (See [Multi-Time Frame Analysis](#)^[84]). Values for **Scale** are Tick, Second, Minute, Daily, Weekly, Monthly, Quarterly, and Yearly. For intraday Scales, the **BarInterval** property returns an integer representing the number of minutes, seconds, or ticks.

How to: Access the current Scale or Interval

When possible, it's always a good idea to write a script in a general fashion so that it can be tested under different scales and intervals without having to edit the code. The following example checks to ensure that **Bars.Scale** is **BarScale.Minute**. If so, it uses **Bars.BarInterval** to determine a time for an end-of-day exit as well as calculates the number of bars that generally make up two trading days for use as a *Period* in the Highest indicator.

Example ([How to run Example code?](#)^[3b])

```

C#
protected override void Execute()
{
    if (Bars.Scale != BarScale.Minute)
    {
        DrawLabel(PricePane, "A minute scale is not selected", Color.Red);
        return;
    }
    // Assumes a 60-Minute interval or less for simplicity
    int exitTime = 1500 + 60 - Bars.BarInterval;
    // Find the Period equivalent to the last 2 days
    int Period = 390 * 2 / Bars.BarInterval;
    DataSeries breakoutHigh = Highest.Series(High, Period);
    PlotStops();

    for(int bar = Period; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            Position p = LastPosition;
            int currentTime = Date[bar].Hour * 100 + Date[bar].Minute;
            if (currentTime >= exitTime)
            {
                SellAtMarket(bar + 1, p, "Market Close");
            }
            else
            {
                SellAtLimit(bar + 1, p, p.EntryPrice * 1.02, "Profit Target");
            }
        }
        else
        {
            // Don't enter on the first bar of the day
            if (!Bars.IsLastBarOfDay(bar))
            {
                BuyAtStop(bar + 1, breakoutHigh[bar]);
            }
        }
    }
}

```

How to: Access the Date and Time

The **Date** property of **Bars** provides access to a *DateTime* object. The previous example demonstrated how to access the individual Hour and Minute properties, whereas the next example shows how to display a fully-qualified date and time in compact format along with

the associated closing price.

[Example \(How to run Example code?\)](#)

```
C#
protected override void Execute()
{
    ClearDebug();
    for(int bar = 20; bar < Bars.Count; bar++)
    {
        PrintDebug(String.Format( "{0:yyyyMMdd HHmm }", Date[bar] ) + Close[bar]);
    }
}
```

The following link is a handy reference for displaying *Custom DateTime Format Strings*.
[http://msdn2.microsoft.com/en-us/library/8kb3ddd4\(VS.71\).aspx](http://msdn2.microsoft.com/en-us/library/8kb3ddd4(VS.71).aspx)

3.3 Symbol Info

How to: Access the current Symbol or Company Name

Symbol and **SecurityName** are string properties of **Bars**.

Example ([How to run Example code?](#)^[3])

```
C#
protected override void Execute()
{
    DrawLabel(PricePane, Bars.Symbol + ": " + Bars.SecurityName, Color.Black);
}
```

How to: Access the Tick Value of a Futures Symbol

Items commonly associated with futures like **Tick**, **Margin**, and **PointValue** are members of the **SymbolInfo** object contained by **Bars**. In Wealth-Lab Version 6, equities (stocks) are assigned a Tick value of 0.01 by default, consequently to determine if a futures symbol is under test, use the **SecurityType** property of **SymbolInfo**.

Example ([How to run Example code?](#)^[3])

```
C#
protected override void Execute()
{
    SymbolInfo si = Bars.SymbolInfo;
    DrawLabel(PricePane, si.Symbol + " Tick = " + si.Tick, Color.Blue);
    int bar = Bars.Count - 1;
    // Create an Alert 1 tick over the last close for Futures, otherwise 5 ticks
    if ( si.SecurityType == WealthLab.SecurityType.Future )
        BuyAtLimit(bar + 1, Close[bar] + si.Tick);
    else
        BuyAtLimit(bar + 1, Close[bar] + si.Tick * 5);
}
```

How to: Access Bars of a secondary symbol

The primary symbol is the one that you clicked, or more generally, the one that the Strategy is operating on. **Bars** defaults to the primary symbol, however, you can access **Bars** of any other symbol by calling **SetContext**, which not only provides access to a secondary symbol's Bars object's properties and methods, but also allows you to explicitly create trades on that symbol. See: [Multi-Symbol Strategies](#)^[67]

4 DataSet

A DataSet is the basic data structure that represents a historical series. It's represented internally as a List<double> with an associated List<DateTime>, which can be managed in the instance or linked to another DataSet or Bars object. Bars OHLC/V Series^[6] and all indicators^[34] are instances of DataSet.

Characteristics of DataSet

1. A DataSet is a series of floating point data values of type *double*. Consequently, each value in the series is double precision (14 to 15 significant digits).
2. All DataSet contain the same number of values as bars in the chart.
Exception: If you specify *false* for the *synchronize* parameter when accessing secondary symbol data via **SetContext** or **GetExternalSymbol**, the returned series can have a different number of bars than the chart.

4.1 Series Operators

How to: Multiply, Divide, Add, or Subtract a **DataSeries** by another

Performing math operation with **DataSeries** are very intuitive in Wealth-Lab Pro Version 6. Simply perform the operation as if a **DataSeries** were any plain-vanilla *double* type. You can multiply, divide, add, or subtract a **DataSeries** with another **DataSeries** to return a result **DataSeries**. In reality, when performing math operations on two **DataSeries**, the operation occurs with the values from the same bar (*DateTime*) in each series. When the operation is between a **DataSeries** and a constant value, then that value operates on each element of the **DataSeries**.

- ➔ When dividing two **DataSeries**, Wealth-Lab suppresses division-by-zero errors and inserts the a *zero value (0d)* for those bars.

Example ([How to run Example code?](#)³⁷)

```
C#
// Create an "Average Price" DataSeries
protected override void Execute()
{
    DataSeries avgPrice = (High + Low) / 2d;
    avgPrice.Description = "Average Price";
    PlotSeries(PricePane, avgPrice, Color.Blue, WealthLab.LineStyle.Solid, 1);
}
```

- ➔ Instead of creating average price yourself, just use the standard indicator, **AveragePrice** instead.

How to: Create the "Typical Price" **DataSeries**

The "Typical Price" is nothing but the average of the High, Low, and Close series.

Example ([How to run Example code?](#)³⁷)

```
C#
protected override void Execute()
{
    DataSeries typPrice = (High + Low + Close) / 3d;
    typPrice.Description = "Typical Price";
    PlotSeries(PricePane, typPrice, Color.Black, WealthLab.LineStyle.Solid, 1);
}
```

- ➔ Instead of creating typical price yourself, just use the standard indicator, **AveragePriceC** instead.

How to: Offset or Delay a **DataSeries**

You can offset a **DataSeries** to the right (delay) or to the left (advance) using the shift operators >> and <<, respectively, followed by an integer number of bars required for the offset. On one end of the series, values are shifted "out of the chart", whereas *zeroes* are shifted into the series at the other extreme. Delaying a **DataSeries** is very useful in calculations that require a look-back Period as in

- ➔ Advancing (<<) a **DataSeries** generally implies usage of future data (peeking), and therefore advanced offset series should not be used in Strategy trading rules.

Example ([How to run Example code?](#)³⁷)

▣ C#

```
protected override void Execute()
{
    // Plot the percent change over the last 5 bar
    DataSeries delayedClose = Close >> 5;
    DataSeries pctChange = 100 * ( Close / delayedClose - 1);
    pctChange.Description = "5-day Pct Change";
    ChartPane changePane = CreatePane(40, true, true);
    PlotSeries(changePane, pctChange, Color.Red, WealthLab.LineStyle.Solid, 1);

    /* We'll cover indicators later, but notice that the same result is obtained
       from the ROC indicator which we'll plot as a histogram */
    DataSeries roc = ROC.Series(Close, 5);
    PlotSeries(changePane, roc, Color.Black, WealthLab.LineStyle.Histogram, 1);
}
```

How to: Return an "Absolute Value Series"

Apply the **DataSeries.Abs** method. In the example, the absolute value of the difference between two **DataSeries** is added to the difference to create zero values for the bars in which the difference is negative.

Example ([How to run Example code?](#)³⁷)

▣ C#

```
using System;
using System.Collections.Generic;
using System.Drawing;
using WealthLab;
using WealthLab.Indicators;

namespace WealthLab.Strategies
{
    public class MyStrategy : WealthScript
    {
        // Returns a DataSeries with the sum of the count that ds1 < ds2
        // in the specified Period
        public DataSeries LessThanCountSeries( DataSeries ds1, DataSeries ds2, int
Period )
        {
            DataSeries LTCSeries = ds2 - ds1;
            LTCSeries = LTCSeries + DataSeries.Abs(LTCSeries);
            LTCSeries = LTCSeries / LTCSeries;
            return Sum.Series(LTCSeries, Period);
        }
        protected override void Execute()
        {
            // Count how many closes are below its 20-Period moving averages
            // in the last 20 Periods
            DataSeries sma = SMA.Series(Close, 20);
            DataSeries ltc = LessThanCountSeries(Close, sma, 20);
            ChartPane ltcPane = CreatePane(40, true, false);
        }
    }
}
```

```
        PlotSeries(ltcPane, ltc, Color.Blue, WealthLab.LineStyle.Histogram, 1);  
        PlotSeries(PricePane, sma, Color.Blue, WealthLab.LineStyle.Solid, 2);  
    }  
}
```

4.2 Accessing a Single Value from a DataSeries

How to: Access a value at a specific bar

Use the square bracket `[]` indexers to access a single **DataSeries** value at the bar number specified in the brackets. We've already seen these used on core series like **Close**, **Volume**, etc. The following example shows how to obtain the most-recent value for the "average price series" that we created in a previous example.

Example ([How to run Example code?](#)³⁴)

```
C#
protected override void Execute()
{
    DataSeries avgPrice = (High + Low) / 2;
    avgPrice.Description = "Average Price";
    PlotSeries(PricingPane, avgPrice, Color.Blue, WealthLab.LineStyle.Solid, 1);

    int lastBarNumber = Bars.Count - 1;
    double lastAvg = avgPrice[lastBarNumber];
    DrawLabel(PricingPane, "The value of avgPrice on the final bar is "
        + lastAvg, Color.Black);
}
```

We'll cover [Technical Indicators](#)³⁴ later, but we'll just point out here that it's often convenient to access an indicator's value at a specified bar directly from the **Series** method, as shown in this simple [screen](#)¹²⁴ for 14-Period **RSI** values under 30.

Example ([How to run Example code?](#)³⁵)

```
C#
protected override void Execute()
{
    int bar = Bars.Count - 1;
    if (RSI.Series(Close, 14)[bar] < 30)
        BuyAtMarket(bar + 1);
}
```

4.3 Filling a Custom DataSeries

How to: Create a DataSeries filled with zeroes

Often it's convenient to create single-use DataSeries or custom indicators "on the fly" with a few lines of code as shown in [How to Return an Absolute Value Series](#)^[3] using series operators. However, other indicators require decisions when assigning values at each bar (if this then that). In this case, start with a DataSeries initialized to all zero values.

Example 1 ([How to run Example code?](#)^[3])

One way is to simply subtract an existing series from itself.

```
C#
protected override void Execute()
{
    DataSeries random = Close - Close;
}
```

Example 2 ([How to run Example code?](#)^[3])

Preferably, create a new DataSeries object by passing the Bars object of the primary series to the DataSeries function. This method provides the ability to give a friendly description to the new series as well.

```
C#
protected override void Execute()
{
    // Create, fill, and, plot a random DataSeries with values between 50 and 100,
    // otherwise zero.
    DataSeries random = new DataSeries(Bars,
        "Initially zeroes, filled by random numbers");
    Random r = new Random();
    for (int bar = 0; bar < Bars.Count; bar++)
    {
        double rVal = 100 * r.NextDouble();
        if (rVal > 50)
            random[bar] = rVal;
    }
    ChartPane rPane = CreatePane( 40, true, true );
    PlotSeries(rPane, random, Color.Blue, WealthLab.LineStyle.Histogram, 1);
}
```

How to: Change the value of items in a DataSeries

You're free to change the values of any **DataSeries** - even the core OHLC/V series as you wish. The example shows how to detect and correct "unreasonable" data spikes that would otherwise distort a chart's y-axis scaling.

Example ([How to run Example code?](#)^[3])

```
C#
protected override void Execute()
{
    const double Sensitivity = 2.5; // less sensitive with bigger numbers
    DataSeries atr = ATR.Series( Bars, 5 );
    DataSeries hi = Highest.Series(High, 3);
    DataSeries lo = Lowest.Series(Low, 3);
    for (int bar = 10; bar < Bars.Count; bar++){
```

```
    if (High[bar] > High[bar-1] + Sensitivity * atr[bar])
        High[bar] = Math.Max( hi[bar-1], Math.Max(Open[bar], Close[bar]) );

    if (Low[bar] < Low[bar-1] - Sensitivity * atr[bar])
        Low[bar] = Math.Min( lo[bar-1], Math.Min(Open[bar], Close[bar]) );
}
```

4.4 Accessing Secondary Symbols

A secondary or *external* symbol is any symbol that is not the primary chart symbol. If the external symbol doesn't exist in the current DataSet, Wealth-Lab searches other DataSets in alphabetical order. If that fails to locate the symbol's data, on-demand and/or streaming requests for data are created as applicable.

How to: Access DataSeries of Secondary Symbols

Method 1: **GetExternalSymbol()** returns a **Bars** object for the specified symbol in the current time frame. Use this method if you don't need to create trades on the secondary symbol, but just need to reference its data for indicators or plotting.

In the example, we access **Bars** of a benchmark symbol and use its **Close** series to calculate a relative index using a *Most Anchored Momentum* technique. Conveniently, using the same **Bars** reference for the benchmark, we can plot the benchmark symbol as well.

Example ([How to run Example code?](#)³⁷)

■ C#

```
protected override void Execute()
{
    // Create a Most-Anchored Momentum Index with a benchmark symbol
    Bars bmBars = GetExternalSymbol(".SPX", true);
    DataSeries mam = Close / bmBars.Close;
    mam = EMA.Series(mam, 10, EMACalculation.Modern) / SMA.Series(mam, 20);
    mam.Description = "Most Anchored Momentum: " + Bars.Symbol + " - "
        + bmBars.Symbol;

    ChartPane mamPane = CreatePane(40, true, true);
    PlotSeries(mamPane, mam, Color.Blue, LineStyle.Solid, 2);

    ChartPane bmPane = CreatePane(40, true, true);
    PlotSymbol(bmPane, bmBars, Color.LightGreen, Color.Black);
}
```

Method 2: **SetContext()** changes the context of the script to a symbol other than the primary or "clicked" symbol such that the **Bars** reference is [temporarily] re-assigned to the new symbol. After a call to **SetContext()** you can create trades with the new symbol in the normal manner. See [Programming Trading Strategies](#)⁴⁵.

Example ([How to run Example code?](#)³⁷)

■ C#

```
protected override void Execute()
{
    SetContext("MSFT", true);
    Bars msft = Bars; // Save the Bars reference to MSFT
    RestoreContext(); // Go back to the primary chart symbol

    // Plot the Close series from MSFT in the main Price Pane
    PlotSeries(PricePane, msft.Close, Color.Blue, LineStyle.Solid, 2);

    // Create an alert for the current symbol
    int bar = Bars.Count - 1;
```



```

BuyAtMarket(bar + 1, Bars.Symbol);

// Create an alert for MSFT
SetContext("MSFT", true);
BuyAtMarket(bar + 1, Bars.Symbol);
RestoreContext();
}

```

- ➔ Normally and in the examples above, you'll set the boolean *synchronize* parameter in `GetExternalSymbol()` and `SetContext()` to *true*. See the next topic on [Secondary Series Synchronization](#)¹⁸ to learn about a case in which you may prefer to disable synchronization in order to create indicators using the raw **Bars** or **DataSeries** of the external symbol.

4.4.1 Secondary Series Synchronization

Whenever you use data from more than a single market in your trading strategy you run into the issue of data synchronization. For example, you might want to make sure an underlying index such as .NDX is rising before pulling the trigger on a long trade in INTC. This topic describes how Wealth-Lab synchronizes multiple data series, and how you can take more control over the process.

How External Data Series are Accessed

You can [access external data series](#)¹⁸ in your scripts via the `GetExternalSymbol()` and `SetContext()` functions. See the QuickRef (*F11*) for details and example use.

Default Synch Behavior

The default synchronization behavior covers most needs, so you usually don't have to even worry about the issue. The default behavior is based on the concept of *Primary Bars*. The Primary **Bars** object is always from the symbol that was used to execute the Strategy. In Wealth-Lab Pro, this is the symbol that you clicked on the Data Panel to execute the system, the symbol you entered in the entry field before you pressed "Go", or the current symbol under test in a Multi-Symbol Backtest.

Default synchronization aligns all external **Bars** and **DataSeries** to the Primary **Bars** by comparing the date-time of each bar.

- Any date times that are outside the range of the Primary **Bars** are eliminated from the external series.
- Dates that exist in the Primary Series but not in the external series are added to the external series. The values are copied from the previous bar of the external series.
- Dates that exist in the external series but not in the Primary **Bars** are eliminated from the external series.

The end result is that all external series are synchronized bar by bar with the Primary **Bars**. This ensures that the **Bars.Count** for the external series matches perfectly. Also, you can create and plot indicators based on the external series without worry. The following illustration shows how the synchronization takes place between a Primary and an external **DataSeries**.

Primary DataSeries

Date	1	2	3	5	8	9	10	11	12	15	16
Value	1	2	3	4	5	6	7	8	9	10	11

External DataSeries (Prior to Synch)

Date	1	2	3	4	5	8	10	11	12	15	16	17
Value	20	21	22	23	24	25	26	27	28	29	30	31

External DataSeries (After Synch)

Date	1	2	3	5	8	9	10	11	12	15	16
Value	20	21	22	24	25	25	26	27	28	29	30

The External Series was affected in the following ways:

- The bar for date "4" was eliminated because it didn't exist in the Primary Series.
- Bar "9" was inserted, and the price value was adopted from Bar "8".
- Bar "17" was eliminated, because it was outside the bounds of the Primary Series.

Impact on Indicator Creation

Let's say that we now want to create a 5-bar moving average on the external series. The moving average will be composed based on the synchronized price bars. This will cause the "25" value from bar 8 to get an extra weighting, since that bar was duplicated to synchronize with bar 9 in the Primary Series. This may or may not be correct. On one hand, you may have a missing bar of data in your external series, and duplicating the previous bar is the best way of handling the situation (short of correcting the data series manually and adding the missing bar). On the other hand, the external series may have actually not traded that day for some reason or another.

Fortunately, Wealth-Lab allows you to cope with either scenario. First, here's the moving average created after using the default synch behavior:

Date	1	2	3	5	8	9	10	11	12	15	16
Value	20	21	22	24	25	25	26	27	28	29	30
SMA5					22.4	23.4	24.4	25.4	26.2	27	28

Changing the Default Behavior

WealthScript provides a way to disable the default behavior and perform the synchronization *after* calculating indicators on the raw external series. This involves disabling secondary symbol synchronization when accessing the a symbol's **Bars** or **DataSeries** and later synchronizing with the **Synchronize()** function.

1. To disable the default synchronization behavior, pass *false* to the *synchronize* parameter in **GetExternalSymbol()** and **SetContext()**.
2. Create the indicators on the unsynchronized (raw) external **Bars** or **DataSeries** reference from Step 1.
3. Call **Synchronize()** to synchronize the external series to create a synchronized **Bars** or **DataSeries** reference.

Synchronize() synchronizes the external series just like the default behavior, but it also

synchronizes the indicators based on the external series. In this way you can be sure that the indicators are accurate if the external series has different trading days than the Primary **Bars**.

Here's how the moving average would have changed if we calculate it using the raw external series data, and then synchronize it. Note that the moving average value for the inserted bar 9 is copied from the previous moving average value.

Date	1	2	3	5	8	9	10	11	12	15	16
Value	20	21	22	24	25	25	26	27	28	29	30
SMA5					23	23	24	25	26	27	28

How to: Create an indicator from a secondary symbol's raw DataSeries

The following script obtains the Nasdaq 100 series, creates a 50 bar moving average, and then synchronizes and plots.

Example ([How to run Example code?](#)^{3b})

```

C#
protected override void Execute()
{
    // Access the secondary symbol, but don't synch
    Bars naz = GetExternalSymbol(".NDX", false);

    // Calculate it's moving average series
    DataSeries nazSMA = SMA.Series(naz.Close, 50);

    // Now synchronize the Bars and average DataSeries
    naz = Synchronize(naz);
    nazSMA = Synchronize(nazSMA);

    ChartPane nazPane = CreatePane(75, true, true);
    PlotSeries(nazPane, nazSMA, Color.Blue, WealthLab.LineStyle.Solid, 1);
    PlotSymbol(nazPane, naz, Color.LightBlue, Color.Black);
}

```

Note on Primary Bars

The **Synchronize()** function synchronizes an external **Bars** or **DataSeries** to the current Primary **Bars**. Be careful, because the Primary **Bars** can be changed via a call to **SetContext()**. If you want to synchronize to the original chart's **Bars**, be sure to call **RestoreContext()** after using **SetContext()**. This behavior is intentional and allows more complex pairs trading analysis scripts to have precise control of external series synchronization.

Related (QuickRef): **Bars.FirstActualBar**

5 Painting the Chart

WealthScript provides a set of functions to control how chart information is displayed. You can plot indicators, create new chart panes, add text, annotations to bars and drawing objects, or even include bitmap images.

Whenever you need to programmatically perform display tasks in the Chart window, open the *Cosmetic Chart* category in the QuickRef (*F1?*) to find the function that serves the purpose.

5.1 Chart Panes

Three types of chart panes exist, but all are **ChartPane** objects.

Price Pane

Bars OHLC series are displayed in the Price Pane according to the selected Chart Style. By default, all Strategy windows have a Price pane, which is referenced in Strategy code as **PricePane**.

Volume Pane

Bars.Volume appears as a histogram in the Volume Pane. By default, all Strategy windows have a Volume pane, which is referenced in Strategy code as **VolumePane**.

Custom Panes

All other panes are custom panes created in Strategy code by calling **CreatePane**, which returns a **ChartPane** object.

How to: Create a Custom Pane

To create a new pane, call the **CreatePane** function. In the example we create three custom panes - two above the Price pane and one below the Volume pane. Panes are "stacked" above/below the Price and Volume panes in the order created.

Example ([How to run Example code?](#) )

 C#

```
protected override void Execute()
{
    bool abovePricePane = true;
    bool displayGrid = true;
    ChartPane myCustomPane1 = CreatePane(40, abovePricePane, displayGrid);
    DrawLabel(myCustomPane1, "myCustomPane1", Color.Black);

    // Create a second pane
    displayGrid = false;
    ChartPane myCustomPane2 = CreatePane(40, abovePricePane, displayGrid);
    DrawLabel(myCustomPane2, "myCustomPane2", Color.Black);

    // Create a third pane
    abovePricePane = false;
    ChartPane myCustomPane3 = CreatePane(40, abovePricePane, displayGrid);
    DrawLabel(myCustomPane3, "myCustomPane3", Color.Black);
}
```

How to: Hide the Volume Pane

To hide the volume pane, call **HideVolume()** *once*, anywhere in your Strategy code.

Example ([How to run Example code?](#) )

 C#

```
protected override void Execute()
{
    HideVolume();
}
```

5.2 Charting Multiple Symbols

How to: Plot a Secondary Symbol

If you don't require access to data of a secondary (external) symbol, but only want to plot the symbol, use the `PlotSymbol()` method. `PlotSymbol()` plots any Bars object using the currently-selected Chart Style (Candlestick, OHLC, etc.).

➔ `PlotSymbol()` is non-functional for Trending Chart Styles (Renko, Kagi, Line Break, and Point and Figure) due to their irregular x-axes.

Example ([How to run Example code?](#))

```
C#
protected override void Execute()
{
    //Plot Microsoft data in a new pane
    Bars msft = GetExternalSymbol("MSFT", true);
    ChartPane msftPane = CreatePane( 100, false, true);
    PlotSymbol( msftPane, msft, Color.DodgerBlue, Color.Red);
}
```

How to: Plot a Synthetic Symbol

Chart a *synthetic symbol* using any four series as the OHLC DataSeries via the `PlotSyntheticSymbol()` method. No checking is performed to ensure properly-formed bars.

Example ([How to run Example code?](#))

```
C#
/* Plot a Heikin-Ashi chart above the main PricePane */
protected override void Execute()
{
    // Add 0 to create a copy of a DataSeries based on a core series
    DataSeries HO = Open + 0;
    DataSeries HH = High + 0;
    DataSeries HL = Low + 0;
    DataSeries HC = (Open + High + Low + Close) / 4;

    for (int bar = 1; bar < Bars.Count; bar++)
    {
        double o1 = HO[ bar - 1 ];
        double c1 = HC[ bar - 1 ];
        HO[bar] = ( o1 + c1 ) / 2;
        HH[bar] = Math.Max( HO[bar], High[bar] );
        HL[bar] = Math.Min( HO[bar], Low[bar] );
    }
    ChartPane haPane = CreatePane(100, true, true);
    PlotSyntheticSymbol(haPane, "Heikin-Ashi", HO, HH, HL, HC, Volume,
        Color.DodgerBlue, Color.Red);
}
```

How to: Compare to a Benchmark Symbol

Re-scale the benchmark symbol's `Bars` to a reference price and then apply the `PlotSyntheticSymbol()` method.

Example ([How to run Example code?](#)³)

■ C#

```
public void PlotBenchMark(string symbol, DateTime fromDate)
{
    if (Bars.Symbol != symbol)
    {
        int bar = Bars.ConvertDateToBar(fromDate, false);
        double refPrice = Close[bar];
        Bars bmBars = GetExternalSymbol(symbol, true);
        double factor = refPrice / bmBars.Close[bar];
        DataSeries dsO = bmBars.Open * factor;
        DataSeries dsH = bmBars.High * factor;
        DataSeries dsL = bmBars.Low * factor;
        DataSeries dsC = bmBars.Close * factor;
        PlotSyntheticSymbol(PricePane, symbol + " (relative)",
            dsO, dsH, dsL, dsC, Volume, Color.Blue, Color.Gray);
    }
}

protected override void Execute()
{
    // Compare to a benchmark symbol from 1/1/2006
    PlotBenchMark("MSFT", new DateTime(2006, 1, 1));
}
```

5.3 Plotting DataSeries

How to Plot a DataSeries

To plot a DataSeries:

1. Create the DataSeries, and if required, finish assigning values to all bars before proceeding to Step 3.
2. Optional: If the DataSeries is not to be plotted in the Price or Volume panes, [create a custom ChartPane](#)^[23].
3. Call one of the following PlotSeries methods: `PlotSeries()`, `PlotSeriesFillBand()`, `PlotSeriesDualFillBand()`, and `PlotSeriesOscillator()`.

➔ `PlotSeries*` statements should be called only once for each plot. They never belong in a `Bars.Count` loop.

In previous topics, we've already seen various examples of plotting a `DataSeries` using the `PlotSeries()`. See the QuickRef for more examples of each type of `PlotSeries*` method. (Category: Cosmetic Chart).

How to: Plot Volume above the PricePane

Hide the volume pane, create a custom pane, and plot `Bars.Volume` in the new pane.

Example ([How to run Example code?](#)^[3])

```
C#
protected override void Execute()
{
    HideVolume();
    ChartPane myVolumePane = CreatePane(40, true, true);
    PlotSeries(myVolumePane, Bars.Volume, Color.Black,
    WealthLab.LineStyle.Histogram, 5);
    for (int bar = 0; bar < Bars.Count; bar++)
        if (Close[bar] > Open[bar])
            SetSeriesBarColor(bar, Bars.Volume, Color.Green);
        else
            SetSeriesBarColor(bar, Bars.Volume, Color.Red);
}
```

How to: Plot Strategy Metrics

Since Wealth-Lab processes Strategies in Raw Profit mode and each trade is arbitrarily assigned a size of 1 share/contract, it's not possible to directly access metrics in Strategy code that are a function of Position Sizing, Equity, Cash, or Drawdown. Instead, metrics such as these are derived during the Strategy's post-processing task in a Performance Visualizer (See: [AP/5](#)^[132]). Nonetheless, you could plot a running metric of "Average Bars Held" or "Win Rate" as the following example shows.

Example ([How to run Example code?](#)^[3])

```
C#
protected override void Execute()
{
    /*
    Your Trading Strategy here
    */

    PlotWinRate();
}
```



```

}
public void PlotWinRate()
{
    DataSeries winRate = new DataSeries(Bars, "Win Rate");
    DataSeries tradeTotal = winRate + 0;
    foreach (Position p in Positions)
    {
        if (p.ExitBar > -1){
            tradeTotal[p.ExitBar] += 1;
            if (p.NetProfitPercent > 0)
                winRate[p.ExitBar] += 1;
        }
    }
    for (int bar = 1; bar < Bars.Count; bar++){
        tradeTotal[bar] = tradeTotal[bar] + tradeTotal[bar-1];
        winRate[bar] = winRate[bar] + winRate[bar - 1];
    }
    winRate = 100 * winRate / tradeTotal;
    winRate.Description = "Win Rate (%)";
    ChartPane winPane = CreatePane(40, true, true);
    PlotSeries(winPane, winRate, Color.Green, WealthLab.LineStyle.Solid, 1);
}

```

5.3.1 Controlling the y-scale

Wealth-Lab automatically scales the y-axis for each pane so that the plots of all price data and indicators are visible. It's not possible to zoom the y-axis scale such that data is visible in one part of the chart, but not another.

On the other hand, by using the `SetPaneMinMax()` method, you can expand a pane's y-axis scale so that it remains constant over a wider range of values. For example, if an **RSI** is oscillating between 50 and 90 for a long period of time, it's likely that Wealth-Lab will automatically rescale the y-axis for the visible chart between those values. But if you're only concerned with the **RSI** when it crosses below 30, you could force the y-axis' minimum scale to 30 or less to gain perspective on the current values.

Example

See `SetPaneMinMax()` in the QuickRef.

5.4 Writing Text

For programmatically writing text on a chart, the following methods are available:

- AnnotateBar()** Provides a method to write a string above/below the high/low of a bar in the **PricePane**. Multiple calls on the same bar result in "stacked text".
- AnnotateChart()** Similar to **AnnotateBar()** but more flexible since you can specify the ChartPane, bar, alignment relative to bar, and price to place text.
- DrawLabel()** Multiple labels are stacked in the upper left corner of the Price pane, below the symbol. Labels are automatically drawn for plotted **DataSet**s using their **Description** property.
- DrawText()** Provides a method to write a string to a position defined by x, y coordinates in pixels, where 0, 0 is the upper left corner of the chart.

How to: Write Vertical Text

When space is tight and you need many annotations, take advantage of the **AnnotateBar()** method to stack text - one letter at a time as demonstrated here:

Example ([How to run Example code?](#) )

```

C#
// A vertical labeling routine
public void VerticalText(int bar, string str, bool abovePrices, Color color)
{
    if (abovePrices)
        for (int n = str.Length - 1; n >= 0; n--)
            AnnotateBar(str.Substring(n, 1), bar, abovePrices, color);
    else
        for (int n = 0; n < str.Length; n++)
            AnnotateBar(str.Substring(n, 1), bar, abovePrices, color);
}

protected override void Execute()
{
    // Make extra room for peak label
    HidePaneLines();
    CreatePane(3, true, false);

    // identify 5% peak and trough bars
    int bar = Bars.Count - 1;
    while (bar > 0) {
        bar = (int)PeakBar.Series(Close, 5, PeakTroughMode.Percent)[bar];
        if (bar > -1) VerticalText(bar, "PEAK", true, Color.Red);
    }
    bar = Bars.Count - 1;
    while (bar > 0) {
        bar = (int)TroughBar.Series(Close, 5, PeakTroughMode.Percent)[bar];
        if (bar > -1) VerticalText(bar, "TROUGH", false, Color.Green);
    }
}

```

How to: Control a DataSet Plot Label

The plot label is the **DataSet.Description** property. Wealth-Lab assigns a default description to all **DataSet**s upon their creation, but you can change it to any "friendly"

string to use as the series' plot label.

Tip: The plot label is given the same color assigned to the series plot.

Example ([How to run Example code?](#) )

■ C#

```
protected override void Execute()
{
    // Create an Average Price DataSeries, delayed by 1 bar
    DataSeries avgPrice = (High + Low) / 2 >> 1;
    string defaultDescription = avgPrice.Description;

    // assign a friendly Description
    avgPrice.Description = "Average Price (1 bar delay)";
    PlotSeries(PricingPane, avgPrice, Color.Blue, WealthLab.LineStyle.Solid, 1);
    DrawLabel(PricingPane, "Default Description: "+defaultDescription, Color.Red);
}
```

5.5 Drawing Objects

When drawing objects, the "hard part" is simply to find the points to connect. The `PeakBar()` and `TroughBar()` functions are convenient for this purpose, but *any method* can be used to find interesting coordinates on a chart for drawing flag formations, triangles, rectangles, etc.

Each of the following methods can be used in any `ChartPane`. See the QuickRef ([F11](#)) for more details and an example for each.

<code>DrawCircle()</code>	Drawing small circles are useful for highlighting key points or targets in a chart.
<code>DrawEllipse()</code>	For an ellipse, you need only to specify opposing corners of a the imaginary rectangle that bounds the ellipse.
<code>DrawLine()</code>	Draws a line that connects two points. The line can be extended with the same slope by finding the y value on a subsequent bar via the <code>LineExtendY()</code> or <code>LineExtendYLog()</code> methods and passing the result to another call to <code>DrawLine()</code> .
<code>DrawHorzLine()</code>	A convenient method for drawing support and resistance lines at a fixed value across the entire chart.
<code>DrawPolygon()</code>	Provides a method to draw any n-sided polygon.

How to: Draw and Extend a Trendline

First draw the line between two known points with `DrawLine()`. Next, find the y value at the bar to which you want to extend the trendline, and use that value in `DrawLine()` again. We can simplify this procedure by wrapping it into the Example's `DrawExtendedTrendline()` method.

Example ([How to run Example code?](#) )

```

C#
protected override void Execute()
{
    // Find the last two 5% peaks and draw a line to the end of the chart
    int bar = Bars.Count - 1;
    int pb2 = (int)PeakBar.Value(bar, High, 5, PeakTroughMode.Percent);
    int pb1 = pb2;
    if (pb2 > -1)
        pb1 = (int)PeakBar.Value(pb2, High, 5, PeakTroughMode.Percent);
    if ((pb2 == -1) || (pb1 == -1))
        DrawLabel(PricePane, "Could not find two 5% peaks", Color.Red);
    else
        DrawExtendedTrendline(PricePane, pb1, High[pb1], pb2, High[pb2], bar,
            Color.Blue);
}

public void DrawExtendedTrendline(ChartPane pane, int bar1, double val1,
    int bar2, double val2, int bar3, Color color)
{
    // draw the line for the 2 known points
    DrawLine(pane, bar1, val1, bar2, val2, Color.Blue, LineStyle.Solid, 2);

    // extend the line to the specified bar3
    double val3 = LineExtendY(bar1, val1, bar2, val2, bar3);
    if (pane.LogScale)

```

```

        val3 = LineExtendYLog(bar1, val1, bar2, val2, bar3);
        DrawLine(pane, bar2, val2, bar3, val3, color, LineStyle.Solid, 1);
    }

```

How to: Draw Speed-Resistance Lines

All of the manual drawing objects can be drawn automatically and without subjectivity, i.e., according to a fixed set of rules that you program. Here's one such example for S-R lines.

[Example \(How to run Example code?\)](#) 

```

C#
using System;
using System.Collections.Generic;
using System.Text;
using System.Drawing;
using WealthLab;
using WealthLab.Indicators;

namespace WealthLab.Strategies
{
    public class SpeedResistanceDemo : WealthScript
    {
        public void SpeedResistance(int bar1, double price1, int bar2, double
price2 )
        {
            // You can extend the lines drawn with this procedure using LineExtendY or
            LineExtendYLog for semi-log charts
            double delta = price1 - price2;
            DrawLine(PricingPane, bar1, price1, bar2, price2, Color.Red,
WealthLab.LineStyle.Solid, 2);
            DrawLine(PricingPane, bar1, price1, bar2, price2 + delta / 3, Color.Blue,
WealthLab.LineStyle.Dots, 1);
            DrawLine(PricingPane, bar1, price1, bar2, price2 + delta * 2 / 3,
Color.Blue, WealthLab.LineStyle.Dots, 1);
        }
        protected override void Execute()
        {
            // Find the latest 8% peak and trough and draw the S-R lines between them
            int bar = Bars.Count - 1;
            int b1 = (int)PeakBar.Value(bar, Close, 8.0,
WealthLab.Indicators.PeakTroughMode.Percent);
            int b2 = (int)TroughBar.Value(bar, Close, 8.0,
WealthLab.Indicators.PeakTroughMode.Percent);
            if (b2 > b1)
                SpeedResistance( b1, Close[b1], b2, Close[b2] );
            else
                SpeedResistance( b2, Close[b2], b1, Close[b1] );
        }
    }
}

```

How to: Highlight an area with a Polygon

Once again, using the convenient peaks and troughs, we can identify 4 (or more) corners that define a polygon.

Example ([How to run Example code?](#)^{3b})

■ C#

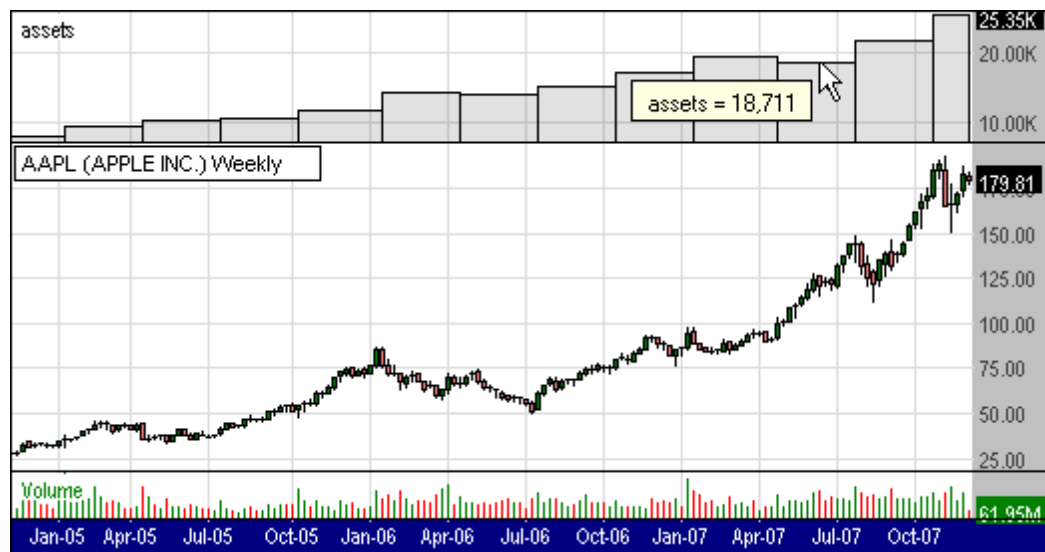
```
protected override void Execute()
{
    // Find the last two 5% peaks
    int bar = Bars.Count - 1;
    int pb2 = (int)PeakBar.Value(bar, High, 5,
WealthLab.Indicators.PeakTroughMode.Percent);
    int pb1 = pb2;
    if (pb2 > -1)
        pb1 = (int)PeakBar.Value(pb2, High, 5,
WealthLab.Indicators.PeakTroughMode.Percent);
    if ((pb2 == -1) || (pb1 == -1)) {
        DrawLabel(PricePane, "Could not find two 5% peaks", Color.Red);
        return;
    }
    int tb2 = (int)TroughBar.Value(bar, Low, 5,
WealthLab.Indicators.PeakTroughMode.Percent);
    int tb1 = tb2;
    if (tb2 > -1)
        tb1 = (int)TroughBar.Value(tb2, Low, 5,
WealthLab.Indicators.PeakTroughMode.Percent);
    if ((tb2 == -1) || (tb1 == -1)) {
        DrawLabel(PricePane, "Could not find two 5% troughs", Color.Red);
        return;
    }
    DrawPolygon(PricePane, Color.LightBlue, Color.LightBlue,
WealthLab.LineStyle.Invisible, 1, true,
        pb1, High[pb1], pb2, High[pb2], tb2, Low[tb2], tb1, Low[tb1]);
}
```

5.6 Plotting Fundamental Items

Wealth-Lab Pro Version 6 has a dedicated plot function, `PlotFundamentalItems()`, for plotting the individual items in the *Fidelity Fundamental Data for Securities* category items such as "assets", "net income", "fiscal year", etc. While you can plot fundamental items using `PlotSeries()` after having converted it to a `DataSeries` with the `FundamentalDataItems()` function, `PlotFundamentalItems()` is superior because divisions in the plot clearly indicate fundamental report dates.

Example ([How to run Example code?](#)³⁷)

```
C#
protected override void Execute()
{
    ChartPane fundPane = CreatePane(40, true, true);
    PlotFundamentalItems(fundPane, "assets", Color.Black,
        WealthLab.LineStyle.Solid, 1);
}
```



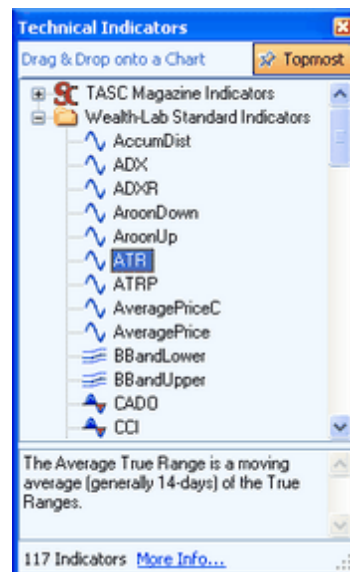
Divisions in the "block plot" indicate reporting dates.

See also: [Fundamental Data Analysis](#)⁷⁴

6 Indicators

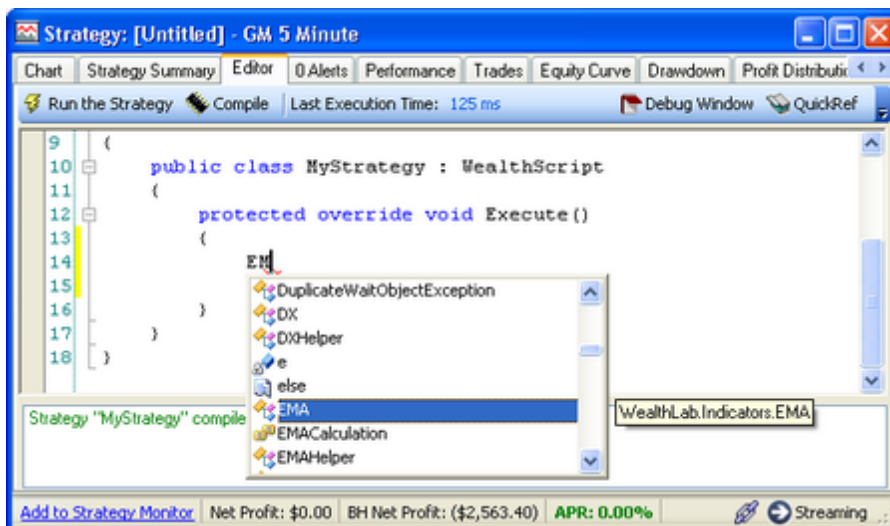
All Wealth-Lab technical indicators that are visible in the Technical Indicators dialog (right) are integrated as *Indicator Libraries*, which are .NET components. Upon startup, Wealth-Lab Pro detects indicator libraries that exist in the main installation directory and displays them as folders in the Technical Indicators dialog.

See [APIS](#) ¹³².



Indicator Syntax

As with other classes, selecting an indicator and viewing its syntax is integrated with the Strategy Editor. Just strike *Ctrl + Space* to bring up the code completion pop up. Type a few letters to narrow down the search as shown. After locating the item, pressing the Tab or Enter key completes the item.



Code Completion

```
protected override void Execute()
{
    EMA ema = EMA.
}
```

As you'd expect, typing the period delimiter automatically lists class properties and methods. As you'll soon learn, all indicators have a **Series** method, and some also have a **Value** method. If the list becomes hidden, place the cursor after the

period and strike *Ctrl + Space* to show the list again.

Upon arriving at the parameter list, typing the opening parenthesis reveals the full indicator syntax. If the tooltip syntax disappears (due to moving the cursor out of the parameter list, for example), place the cursor back in the parameter list and strike *Ctrl + Shift + Space* to view the tooltip again. The bold type shows indicates the current parameter at the cursor's position.

```
protected override void Execute()
{
    EMA ema = EMA.Series(Close, 20
    WealthLab.Indicators.EMA EMA.Series (WealthLab.DataSeries source, int period, WealthLab.Indicators.EMACalculation calcType)
```

Tip:

Make it a habit to type a white space after each comma in a parameter list. This causes enumerations to appear automatically for enumerated parameters as shown below.

```
EMA ema = EMA.Series(Close, 20, |
    WealthLab.Indicators.EMACalculation.Legacy
    WealthLab.Indicators.EMACalculation.Modern
```

6.1 Series Method

Indicator classes implement a static method called **Series** that returns an instance of the indicator, which is derived from the **DataSeries** class. This will either be a new instance, or an instance that was found in the Cache. The caching mechanism saves computer resources by ensuring that multiple copies of the same indicator are not created.

How to: Access a complete Indicator DataSeries

Two methods are possible for returning an indicator's DataSeries; either method is equally acceptable.

- Method 1: Use the **new** operator to return an instance of the indicator derived from **DataSeries**. In addition to being more "professional", Method 1 allows you to provide a friendly string Description in the same call.
- Method 2: Pre-.NET legacy Wealth-Lab customers may find comfort in the static **Series** notation method to return a **DataSeries** type.

Example ([How to run Example code?](#)³)

■ C#

```
protected override void Execute()
{
    /* Comparison of the two methods */

    // Method 1
    SMA sma = new SMA(Close, 50, "50-day SMA");
    PlotSeries(PricePane, sma, Color.Blue, WealthLab.LineStyle.Solid, 2);

    // Method 2
    DataSeries sma2 = SMA.Series(Close, 20);
    sma2.Description = "20-day SMA";
    PlotSeries(PricePane, sma2, Color.Red, WealthLab.LineStyle.Solid, 2);
}
```

6.2 Static Value Method

The **Value** method is optional for indicators. A static **Value** method allows indicators to calculate and return a value "on the fly" for a specified bar number, without having to go through the overhead of creating a complete instance of the indicator. This corresponds to the *SingleCalcMode* that was available in the pre-.NET version of Wealth-Lab. Since it's not cached, the **Value** method can also recalculate an indicator whose **DataSet** parameters have changed value.

If supported by the indicator, the **Value** method will accept the same parameters as the indicator's **Series** method with the addition of an *int* parameter to specify a bar number. The result is a type *double* value.

How to: Calculate a single indicator value

Let's give an example using both the **Series** and **Value** methods. Assume that you're only interested in determining if a stock is currently trading above its 50-day **SMA**.

Example, Method 1 ([How to run Example code?](#)³¹)

```

C#
protected override void Execute()
{
    int bar = Bars.Count - 1;
    DataSet sma = SMA.Series(Close, 50);
    if (Close[bar] > sma[bar])
        DrawLabel(PricePane, "Above 50-day SMA", Color.Blue);
    else
        DrawLabel(PricePane, "Below 50-day SMA", Color.Red);
}

```

Example, Method 2 ([How to run Example code?](#)³¹)

```

C#
protected override void Execute()
{
    int bar = Bars.Count - 1;
    double smaVal = SMA.Value(bar, Close, 50);
    if (Close[bar] > smaVal)
        DrawLabel(PricePane, "Above 50-day SMA", Color.Blue);
    else
        DrawLabel(PricePane, "Below 50-day SMA", Color.Red);
}

```

While both methods produce equivalent results, recall that the **Series** method returns a **DataSet** with indicator values calculated for every bar. Since we only need one value in the example, there's no reason to commit to the processing and memory overhead of the **Series** method when only a single **SMA** value is required.

6.3 Custom Indicators

In Wealth-Lab Pro Version 6 only formal, compiled indicator libraries exist that conform to [Indicator Library](#)^[132] specifications. You can, of course, create your own conforming custom library namespace for indicators that you frequently use. In many cases, however, it's simply not worth the effort to formalize an indicator that can be created by a few simple, easy-to-remember math operations.

For this reason, we refer to *custom indicators* as **DataSeries** that are created "on the fly" rather than those that are formalized in a compiled library. The ["typical price" series](#)^[12] is one such example that we've already seen.

How to: Create a "semi-formal" Custom Indicator

A semi-formal custom indicator doesn't exist in a proper [Indicator Library](#)^[132], but it's one that derives from the **DataSeries** class in the same way as a fully-integrated indicator. You might choose the semi-formal method, as shown by **SumTest** in the following example, if you need to create a **DataSeries** and later fill its elements with values. The alternative is just as effective without the formalities of creating a class for the indicator by starting with a **DataSeries** of zeroes created by subtracting one of the core **DataSeries** from itself. Both methods are demonstrated below.

Example ([How to run Example code?](#)^[37])

```
C#
/* Replicating the Sum.Series indicator */
using System;
using System.Collections.Generic;
using System.Text;
using System.Drawing;
using WealthLab;
using WealthLab.Indicators;

namespace WealthLab.Strategies
{
    public class SumTest : DataSeries
    {
        public SumTest ( DataSeries ds, int period ) : base(ds, "My Sum.Series")
        {
            double sum = 0;
            for (int bar = 0; bar < period; bar++)
                sum += ds[bar];
            this[period - 1] = sum;          // sum for initial period

            for (int bar = period; bar < ds.Count; bar++)
                this[bar] = this[bar - 1] - ds[bar - period] + ds[bar];
        }
        public static SumTest Series( DataSeries ds, int period )
        {
            SumTest _SumTest = new SumTest( ds, period );
            return _SumTest ;
        }
    }

    public class MyStrategy : WealthScript
    {
        protected override void Execute()
        {
            int period = 20;
        }
    }
}
```

```

// create and plot the semi-formal Sum.Series
DataSeries sum1 = SumTest.Series(Close, period);
ChartPane cp = CreatePane(40, true, true);
PlotSeries(cp, sum1, Color.Blue, LineStyle.Solid, 1);

// create the same series "on the fly", starting with a DataSeries of
zeroes to fill
DataSeries sum2 = Close - Close;
sum2.Description = "Sum (on-the-fly)";
double sum = 0d;
for (int bar = 0; bar < period; bar++)
    sum += Close[bar];
sum2[period - 1] = sum;           // sum for initial period
for (int bar = period; bar < Bars.Count; bar++)
    sum2[bar] = sum2[bar - 1] - Close[bar - period] + Close[bar];

// plot as a histogram in the same pane for comparison
PlotSeries(cp, sum2, Color.Black, LineStyle.Histogram, 1);
    }
}
}

```

How to: Create a MACD with Custom Periods

The classical MACD, Wealth-Lab's **MACD** indicator, is based on the difference between two **EMA** indicators with exponents 0.075 and 0.15. Since a 26-period **EMA** has an exponent of 0.074074 and the 12-period has 0.153846, you can *approximate* **MACD** instead of using the classical indicator by taking the difference of 12 and 26-period **EMAs** (or any other combination of periods).

[Example \(How to run Example code?\)](#)^{3b}

```

C#
// Note: The modern calculation for the EMA exponent is 2/(1 + Period)
protected override void Execute()
{
    // Create a custom MACD indicator
    DataSeries macdX = EMA.Series(Close, 12, EMACalculation.Modern) - EMA.Series(Clo

    // Plot the custom MACD and compare to the standard one
    ChartPane macdPane = CreatePane(40, true, true);
    MACD standardMACD = new MACD(Close, "Standard MACD");
    PlotSeries(macdPane, macdX, Color.Blue, LineStyle.Solid, 1);
    PlotSeries(macdPane, standardMACD, Color.Red, LineStyle.Histogram, 1);
}

```

How to: Create a Weighted Close Indicator

The weighted close is similar to the typical price series, but gives twice the weight (influence) to the closing price of **Bars**.

[Example \(How to run Example code?\)](#)^{3b}

```

C#
protected override void Execute()
{
    DataSeries weightedClose = ( (2 * Close) + High + Low ) / 4;
    PlotSeries(PricePane, weightedClose, Color.Blue, LineStyle.Solid, 1);
}

```

How to: Create a Spread Indicator

A spread is the difference in price between two symbols. Note the use of `GetExternalSymbol()` to access `Bars` of [the secondary symbol](#)^[18], which is immediately qualified to return its `Close`.

Example ([How to run Example code?](#)^[3])

```
C#
protected override void Execute()
{
    DataSeries spread = Close - GetExternalSymbol("MSFT", true).Close;
    ChartPane sPane = CreatePane(40, true, true);
    PlotSeries(sPane, spread, Color.Blue, LineStyle.Solid, 2);
}
```

How to: Create the Elder Force Oscillator (EFO)

Dr. Elders tells us in *Come Into My Trading Room* that when a stock is trending higher and the 2-day EMA of the EFO declines below zero, it's a buy signal. Likewise, it's a sell when the trend is down and the 2-day EMA of EFO rallies above zero.

Note the use of the [offset operator](#)^[12] (`>>`) to shift the `Close` one bar to the right, allowing the creation of a DataSeries that's the difference between today's and yesterday's close. The result is multiplied by `Volume` - another 1-line indicator!

Example ([How to run Example code?](#)^[3])

```
C#
protected override void Execute()
{
    DataSeries efo = (Close - (Close >> 1)) * Volume;
    efo.Description = "Elder Force";
    DataSeries efEma = EMA.Series(efo, 2, EMACalculation.Modern);
    ChartPane efPane = CreatePane(40, true, true);
    PlotSeries(efPane, efo, Color.Black, LineStyle.Histogram, 1);
    PlotSeries(efPane, efEma, Color.Green, LineStyle.Solid, 2);
}
```

How To: Create the NRTR_WATR indicator

Russian trader Konstantin Kopyrkin has created an indicator called "NRTR_WATR": an adaptive variation of the trailing reverse technique. To gauge current volatility, the true range value is smoothed by Weighted Moving Average, thus the trailing reverse level is made adaptive to the market conditions. Crossovers and crossunders of the NRTR line can be used to trigger trend trades, as in this example, or as the basis of a stop-and-reverse (SAR) strategy.

Example ([How to run Example code?](#)^[3])

```
C#
protected override void Execute()
{
    int Period = 20;
    double Multiple = 3.0;
    int Trend = 0;
    double Reverse = 0;

    DataSeries HPrice = Highest.Series(Close, 20);
    DataSeries LPrice = Lowest.Series(Close, 20);
}
```

```

DataSeries K = WMA.Series( TrueRange.Series( Bars ), Period ) * Multiple;
DataSeries NRTR_WATR = new DataSeries( Bars, "NRTR_WATR" );

for(int bar = K.FirstValidValue; bar < Bars.Count; bar++)
{
    // Calculate NRTR_WATR Series
    if( Trend >= 0 )
    {
        HPrice[bar] = Math.Max( Bars.Close[bar], HPrice[bar] );
        Reverse = HPrice[bar] - K[bar];
        if( Bars.Close[bar] <= Reverse )
        {
            Trend = -1;
            LPrice[bar] = Bars.Close[bar];
            Reverse = LPrice[bar] + K[bar];
        }
    }
    if( Trend <= 0 )
    {
        LPrice[bar] = Math.Min( Bars.Close[bar], LPrice[bar] );
        Reverse = LPrice[bar] + K[bar];
        if( Bars.Close[bar] >= Reverse )
        {
            Trend = 1;
            HPrice[bar] = Bars.Close[bar];
            Reverse = HPrice[bar] - K[bar];
        }
    }
    NRTR_WATR[bar] = Reverse;
}
// Display the resulting NRTR_WATR data series
PlotSeries( PricePane, NRTR_WATR, Color.Teal, WealthLab.LineStyle.Dotted, 3 );

for(int bar = NRTR_WATR.FirstValidValue+1; bar < Bars.Count; bar++)
{
    if (IsLastPositionActive)
    {
        if( CrossUnder( bar, Close, NRTR_WATR ) )
            SellAtMarket( bar+1, LastPosition, "NRTR Exit" );
    }
    else if( CrossOver( bar, Close, NRTR_WATR ) )
        BuyAtMarket( bar+1, "NRTR Entry" );
}
}

```

See the Glitch Index Strategy code for more examples.

6.4 Indicators of Secondary Symbols

All indicators require a symbol's **Bars** or **DataSeries** object, consequently in either case you must use [one of the two methods](#)^[18] to access **Bars** of the secondary symbol.

How to: Create Indicators of Secondary Symbols

Once you have a reference to the secondary symbol's **Bars**, you can use it as the *source Bars or DataSeries* for any technical indicator.

Example ([How to run Example code?](#)^[3])

■ C#

```
protected override void Execute()
{
    Bars msftBars = GetExternalSymbol("MSFT", true);

    // Plot MSFT and it's 20-period SMA in a new pane
    ChartPane msftPane = CreatePane(60, true, true);
    DataSeries smaMSFT = SMA.Series(msftBars.Close, 20);
    PlotSymbol(msftPane, msftBars, Color.LightGreen, Color.Black);
    PlotSeries(msftPane, smaMSFT, Color.Blue, LineStyle.Solid, 2);

    // Plot the 10-period ATR of MSFT in another pane
    ChartPane atrPane = CreatePane(40, true, true);
    DataSeries atrMSFT = ATR.Series(msftBars, 10);
    PlotSeries(atrPane, atrMSFT, Color.Red, LineStyle.Solid, 2);
}
```


6.5 Stability of Indicators

Traders new to technical analysis (and some not so new) often assume that they can apply a technical indicator as soon as it begins churning out "valid" values. For many indicators such as **SMA**, **WMA**, **StochK**, etc., the assumption is fine. For example, the value of a 20-period Simple Moving Average (**SMA**) will *always* be the same at the end of the same 20 bars. Values of these indicators at the end of the Period are not affected by data prior to the Period.

Another large group of indicators such as **EMA**, **WilderMA**, **RSI**, **MACD**, Kalman (more below) are in fact, calculated using the indicator's own previous value(s). You're likely to find that *progressively-calculated indicators* like these generally have reduced lag, but that the *initial*/short-term values that they produce are unstable.

What does that mean?

It's a good question for which we've prepared an example using four indicators with nominal 20-period lengths. To run the example, start by setting the Data Range control to 21 fixed bars and click any symbol. Progressively change the Data Range, increasing the number of bars to 30, 40, 60, 100, and 120. With each change the script will execute and print a new set of values for the final bar in the debug window. You'll notice along the way that only one of the indicators (**SMA**) maintains a single precise value, while the others fluctuate until they stabilize on a more precise value.

Example ([How to run Example code?](#) ³⁷)

 C#

```
protected override void Execute()
{
    HideVolume();
    ChartPane pane1 = CreatePane( 25, true, true );
    ChartPane pane2 = CreatePane( 25, true, true );
    ChartPane pane3 = CreatePane( 25, true, true );
    ChartPane pane4 = CreatePane( 25, true, true );
    PrintDebug( "" );
    PrintDebug( "Bar count = " + Bars.Count );

    DataSeries ema = EMA.Series(Close, 20,
    WealthLab.Indicators.EMACalculation.Modern);
    DataSeries macd = MACD.Series(Close);
    DataSeries rsi = RSI.Series(Close, 20);
    DataSeries sma = SMA.Series(Close, 20);

    // Print the last value of each indicator to the debug window
    int bar = Bars.Count - 1;
    PrintDebug("ema = " + ema[bar]);
    PrintDebug("macd = " + macd[bar]);
    PrintDebug("rsi = " + rsi[bar]);
    PrintDebug("sma = " + sma[bar]);

    PlotSeries(pane1, ema, Color.Red, WealthLab.LineStyle.Solid, 1);
    PlotSeries(pane2, macd, Color.Blue, WealthLab.LineStyle.Solid, 1);
    PlotSeries(pane3, rsi, Color.Green, WealthLab.LineStyle.Solid, 1);
    PlotSeries(pane4, sma, Color.Fuchsia, WealthLab.LineStyle.Solid, 1);
}
```

Avoid Using Unstable Indicator Values

As we demonstrated, some indicators can produce wildly different results in the short

term. The trick is to understand the indicators that you're using, and to ignore their values until they're stable.

➔ For meaningful and reproducible results from a trading strategy, you must use indicators when they're *stable*.

The best way to ignore an indicator is to start the trading loop on a bar after which the longest indicator is estimated to be stable. For the four 20-period indicators in the example, you may have noticed that the **RSI** requires a significant amount of "seed data" to stabilize. Consequently, we would choose to start our trading loop no earlier than bar 60 and preferably bar 120 if sufficient test data is available. A reasonable rule of thumb is to ignore progressively-calculated indicators for 3 to 4 times their period.

```
protected override void Execute()
{
    /* begin trading loop */
    for(int bar = 60; bar < Bars.Count; bar++)
    {
        /* trading strategy code or rules */
    }
}
```

Which Indicators?

The following list from the Wealth-Lab Standard Indicator Library are calculated using previous values and consequently have the potential to produce unstable values at the beginning of the series.

ADX, ADXR, ATR, ATRP, CADO, DIMinus, DIPlus, DSS, EMA, FAMA, Kalman, KAMA, KST, MACD, MAMA, OBV, Parabolic, Parabolic2, RSI, StochRSI, TRIX, UltimateOsc, Vidya, Volatility, WilderMA

➔ **AccumDist** is also unstable in the sense that its calculation "accumulates" from a specific starting point. When that starting point changes, the calculation also changes for all bars that follow.

7 Programming Trading Strategies

The most powerful feature of WealthScript is the ability to embed Trading System rules in your Strategy scripts. Whenever your Strategy executes, Wealth-Lab displays all of the trades that your Strategy generated using clear buy and sell arrows on the chart. The Strategy Performance Results view also lists the overall Strategy performance, and the Trades view contains a detailed listing of all theoretical trades that were generated. See Preferences (*F12*) for Additional Performance Visualizers.

The Strategy Class



All Wealth-Lab Strategies are public classes that derive from *WealthScript*. You can add functions, procedures, variables, classes, etc. to your Strategy class, but it must contain WealthScript's `Execute()` method, which is the entry point for a Strategy's execution and where the bulk of most Strategy code is typically programmed. The installed template code, shows the typical structure of Strategy code.

➔ You can change the code template by modifying it and selecting **Edit > Set as Default Template Code**.

```
/* Strategy Code Template */
using System;
using System.Collections.Generic;
using System.Text;
using System.Drawing;
using WealthLab;
using WealthLab.Indicators;



namespace WealthLab.Strategies
{
    public class MyStrategy : WealthScript
    {
        protected override void Execute()
        {
            for(int bar = 20; bar < Bars.Count; bar++)
            {
                if (IsLastPositionActive)
                {
                    //code your exit rules here
                }
                else
                {
                    //code your entry rules here
                }
            }
        }
    }
}
```

Compiling Strategies

A script's Strategy class is compiled and instantiated when you: click  Run the Strategy,  Compile, or on demand when you open a non-compiled Strategy from the Strategy Explorer or when a Strategy Window is restored with a Workspace.

Consequently, class-scope variables are initialized only when the class is instantiated.

Run the example script by first clicking

 Run the Strategy, and thereafter by clicking , striking *F5*, or by clicking another symbol. You should notice that the variable `_scriptRunCountSinceCompile` is initialized to 0 only upon compilation and thereafter it retains the value assigned inside the Execute method.

Example ([How to run Example code?](#) )

```
C#
using System;
using WealthLab;

namespace WealthLab.Strategies
{
    public class StrategyTest : WealthScript
    {
        private int _scriptRunCountSinceCompile = 0;

        protected override void Execute()
        {
            _scriptRunCountSinceCompile++;
            PrintDebug("Runs since compile = " + _scriptRunCountSinceCompile);
        }
    }
}
```

Strategy Code Limitations

Strategies based on Portfolio Equity, Cash, or Drawdown

Due to the Wealth-Lab Pro Version 6 architecture, it is not possible to create trading strategies that enter or exit trades as a function of Portfolio Equity, which includes Portfolio Cash and Drawdown. Refer to the QuickRef when selecting WealthScript functions or object properties for information regarding their use in Strategy code.

Single-Position Strategies with multiple triggers

In one very special case for both Raw Profit and Portfolio Simulation backtest modes, it's possible for Wealth-Lab not to trigger an entry signal when expected. The following conditions are required:

1. the script uses single-position logic, i.e., can hold only one position for a particular symbol, and,
2. a trade is rejected due to Position Sizing rules, and,
3. a new entry signal is programmed to occur prior to the condition that would have exited the trade in item 2 (had that trade not been rejected).

Since Wealth-Lab always executes a Strategy's trades in Raw Profit mode, the first trade in a series of entry triggers will always result in creating a Position. If that Position is later rejected due to sizing rules or insufficient cash, *when using single-Position logic* it's not possible for any of the other potential trade triggers to have occurred until the programmed exit of the first trade because the entry logic would not be executed until that time.

As a generic example, imagine a strategy whose single-Position entry logic buys a Position on any Tuesday or Wednesday and whose exit logic sells on Friday. Raw profit processing will always result in picking up the Position on Tuesday, which causes only the exit logic to

be processed until the Position is sold on Friday. The test to enter a Position on Wednesday will never be executed - even if the simulated Position from Tuesday is rejected.

7.1 Trading Signals

At the heart of backtesting is the ability to create theoretical trades according to specified criteria. WealthScript trading signals are what you would think of as *orders* that you place with a broker. Refer to the following table.

Buy (open long Position)	AtMarket (execute at opening price)
Sell (exit long Position)	AtLimit (execute at Limit price <i>or better</i>)
Short (open short Position)	AtStop (execute at Stop price <i>or worse</i>)
Cover (exit short Position)	AtClose (execute at closing price)
Exit (can be used in place of Sell or Cover)	

All combinations of the first and second columns are possible in Wealth-Lab for a total of 16 order types (20 counting Exit); e.g., **BuyAtMarket** or **SellAtLimit** are WealthScript functions used to create and exit a long **Position**.

Signal Parameters

<i>bar</i>	All trading signals have a <i>bar</i> parameter to specify the bar on which the trade is placed. In general, AtMarket and AtClose orders are executed on the bar placed. Exceptions deal with insufficient Position size or trading equity.
<i>pos</i>	For all exit signals (Sell, Cover, or Exit) you must specify a Position . For most [single-Position] scripts, you'll most often use LastPosition . To indiscriminately exit all Positions, pass Position.AllPositions for the <i>pos</i> parameter.
<i>limitPrice</i> or <i>stopPrice</i>	You must specify a limit/stop price for AtLimit/AtStop orders just as you would for live trading. Wealth-Lab AtStop orders function as "stop market" orders. In other words, the <i>stopPrice</i> is the activation price.
<i>signalName</i>	Optional for all signals and is displayed in the Trades list's Signal Name column, making it a useful for sorting trades.

How to: Create a [Theoretical] Trade

To illustrate how trading functions work, run the following script in any chart, any scale. The script sets the "current bar" to the 10th to last bar of the chart and then executes a **BuyAtMarket** signal on the *next bar*. As we'll see, you should get into a habit of always passing *bar + 1* to trading signals, which properly simulates the way you trade in real life. In the case of this example and in backtesting in general, "trades" created on historical data are only *theoretical* in nature.

Example ([How to run Example code?](#)³)

```

C#
protected override void Execute()
{
    /* Illustrative test only */

```

```

ClearDebug();
PrintDebug("Positions.Count = " + Positions.Count);

// assign a variable 'bar' to the 10th to last bar in the chart
int bar = Bars.Count - 10;
BuyAtMarket(bar + 1);
PrintDebug("Positions.Count = " + Positions.Count);
PrintDebug("LastPosition.Active = " + LastPosition.Active);

// increment the bar and exit the Position AtClose
bar++;
SellAtClose(bar + 1, LastPosition);
PrintDebug("Positions.Count = " + Positions.Count);
PrintDebug("LastPosition.Active = " + LastPosition.Active);
}

```

In the example, note that `Positions.Count` is incremented *immediately* upon the creation of a `Position`, which initially is "active". After selling the `Position`, it remains in the list, but its active status changes to *false*.

- ➔ The example subtlety illustrates that Wealth-Lab is *Position-based*. Notice that the details of `Position` size (shares) are not specified in the Strategy code. Instead, the size is determined by the Position Size control in the Data Panel (User Guide).

How to: Create a Trade on a Specified Date

Trades can be created only on bars that exist in the chart. Fortunately the `DateTimeToBar` function makes this a simple task by converting a `DateTime` value to a bar number, that can be passed to any trading signal.

Example ([How to run Example code?](#) )

```

C#
protected override void Execute()
{
    // Create a trade on or as close to 12/18/2007 as possible
    DateTime dt = new DateTime(2007, 12, 18);
    int bar = DateTimeToBar(dt, false);
    if (bar > -1)
        BuyAtMarket(bar, "Trade on " + dt.ToString());
    else
        DrawLabel(PricePane, "No bar on or after " + dt.ToString(), Color.Red);
}

```

7.2 Trading Loop

When screening, we're interested only in identifying symbols that *currently meet* a specified criteria. That's why we're [generally] only interested in values at the last bar, or **Bars.Count - 1**, for screens. Conversely, when backtesting, we want to test criteria on *every bar* after all our indicators become valid.

Every trading Strategy should have a main loop that cycles through the data in your chart. This is accomplished with a typical for statement and allows you to "look" the data one bar at a time, just as you would in real trading. We often refer to the main loop as the *trading loop*.

Example (illustrative)

```
protected override void Execute()
{
    for(int bar = 50; bar < Bars.Count; bar++)
    {
        // Trading rules here
    }
}
```

Here, the for loop starts at the 50th bar of data, but you should set your main loop's starting point based on the technical indicators that you're using. For example, if you use a 50-bar **SMA** and a 14-bar **RSI** in your rules, choose the longer of the two indicators and set your starting for loop value to 50. The main loop should end by processing the last bar at **Bars.Count - 1**. For more information about when to start the main loop, see [Stability of Indicators](#)⁴³.

How to: Convert Strategy Code into a Screen

Quick answer: Remove the main looping statement and assign the variable *bar* to the last chart bar.

You can actually use the Strategy Builder to create a simple screen. First, select the **Convert to Code-Based Strategy** action in the Rules view. By default, the Strategy Builder includes a for loop as shown above. To turn it into a Screen, you need only to replace the "for" looping statement by assigning the variable *bar* to the last chart bar as shown:

Example (illustrative)

```
/* Strategy with for loop */
protected override void Execute()
{
    for(int bar = 50; bar < Bars.Count; bar++) /* CHANGE THIS */
    {
        // Trading rules here
    }
}

/* Screen: assign bar to the final chart bar */
protected override void Execute()
{
    int bar = Bars.Count - 1; /* TO THIS */
    {
        // Trading rules here
    }
}
```


7.3 Single-Position Strategies

Strategies that trade one Position at a time are *Single-Position (SP) Strategies*. Generally, this means that the Strategy can either hold one active Position (long or short) or no active Positions for any symbol.

➔ When you run a SP Strategy in a Multi-Symbol Backtest (see User Guide), the "total simulation" can hold multiple Positions simultaneously but only one active Position per symbol, at most.

SP Strategy Code Template

The code template installed with Wealth-Lab is a template for SP Strategies. It consists of the main loop and logic that makes entries and exits mutually-exclusive on the same bar. In other words, by following the template you'll automatically write code that only opens Positions *or* closes Positions on a given bar.

Example ([How to run Example code?](#)³⁷)

```
C#
// SP Strategy Template
protected override void Execute()
{
    for(int bar = 20; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            //code your exit rules here
        }
        else
        {
            //code your entry rules here
        }
    }
}
```

Positions

A key concept for programming Strategies and processing results is that of the **Positions** list. As a script creates trades, they are added to the **Positions** list. You can access **Position** objects from the list at any time during a Strategy's execution, and it's often required to do so to determine which **Position** to pass as the *pos* parameter in an exit signal. The **LastPosition** property conveniently accesses the most-recent **Position** added to **Positions**, and therefore is most useful in SP Strategies. Refer to the Position Management functions in the QuickRef ([F77](#)) for more details.

How to: Test for an Open Position

Since we work with only one open (active) Position in SP Strategies, we can use the *LastPosition logic* shortcuts provided by WealthScript. If a **Position** is active, we can be sure that it will be the **LastPosition** in the **Positions** list. In the code template above, if the last Position is active, the **IsLastPositionActive** function will return *true* and we branch to our exit rules. Otherwise, we test if our entry rules present an opportunity to initiate a new **Position**. Here's a complete SP Strategy to demonstrate the concepts.

Example ([How to run Example code?](#)³⁷)

```
C#
protected override void Execute()
```

```

{
    const int Period = 14;
    PlotSeries(PricePane, SMA.Series(Close, Period), Color.Blue, LineStyle.Solid,
1);

    for(int bar = Period; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            if( CrossUnder(bar, Close, SMA.Series(Close, Period)) )
                SellAtMarket(bar + 1, LastPosition);
        }
        else
        {
            if( CrossOver(bar, Close, SMA.Series(Close, Period)) )
                BuyAtLimit(bar + 1, Close[bar]);
        }
    }
}

```

Closing out a Position

You can see from the example above that the **SellAtMarket** function is one way to close out an open long Position. The function takes two parameters. The first parameter is the bar in which to close out the **Position**, and the second is a reference to the **Position** that we want to sell. Since WealthScript can support trading systems that [manage multiple positions](#)⁶⁵ we need a way to tell the exit rule which **Position** we want to sell. For systems that manage a single Position at a time we can use the **LastPosition** for this purpose.

See also: [Multiple-Position Strategies](#)⁶⁵

7.3.1 AtLimit, AtStop Orders

Think about the process that you use to place a stop or limit order when trading live. After updating your chart data, you'll determine a price at which you're willing to enter a Position, and subsequently you'll place the order in the market on the *next bar*.

The process is precisely the same when programming Strategies in Wealth-Lab: access data as of the current *bar*, calculate stop or limit prices, and place the order to execute on the next bar, *bar + 1*.

How to: Trigger a Limit or Stop Buy Order

Use the **BuyAtStop** and **BuyAtLimit** to simulate stop and limit buy orders (likewise for Sell, Short, and Cover). As with live trading, since there's a chance that these orders might not be filled, BuyAt and ShortAt functions return either a **Position** or **null**, indicating whether or not the trades were executed; SellAt and CoverAt function return boolean *true* or *false* for the same purpose.

Going back to the previous example for [Testing for an Open Position](#)⁵¹, the entry logic uses a **BuyAtLimit** order, which is placed on the next bar (*bar + 1*), but only if the **Close** crosses over its moving average. The Strategy could have just as easily used a **BuyAtMarket** order to guarantee taking on a Position after a cross over, but the limit order specifies a maximum price that you're willing to pay. That price happens to be the

closing price of the *current bar*.

Example (illustrative)

```
else
{
    if( CrossOver(bar, Close, SMA.Series(Close, Period)) )
        BuyAtLimit(bar + 1, Close[bar]);
}
```

How to: Create Good-Til-Canceled Orders

In reality, GTC orders are a natural consequence of programming stop or limit orders - Wealth-Lab places, replaces, or continues to hold open orders on the *bar* that you specify. Consequently, GTC orders in Strategies are stop or limit orders that are repeated until: 1) the order is filled, or, 2) the order is removed logically; in other words, GTC orders are canceled by not calling the [trading signal](#)^[48].

The next example is a simplified version of the *Channel Breakout VT* Strategy. Instead of plotting the breakout series, using the `PlotStops()` method we can visually identify the bars and prices where orders are placed, replaced, or canceled.

Example ([How to run Example code?](#)^[3])

■ C#

```
protected override void Execute()
{
    int longPer = 55;
    int shortPer = 21;
    DataSeries H55 = Highest.Series(High, longPer);
    DataSeries L55 = Lowest.Series(Low, longPer);
    DataSeries H21 = Highest.Series(High, shortPer);
    DataSeries L21 = Lowest.Series(Low, shortPer);
    PlotStops();

    for (int bar = longPer; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            Position Pos = LastPosition;
            if (Pos.PositionType == PositionType.Long)
                SellAtStop(bar + 1, LastPosition, L21[bar]);
            else
                CoverAtStop(bar + 1, LastPosition, H21[bar]);
        }
        else
        {
            RiskStopLevel = L21[bar];
            if (BuyAtStop(bar + 1, H55[bar]) == null)
            {
                RiskStopLevel = H21[bar];
                ShortAtStop(bar + 1, L55[bar]);
            }
        }
    }
}
```

How to: Plot Stops

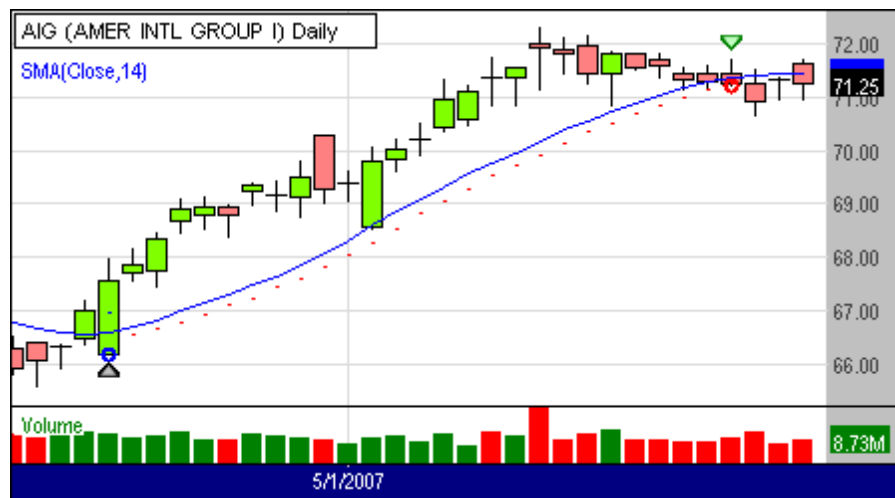
While the previous example contains a good demonstration of plotting stops using the `PlotStops()` method, let's take another look at it using a **SMA** (simple moving average) as

the stop price. When you run the code, you'll notice that the stops are not plotted exactly on the **SMA** line. That's because **PlotStops()** shows you the bar and price at which a stop (or limit) order is active. The **SMA** is calculated and plotted on the current *bar*, but the resulting value is used as the stop price for the *next bar*, i.e., *bar + 1*.

Example ([How to run Example code?](#)³)

```
C#
protected override void Execute()
{
    const int Period = 14;
    PlotSeries(PricePane, SMA.Series(Close, Period), Color.Blue, LineStyle.Solid,
1);
    PlotStops();

    for(int bar = Period; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            SellAtStop(bar + 1, LastPosition, SMA.Series(Close, Period)[bar]);
        }
        else
        {
            if( CrossOver(bar, Close, SMA.Series(Close, Period)) )
                BuyAtLimit(bar + 1, Close[bar]);
        }
    }
}
```



The small colored dots, automatically plotted by one call to **PlotStops()**, represent the bar and price on which a stop is active.

7.3.2 Time-Based Exits

How to: Perform a Time-Based Exit

Generally speaking, a time-based exit strategy is a market order that is triggered after a specified number of bars. In a SP Strategy, the general pattern for a time-based exit is as follows:

```
int numberOfBars = 5; // specify max number of Bars to hold Position
```

```

Position p = LastPosition;
if( bar + 1 - p.EntryBar >= numberOfBars )
    ExitAtMarket(bar + 1, p, "Time-based");

```

Here's a look at a time-based exit in a Strategy that buys at a limit price 3% below the most-recent close and sells after 5 bars.

Example ([How to run Example code?](#)³⁴)

```

C#
protected override void Execute()
{
    int numberOfBars = 5;
    for(int bar = 1; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            Position p = LastPosition;
            if( bar + 1 - p.EntryBar >= numberOfBars )
                ExitAtMarket(bar + 1, p);
        }
        else
            BuyAtLimit(bar + 1, Close[bar]* 0.97);
    }
}

```

How to: NOT Perform a Time-Based Exit

You may be tempted to try to simplify the code above by immediately specifying the future bar on which to sell a **Position**. However, when you use `Bar + n`, where $n > 1$ in trading signals, Wealth-Lab executes those trades immediately during backtests. Indeed, those trades will have taken place in the future, but the active status of those Positions are changed too early. Most often that causes the code's logic to start processing entry or exit logic too soon (because it's usually based on Position status) and therefore in an unrealistic way.

Example ([How to run Example code?](#)³⁴)

```

C#
protected override void Execute()
{
    int numberOfBars = 5;
    for(int bar = 1; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
            // THE WRONG WAY TO CODE A TIME-BASED EXIT (OR ANY TRADING SIGNAL)
            ExitAtMarket(bar + numberOfBars, LastPosition);
        else
            BuyAtLimit(bar + 1, Close[bar]* 0.97);
    }
}

```

Specifying a trading signal to occur on *bar + n*, where $n > 1$ is never correct for backtesting or trading because:

1. Wealth-Lab Pro executes trading signals immediately in backtests. Exiting a **Position** on *bar + n*, where $n > 1$ causes a Position's status to change to 'not active' prematurely, which has the effect of disrupting a Strategy's intended logic (see [Peeking](#)⁵⁷).
2. It's impossible to trade a Strategy that generates Alerts on *bar + n*, where $n > 1$.

Depending on the Strategy, the result will be either getting an Alert $n - 1$ days too early, or, getting an Alert for each of n days to exit the same **Position**.

7.4 Peeking

Being a programming environment, Wealth-Lab doesn't artificially limit what you can do in Strategy code, such as peeking - intentionally or unintentionally. (In *Trade Your Way to Financial Freedom*, Van Tharp called peeking "postdictive" errors.) While some [valid uses for peeking](#)⁵⁹ exist, generally it's something that you want to avoid. Unintentional peeking is usually not a concern with logically-designed code, but reviewing the following subtopics should help you avoid some common pitfalls.

Don't Access Future Data

Be sure that your Strategy doesn't take advantage of information that it would have no way of accessing in the real world. It's easy to avoid this type of error by always accessing the value of a **DataSeries** or indicator in the main loop at the current (*or previous*) *bar* and not attempting to access data at *bar + 1* or any other future bar.

More often, subtle mistakes are made *prior to* the main loop by extracting some statistic or indicator based on the most-recent value(s) and then using that information later in the [main loop](#)⁵⁰. In the example, we obtain the standard deviation of the entire **Close** **DataSeries**, but then [incorrectly] use its value in the trading Strategy.

Example (Illustrating what not to do)

```
protected override void Execute()
{
    double sd = StdDev.Value(Bars.Count - 1, Close, Bars.Count,
        StdDevCalculation.Sample);
    for(int bar = 20; bar < Bars.Count; bar++)
    {
        // Trading logic here CANNOT use sd
    }
}
```

Execute Trading Signals on bar + 1

While you should not access data on a future bar, Strategies should process the current *bar* and create trading signals for the *next bar*, i.e., *bar + 1*. All Strategies installed with Wealth-Lab Pro are good examples to review. (See also: [Time-Based Exits](#))⁵⁴

Exception: AtClose Signals

Although it's difficult (perhaps impossible for most) to access a bar's closing data *and* then execute a trade at the closing price using an **AtClose** signal, it's an accepted backtesting technique for end-of-day Strategies. (See [How to Alert for AtClose Signals](#))⁶⁰ In general, intraday Strategies *should not* use **AtClose** signals, except perhaps for exiting Positions at the session close.

Order of Trading Signals

Strategies that using multiple order types for entries or exits should test those signals in the proper, realistic order, which is:

1. **AtClose** signals executed on the current *bar*,
followed by signals executed on *bar + 1*;
2. **AtMarket** signals, and then,

3. **AtStop** signals (stop losses), and finally,
4. **AtLimit** signals (profit targets)

Stop losses should be tested before profit targets in order to give the most pessimistic result in case both signals would have triggered in the same bar since it's usually not possible to precisely determine which signal would have triggered first.

Example (Illustrative)

■ C#

```
protected override void Execute()
{
    for(int bar = 50; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            Position p = LastPosition;
            if( CumUp.Series(Close, 5)[bar] >= 1 )
                ExitAtClose(bar, p);
            else if( p.MAEAsOfBarPercent(bar) < -8)
                ExitAtMarket(bar + 1, p);
            else if( !ExitAtTrailingStop(bar + 1, p, SMA.Series(Close, 50)[bar]) )
                ExitAtLimit(bar + 1, p, p.EntryPrice * 1.25);
        }
        else
        {
            //code your entry rules here
        }
    }
}
```

Trading based on Position Active Status

Probably the most subtle peeking error that you can make when programming a trading strategy is to base logic that executes on the *current bar* on the results of a trade that may or may not occur on the *next bar*. The general rule is to simply not test a Position's active status *after* executing a trading signal. For example, in the [SP Strategy Code Template](#)^[5], all the trading logic is contained with the the **if** statement that determines if the last Position is active *before* executing trading signals.

Consider the following "problematic Strategy" that enters a Position using a limit order on the next bar, but then checks to exit that Position **AtMarket** (on the open) of that same bar.

Example (Illustrating what not to do)

```
protected override void Execute()
{
    for(int bar = 5; bar < Bars.Count; bar++)
    {
        if( !IsLastPositionActive )
            BuyAtLimit(bar + 1, Close[bar] * 0.97);

        /* DO NOT CHECK A POSITION'S ACTIVE STATUS AFTER A TRADING SIGNAL */
        if (IsLastPositionActive)
            ExitAtMarket(bar+1, LastPosition);
    }
}
```


Survivorship Bias

Survivorship bias refers to backtesting with data only from companies that survived severe market downturns. *Excluding* data from companies that no longer exist or trade publicly can make Strategies appear to perform better (or in some cases, worse) in a simulation than they would have had they been actually traded. Survivorship bias may or may not have a significant effect on the performance of your Strategy, but it's good to be aware that it exists.

Noting that it can be difficult to obtain price data from companies that are no longer in business, it's definitely a worthwhile exercise to test with such data if you can find it. Conversely, it's worth mentioning that a good number of companies are purchased by others at large premiums, which create price shocks in a positive direction.

7.4.1 Valid Peeking

It may be surprising to you to find code that appears peek does not actually qualify as peeking. Some common cases follow, but more may apply.

Valid Peeking #1

Checking if a known event (options expiry, last trading day of the month, split, etc.) will to occur on a "future bar".

In the same way that you can look at a calendar and determine if tomorrow is a new month, the example's *NextBarIsNewMonth* function checks the date of the *next bar*. If it's a new month, a new Position is created. By running the script on you'll see that indeed Positions are entered on the first trading day of each month. Note, however, that the script can check the next bar's date only if that next bar actually exists in the chart. For that reason (and simplicity) when processing the final bar of the chart, the function *assumes* that the next bar is a new month. Consequently, when a Position is not active, the Strategy will always generate an Alert to enter a new trade, and you must look at a calendar to find out if you should actually place the order.

Example ([How to run Example code?](#) ³⁷)

```
C#
public bool NextBarIsNewMonth(int bar)
{
    if (bar < Bars.Count - 1)
        return Date[bar + 1].Day < Date[bar].Day;
    else
        /* Assume that next bar is a new month.
         * Look at a calendar and just ignore any entry Alerts if it is not! */
        return true;
}

protected override void Execute()
{
    for(int bar = 1; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            // exit after 3 trading days
            Position p = LastPosition;
            if (bar + 1 - p.EntryBar >= 3)
                SellAtMarket(bar + 1, p, "Time-based");
        }
    }
}
```

```

    }
    else if (NextBarIsNewMonth(bar)) // enter on the first trading day of a new
month
        BuyAtMarket(bar + 1);
    }
}

```

Valid Peeking #2

Determine if an AtLimit/AtStop order occurred "on the open" so as to give the Position a higher priority.

Generally speaking, **Position.Priority** is difficult to apply when backtesting end-of-day stop and limit Strategies. That's because the actual trade sequence should be prioritized by the *time of day* at which the actual price attains the order price. Nonetheless, it's easy to determine if a stop or limit order would have occurred *at market* on the opening of a session. In this case, you can assign a greater-than-zero priority so that Wealth-Lab processes those trades first.

[Example \(How to run Example code?\)](#)

```

C#
protected override void Execute()
{
    for(int bar = 1; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            // exit logic
        }
        else
        {
            double limitPrice = Close[bar] * 0.98; // 2% drop
            if ( BuyAtLimit(bar + 1, limitPrice, "") != null )
                if ( bar < Bars.Count - 1 && Open[bar+1] <= limitPrice )
                    LastActivePosition.Priority = 1.0;
        }
    }
}

```

Valid Peeking #3

Determine when an exit condition occurs to Split and exit a portion of a Position

When trading in real life, you don't have to think in terms of "splitting Positions" - to exit half of a 1000-share Position, you'd simply enter an order for 500 shares. When backtesting in Wealth-Lab, the same operation required that you split the initial Position into two individual Positions and then exit when required. But what if you want to split the Position to lock in profits on the way up, but exit the full Position at a stop loss?

If you always split the Position and sell both halves for a loss, you incur the penalty of an additional commission charge. Instead, you can wait for the exit condition to occur (just as in real trading) and decide at that moment if a split is required. In the example, we look ahead to the High of the next bar to lock in a 5% profit on half of the initial Position.

[Example \(How to run Example code?\)](#)

```

C#
protected override void Execute()
{
    bool hasSplit = false;

```

```
for(int bar = 1; bar < Bars.Count; bar++)
{
    if (IsLastPositionActive)
    {
        Position p = LastPosition;
        // test stop loss first
        if (!SellAtStop(bar + 1, p, p.EntryPrice * 0.975, "Stop Loss"))
            if (!hasSplit)
            {
                // exit half of the Position at a 5% profit target
                double target = p.EntryPrice * 1.05;
                if( bar < Bars.Count - 1 && target <= High[bar+1] )
                {
                    Position s = SplitPosition( p, 49.99 );
                    // s is now the LastPosition, so sell off p to continue to use
                    LastPosition logic
                    hasSplit = SellAtLimit(bar + 1, p, target, "5% Profit");
                }
            }
        else
            // Sell the rest at 20% profit target
            SellAtLimit(bar + 1, p, p.EntryPrice * 1.2, "20% Profit");
    }
    else
    {
        BuyAtLimit(bar + 1, Close[bar] * 0.97);
        hasSplit = false;
    }
}
```


7.5 Strategy Parameters

Strategy Parameters (*Straps* for short) provide the ability to perform quick, on-demand optimizations to Strategies by adjusting visual sliders at the bottom of the Data Panel. See the User Guide for more details about their application.

How to: Add Strategy Parameters

Adding Straps to Strategy code is a simple task - just follow the design patterns outlined here.

Method 1 (automatic)

From a Chart or Strategy window, drag and drop one or more indicators. To automatically generate the plot code and sliders, simply click the  button in the **Function Toolbar**.

See Pushing Dropped Indicators in the User Guide for details.

Method 2 (semi-automatic)

From any Strategy window, click *Optimize* link in the lower left status bar. Use the Optimizer's interface to *Add New Parameters*, adjust their default, start, end, and step values, and finally *Apply Changes to Code*. In the code, access the value of *strategyParameterN* using the **Value** or **ValueInt** property for floating point or integer parameters, respectively.

See Optimization in the User Guide for details.

Method 3 (manual)

Assume that you're working with a simple moving average (**SMA**) crossover Strategy and want an easy way to change the **SMA** periods. Below we start with the basic (Strapless) crossover Strategy.

```
namespace WealthLab.Strategies
{
    public class MyStrategy : WealthScript
    {
        protected override void Execute()
        {
            int fastPer = 20;
            int slowPer = 55;

            SMA smaFast = SMA.Series(Close, fastPer);
            SMA smaSlow = SMA.Series(Close, slowPer);
            PlotSeries(PricePane, smaFast, Color.Green,
                LineStyle.Solid, 2);
            PlotSeries(PricePane, smaSlow, Color.Red,
                LineStyle.Solid, 2);

            for (int bar = Math.Max(fastPer, slowPer); bar <
                Bars.Count; bar++)
            {
                if (IsLastPositionActive)
                {
                    if (CrossUnder(bar, smaFast, smaSlow))
                        SellAtMarket(bar + 1, LastPosition);
                }
            }
        }
    }
}
```

```

else
{
    if (CrossOver(bar, smaFast, smaSlow))
        BuyAtMarket(bar + 1);
}
}
}
}
}

```

Step 1

Focusing on the MyStrategy class, change its name to something more meaningful, like *SMACrossover*, and declare any number of Straps as **StrategyParameter** types as private class variables. Since we want to adjust the fast and slow periods, we identify a variable for each one.

```

namespace WealthLab.Strategies
{
    public class SMACrossover : WealthScript
    {
        /* Declare parameters */
        private StrategyParameter slowPeriod;
        private StrategyParameter fastPeriod;

        protected override void Execute()
        {
            int fastPer = 20;
            int slowPer = 55;

```

Step 2

The next step is to add a public class *constructor*. A constructor's method name is the same as its class: *SMACrossover()*.

Within the constructor assign the **CreateParameter()** method for each one of the variables that you declared in Step 1. Refer to the syntax for **CreateParameter()**:

```

namespace WealthLab.Strategies
{
    public class SMACrossover : WealthScript
    {
        private StrategyParameter slow;
        private StrategyParameter fast;

        /* Add a public constructor (same name as the class) */
        public SMACrossover()
        {
            fast = CreateParameter("Fast Per",
20, 4, 100, 1);
            slow = CreateParameter("Slow Per",
55, 5, 300, 5);
        }

        protected override void Execute()
        {
            int fastPer = 20;
            int slowPer = 55;

```

CreateParameter(string name, double value, double start, double stop, double step);

The specified **name** appears next to the slider in the Data Panel to identify the parameter, **value** is the initial default value for the Strategy Parameter, and **step** controls the increments between the **start** and **stop** minimum and maximum bounds of the parameter.

Step 3

Finally, convert the **StrategyParameter** types to their corresponding integer or double parameter inside the execute method using the **ValueInt** or **Value** properties, respectively.

```

namespace WealthLabCompile
{
    class MovingAverageCrossover :
WealthScript
    {
        private StrategyParameter slow;
        private StrategyParameter fast;

        public MovingAverageCrossover()
        {

```

For the final result of this process,

please see the pre-built "Moving Average Crossover" Strategy.

```
        fast = CreateParameter("Fast Per",
                                20, 4, 100, 1);
        slow = CreateParameter("Slow Per",
                                55, 5, 300, 5);
    }

    protected override void Execute()
    {
        /* Obtain periods from parameters */
        int fastPer = fast.ValueInt;
        int slowPer = slow.ValueInt;
    }
}
```

7.6 Multi-Position Strategies

All examples up to this point have used a design pattern that ensure that Strategies can enter and hold only one Position at a time. *Multiple-Position (MP) Strategies* are design to manage one or more Positions simultaneously. For example, a Strategy that pyramids Positions (adds more shares) of the same stock has to manage each new Position separately.

➔ While you can use `SplitPosition()` to split a `Position` into two parts, it's not currently possible to "merge" multiple Positions into one.

Some Guidelines for Managing Multiple Positions

WealthScript contains several properties to help you work with information about [Strategy Positions](#)^[51]. `Positions.Count` returns the total number of `Positions` that have been created, whereas `ActivePositions.Count` returns the total of currently-active Positions.

Each `Position` added to `Positions` has its own set of properties (e.g., `.Active`, `.BarsHeld`, `.EntryPrice`, `.EntryBar`, `.ExitBar`, etc.) containing information about a specific Position. For details, refer to the *QuickRef: Position Object*.

When working with multiple Positions, you typically have a secondary loop within your main loop that processes each active `Position` to determine if it should be closed out. See [How to Use ActivePositions](#)^[66]. However, if the Strategy logic is such that all active Positions are always closed simultaneously, use the [Position.AllPositions shortcut](#)^[66] instead.

Programming notes:

- Place exit logic above entry trading rules to prevent the exit logic from being applied to Positions that are opened on the very same bar.
- It's usually a mistake to use `LastPosition` logic (especially `IsLastPositionActive`) in MP Strategies. You can unwittingly close the last Position, leaving Positions that were opened earlier unprocessed.

MP Strategy Template

Multiple Positions are typically opened on different bars for the same symbol, so the entry and exit logic cannot be mutually-exclusive as with [SP Strategies](#)^[51]. For this reason you must be extra-careful to ensure that you don't write code that [peeks](#)^[57] by testing Position-active status following the exit logic. In other words, make sure that entry logic is independent of exits that occur on the *next bar*.

Example ([How to run Example code?](#)^[3])

```
C#
protected override void Execute()
{
    // main loop
    for(int bar = 20; bar < Bars.Count; bar++)
    {
        //exit logic - test each active position
        for(int pos = ActivePositions.Count - 1; pos >= 0; pos--)
        {
            // assign the Position using the list index
        }
    }
}
```

```

        Position p = ActivePositions[pos];

        // Example: Process the currently-assigned Position, p
        if (bar - p.EntryBar > 10)
            ExitAtMarket(bar + 1, p, "Time-Based");
    }

    /* entry rules here */
}

```

- ➔ Don't use the *foreach* statement with **ActivePositions** in the exit logic. It's an error in C# to remove items from a collection like **ActivePositions**, which occurs by closing a **Position**. The proper method is to work backwards in the collection, as shown above.

How to: Use ActivePositions

Wealth-Lab Pro Version 6 provides the ability to directly access a list of only active **Positions**, appropriately named **ActivePositions**. See the [MP Strategy Template](#)^[65] for example use.

How to: Use Position.AllPositions

If your MP Strategy uses the same exit trigger for all active Positions, then pass **Position.AllPositions** as the Position parameter in the exit signal. The simplification allows you to forego looping through **ActivePositions** to exit each **Position** separately.

Example ([How to run Example code?](#)^[3])

```

C#
/* This Trading System buys whenever RSI crosses above 30, and closes all open
positions when it crosses below 70. */
protected override void Execute()
{
    RSI rsi = RSI.Series(Close, 14);

    for(int bar = 42; bar < Bars.Count; bar++)
    {
        if( ActivePositions.Count > 0 )
            if( CrossUnder(bar, rsi, 70) )
                SellAtMarket(bar + 1, Position.AllPositions);

        if( CrossOver(bar, rsi, 30 ) )
            BuyAtMarket(bar + 1);
    }
}

```

See also: [Single-Position Strategies](#)^[5]

7.7 Multi-Symbol Strategies

With Wealth-Lab's Multi-Symbol Backtest feature, it's generally not required to explicitly create trades on specific symbols within a script. In other words, by default a script trades the primary symbol - the symbol under test.

However, when a Strategy involves trading rules that depend on price action of other symbols and vice-versa, then you must take full control of the simulation by explicitly creating trades on secondary symbols after calling `SetContext()`. Examples of these more-complex Strategies are *Pairs Trading* and *Symbol Rotation* Strategies. For examples, see the pre-built "RSI Rotation" or "Dogs of the Dow" scripts.

7.8 Alerts

Trading signals are used in two ways:

1. To create theoretical trades for backtests, and,
2. To generate *Alerts* for orders that should be placed on the bar following the last bar of the chart.

Alerts are what allow you to trade your strategy "live". They're the orders that you place on the next bar (intraday interval, day, week, etc.)

For example, when trading daily bars you'll run the Strategy each evening after updating data following the market's close. If the script generates an Alert, you would place a corresponding order the next day. On the other hand, when trading intraday, you must place the order immediately after receiving an Alert.

When coding a trading system in WealthScript the only requirement to generate an Alert is to pass $bar + 1$ to one of the [trading signals](#)^[48] such that the signal occurs on a bar beyond the last bar in the chart. During the final iteration of the trading loop, when bar is assigned the value `Bars.Count - 1`, a trading signal such as `BuyAtMarket(bar + 1);` will create an alert since the bar number passed to the signal is greater than `Bars.Count - 1`.

Warning!

Never pass $bar + n$, where $n > 1$ to a trading signal. *Never*. *Always* pass $bar + 1$ to trading signals, with the single exception being that trades can be created on the current bar for AtClose orders (See [Alerts for AtClose Orders](#)^[69]).

We've already made use of Alerts for Creating a Screener. For more examples, review some of the pre-existing Strategies that come with the Wealth-Lab installation.

- ➔ While you should use $bar + 1$ in trading signals to create a trade condition for the next bar, you should not attempt to access data on the next bar. That would be an obvious [peeking](#)^[57] error and would result in a script runtime when processing the chart's last bar.

How to: Save Strategy Alerts to a File

Although the majority of Strategies will at most generate one Alert per symbol due to the use of *single-Position logic*, it's useful to know that Wealth-Lab maintains an **Alerts** list that you can use to iterate through any and all Alerts generated by a script. The example's *SaveAlerts* method can be reused to save Alerts to a text file at a specified path.

Example ([How to run Example code?](#)^[3])

```
C#
/* Must be used in Raw Profit mode for proper Alert Shares (size) */
public void SaveAlerts(string path)
{
    string dateFormat = "yyyyMMdd";
    if (Bars.IsIntraday) dateFormat = "yyyyMMdd HHmm";
    if ( Alerts.Count > 0 )
```

```

    {
        for( int i = 0; i < Alerts.Count; i++ )
        {
            Alert a = Alerts[i];
            string s = a.AlertDate.ToString(dateFormat);
            s = s + ";" + a.AlertType.ToString();
            s = s + ";" + a.Shares.ToString(); // always 1 in Portfolio Simulation

Mode
            s = s + ";" + a.Symbol;
            s = s + ";" + a.OrderType.ToString();
            if (a.OrderType == OrderType.Limit || a.OrderType == OrderType.Stop)
                s = s + ";" + a.Price.ToString();
            else
                s = s + ";" + a.BasisPrice.ToString();
            s = s + ";" + a.SignalName + "\n";

            System.IO.TextWriter tw = new System.IO.StreamWriter(path, true);
            tw.Write(s);
            tw.Close();
        }
    }
}

protected override void Execute()
{
    // Generate 5 Alerts, each 1% lower than the previous
    int bar = Bars.Count - 1;
    int pct = 99;
    for(int numAlerts = 1; numAlerts <= 5; numAlerts++)
        BuyAtLimit(bar + 1, Close[bar] * (pct - numAlerts)/100,
numAlerts.ToString());

    // Call the SaveAlerts method after the main trading loop
    SaveAlerts(@"C:\Alerts.txt");
}

```

How to: Alert for AtClose Signals

Strategies that use AtClose orders generally intend to open or close a **Position** on the close of the *current bar*, i.e., the one just processed. Consequently, you'll generally pass *bar* instead of *bar + 1* to an AtClose signal, and therefore it's not possible for Wealth-Lab to Alert you for a trade. Theoretically, Wealth-Lab has no trouble executing such a trade, however, in practice it's actually quite impossible to process data from a bar that has just completed *and then* execute a trade on the close of that same bar.

However, if you were able to access partial bar data (Wealth-Lab User Guide: Yahoo! Data Provider), it would be possible to update a Daily bar just prior to the market close to get an estimate of the final closing price with which to execute a trade. The trick, however, is to use an AtMarket order in place of the AtClose order when processing the last bar as demonstrated below.

Example ([How to run Example code?](#)³⁷)

■ C#

```

protected override void Execute()
{
    SMA smaFast = new SMA(Close, 8, "Fast");
    SMA smaSlow = new SMA(Close, 16, "Slow");
    PlotSeries(PricingPane, smaFast, Color.Green, LineStyle.Solid, 1);
    PlotSeries(PricingPane, smaSlow, Color.Red, LineStyle.Solid, 1);
}

```

```

for(int bar = 16; bar < Bars.Count; bar++)
{
    if (IsLastPositionActive)
    {
        Position p = LastPosition;
        if (CrossUnder(bar, smaFast, smaSlow))
        {
            /* Exit condition is true. If last bar is current bar, then Alert AtMarket
            */
            if (bar == Bars.Count - 1)
                SellAtMarket(bar+1, p, "Alert");
            else
                SellAtClose(bar, p, "AtClose");
        }
    }
    else
    {
        if (CrossOver(bar, smaFast, smaSlow))
            BuyAtMarket(bar + 1);
    }
}

```



On intraday scales, the only difference between AtClose and AtMarket orders is 1 trade. Since intraday AtClose signals cannot be realized by any practical means, they should rarely - if ever - be used in intraday Strategies; AtMarket signals are preferred. An exception would be to close intraday Positions at the end of the trading day in the same manner as described above.

7.9 Options Strategies

By constructing synthetic option contracts based on the underlying stock price and the Black-Scholes model Wealth-Lab Pro has the capability to simulate and backtest options trading strategies and applying option contract leverage.

WealthScript Methods to Support Option Strategies

The following WealthScript methods support the capability: `CreateSyntheticOption()` (two overloads), `NextExpiryDate()`, `IsOptionsExpiryDate()`, and a single-parameter overload for `SetContext()`. The syntax of each method is shown below, but for more details see the Options category of functions in the QuickRef (F11).

```
public Bars CreateSyntheticOption(DateTime startDate, DateTime expiryDate, double strikePrice,
public Bars CreateSyntheticOption(int asOfBar, int atLeastXDaysTilExpiration, int daysToPlotBe
```

➔ The resulting bars object for `CreateSyntheticOption()` generates a symbol with the format:

/symbol_strike_YYMMDD_optionType

where,

symbol underlying symbol of the current Bars context,

strike option's strike price,

YYMMDD date of the monthly expiry, and,

optionType CALL or PUT

```
public bool IsOptionExpiryDate(DateTime dt)

public DateTime NextOptionExpiryDate(int bar)

public void SetContext(WealthScript.Bars bars)
```

See also: [OptionsExpiryDate](#)¹¹⁹

Sample Option Strategies

The strategy below demonstrates the simplicity to generate a hypothetical option contract based on the underlying stock price at the specified bar. Note the use of the second overload method of `CreateSyntheticOption()`. The strike price of the synthetic option contract is the integer part of the closing price of the underlying stock on the specified *asofBar*.

[Example \(How to run Example code?\)](#)

■ C#

```
protected override void Execute()
{
    ChartPane pane = CreatePane(75, true, true);
    HideVolume();

    //contract 1
    int b = Bars.Count - 120;
    Bars oc = CreateSyntheticOption(b, 30, 30, true);
    PrintDebug(oc.Count);
    Color c = Color.Black;
    PlotSymbol(pane, oc, c, c);

    //contract 2
    b = Bars.Count - 80;
    Bars oc2 = CreateSyntheticOption(b, 30, 30, true);
    c = Color.Purple;
    PlotSymbol(pane, oc2, c, c);

    //contract 3
    b = Bars.Count - 40;
    Bars oc3 = CreateSyntheticOption(b, 30, 30, true);
    c = Color.Blue;
    PlotSymbol(pane, oc3, c, c);
}
```

The following code uses a “4x2” approach. When the underlying is down 4 days in a row, the strategy buys a synthetic call with an expiry date of at least 30 days out. When the underlying is up 2 days in a row, the call is sold.

[Example \(How to run Example code?\)](#)

■ C#

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Drawing;
using WealthLab;
using WealthLab.Indicators;

namespace WealthLab.Strategies
{
    public class MyStrategy : WealthScript
    {
        private Color[] PlotColors = { Color.FromArgb(255, 128, 128),
            Color.FromArgb(128, 255, 128),
            Color.FromArgb(128, 128, 255) };
        private int PlotIndex;
        private List<string> PlottedSymbols = new List<string>();

        protected override void Execute()
```

```

{
    ChartPane optionsPane = CreatePane(75, true, true);

    SetBarColors(Color.Gray, Color.Gray);
    for(int bar = 30; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            if (CumUp.Value(bar, Close, 1) >= 2)
            {
                //sell the call
                SetBarColor(bar, Color.Red);
                Bars contract = LastPosition.Bars;
                SetContext(contract);
                SellAtMarket(bar + 1, LastPosition, "Sell Call");
                RestoreContext();
            }
        }
        else
        {
            if (CumDown.Value(bar, Close, 1) == 4)
            {
                //buy a call
                SetBarColor(bar, Color.Blue);
                Bars contract = CreateSyntheticOption(bar, 30, 30, true);
                SetContext(contract);
                BuyAtMarket(bar + 1, "Buy Call");
                RestoreContext();

                //plot it if we have not done so already
                if (!PlottedSymbols.Contains(contract.Symbol))
                {
                    Color c = PlotColors[PlotIndex];
                    PlotIndex++;
                    if (PlotIndex >= 3) PlotIndex = 0;
                    PlotSymbol(optionsPane, contract, c, c);
                }
            }
        }
    }
}

```

8 Fundamental Analysis

You may have used company fundamentals to whittle down a list of trading candidates, but how about using *time-series fundamental and economic data* in a trading strategy combined with technical analysis? Fundamental Data Analysis is fully integrated in WealthScript using a set of functions to access fundamental data collections for Fidelity DataSets.

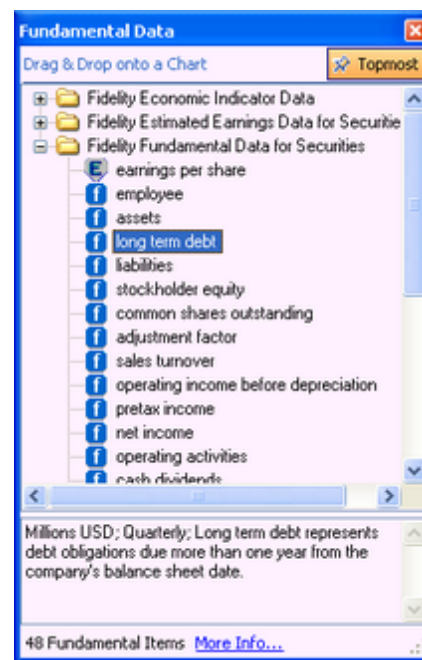
WealthScript Fundamental Data Functions

For a list of functions, details, and examples, refer to the *Fundamental Data* and *FundamentalItem Object* function categories in the Wealth-Lab QuickRef (*F17*).

Fundamental Data and Economic Indicator Definitions Guide

The Fundamental Data and Economic Indicator Definitions Guide is a separate PDF document for updates, changes, and more detail of fundamental *Items* and their descriptions. Along with the Strategy Builder, it contains the information necessary to access these items as well as *fundamental ratios* such as P/E, ROA, and "per Share" data using standard WealthScript operations.

Tip:
Click the [More Info..](#) link at the bottom of the Fundamental Data dialog to launch the Fundamental Data and Economic Indicator Definitions Guide.



Collecting Fundamental Data

Updating and Refreshing Fundamental Data

To include Fundamental data in updates, be sure to check the Fidelity Data Providers for the Fundamental items that you wish to update on the the Data Manager > Update Data tab. Fidelity fundamental data is always refreshed any time that it is retrieved.

➔ Collecting Fidelity fundamental data can result in much longer data update times. For this reason, we recommend to opt out if you do not actually use the data.

To completely refresh all Fidelity fundamental data:

1. Delete or rename the FidelityWSOD*Provider folder(s) in the Data directory, where * represent the type of fundamental data.
2. Restart Wealth-Lab Pro

- ➔ The availability of corporate fundamental data has been observed to lag quarterly reports by several weeks. Since it's possible that an update will include only a partial report (e.g., missing Balance Sheet items), Wealth-Lab refreshes corporate fundamental data for up to 6 weeks after the report date to ensure the entire most-recent report is gathered when it becomes available.

See also: [*Plotting Fundamental Items*](#)³³

8.1 Corporate Fundamental Data

The following fundamental items are available from Fidelity Fundamental Data Provider and can be passed as the parameter for any WealthScript Fundamental Data function that has a *itemName* string parameter. Except where noted, fundamental items from the first four groups below are updated quarterly. See the Fundamental Data and Economic Indicator Definitions Guide (Help menu) for more information.

Balance Sheet items

```
"accounts payable"
"assets"
"current assets"
"current liabilities"
"goodwill"
"stockholder equity"
"total receivables"
"total inventories"
"property plant and equipment"
"liabilities"
"long term debt"
```

➔ Balance Sheet items provide "as is" current totals.

Cash Flow items

```
"operating activities"
"cash"
```

Income Statement items

```
"net income"
"interest expense"
"ebit"
"stock compensation expense"
"operating income before depreciation"
"research and development expense"
"pretax income"
"sales turnover"
```

Corporate items

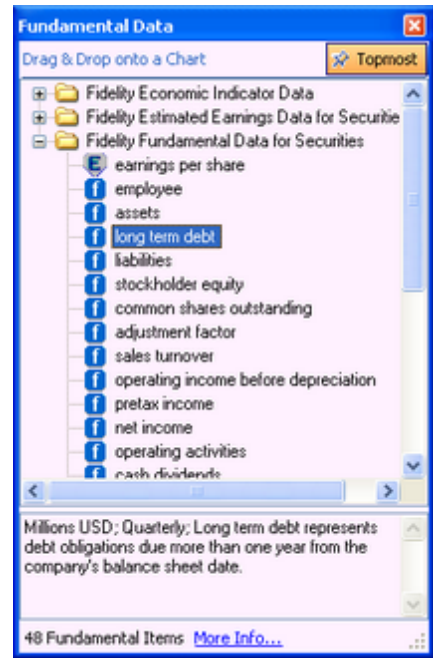
```
"adjustment factor" /* updates as required */
"common shares used to calculate eps diluted" /* updates annually */
"common shares outstanding"
"cash dividends"
"dividend" /* per share, split-adjusted */
"employee"
"fiscal quarter"
"fiscal year"
"split" /* updates as required */
```

[Equity Summary Score](#)^[82] ([click to topic](#)^[82])

```
"equity summary score"
"equity summary category"
```

Estimated Earnings

```
"estimated earnings"
```



A convenient item name reference is the Fundamental Data dialog.

Sentiment

Analyst Ratings

"all analyst ratings", "analyst upgrade", or "analyst downgrade".

- ➔ See Analyst Upgrades Downgrades Data in the Fundamental Data and Economic Indicator Definitions Guide (Help menu) for more information.

Security Sentiment


"net insider transactions", "insider buy", or "shares short"

"days to cover"

"short interest as a % of shares outstanding"

- ➔ Short interest data points are updated twice a month (semi-monthly) in accordance with SEC requirements.



Create Strategies with the Insider and Short Interest sentiment items by employing rules from the General Fundamental folder in the Strategy Builder. You can also drag, drop, and  push Short Interest items into a Chart or Strategy Window.

8.2 Economic Indicator Data

➔ Economic Indicators have been decommissioned. Cached data on local machines may still be accessed, but the indicators will no longer update.

Items can be accessed regardless of the primary charted symbol using `GetExternalSymbol` or `SetContext`. Also see [Plotting Fundamental Items](#)³³.

"capacity utilization"	Capacity Utilization (% of Capacity)
"consumer confidence"	Consumer Confidence (Value, Hundreds)
"core cpi"	Core CPI (Percent)
"core ppi"	Core PPI (Percent)
"cpi"	CPI (Percent)
"existing home sales"	Existing Home Sales (Millions)
"gdp"	GDP-Final (% Change)
"gdp-forecast"	GDP-Forecast (% Change)
"housing starts"	Housing Starts (% Change or Millions)
"initial jobless claims"	Initial Jobless Claims (Thousands)
"ism index"	ISM Index (Percent)
"michigan sentiment-final"	Michigan Sentiment-Final (Value, Hundreds)
"new home sales"	New Home Sales (Millions)
"non-farm payroll"	Non-Farm Payroll (Thousands)
"personal spending"	Personal Spending (% Change)
"ppi"	PPI (Percent)
"retail sales (excl autos)"	Retail Sales (% Change)
"retail sales"	Retail Sales (Excl Autos) (% Change)
"trade balance"	Trade Balance (Billions)
"unemployment rate"	Unemployment Rate (Percent)

Tip:

If you don't need to access its data in a script, the quickest way to plot an item is to drag it from the Fundamental Data dialog (*Ctrl+U*) and drop it in a chart window.

8.3 Market Sentiment Data

A DataSet named *Market Sentiment* is pre-loaded with your Wealth-Lab Pro installation containing the following market sentiment indicators:

Indicator Description	NYSE	NASDAQ	AMEX
Advancing Issues / Volume	.MB_ADV.N	.MB_ADV.Q	.MB_ADV.A
Declining Issues / Volume	.MB_DEC.N	.MB_DEC.Q	.MB_DEC.A
Unchanged Issues / Volume	.MB_UNC.N	.MB_UNC.Q	.MB_UNC.A
New 52-Week Highs	.MB_NH.N	.MB_NH.Q	.MB_NH.A
New 52-Week Lows	.MB_NL.N	.MB_NL.Q	.MB_NL.A
Total Volume	.MB_TV.N	.MB_TV.Q	.MB_TV.A
Arms Index	.STI.N	.STI.O	.STI.A

Market sentiment symbols are not tradeable, but you can access their data in your trading strategies in the normal way using `GetExternalSymbol` or `SetContext`. Note that advancing, declining, and unchanged issues have an associated Volume series representing the corresponding share volume. The following example will clarify (run on any symbol in the Dow 30, for example):

Example ([How to run Example code?](#)³)

```

C#
protected override void Execute()
{
    //Access the NYSE advancing issues and their volume for a trading system
    Bars advN = GetExternalSymbol(".MB_ADV.N", true );

    //Now you can use advN.Close and advN.Volume as references to series containing
    advancing issues/volume }
    ChartPane cp = CreatePane(40, true, true);
    PlotSeries(cp, advN.Close, Color.Blue, LineStyle.Histogram, 2);
    PlotSeries(VolumePane, advN.Volume, Color.Green, LineStyle.Solid, 2);
}

```

More examples of how to access and use these data can be found in the Strategy Builder as "Market Sentiment Conditions". Note, however, that the Strategy Builder use methods from the WealthLab.Rules component, which hides the complexity of specifying the exact market sentiment symbol as in the example above. Instead, the WealthLab.Rules methods require you, for example, to specify the exchange for the Arms Index - the Strategy Builder are programmed to use the corresponding symbol.

8.4 GICS Data

Download and keep Global Industry Classification Standard (GICS) Industry data updated using the DataSource Manager (*Ctrl+M*) to Create a New [Fidelity] DataSet; it's not required to complete creating a new DataSet to update GICS data. Choose the option for Industry Classification groups and finally click the Update Industry Data button.

Industry data is stored in the Gics.xml data in your Wealth-Lab Data folder. The GICS database provides an 8-digit code that classifies every security traded, and the code is divided into four sections (2 digits each):

- Sector
- Industry Group
- Industry
- Sub-Industry

This structure provides a hierarchy that can be used as a basis for analyzing securities in various industries. The Gics static class in the {}WealthLab.DataProviders namespace provides the necessary functions to easily extract GICS data for any symbol in the database.

For a complete map of the GICS structure, refer to the Standard & Poor's website.

Members of WealthLab.DataProviders.Gics

public static [string](#) **DataPath** { set; get; }

Wealth-Lab sets the DataPath to the Gics.xml file and you can retrieve this property in your scripts. The DataPath is used internally for the other members in this group, so there's no need for you to use this property in your scripts.

public static [string](#) **GetIndustry**([string](#) *symbol*)

Returns the 6-digit GICS industry code of the specified *symbol* as a string. Pass the result of **GetIndustry** to **GetDesc** to return a textual description of a GICS Industry.

public static [string](#) **GetIndustryGroup**([string](#) *symbol*)

Returns the 4-digit GICS (Global Industry Classification Standard) industry group code of the specified Symbol as a string. Pass the result of **GetIndustryGroup** to **GetDesc** to return a textual description of a GICS Industry Group.

public static [string](#) **GetSubIndustry**([string](#) *symbol*)

Returns the 8-digit GICS (Global Industry Classification Standard) sub-industry code of the specified *Symbol* as a string. Pass the result of **GetSubIndustry** to **GetDesc** to return a textual description of a GICS Sub-Industry.

public static [string](#) **GetSector**([string](#) *symbol*)

Returns the 2-digit GICS (Global Industry Classification Standard) sector code of the specified Symbol as a string. Pass the result of **GetSector** to **GetDesc** to return a textual description of a GICS Sector.

public static [string](#) **GetDesc**([string](#) *gicsCode*)

Returns a textual description of the specified *gicsCode*. Use the result of any of the functions above as the *gicsCode* parameter: GetSector, GetIndustryGroup, GetIndustry, or GetSubIndustry.

- ➔ If the *symbol* or *gicsCode* parameter does not correspond to a valid description, a runtime exception occurs with the message: "The given key was not present in the dictionary".

Example ([How to run Example code?](#)³⁾)

☐ C#

```
using System;
using WealthLab;
using WealthLab.DataProviders;      /* Gics is here */

namespace WealthLab.Strategies
{
    public class MyStrategy : WealthScript
    {
        protected override void Execute()
        {
            ClearDebug();
            string sym = Bars.Symbol;
            try {
                string sector = Gics.GetSector(sym);
                string industryGroup = Gics.GetIndustryGroup(sym);
                string industry = Gics.GetIndustry(sym);
                string subIndustry = Gics.GetSubIndustry(sym);

                // Print out the GICS codes
                PrintDebug( sym );
                PrintDebug( Gics.DataPath );
                PrintDebug( String.Format("{0} = {1}", sector, Gics.GetDesc( sector ) )
            );
                PrintDebug( String.Format("{0} = {1}", industryGroup,
Gics.GetDesc( industryGroup ) ) );
                PrintDebug( String.Format("{0} = {1}", industry, Gics.GetDesc( industry
) ) );
                PrintDebug( String.Format("{0} = {1}", subIndustry,
Gics.GetDesc( subIndustry ) ) );
            }
            catch (Exception e) {
                PrintDebug( e.Message );
            }
        }
    }
}
```

8.5 Equity Summary Score

Equity Summary Scores consolidate the leading independent stock research providers' relative, historical recommendation performance along with other factors to provide an aggregate, accuracy-weighted indication of the independent research firms' stock sentiment. It is calculated by StarMine, an independent third-party, using a proprietary model based on the daily opinions of independent research firms.

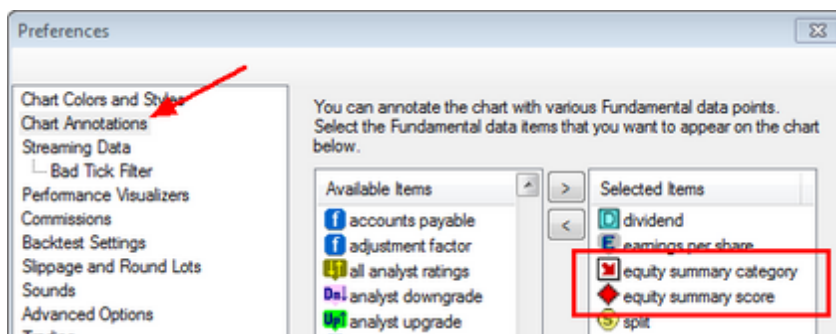
Equity Summary Score has three main factors in its calculation. The first is normalization of data where scarce ratings are given more importance by adding weight to them. StarMine adjusts their recommendation values by overweighting the scarce values and underweighting the plentiful ratings. The second factor is weighted historical accuracy. StarMine uses the previous 24 month relative firm/sector ratings accuracy to determine which firms' ratings have the most weight in the aggregated ESS. The final factor used in ESS is forced ranking between the 1,500 largest market capitalization stocks. Equity Summary Score is on a scale of 0.1 to 10.0, and by ranking the largest stocks. This ensures there will be uniform distribution of scores of the larger stocks, and then the smaller cap stocks are slotted into the distribution without ranking.

For more information see Research Firm Details for StarMine at fidelity.com.

The following table summarizes the Equity Summary Score, chart icons, and associated sentiment ratings (up/downgrade) by StarMine.

Icon	Score	Sentiment Rating	Downgrade To	Upgrade To
	0.1 to 1.0	Very Bearish		
	1.1 to 3.0	Bearish		
	3.1 to 7.0	Neutral		
	7.1 to 9.0	Bullish		
	9.1 to 10.0	Very Bullish		

To enable the Equity Summary Score and Category charting icons, select them for display in Preferences (*F12*) > Chart Annotations



Accessing Equity Summary Score in a Strategy

While you can create screens for stocks using current Equity Summary Scores (ESS) at fidelity.com, Wealth-Lab Pro has access to historical ESS data points as well as the number of firms participating in each score for use in trading strategies.

You access fundamental DataSeries of Equity Summary Scores/Category in the same way as other fundamental data: by passing the strings *"equity summary score"* or *"equity summary category"*

summary category" as parameters to the functions `FundamentalDataSeries` or `GetFundamentalItem`, for example. In addition, the *"equity summary score"* `FundamentalItem` returns the number of firms participating in the score by passing "firms" to the `FundamentalItem.GetDetail` method.

The Strategy Builder provides the fastest way to start a strategy that employs Equity Summary Score. When you add a condition using Equity Summary Score, the wizard creates a pane with colored ranges representing the Equity Summary categories. The result provides a fine visual reference for a company's ESS history.

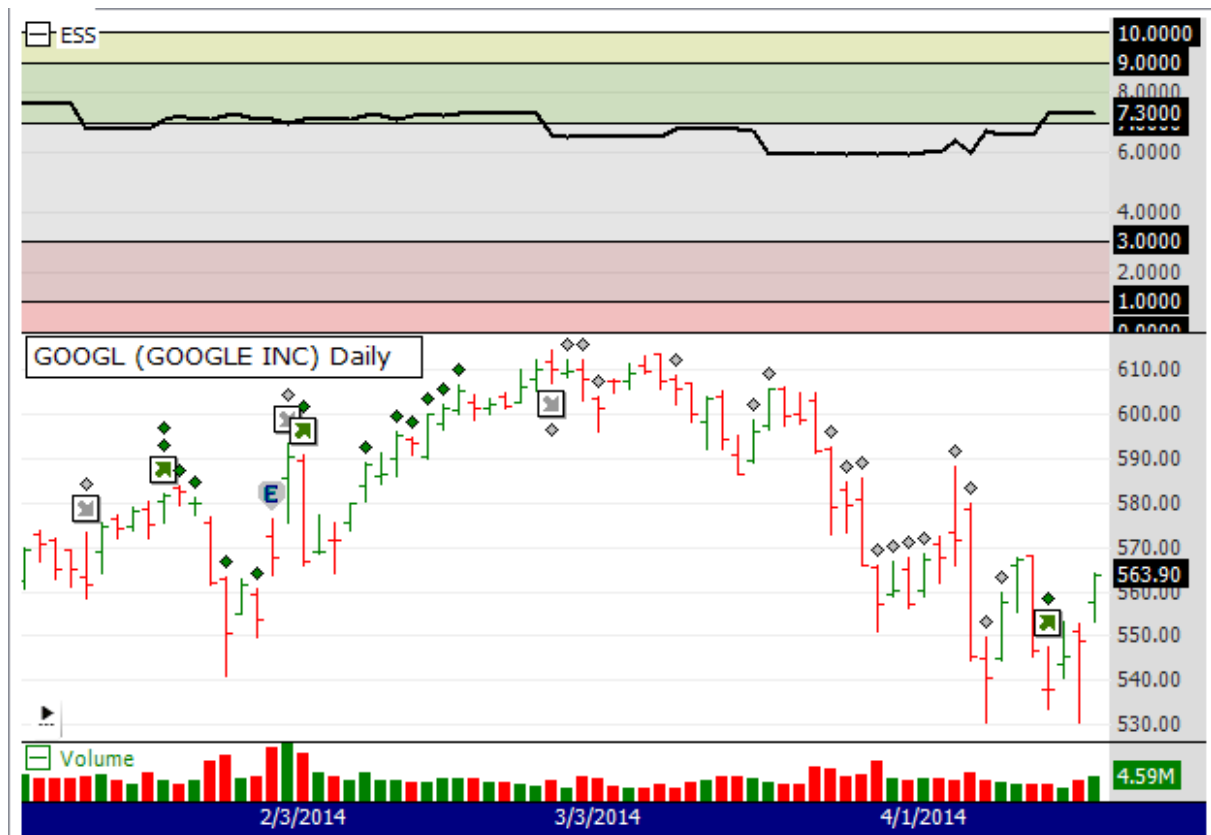


Chart showing Equity Summary Score and Category annotations in addition to the ESS pane with category coloring.

Your Wealth-Lab Pro® installation includes Equity Summary Score-based Strategies in the Equity Summary Score Strategy folder for you to reference as additional programming examples.

9 Multi-Time Frame Analysis

A number of popular trading techniques rely on the ability to access data from higher timeframes, and WealthScript provides a set of functions specifically for this purpose. From underlying daily data, indicators for weekly or monthly charts could be created, just as intraday chart data could be compressed into higher intraday scales - or even into daily, weekly and monthly bars.

Multi-Time Frame Analysis allows you to access data and indicators based on higher timeframes, but create trades in the chart's base scale.

For example, let's say that you'd like your trade setup to be based on an underlying trend measured by a moving average of weekly bars. In Wealth-Lab Version 6 you can identify that trend and trade it on a daily (or even intraday) basis.

How to: Programmatically determine the Chart Scale

To determine the chart's (**Bars**) timeframe, you can use the **BarDataScale** struct to return a simple value-type that combines the **Bars.Scale** (Tick, Second, Minute, Daily, Weekly, Monthly, Quarterly or Yearly) with the **Bars.BarInterval** (e.g., 20-tick, 30-second, 60-minute). **BarInterval** returns zero for non-intraday scales.

[Example \(How to run Example code? ³\)](#)

```

C#
protected override void Execute()
{
    // Returns the chart bar interval and data scale
    BarDataScale ds = Bars.DataScale;
    if( ds.BarInterval == 0 )
        PrintDebug( ds.Scale );
    else
        PrintDebug( ds.BarInterval + "-" + ds.Scale );
}
```

9.1 Intraday/Intraday

To access data from a higher time frame in an intraday chart, use `SetScaleCompressed()` method with the required target period. After you're finished operating in the higher time frame, call the `RestoreScale()` method. The next step is to return a new synchronized `DataSet` or `Bars` object which is implemented by calling the `Synchronize()` method.

How to: Access Data in a Higher Intraday Scale

This example demonstrates how to access the data in a higher intraday scale (120-minute for the illustration purposes) from a lower scale (e.g. 15-minute). We obtain two simple moving averages of different periods on each scale and detect the crossovers/crossunders.

Example ([How to run Example code?](#)³)

■ C#

```
protected override void Execute()
{
    // Strategy requires intraday data
    if( Bars.BarInterval == 0 ){
        DrawLabel(PricePane, "Strategy requires intraday data", Color.Red);
        return;
    }
    // Enough data to create a 60-bar 120-minute indicator?
    int barsRequired = 120 * (60 + 1) / Bars.BarInterval;
    if (Bars.Count < barsRequired){
        DrawLabel(PricePane, "Strategy requires " + barsRequired.ToString() + " bars, minimum", Color.Red);
        return;
    }

    // Get 20- and 60-bar SMAs in original scale
    DataSet SMA20 = SMA.Series( Close, 20 );
    DataSet SMA60 = SMA.Series( Close, 60 );

    //Switch to 120-minute scale
    SetScaleCompressed( 120 );

    //Obtain 20-and 60-bar 120-minute SMAs
    DataSet SMA20_120 = SMA.Series( Close, 20 );
    DataSet SMA60_120 = SMA.Series( Close, 60 );

    //Switch back to the original time frame
    RestoreScale();
    SMA20_120 = Synchronize( SMA20_120 );
    SMA60_120 = Synchronize( SMA60_120 );

    //Plot the SMAs
    PlotSeries( PricePane, SMA20, Color.DarkRed, WealthLab.LineStyle.Solid, 2 );
    PlotSeries( PricePane, SMA60, Color.DarkGreen, WealthLab.LineStyle.Solid, 2 );
    PlotSeries( PricePane, SMA20_120, Color.Red, WealthLab.LineStyle.Solid, 2 );
    PlotSeries( PricePane, SMA60_120, Color.Green, WealthLab.LineStyle.Solid, 2 );

    // Highlight crossovers and crossunders
    for(int bar = barsRequired; bar < Bars.Count; bar++)
    {
        if( CrossOver( bar, SMA20, SMA60 ) )
            AnnotateChart( PricePane, Bars.DataScale + " Crossover", bar, SMA20[bar], Color.Red );
    }
}
```

```
        if( CrossOver( bar, SMA20_120, SMA60_120 ) )
            AnnotateChart( PricePane, "120 Min Crossover", bar, SMA20_120[bar],
Color.Green);
        if( CrossUnder( bar, SMA20, SMA60 ) )
            AnnotateChart( PricePane, Bars.DataScale + " Crossunder", bar, SMA20[bar],
Color.Red );
        if( CrossUnder( bar, SMA20_120, SMA60_120 ) )
            AnnotateChart( PricePane, "120 Min Crossunder", bar, SMA20_120[bar],
Color.Green);
    }
}
```

9.2 Intraday/Daily

How to: Access Daily Data from an Intraday Chart

The example demonstrates how to calculate the so called "Floor Trader Pivots", also known as daily support and resistance lines. These series use intraday data compressed to the Daily scale. As always, when working with data in multiple timeframes, plotting and trading must be done on the chart's base scale.

Delayed Synchronization

In this example, the Daily data are [delayed](#) by one bar in the intraday timeframe so that the current intraday bar, especially the final bar for each day, always synchronizes with the previous day's data. Otherwise new pivots for the current day would be displayed on the last bar in a Streaming chart. Wealth-Lab Version 4 customers will recognize this as "Option 3" synchronization.

Example ([How to run Example code?](#))

```
C#
protected override void Execute()
{
    const int NumTicks = 2; // Number of ticks past the pivot
    SetScaleDaily();
    DataSeries hPivot = AveragePriceC.Series(Bars);
    DataSeries hDayHigh = High;
    DataSeries hDayClose = Close;
    DataSeries hDayLow = Low;
    RestoreScale();
    hPivot = Synchronize( hPivot ) >> 1;
    hDayHigh = Synchronize( hDayHigh ) >> 1;
    hDayClose = Synchronize( hDayClose ) >> 1;
    hDayLow = Synchronize( hDayLow ) >> 1;

    hPivot.Description = "Pivot";
    DataSeries hR1 = ( 2 * hPivot - hDayLow );
    hR1.Description = "R1";
    DataSeries hS1 = ( 2 * hPivot - hDayHigh );
    hS1.Description = "S1";
    DataSeries hR2 = ( hPivot - ( hS1 - hR1 ) );
    hR2.Description = "R2";
    DataSeries hS2 = ( hPivot - ( hR1 - hS1 ) );
    hS2.Description = "S2";

    HideVolume();
    PlotSeries( PricePane, hR2, Color.Maroon, LineStyle.Dotted, 1 );
    PlotSeries( PricePane, hR1, Color.Olive, LineStyle.Solid, 1 );
    PlotSeries( PricePane, hPivot, Color.Blue, LineStyle.Solid, 1 );
    PlotSeries( PricePane, hS1, Color.Fuchsia, LineStyle.Dotted, 1 );
    PlotSeries( PricePane, hS2, Color.Red, LineStyle.Solid, 1 );

    // Breakout value to be added/subtracted from resistance/support
    double boVal = Bars.SymbolInfo.Tick * NumTicks;

    // Find the StartBar of the 2nd day
    int StartBar = 0;
    int cnt = 0;
    for(int bar = 0; bar < Bars.Count; bar++)
    {
        if( Bars.IntradayBarNumber(bar) == 0 ) cnt++;
        if( cnt == 2 )
    }
```

```

        {
            StartBar = bar;
            break;
        }
    }
    for(int bar = StartBar; bar < Bars.Count; bar++)
    {
        // Don't trade on the first 2 bars of the day, 0 and 1
        if( Bars.IntradayBarNumber(bar) < 2 ) continue;

        if (IsLastPositionActive)
        {
            Position p = LastPosition;
            if( p.PositionType == PositionType.Long )
            {
                if( !SellAtStop(bar + 1, p, hS1[bar], "StopLoss" ) )
                    SellAtLimit(bar + 1, p, hR2[bar], "PftTgt" );
            }
            else if( !CoverAtStop(bar + 1, p, hR1[bar], "StopLoss" ) )
                CoverAtLimit(bar + 1, p, hS2[bar], "PftTgt" );
        }
        else if( Bars.IsLastBarOfDay( bar ) == false )
        {
            bool NewPositionEntered = false;
            if( Close[bar] < hR1[bar] )
                NewPositionEntered = BuyAtStop(bar + 1, hR1[bar] + boVal) != null;
            if( !NewPositionEntered )
            {
                if( Close[bar] > hS1[bar] )
                    ShortAtStop(bar + 1, hS1[bar] - boVal);
            }
        }
    }
}

```

How to: Overlay Daily Candlesticks on an Intraday Chart

The next example presents artificial Daily candlesticks plotted on an intraday chart.

[Example \(How to run Example code?\)](#)

■ C#

```

protected override void Execute()
{
    HideVolume();
    // Ensure intraday data
    if ( !Bars.IsIntraday ) return;

    // Get daily bars
    SetScaleDaily();
    Bars iDay = Bars;
    RestoreScale();
    iDay = Synchronize( iDay );    // Standard (fast) synch

    for(int bar = 1; bar < Bars.Count; bar++)
    {
        if ( Bars.IsLastBarOfDay(bar) || (bar == Bars.Count-1) )
        {
            int o = bar - Bars.IntradayBarNumber( bar ); // first bar of the day
            double p1 = Open[o];
            double p2 = Close[bar];
            double p3 = iDay.High[bar];
            double p4 = iDay.Low[bar];
            double[] rectangle = { o, p1, o, p2, bar, p2, bar, p1 }; // counter-clockw

```

```
int avgBar = bar - Bars.IntradayBarNumber(bar)/2;
if( p2 > p1 ){
    DrawPolygon( PricePane, Color.Silver, Color.Empty, WealthLab.LineStyle.
    DrawLine( PricePane, avgBar, p4, avgBar, p1, Color.Silver, WealthLab.Li
    DrawLine( PricePane, avgBar, p3, avgBar, p2, Color.Silver, WealthLab.Li
}
else {
    DrawPolygon( PricePane, Color.Silver, Color.Silver, WealthLab.LineStyle
    DrawLine( PricePane, avgBar, p3, avgBar, p4, Color.Silver, WealthLab.Li
}
}
}
```

9.3 Intraday/Weekly,Monthly

Intraday can be scaled to Daily or even Weekly and Monthly scales - provided that a sufficient amount of data exists in the chart's intraday base time frame. For example, you need over 8,000 1-Minute bars for a single Monthly bar.

How to: Plot Weekly Support/Resistance on an Intraday Chart

Not only it's possible to reference a higher intraday scale or daily data from the strategy. One step further is to utilize Weekly and Monthly data. Here, we demonstrate how to look up for weekly support and resistance levels, which are the moving averages of lows/highs, on intraday charts.

Example ([How to run Example code?](#) ³⁷)

```

C#
protected override void Execute()
{
    // Search for weekly support/resistance levels on intraday charts
    // Support/resistance is the moving average of lows/highs
    if( Bars.Scale != BarScale.Minute ) return;

    SetScaleWeekly();
    DataSeries WeeklySup = SMA.Series( Bars.Low, 5 );
    DataSeries WeeklyRes = SMA.Series( Bars.High, 5 );
    RestoreScale();
    WeeklySup = Synchronize( WeeklySup );
    WeeklySup.Description = "Weekly Support";
    WeeklyRes = Synchronize( WeeklyRes );
    WeeklyRes.Description = "Weekly Resistance";

    HideVolume();
    for(int bar = 0; bar < Bars.Count; bar++)
        SetBackgroundColor( bar, Color.LightSkyBlue );
    PlotSeriesFillBand(PricePane, WeeklySup, WeeklyRes, Color.Black, Color.White,
LineStyle.Solid, 2);
}

```


9.4 Daily/Weekly

How to: Access Weekly Indicator Series from a Daily Chart

To illustrate the concept, the following strategy buys when Daily **MACD** crosses over zero, and sells when Weekly **MACD** crosses under zero.

Example ([How to run Example code?](#)³⁴)

```

C#
protected override void Execute()
{
    /*
    Long position is opened when Daily MACD crosses over zero
    The position is closed when Weekly MACD crosses below zero
    */
    if( Bars.Scale != BarScale.Daily ) return;

    SetScaleWeekly();
    DataSeries WeeklyMACD = MACD.Series( Bars.Close );
    RestoreScale();
    WeeklyMACD = Synchronize( WeeklyMACD );
    WeeklyMACD.Description = "Weekly MACD";

    for(int bar = 60; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            if( CrossUnder( bar, WeeklyMACD, 0 ) )
                SellAtMarket( bar+1, LastPosition, "Weekly MACD xu 0" );
        }
        else
        {
            if( CrossOver( bar, MACD.Series( Bars.Close ), 0 ) )
                BuyAtMarket( bar+1, "Daily MACD xo 0" );
        }
    }

    HideVolume();
    ChartPane WeeklyMACDPane = CreatePane( 30, false, true );
    DrawHorzLine( WeeklyMACDPane, 0, Color.Red, WealthLab.LineStyle.Solid, 2 );
    PlotSeries( WeeklyMACDPane, WeeklyMACD, Color.Blue,
    WealthLab.LineStyle.Histogram, 2 );
}

```

9.5 Daily/Monthly

How to: Access Monthly Indicator Series from a Daily Chart

To access Weekly and Monthly data from a strategy working on Daily data, call the `SetScaleWeekly()` and `SetScaleMonthly()` methods respectively. After you're finished operating in the higher time frame, call the `RestoreScale()` method. The next step is to return a new synchronized `DataSeries` or `Bars` object which is achieved by calling the `Synchronize()` method.

Here is a sample strategy that utilizes the three time frames: Daily, Weekly and Monthly in order to arrive at trading decision. If both the Weekly and Monthly `CCI` indicators are greater than 100, and the Daily `CCI` crosses over 100, an order is generated to enter next bar at market. This long position is exited when Daily `CCI` crosses under -200.

Example ([How to run Example code?](#) )

```
C#
using System;
using System.Collections.Generic;
using System.Text;
using System.Drawing;
using WealthLab;
using WealthLab.Indicators;

namespace WealthLab.Strategies
{
    public class CCITriplex : WealthScript
    {
        private StrategyParameter cciPeriod; // parameter for period

        public CCITriplex()
        {
            cciPeriod = CreateParameter("CCI Period", 10, 10, 60, 5 );
        }
        protected override void Execute()
        {
            int Period = cciPeriod.ValueInt;
            if( Bars.Scale != BarScale.Daily ) return;

            DataSeries cciD = CCI.Series( Bars, Period );
            SetScaleWeekly();
            DataSeries cciW = CCI.Series( Bars, Period );
            RestoreScale();
            SetScaleMonthly();
            DataSeries cciM = CCI.Series( Bars, Period );
            RestoreScale();
            cciW = Synchronize( cciW ); cciW.Description = "Weekly CCI";
            cciM = Synchronize( cciM ); cciM.Description = "Monthly CCI";

            ChartPane cciPane = CreatePane( 50, true, true );
            PlotSeries( cciPane, cciD, Color.Green, WealthLab.LineStyle.Solid, 1 );
            PlotSeries( cciPane, cciW, Color.Blue, WealthLab.LineStyle.Solid, 1 );
            PlotSeries( cciPane, cciM, Color.Red, WealthLab.LineStyle.Solid, 1 );

            for(int bar = Period; bar < Bars.Count; bar++)
            {
                if (IsLastPositionActive) {
                    if( CrossUnder( bar, cciD, -200 ) )
```

```
        SellAtMarket( bar+1, LastPosition );
    }
    else if( ( cciW[bar] >= 100 ) && ( cciM[bar] >= 100 ) &&
CrossOver( bar, cciD, 100 ) )
        BuyAtMarket( bar+1 );
    }
}
}
```

10 Backtesting with ChartStyles.Trending

The Trending Chart Styles (Kagi, Renko, Line Break, and Point & Figure) exist in the {} WealthLab.ChartStyles.Trending namespace. The Trending Chart Style class names are:

```
KagiChartStyle  
RenkoChartStyle  
LineBreakChartStyle  
PnFChartStyle
```

In WealthScript, you can gain access to the current Chart Style utilizing the **ChartStyle** property and casting it as the underlying Chart Style in use. For example, the *Kagi Basic* Strategy accesses the instance of a KagiChartStyle with the following line of code:

```
KagiChartStyle kagiCS = (KagiChartStyle)ChartStyle;
```

In the current version of Wealth-Lab Pro (6.2 and up), however, the Trending Chart Styles and scripts now both have access to and can use the following classes:

```
TKagi  
TRenko  
TLineBreak  
TPnF
```

In other words, when you view a Kagi Chart, the KagiChartStyle creates a TKagi object. Likewise, when you want to create a script that uses Kagi methods, you will create its own TKagi object in your WealthScript code. The object the you create in a script can be completely different based on the Bars object (which can be from any symbol) and properties that you use to instantiate TKagi. This arrangement gives you maximum flexibility since you do not even need to employ the visual Chart Style whose objects your script references!

Strategies create (instantiate) their own Chart Style objects and are therefore *disconnected* from the properties that you set for a Chart Style. For example, if you use a 1% reversal in your Kagi *chart*, your code that uses Kagi methods should also use 1% so that the signals are coordinated with the chart.

10.1 Column Class (Reference)

The Trending Chart Styles are similar in that multiple prices are displayed in the same *column*. Renko and Line Break are special cases of columns, and these can be thought of as *horizontally-extended columns*. Accordingly, Column is the base class name of column objects for Kagi, Renko, Line Break, and Point and Figure (PnF).

Each bar number in a traditional OHLC chart has an associated column object. In this way, it's possible to access the column properties as of a specific bar. Note that another characteristic of all Trending Chart Styles is that price movement may or may not contribute to a column's high or low price value. Consequently, it's possible to advance from bar to bar without any corresponding movement in a Trending Chart Style, each of which is dependent on the Chart Style's settings.

- ➔ For Kagi and PnF charts, a single plotted column represents an entire bullish or bearish move. For Renko and Line Break charts, several consecutive columns can represent bullish or bearish periods, which are plotted using "bricks" or bars in a stair-step fashion.

Members of WealthLab.ChartStyles.Trending.Column

public **int** **Bar** { get; }

For Kagi, Line Break, and Renko, new Column objects are added for each bar in the chart, which is reflected in this value. However, for Point and Figure, new PnF Columns are created only when one of its property values change. For backtesting, since the bar number is always known, you'll normally pass the current bar to a Columns collection object to access the current state of a column; consequently it's usually never necessary to access this property.

public **int** **Col** { get; }

Col is the column number of the object. Just as bars are numbered from 0 to Bars.Count - 1, Trending ChartStyle columns are numbered from 0 to Columns.Count - 1.

public **bool** **DirectionUp** { get; }

The current Chart Style trend direction as of the specified bar. If true, the trend is bullish, and if false the trend is bearish.

public **double** **High** { get; }

The high of the column as of the specified Bar. Note that the most-recent closing price can always be accessed directly from the Bars object.

- ➔ For Point & Figure log method, this is the natural log value of price. Convert to the the arithmetic price using Math.Pow(value, Math.E).

public **double** **Low** { get; }

The low of the column as of the specified Bar. Note that the most-recent closing price can always be accessed directly from the Bars object.

- ➔ For Renko and Line Break charts the low/high of a column refers to the low and high of 1 or more bricks/lines for a particular bar. The Low and High Column properties continue to report the low/high of the last bricks/lines drawn. If no new brick/line for a particular bar, then Low and High are the most-recent value that the last brick/line drawn.

➔ For Point & Figure log method, this is the natural log value of price. Convert to the arithmetic price using `Math.Pow(value, Math.E)`.

public **bool Plotted** { set; get; }

Indicates true if the Chart Style column object should be plotted on the specified bar and must be reset for the following bar (new object required). The plotted property is read/write for Kagi and PnF styles which need to set plotted retroactively after a Column object has been added to the collection.

public **bool Reversed** { get; }

Indicates true if the Chart Style trend direction reversed on the specified bar. Note that even if the next Bar does not change the chart after a reverse, the reversed property will toggle to false since a reverse will occur only for one bar in a column.

public **double ReverseLevel** { get; }

The price at which the column will reverse (in the future). Can be used as a stop for the next bar, for example.

➔ For Point & Figure log method, this is the natural log value of price. Convert to the arithmetic price using `Math.Pow(value, Math.E)`.

public **double Volume** { get; }

The cumulative volume of the column as of the specified Bar. For Renko charts, the volume is cumulative for the current trend.

➔ This field is not valid for Point & Figure.

10.1.1 Kagi

Members of `WealthLab.ChartStyles.Trending.Kagi`

The following members are unique to the Kagi Chart Style. They complete the properties for a *Kagi Column*.

public **bool IsBullish** { get; }

`IsBullish` is not the same as `DirectionUp`. A Kagi chart `IsBullish` when the thick yang line is being drawn. This condition can and does continue when `DirectionUp` is false for some columns.

public **double YangLevel** { get; }

The level at which the Kagi chart turns bullish. This is the level of the most-recent "shoulder" (peak).

public **double YinLevel** { get; }

The level at which the Kagi chart turns bearish. This is the level of the most-recent "waist" (trough).

10.1.1.1 TKagi

The `TKagi` class serves up the object used by `KagiChartStyle`, and, you can create your own script-based `TKagi` objects without the requirement to actually use the Kagi Chart Style. Furthermore, you can create multiple instances of `TKagi` using not only different settings, but also for external symbols by passing their `Bars` object.

Members of WealthLab.ChartStyles.Trending.TKagi

public enum KagiReverseType { Points, Percent, ATR }

The Kagi reversal method that determines how to interpret the *reversalAmount* parameter in the TKagi constructor (below).

public TKagi([WealthLab.Bars](#) bars, [KagiReverseType](#) kagiReverseType, **double** reversalAmount, **int** kagiATRPeriod)

The TKagi constructor. Pass a Bars object, a KagiReverseType enumeration, the *reversalAmount*, and an integer for *kagiATRPeriod*, the ATR period, which is used only when *kagiReverseType* = KagiReverseType.ATR.

public IDictionary<int, PnF> Columns;

Columns holds the collection of Kagi Column objects, indexed by bar number.

For an example, use the ChartScript Explorer's Download feature to download the *Kagi Basic [Rev A]* Strategy to your *Trend Following* Strategy folder.

10.1.2 Renko

Members of WealthLab.ChartStyles.Trending.Renko

The following members are unique to the Renko Chart Style. They complete the properties for a *Renko Column*.

public int BricksInTrend { get; }

The number of total bricks in the current trend, which can be thought of as a horizontally-extended column.

public double NextBrick { get; }

The price level at which the next brick will be drawn in the current trend.

10.1.2.1 TRenko

The TRenko class serves up the object used by RenkoChartStyle, and, you can create your own script-based TRenko objects without the requirement to actually use the Renko Chart Style. Furthermore, you can create multiple instances of TRenko using not only different settings, but also for external symbols by passing their Bars object.

Members of WealthLab.ChartStyles.Trending.TRenko

public TRenko([WealthLab.Bars](#) bars, **double** rkoAmount)

The TRenko constructor. Pass a Bars object and the *rkoAmount*.

public IDictionary<int, PnF> Columns;

Columns holds the collection of Renko Column objects, indexed by bar number.

For an example, use the ChartScript Explorer's Download feature to download the *Renko Basic [Rev A]* Strategy to your *Trend Following* Strategy folder.

10.1.3 Line Break

Members of `WealthLab.ChartStyles.Trending.LineBreak`

The following members are unique to the Line Break Chart Style. They complete the properties for a *LineBreak Column*.

public **int** **LinesInTrend** { get; }

`LinesInTrend` are the number of consecutive "white" or "black" lines drawn in the same direction. If the number is less than the `_lineBreakAmount`, then the trend has not been confirmed, i.e., non-trending.

For an example, use the ChartScript Explorer's Download feature to download the *Line Break Basic [Rev A]* Strategy to your *Trend Following* Strategy folder.

10.1.3.1 TLineBreak

The `TLineBreak` class serves up the object used by `LineBreakChartStyle`, and, you can create your own script-based `TLineBreak` objects without the requirement to actually use the Line Break Chart Style. Furthermore, you can create multiple instances of `TLineBreak` using not only different settings, but also for external symbols by passing their `Bars` object.

Members of `WealthLab.ChartStyles.Trending.TLineBreak`

public **TLineBreak**(**WealthLab.Bars** bars, **int** lines)

The `TLineBreak` constructor. Pass a `Bars` object and the number of `lines` to break, 2 or greater.

public **IDictionary**<**int**, **PnF**> **Columns**;

`Columns` holds the collection of `LineBreak Column` objects, indexed by bar number.

For an example, use the ChartScript Explorer's Download feature to download the *Line Break Basic [Rev A]* Strategy to your *Trend Following* Strategy folder.

10.1.4 Point and Figure

Members of `WealthLab.ChartStyles.Trending.PnF`

The following members are unique to the Point and Figure Chart Style. They complete the properties for a *PnF Column*.

public **int** **Boxes** { get; }

Returns the number of boxes drawn for the current column as of the specified bar

public **bool** **OneStepBack** { get; }

Returns true if the column is a confirmed one-step-back condition (1-box charts only). One step back occurs if, say, if price is trending in the X direction, reverses for one O only, and then resumes the X trend. This field remains true from the time the condition is confirmed until the next reversal.

public **bool** **OneStepBackProbationary** { get; }

Returns true if the current state of the column at Bar is a probationary one-step-back condition (1-box charts only). The condition is probationary while only one box is drawn on a reversal.

See also: [TPnF Class](#)^[99], [TrendLine Class](#)^[100]

10.1.4.1 TPnF Class

The TPnF class serves up objects used by PnFChartStyle, and, you can create your own script-based TPnF objects without the requirement to actually use the Point and Figure Chart Style. Furthermore, you can create multiple instances of TPnF using not only different settings, but also for external symbols.

Members of WealthLab.ChartStyles.Trending.TPnF

public **TPnF**(Bars bars, **ControlPrice** priceField, **double** boxSize, **int** reversalBoxes, **bool** logMethod)

The TPnF constructor. Pass a Bars object, a ControlPrice enum (below), the box size, the number of reversal boxes, and true to use the logarithmic method.

When *logMethod* = true

- *boxSize* is interpreted as a percentage (otherwise as points), and,
- *boxSize* cannot be smaller than 0.01, otherwise 0.01% is used, i.e., $\text{Ln}(1 + 0.01/100)$

public **enum** **ControlPrice** { Close, HighLow }

The price field use to generate the Point and Figure chart.

public **ControlPrice** PriceField

public **int** ReversalBoxes

public **bool** LogMethod

These properties are loaded with the values that you pass to the TPnF constructor, and consequently it is unlikely that you'll need to access them during script execution. Note that after creating TPnF, explicitly writing new values will have no effect on the Column objects.

public **double** BoxSize

Like the properties above, you pass the BoxSize to the TPnF constructor. However in the case of the log method, the BoxSize property returns the "actual" log value box size. For example, if you pass 2.0 as *boxSize*, the BoxSize property will return $\text{Ln}(1 + 2.0/100) = 0.0198026272961797$.

public **IDictionary**<**int**, **PnF**> **Columns**;

Columns holds the collection of PnF Column objects, indexed by bar number.

public **IDictionary**<**int**, **int**> **ReversalBars**;

The ReversalBars collection allows you to obtain the bar number at which the reversal occurred for a specified column number. (key, value = columnNumber, bar)

public **List**<**PnFTrendLine**> **TrendLines**;

The TrendLines list contains all Trendline information. Currently, TPnF finds 45° lines only. Note that 45° trendlines can be projected immediately following each reversal. See: [TrendLine Class](#)^[100]

public **int**[] **DbITopDbIBottom**

Returns a value > 0 if the last Point and Figure signal was a double-top buy (value < 0 double-bottom

sell). The value is 2 on the bar that triggers a double-top buy and -2 for a double-bottom sell trigger. Thereafter, the value is 1 or -1 indicating the direction of the last signal. The signal is not dependent on the prevailing trend. A double-top (bottom) signal occurs when there is one more box of X's (O's) in the current column than O's (X's) in the previous column. This indicator facilitates the creation of the Bullish Percent Index (BPI), but is not valid until the first buy or sell signal.

public void PaintContributorBars(WealthScript ws, Color XBarColor, Color OBarColor, Color NullBarColor)
Sets the colors of the charted bars that contribute to creating new Point and Figure boxes in a regular OHLC/Candlestick chart. Pass the X, O, and Null (non-contributor) colors. This method is not valid for secondary symbols.

public int Boxes(double high, double low)
Returns the number of boxes between the high and low values [of a column].

public double RoundToBox(double x)
Returns the box price corresponding to the box closest the price passed as the parameter. This is used for fine-rounding after a calculation near a box price.

public double CeilingToBox(double x)
Returns the box price corresponding to the box above the price passed as the parameter.

public double FloorToBox(double x)
Returns the box price corresponding to the box below the price passed as the parameter.

For example use, see the *Point and Figure Basic* Strategy found in your Wealth-Lab Pro installation's *Trend Following** Strategy folder.

* The Strategy may be located in the *Newly Installed Strategies* folder

10.1.4.2 TrendLine Class

The TrendLine Class is used to store the properties of trendlines in any chart. TPnF builds PnFTrendLines, which are optionally displayed by the Point and Figure ChartStyle. You can use the trendline information in your PnF Strategies. See the TrendLines collection in the [TPnF class](#)^[99].

Members of WealthLab.ChartStyles.Trending.TrendLine

public TrendLine(Bars bars, int bar1, double price1, int bar2, double price2, int idBar)
TrendLine constructor.

public int Bar1 = -1;
public double Price1 = 0d;
Bar1 and Price1 are bar and price that defines the start of the trend.

public int Bar2 = -1;
public double Price2 = 0d;
Bar2 and Price2 are the second defining bar of a general trendline. For 45 deg PnF trendlines, IdBar and Bar2 will equal the StartBar.

public int EndBar = -1;
EndBar is the bar at which trendline ends (used for drawing). If EndBar = Bars.Count - 1, then the trend defined by the trendline is currently intact.

public int IdBar = -1;

IdBar is the bar at which a trendline is confirmed/detected. For a PnFTrendLine, IdBar equals Bar1 since these special type of trendlines can be drawn immediately following a PnF reversal. See [PnFTrendLine](#) below.

public int Accel = 0;

Accel is the number of "accelerated" trendlines that are redrawn after a primary/underlying trend is identified. For PnF, the Accel represents the trendline number in the same direction as an underlying trend; e.g., if a second +45° trendline is drawn at a higher level, its Accel value is 1.

public bool IsRising = true;

When a trendline is instantiated, IsRising is set to true if Price2 > Price1.

Members of WealthLab.ChartStyles.Trending.PnFTrendLine

PnFTrendLine inherits from TrendLine; some of the

public PnFTrendLine(Bars bars, int bar1, double price1, int bar2, double price2, int idBar, int startCol, PnFAngle ang, bool risingTrendLine)
PnFTrendLine constructor.

public enum PnFAngle { a27, a34, a45, a56, a63 }

Enumerations for predefined trendline angles. Currently, only 45° PnFTrendLines are found by TPnF.

public PnFAngle Angle = PnFAngle.a45;

The angle of the PnFTrendLine.

public int StartColumn = -1;

public int EndColumn = -1;

The starting and ending columns of a PnFTrendLine in a Point and Figure Chart. EndColumn = -1 if the trend defined by a PnFTrendLine is still intact. For simplicity (at least initially) TPnF considers the end of a trend when the line "touches" a Point and Figure box. Another, perhaps more popular, interpretation of PnF trendlines is that the trend does not end until a double top/bottom signal occurs near the crossing.

public int Reversals = 0;

This field increments if a reversal occurs precisely at the value of the existing trend, and, in this case a new line is not added to the TrendLines collection.

11 WealthLab.Rules Classes

A number of classes found in the `WealthLab.Rules{ }` and `WealthLab.Rules.CandleSticks{ }` namespaces support the implementation of fundamental-based and other rules. The classes are not actually necessary to create the rules, but their use provides *indirection*, which hides the implementation details helping to keep your strategy code saner. Of course, programmers can access any of the public methods without resorting to the Strategy Builder.

11.1 ArmsIndex

Members of WealthLab.Rules.ArmsIndex

Applies to: Fidelity Wealth-Lab Pro

The *ArmsIndex* class simplifies accessing and plotting a market's Arms Index (TRIN). The Market Sentiment data for NYSE, Nasdaq, and AMEX are found in the [Market Sentiment DataSet](#)^[79].

public **ArmsIndex**([WealthLab.WealthScript](#) ws, [string](#) exchange, [double](#) level)
Constructor.

ws An instance of the WealthScript class, i.e., *this*
exchang NYSE, Nasdaq, or AMEX
e
level Specifies the level at which to draw a horizontal line in the plotted pane

public [WealthLab.DataSeries](#) **ArmsIndexSeries** { get; }
Arms Index DataSeries property.

The plot functions plot the Arms Index DataSeries, automatically creating a ChartPane, if required.

public **void plotArmsIndexSeries**()
public **void plotArmsIndexSeries**([WealthLab.ChartPane](#) paneSTI)
public **void plotArmsIndexSeries**([int](#) height, [bool](#) abovePrice, [bool](#) showGrid)

Example ([How to run Example code?](#)^[37])

```
C#
using System;
using System.Drawing;
using WealthLab;
using WealthLab.Rules;

namespace WealthLab.Strategies
{
    public class MyStrategy : WealthScript
    {
        protected override void Execute()
        {
            ArmsIndex ad = new ArmsIndex(this, "NYSE", 5 );
            DataSeries dsSTI = ad.ArmsIndexSeries;
            ad.plotArmsIndexSeries();
        }
    }
}
```

11.2 Candlesticks

Member of `WealthLab.Rules.Candlesticks.CandlePattern`

The *CandlePattern* methods, found in the `WealthLab.Rules.Candlesticks{ }` namespace, greatly simplify identifying candlestick patterns for charting and for strategy rules. All methods have the same parameter list as shown in the following typical syntax:

```
public static void ReversalLongLeggedDoji(WealthLab.WealthScript w, string annotateText, bool highlight, out bool\[\] arr)
```

w An instance of the `WealthScript` class, i.e., *this*
annotateText Text to annotate the candlestick pattern. For no annotation pass a blank string, i.e., "".
ext
highlight Pass *true* for a Fuchsia highlight around the last bar of the pattern and *false* for no highlight.
arr Declare a `bool` array to pass to the method. The array returned is sized to `Bars.Count` and indicates *true* for bars on which the candle pattern is detected. (See example.)

Being a static class, you must qualify methods by using the type name *CandlePattern* instead of using an instance reference (see example). Note that many patterns use a 10-period ATR indicator in their detection logic, and for this reason you should ignore patterns detected in approximately the first 25 bars of a chart.

The following is an example of a complete trading strategy based on two bullish and two bearish candlestick patterns. The strategy buys when either bullish pattern is detected and sells when a bearish pattern is found.

Example ([How to run Example code?](#)³⁾)

```

C#
using System;
using WealthLab;
using WealthLab.Rules.Candlesticks;

namespace WealthLab.Strategies
{
    public class CandlePatternStrategy : WealthScript
    {
        protected override void Execute()
        {
            bool[] bullishThreeWhiteSoldiers;
            CandlePattern.BullishThreeWhiteSoldiers(this, "+3White Soldiers", true,
out bullishThreeWhiteSoldiers);
            bool[] bullishInvertedHammer;
            CandlePattern.BullishInvertedHammer(this, "+Inverted Hammer", true, out
bullishInvertedHammer);
            bool[] bearishThreeBlackCrows;
            CandlePattern.BearishThreeBlackCrows(this, "-3Black Crows", true, out
bearishThreeBlackCrows);
            bool[] bearishTriStar;
            CandlePattern.BearishTriStar(this, "-Tri Star", true, out bearishTriStar);

            for(int bar = 25; bar < Bars.Count; bar++)
            {
                if (IsLastPositionActive)

```

```
{
    {
        {
            {
                Position p = LastPosition;
                if (bearishThreeBlackCrows[bar] || bearishTriStar[bar])
                    SellAtMarket(bar + 1, p);
            }
            else
            {
                if (bullishThreeWhiteSoldiers[bar] || bullishInvertedHammer[bar])
                    BuyAtMarket(bar + 1);
            }
        }
    }
}
```

Clarification for terminology used in the Candlestick definitions in the following topics:

<i>White Line/ Candle</i>	A candle whose close price is greater than the open price.
-------------------------------	--

<i>Black Line/ Candle</i>	A candle whose closing price is less than the open price.
-------------------------------	---

stable at bar 25A 10-period Average True Range (ATR) is used to determine relative candle length for many of the patterns. ATR indicator values are unstable until approximately 2.5 times the period, or 25 bars.

11.2.1 Bearish

Bearish Candlestick Method Names

Typical syntax and example: [Candlesticks](#)

Bearish Belt Hold

(2-bar pattern, stable at bar 25) A significant gap up occurs but prices are heading downwards only. After a significant gap up causes large profit-taking. Expect lower prices.

Bearish Dark Cloud Cover

(2-bar pattern, stable at bar 25) The first day is a long white day, the second day is a black day which opens above the first day's high and closes within the first day but below the midpoint. Although a gap-up indicates a uptrend, prices are heading downwards and probabilities are high that prices continue to fall.

Bearish Doji Star

(2-bar pattern, stable at bar 25) The first day is a long white day. The second day is a doji star that gaps above the first day. A star indicates a reversal and a doji indicates indecision. This pattern usually indicates a reversal following an indecisive period. You should wait for a confirmation before trading a doji star.

Bearish Engulfing Lines

(2-bar pattern, stable at bar 25) A small white line (first day) is engulfed by a large black line (second day). This pattern is strongly bearish if it occurs after a significant up-trend (i.e., it acts as a reversal pattern).

Bearish Evening Star

(3-bar pattern, stable at bar 25) The first day is a long white day, the second day gaps above the first

day's close and the third day is a long black day. The "star" indicates a possible reversal and the black line confirms this. The star can be empty or filled-in. This is a bearish pattern signifying a potential top.

BearishHangingMan

(2-bar pattern, stable at bar 25) A small real body (i.e., a small range between the open and closing prices) and a long lower shadow (i.e., the low was significantly lower than the open, high, and close). This pattern is bearish if it occurs after a significant uptrend, but detection requires only a minor uptrend measured by a 5-period SMA.

BearishHarami

(2-bar pattern, stable at bar 25) A line with a small black body (second day) falls within the area of a larger white body (first day). This pattern indicates a decrease in momentum.

BearishHaramiCross

(2-bar pattern, stable at bar 25) First day is a long white day and the second day is a Doji day that is engulfed by the first day's body. It is similar to a Harami, except the second line is a Doji (signifying indecision). This pattern indicates a decrease in momentum.

BearishKicking

(2-bar pattern, stable at bar 25) A white marubozu followed by a black marubozu where the black marubozu gaps below the 1st day's open. The gap between the 2nd and 3rd day is a resistance area, a downtrend is expected.

BearishLongBlackLine

(1-bar pattern, stable at bar 25) Prices open near the high and close significantly lower near the period's low. This is a bearish line.

BearishSeparatingLines

(2-bar pattern) The 1st day is a long white day followed by a black day that opens at the opening price of the 1st day. Although the 1st day prices are heading upwards, the second day shows significant downturn with an insignificant upper shadow. The downtrend should continue.

BearishShootingStar

(2-bar pattern, stable at bar 25) Price gap up to the upside with a small real body. The star's body must appear near the low price and the line should have a long upper shadow. This pattern suggests a minor reversal when it appears after a uptrend.

BearishSideBySideWhiteLines

(3-bar pattern) A black day followed by a gap down of two almost identical white days. No trend-reversal occurs on the 2nd and 3rd day, thus the downtrend is still intact.

BearishThreeBlackCrows

(3-bar pattern, stable at bar 25) Three black days with lower closes each day where each day opens lower but within the body of the previous day. Strong down-trend indicates a continuation of falling prices.

BearishThreeInsideDown

(3-bar pattern, stable at bar 25) The first two days are a Bearish Harami pattern followed by a black day that closes below the second day. The 3rd day is the confirmation of the Bearish Harami pattern.

BearishThreeOutsideDown

(3-bar pattern, stable at bar 25) The first two days are a Bearish Engulfing pattern followed by a black day that closes below the second day. The third day is the confirmation of the Bearish Engulfing pattern.

BearishTriStar

(3-bar pattern, stable at bar 25) Three Dojis where the second Doji gaps above the 1st and the third. High indecision during three days suggests that the trend is about to change. This pattern appears very seldom on securities with high volume.

11.2.2 Bullish

Bullish Candlestick Method Names

Typical syntax and example: [Candlesticks](#)¹⁰⁴

BullishBeltHold

(2-bar pattern) A significant gap down occurs but prices are heading upwards only. After a significant down gap causes a short-covering and strong buying. Expect higher prices.

BullishDojiStar

(2-bar pattern, stable at bar 25) First day is a long black day, the second day is a Doji day that gaps below the first day. A "star" indicates a reversal and a Doji indicates indecision. This pattern usually indicates a reversal following an indecisive period. You should wait for a confirmation before trading a Doji star.

BullishEngulfingLines

(2-bar pattern, stable at bar 25) A small black line (first day) is engulfed by a large white line (second day). This pattern is strongly bullish if it occurs after a significant downtrend (i.e., it acts as a reversal pattern).

BullishHammer

(1-bar pattern, stable at bar 25) A small real body (i.e., a small range between the open and closing prices) and a long lower shadow (i.e., the low was significantly lower than the open, high, and close). Detection requires at least a minor downtrend measured by a 5-period SMA.

BullishHarami

(2-bar pattern, stable at bar 25) A line with a small white body (second day) falls within the area of a larger black body (first day). This pattern indicates a decrease in momentum.

BullishHaramiCross

(2-bar pattern, stable at bar 25) First day is a long black day and the second day is a Doji day that is engulfed by the first day's body. It is similar to a Harami, except the second line is a Doji (signifying indecision). This pattern indicates a decrease in momentum.

BullishInvertedHammer

(2-bar pattern, stable at bar 25) Price gap down to the downside with a small real body. The star's body must appear near the low price and the line should have a long upper shadow. This pattern suggests a minor reversal when it appears after a downtrend.

BullishKicking

(2-bar pattern, stable at bar 25) A black marubozu followed by a white marubozu where the white marubozu gaps up the 1st day's open. The gap between the 2nd and 3rd day is a support area, an uptrend is expected.

BullishLongWhiteLine

(1-bar pattern, stable at bar 25) The prices open near the low and close significantly lower near the period's high. This is a bullish line.

BullishMorningStar

(3-bar pattern, stable at bar 25) The first day is a long black day, the second day gaps below the first day's close and the third day is a long white day. The "star" indicates a possible reversal and the white line confirms this. This is a bullish pattern signifying a potential bottom.

BullishPiercingLine

(2-bar pattern, stable at bar 25) The first day is a long black day, the second day is a white day which opens below the first day's low and closes within the first day but above the midpoint. This is a bullish pattern and the opposite of a Dark Cloud Cover.

BullishSeparatingLines

(2-bar pattern) The 1st day is a long black day followed by a white day that opens at the opening price of the 1st day. Although the 1st day prices are heading downwards, the second day shows significant upturn with an insignificant lower shadow. The uptrend should continue.

BullishSideBySideWhiteLines

(3-bar pattern) A white day followed by a gap up of two almost identical white days. No trend-reversal occurs on the 2nd and 3rd day, thus the uptrend is still intact.

BullishThreeInsideUp

(3-bar pattern, stable at bar 25) The first two days are a Bullish Harami pattern followed by a white day that closes above the second day. This is usually the confirmation of the Bullish Harami pattern.

BullishThreeOutsideUp

(3-bar pattern, stable at bar 25) The first two days are a Bullish Engulfing pattern followed by a white day that closes above the second day. The third day is the confirmation of the Bullish Engulfing pattern.

BullishThreeWhiteSoldiers

(3-bar pattern, stable at bar 25) Three white days with higher closes each day and where each day opens above but within the body of the previous day. This pattern represents strong uptrend.

BullishTriStar

(3-bar pattern, stable at bar 25) Three Dojis where the second Doji gaps below the 1st and the third. High indecision during three days suggests that the trend is about to change. This pattern appears very seldom on securities with high volume.

11.2.3 Neutral and Reversals

Neutral and Reversal Candlestick Method Names

Typical syntax and example: [Candlesticks](#)¹⁰⁴

NeutralDoji

(1-bar pattern, stable at bar 25) The price opens and closes at the same price. This pattern implies indecision. Double doji lines (two adjacent doji lines) imply that a forceful move will follow a breakout from the current indecision.

NeutralMarubozu

(1-bar pattern, stable at bar 25) A candlestick that has no upper and lower shadows.

NeutralSpinningTop

(1-bar pattern) The distance between the high and low, and the distance between the open and close, are relatively small.

ReversalDragonflyDoji

(1-bar pattern) The open and close are the same, and the low is significantly lower than the open, high, and closing prices. This pattern signifies a turning point.

ReversalGravestoneDoji

(1-bar pattern) The open, close, and low are the same, and the high is significantly higher than the open, low, and closing prices. This pattern signifies a turning point.

ReversalLongLeggedDoji

(1-bar pattern, stable at bar 25) The open and close are the same, and the range between the high and low is relatively large. This pattern often signifies a turning point.

11.3 DateRules

Members of WealthLab.Rules.DateRules

➔ DateRules functions are static members whose syntax is self-descriptive.

public static **bool** **IsLastTradingDayOfMonth**(**System.DateTime** dt)

public static **bool** **IsLastTradingDayOfQuarter**(**System.DateTime** dt)

Example ([How to run Example code?](#) ³⁷)

```
C#
// Highlight the last trading day of the month or quarter
using System;
using System.Drawing;
using WealthLab;
using WealthLab.Rules;

namespace WealthLab.Strategies
{
    public class MyStrategy : WealthScript
    {
        protected override void Execute()
        {
            DateRules dr = new DateRules();
            for(int bar = 1; bar < Bars.Count; bar++)
            {
                if( DateRules.IsLastTradingDayOfQuarter(Date[bar]) )
                    SetBackgroundColor(bar, Color.LightSalmon);
                else if( DateRules.IsLastTradingDayOfMonth(Date[bar]) )
                    SetBackgroundColor(bar, Color.LightCyan);
            }
        }
    }
}
```

11.4 DataSeriesOp

Members of WealthLab.Rules.DataSeriesOp

DataSeriesOp contains a single function, SplitReverseFactor, that populates a DataSeries *ds* with the reverse split adjustment factor. Multiply prices or divide volume by the DataSeries returned to obtain a DataSeries that indicates price (or volume) at the actual levels traded historically, i.e., not split-adjusted. This method is used in the Price (or Volume) Action rules specified for '(backtest)'.

public static **void** **SplitReverseFactor**([WealthLab.WealthScript](#) ws, [string](#) splitItem, out [WealthLab.DataSeries](#) ds)

ws An instance of the WealthScript class, i.e., *this*
 splitItem Source of split data: Use "split" for Fidelity, "Split (Yahoo! Finance)", or "Split (MSN)"
 ds The reverse split factor DataSeries

Example ([How to run Example code?](#)^{3b})

```
C#
using System;
using System.Drawing;
using WealthLab;
using WealthLab.Rules;

namespace WealthLab.Strategies
{
    public class MyStrategy : WealthScript
    {
        protected override void Execute()
        {
            DataSeries reverseAdjustment;
            DataSeriesOp.SplitReverseFactor( this, "Split (Yahoo! Finance)", out
reverseAdjustment);

            // Reverse adjust the Close and Volume DataSeries
            DataSeries closeAdjusted = Close * reverseAdjustment;
            DataSeries volAdjusted = Volume / reverseAdjustment;
            PlotSeries(PricePane, closeAdjusted, Color.Blue,
WealthLab.LineStyle.Solid, 1);
            PlotSeries(VolumePane, volAdjusted, Color.Blue,
WealthLab.LineStyle.Solid, 1);
        }
    }
}
```

11.5 FundamentalsDateRules

Members of WealthLab.Rules.FundamentalsDateRules

public **FundamentalDateRules**([WealthLab.WealthScript](#) ws, [WealthLab.Bars](#) bars, [string](#) symbol, [string](#) itemName)
 Constructor.

public **bool** **DaysSinceFundamentalItem**([int](#) bar, out [int](#) days)

public **bool** **DaysToFundamentalItem**([int](#) bar, out [int](#) days)

DaysSince/DaysTo functions indicate calendar days item (not number of bars) since/to the previous/next fundamental item specified. If the item is not found, the functions return false and days is assigned 0.

Example ([How to run Example code?](#)^{3b})

☐ C#

```
// Annotate every 5th bar with the days since/to the previous/next item specified.
using System;
using System.Drawing;
using WealthLab;
using WealthLab.Rules;

namespace WealthLab.Strategies
{
    public class MyStrategy : WealthScript
    {
        protected override void Execute()
        {
            int daysTo = 0;
            int daysSince = 0;
            Font font = new Font("Arial", 8, FontStyle.Regular);
            FundamentalDateRules fdr = new FundamentalDateRules(this, Bars,
            Bars.Symbol, "dividend");

            for(int bar = 1; bar < Bars.Count; bar++)
            {
                if( bar % 5 == 0 )
                {
                    if( fdr.DaysToFundamentalItem(bar, out daysTo) )
                        AnnotateBar(daysTo.ToString(), bar, true, Color.Black,
            Color.Transparent, font);
                    if( fdr.DaysSinceFundamentalItem(bar, out daysSince) )
                        AnnotateBar(daysSince.ToString(), bar, false, Color.Black,
            Color.Transparent, font);
                }
            }
        }
    }
}
```

11.6 FundamentalsRatio

Members of WealthLab.Rules.FundamentalsRatio

Applies to: Fidelity Wealth-Lab Pro fundamental data items, except where specified

The following methods are available in the FundamentalsRatio() static class. Being a static class, you must qualify the methods using the type name *FundamentalsRatio* instead of using an instance reference (see examples). Each method takes a WealthScript instance, which you pass as *this* in Strategy code. For external (secondary) symbols, call **SetContext()** prior to using one of these functions, and **RestoreContext()** after.

Abbreviations:

ttr = Trailing 12 Months

mrq = Most-Recent Quarter

➔ In general, functions that return a **DataSeries** apply to quarterly or annual fundamental data found in a 10-Q filing. Consequently, functions that return a **DataSeries** type do not apply to discrete corporate actions such as "split" or "dividend".

public static **System.DateTime** BarFundamentalItemDate(**WealthLab.WealthScript** _ws, **string** _name, **int** _bar)

Returns the date of the most-recent fundamental item (_name parameter) with respect to the specified _bar. Returns a System.ArgumentOutOfRangeException if the fundamental item _name is not found. Valid for use with items from any Fundamental provider.

Example ([How to run Example code?](#)³)

```
C#
// Highlight the dates
using System;
using System.Drawing;
using WealthLab;
using WealthLab.Rules;

namespace WealthLab.Strategies
{
    public class MyStrategy : WealthScript
    {
        protected override void Execute()
        {
            for(int bar = 0; bar < Bars.Count; bar++)
                if( FundamentalsRatio.BarFundamentalItemDate(this, "assets", bar)
                    == Date[bar])
                    SetBackgroundColor(bar, Color.LightSalmon);
        }
    }
}
```

public static **System.DateTime** FundamentalItemDate(**WealthLab.WealthScript** _ws, **string** _name)
 public static **System.DateTime** FundamentalItemDate(**WealthLab.WealthScript** _ws, **string** _name, **int** _cnt)

Returns the date of the first fundamental item specified by the _name parameter or the date of the item identified by _cnt in the FundamentalDataItems list.

Returns a System.ArgumentOutOfRangeException if the fundamental item _name is not found. Valid for use with items from any Fundamental provider.

Example (How to run Example code?)

 C#

```
using System;
using System.Drawing;
using System.Collections.Generic;
using WealthLab;
using WealthLab.Rules;

namespace WealthLab.Strategies
{
    public class MyStrategy : WealthScript
    {
        protected override void Execute()
        {
            const string item = "assets";
            ClearDebug();
            PrintDebug("First date: " + FundamentalsRatio.FundamentalItemDate(this,
item));
            PrintDebug("All dates:");

            IList<FundamentalItem> fList = FundamentalDataItems(item);
            for(int j = 0; j < fList.Count; j++)
                PrintDebug(FundamentalsRatio.FundamentalItemDate(this, item, j));
        }
    }
}
```

public static [WealthLab.DataSeries](#) ConsecutiveGrowthSeries([WealthLab.WealthScript](#) ws, [string](#) _name, [int](#) _periods, [double](#) _target, [bool](#) _above, [bool](#) _annual)

Returns a series with the number of consecutive quarters whose growth is above the specified _target growth rate percentage, which can be a positive or negative number. Growth is relative to the the _periods parameter, which represents quarter over quarter growth (*annual* = false) or fiscal year over fiscal year (*annual* = true). Recommended Fundamentals items for the _name parameter are "cash" (Cash Flow), "net income", and "sales turnover".

Example (How to run Example code?)

 C#

```
using System;
using System.Drawing;
using WealthLab;
using WealthLab.Rules;

namespace WealthLab.Strategies
{
    public class MyStrategy : WealthScript
    {
        protected override void Execute()
        {
            DataSeries dsCPG = FundamentalsRatio.ConsecutiveGrowthSeries( this, "sales
turnover", 1, 12, true, true );
            dsCPG.Description = "Consecutive years of Sales Growth above 12%";
            ChartPane paneCPG = CreatePane(50, true, false);
            PlotSeries(paneCPG, dsCPG, Color.Purple, LineStyle.Histogram, 50);
        }
    }
}
```

public static [WealthLab.DataSeries](#) GrowthRateSeries([WealthLab.WealthScript](#) ws, [string](#) itemName, [int](#) periods, [bool](#) annual)

Returns the growth rate expressed as percentage growth (e.g. 10 = 10%) with respect to the specified number (*periods*) of years ago (*annual=true*) or quarters ago (*annual=false*). Recommended values for *itemName*: "cash" (Cash Flow), "net income", and "sales turnover".

Example ([How to run Example code?](#)³)

```

C#
using System;
using System.Drawing;
using WealthLab;
using WealthLab.Rules;

namespace WealthLab.Strategies
{
    public class MyStrategy : WealthScript
    {
        protected override void Execute()
        {
            DataSeries dsRatio = FundamentalsRatio.GrowthRateSeries( this, "sales
turnover", 1, true );
            dsRatio.Description = "Sales Growth Rate % over 1 year";
            ChartPane paneRatio = CreatePane(50, true, false);
            PlotSeries(paneRatio, dsRatio, Color.Purple, LineStyle.Histogram, 50);
        }
    }
}

```

public static [string](#) **ConvertName**([string](#) name)

The Strategy Builder uses this functions to convert parameters to a corresponding Fundamental item string. For example, passing "Shares" to ConvertName returns "common shares outstanding". If not found, the *name* passed is returned.

public static [WealthLab.DataSeries](#) **DilutedSharesSeries**([WealthLab.WealthScript](#) ws)

Number of shares (millions) used to calculate diluted eps on an annual basis, adjusted for splits.

public static [WealthLab.DataSeries](#) **ShareSeries**([WealthLab.WealthScript](#) ws)

Common shares outstanding (millions) on a ttm basis, adjusted for splits.

public static [WealthLab.DataSeries](#) **MarketCapSeries**([WealthLab.WealthScript](#) ws)

Market Capitalization (millions of dollars) for the trailing 12 months (ttm) is calculated by multiplying the company's stock price by the number of common shares outstanding. For example: 12,345.67 = 12 billion, 345 million, 67 thousand dollars

public static [WealthLab.DataSeries](#) **ReturnOnCapitalSeries**([WealthLab.WealthScript](#) ws)

Return on Capital (percentage) for the trailing 12 months (ttm). Inventories, Receivables, Liabilities and Assets are aggregated and averaged for the previous 4 quarters.

public static [WealthLab.DataSeries](#) **SalesPerEmployeeSeries**([WealthLab.WealthScript](#) ws)

Sales per employee is an aggregate measure (ttm) in thousands of dollars.

Example ([How to run Example code?](#)³)

```

C#
using System;
using System.Drawing;
using WealthLab;
using WealthLab.Rules;

namespace WealthLab.Strategies
{
    public class MyStrategy : WealthScript
    {

```

```

        protected override void Execute()
        {
            DataSeries spe = FundamentalsRatio.SalesPerEmployeeSeries( this );
            spe.Description = "Sales per Employee (ttm)";
            ChartPane spePane = CreatePane(50, true, true);
            PlotSeries(spePane, spe, Color.Purple, LineStyle.Histogram, 50);
        }
    }
}

```

public static [WealthLab.DataSeries](#) EnterpriseValueSeries([WealthLab.WealthScript](#) ws)

➔ Enterprise Value is not calculated correctly due to the use of Cash Flow *cash* instead of Balance Sheet *cash*, the latter of which is currently not an available fundamental data item.

public static [WealthLab.DataSeries](#) FundamentalSeries([WealthLab.WealthScript](#) ws, [string](#) itemName, [bool](#) annual)

Returns the Fundamental DataSeries for the *itemName* synchronized with the chart. Pass *false* to *annual* for mrq (raw) data values. Pass *true* to *annual* for annualized values. For balance sheet items like "assets", "liabilities", "long term debt", etc. the annualized value is the average of the reports for the trailing twelve months (ttm). Income statement items like "net income", "interest expense", "sales turnover", etc. are summed for the ttm value.

Example ([How to run Example code?](#) ³)

```

C#
using System;
using System.Drawing;
using WealthLab;
using WealthLab.Rules;

namespace WealthLab.Strategies
{
    public class MyStrategy : WealthScript
    {
        protected override void Execute()
        {
            // A Balance Sheet item
            DataSeries assets_mrqr = FundamentalsRatio.FundamentalSeries(this,
"assets", false);
            DataSeries assets_ttm = FundamentalsRatio.FundamentalSeries(this,
"assets", true);
            assets_mrqr.Description = "Assets (mrq)";
            assets_ttm.Description = "Assets (ttm)";
            ChartPane assetsPane = CreatePane(50, true, true);
            PlotSeries(assetsPane, assets_ttm, Color.Green, LineStyle.Solid, 2);

            PlotSeries(assetsPane, assets_mrqr, Color.Purple, LineStyle.Histogram, 50);

            // An Income Statement item
            DataSeries sales_mrqr = FundamentalsRatio.FundamentalSeries(this, "net
income", false);
            DataSeries sales_ttm = FundamentalsRatio.FundamentalSeries(this, "net
income", true);
            sales_mrqr.Description = "Sales (mrq)";
            sales_ttm.Description = "Sales (ttm)";
            ChartPane salesPane = CreatePane(50, true, true);

```

```

        PlotSeries(salesPane, sales_ttm, Color.Green, LineStyle.Solid, 2);

        PlotSeries(salesPane, sales_mrq, Color.Purple, LineStyle.Histogram, 50);
    }
}

```

public static [WealthLab.DataSeries](#) **PerEmployeeSeries**([WealthLab.WealthScript](#) ws, [string](#) itemName)
Returns a ttm *itemName*/Employee ratio series.

public static [WealthLab.DataSeries](#) **PerShareSeries**([WealthLab.WealthScript](#) ws, [string](#) itemName, [bool](#) annual)
A Fundamental Item per Share for the trailing twelve months (*annual = true*) or most-recent quarter (*annual = false*). Possible *itemName* values: "net income", "assets", "sales turnover", "stockholder equity", "total inventories", "total receivables". Enter the value in dollars per share; e.g. 20 = \$20/share

public static [WealthLab.DataSeries](#) **PriceToRatioSeries**([WealthLab.WealthScript](#) ws, [string](#) itemName)
Price/Fundamental ratio annualized for the trailing 12 months (ttm). Valid fundamentals: "cash" (Cash Flow), "net income", and "sales turnover".

public static [WealthLab.DataSeries](#) **RatioSeries**([WealthLab.WealthScript](#) ws, [string](#) itemName1, [string](#) itemName2, [bool](#) annual)
Returns a the ttm (*annual = true*) or mrq (*annual = false*) ratio series between *itemName1* and *itemName2*.

public static [WealthLab.DataSeries](#) **ReturnOnSeries**([WealthLab.WealthScript](#) ws, [string](#) itemName, [bool](#) annual)
itemName: "stockholder equity" or "assets" returns a Return on equity or assets series, respectively. Pass true to the *annual* parameter for a trailing twelve months (ttm) or *false* for most-recent quarter (mrq) return-on data.

[Example \(How to run Example code?\)](#)³⁾

```

C#
using System;
using System.Drawing;
using WealthLab;
using WealthLab.Rules;

namespace WealthLab.Strategies
{
    public class MyStrategy : WealthScript
    {
        protected override void Execute()
        {
            DataSeries dsRatio = FundamentalsRatio.ReturnOnSeries( this, "stockholder
equity", false );
            dsRatio.Description = "Return on Equity (mrq)";
            ChartPane paneRatio = CreatePane(50, true, false);
            PlotSeries(paneRatio, dsRatio, Color.Purple, LineStyle.Histogram, 50);
        }
    }
}

```

public static [WealthLab.DataSeries](#) **YieldSeries**([WealthLab.WealthScript](#) ws, [string](#) itemName)
itemName: "cash dividends" or "net income" returns a dividend or earnings yield series, respectively.

[Example \(How to run Example code?\)](#)³⁾

```

C#
using System;
using System.Drawing;

```

```
using WealthLab;
using WealthLab.Rules;

namespace WealthLab.Strategies
{
    public class MyStrategy : WealthScript
    {
        protected override void Execute()
        {
            DataSeries divYield = FundamentalsRatio.YieldSeries(this, "cash
dividends");
            divYield.Description = "Dividend Yield (ttm)";
            ChartPane paneRatio = CreatePane(50, true, false);
            PlotSeries(paneRatio, divYield, Color.Green, LineStyle.Histogram, 50);
        }
    }
}
```

11.7 OptionExpiryDate

Members of `WealthLab.Rules.OptionExpiryDate`

The *OptionExpiryDate* function names and syntax are self-descriptive.

```
public OptionExpiryDate(WealthLab.Bars bars)
    OptionExpiryDate Class constructor.
```

```
public int DaysSinceOptionExpiryDate(int bar)
```

```
public int DaysSinceTripleWitchingDate(int bar)
```

```
public int DaysToOptionExpiryDate(int bar)
```

```
public int DaysToTripleWitchingDate(int bar)
```

DaysSince and **DaysTo** functions indicate calendar days since/to expiry dates (not number of bars).

```
public System.DateTime NextOptionExpiryDate(int year, int month)
```

```
public System.DateTime NextTripleWitchingDate(int year, int month)
```

Example ([How to run Example code?](#)³)

▣ C#

```
using System;
using WealthLab;
using WealthLab.Rules;

namespace WealthLab.Strategies
{
    public class MyStrategy : WealthScript
    {
        protected override void Execute()
        {
            OptionExpiryDate oed = new OptionExpiryDate(Bars);
            for (int y = 2008; y <= 2010; y++)
                for (int m = 1; m <= 12; m++)
                {
                    DateTime dt = oed.NextOptionExpiryDate(y, m);
                    PrintDebug(dt.ToShortDateString());
                }
        }
    }
}
```

11.8 NewHighLow

Members of `WealthLab.Rules.NewHighLow`

Applies to: Fidelity Wealth-Lab Pro

The `NewHighLow` class simplifies accessing a market's new highs-to-lows ratio as well as the convergence and divergence set of Market Sentiment rules. The Market Sentiment data for NYSE, Nasdaq, and AMEX are found in the [Market Sentiment DataSet](#)^[79].

```
public NewHighLow(WealthLab.WealthScript ws, string exchange, string index, int period, int smaPeriod)
```

`ws` An instance of the `WealthScript` class, i.e., *this*
`exchange` NYSE, Nasdaq, or AMEX
`e`
`index` .NYA (NYSE Composite), .IXIC (Nasdaq Composite), or .XAX (AMEX Composite)
`period` Momentum lookback period
`smaPeriod` Moving Average period. The average of the Momentum series is used to detect convergence/divergence.

```
public string Exchange { set; }  
The Exchange ("NYSE", "Nasdaq", or "AMEX") determines which New High/Low data are used from the Market Sentiment DataSet[79].
```

```
public string Index { set; }  
The Composite Index symbol (".NYA", ".IXIC", or ".XAX") used for the convergence/divergence detection.
```

```
public int Period { set; }  
The momentum lookback period.
```

```
public int SMAPeriod { set; }  
The simple moving average period used to average the momentum series.
```

```
public double Threshold { set; }  
The plotNewHighLowRatio method plots a dotted horizontal line at the Threshold value. The value is not used in any calculations.
```

The following group of functions return the series indicated by the function name. The source of the ratio and index are specified when instantiating the class or after changing one of the properties above (Exchange, Index, etc.).

```
public WealthLab.DataSeries MomentumIndexSeries { get; }  
public WealthLab.DataSeries MomentumRatioSeries { get; }  
public WealthLab.DataSeries MomentumSMAIndexSeries { get; }  
public WealthLab.DataSeries MomentumSMARatioSeries { get; }  
public WealthLab.DataSeries NewHighLowRatioSeries { get; }
```

The following group of functions create the series indicated by the function name. It's not required to explicitly call these functions. Instead, they are implicitly created by the previous set of functions that return a `Wealth-Lab DataSeries`.

```
public bool createMomentumIndexSeries()  
public bool createMomentumRatioSeries()  
public bool createMomentumSMAIndexSeries()  
public bool createMomentumSMARatioSeries()
```

```
public bool createNewHighLowRatioSeries()
public bool createNewHighSeries()
public bool createNewLowSeries()
```

The plot functions plot the indicated DataSeries, automatically creating a ChartPane, if required.

```
public void plotIndex()
public void plotIndex(WealthLab.ChartPane paneIndex)
public void plotIndex(int height, bool abovePrice, bool showGrid)

public void plotMomentumRatioToIndex()
public void plotMomentumRatioToIndex(WealthLab.ChartPane paneMomentum)
public void plotMomentumRatioToIndex(int height, bool abovePrice, bool showGrid)
```

```
public void plotMomentumSMA()
public void plotMomentumSMA(WealthLab.ChartPane paneMomentumSMA)
public void plotMomentumSMA(int height, bool abovePrice, bool showGrid)
```


```
public void plotNewHighLowRatio()
public void plotNewHighLowRatio(WealthLab.ChartPane paneRatio)
public void plotNewHighLowRatio(int height, bool abovePrice, bool showGrid)
```

```
public void plotNewHighs()
public void plotNewHighs(WealthLab.ChartPane paneNH)
public void plotNewHighs(int height, bool abovePrice, bool showGrid)
```

```
public void plotNewLows()
public void plotNewLows(WealthLab.ChartPane paneNL)
public void plotNewLows(int height, bool abovePrice, bool showGrid)
```

Simply specify the exchange of the desired Market Sentiment data to easily plot the Highs, Lows, and ratio series as in the following example.

[Example](#) ([How to run Example code?](#)³⁴)

```
 C#
using System;
using WealthLab;
using WealthLab.Indicators;
using WealthLab.Rules;

namespace WealthLab.Strategies
{
    public class MyStrategy : WealthScript
    {
        protected override void Execute()
        {
            NewHighLow nhl = new NewHighLow(this, "Nasdaq", "", 0, 0);
            nhl.Threshold = 5;
            DataSeries dsNewHighLowRatio = nhl.NewHighLowRatioSeries;
            nhl.plotNewLows();
            nhl.plotNewHighs();
            nhl.plotNewHighLowRatio();
        }
    }
}
```

In the following example, the Strategy buys when the NYSE High/Low ratio ≥ 10 and sells on bearish convergence with the NYSE Composite Index.

[Example \(How to run Example code?\)](#)³⁾

```

C#
using System;
using WealthLab;
using WealthLab.Rules;

namespace WealthLab.Strategies
{
    public class MyStrategy : WealthScript
    {
        protected override void Execute()
        {
            NewHighLow nhl = new NewHighLow(this, "NYSE", ".NYA", 5, 10 );
            nhl.Threshold = 10;
            DataSeries dsNewHighLowRatio = nhl.NewHighLowRatioSeries;
            DataSeries dsMomentumSMAIndex = nhl.MomentumSMAIndexSeries;
            DataSeries dsMomentumSMARatio = nhl.MomentumSMARatioSeries;

            nhl.plotNewLows();
            nhl.plotNewHighs();
            nhl.plotNewHighLowRatio();
            nhl.plotMomentumSMA();
            nhl.plotIndex();

            for(int bar = 11; bar < Bars.Count; bar++)
            {
                if (IsLastPositionActive)
                {
                    Position p = LastPosition;
                    if(( dsMomentumSMAIndex[bar] < 0 )&&
( dsMomentumSMARatio[bar] < 0 ))
                        SellAtMarket(bar + 1, p);
                }
                else
                {
                    if( dsNewHighLowRatio[bar] >= 10 )
                        BuyAtMarket(bar + 1);
                }
            }
        }
    }
}

```

12 Techniques

This chapter will be dedicated to widely-used testing techniques and how to accomplish them with WealthScript. Check back for more content with each successive Wealth-Lab Pro software upgrade.

12.1 Creating a Screener

A Screen, or Filter, is the simplest form of a Strategy that identifies one or more symbols that currently meet a specified criteria; that is, using the most recently-available completed bar. You can do some extremely sophisticated screening with Wealth-Lab, much more elaborate than most other software packages. The key is writing a "Screening Script" and then running it in a Strategy Window in Multi-Symbol Backtest mode.

➔ Generally, we think of screens as an end-of-day activity, but there's no reason you can't use Wealth-Lab's Strategy Monitor to generate Alerts for DataSets screening using intraday data.

The Mechanics of a Stock Screen

To build a Screen we'll take advantage of the following fact: Strategies issue an [Alert](#)^[68] whenever an order is placed for the trading day after the last bar on the chart. In WealthScript lingo, this idea can be expressed as:

```
BuyAtMarket( Bars.Count, 'Alert' );
```

The bottom line is that if you want the symbol to appear in the screen's results, issue a **BuyAtMarket** at bar number **Bars.Count** ([How to count bars in a chart](#)^[4]).

An Example Screen

The following example will implement a screen based on the following concepts:

- Prices are above the 200 day simple moving average
- Avoid price shocks, largest 5 day loss within 200 days < 20%
- 20 day average Volume must be above 100,000
- Avoid stocks that are short-term overbought, 10 day RSI must be below 50

[Example](#) ([How to run Example code?](#)^[3])

Save this as a new Strategy called "Example Screen" in a new folder called "Screens", where you can store all of your future screening scripts.

```
protected override void Execute()
{
    int bar = Bars.Count - 1; // the last chart bar number
    if (bar < 199) return;     // chart must have at least 200 bars to continue
    if (
        Close[bar] > SMA.Value(bar, Close, 200)
        && Lowest.Value(bar, ROC.Series(Close, 5), 200) > -20
        && SMA.Value(bar, Volume, 20) > 100000
        && RSI.Series(Close, 10)[bar] < 50
    )
        BuyAtMarket(bar + 1);
}
```

➔ The example above uses familiar technical indicators, but you can apply [fundamental criteria](#)^[74] as well.

Running the Screen

To run the screen, open it in a Strategy Window. Click the DataSet that you want to screen. Before clicking "Go" or the "Backtest" button, select the Data Range dialog. The screen uses at most 200 bars of data (a 200 bar **SMA**) so we can safely change the number of bars to load to 200. By reducing the number of bars, the screening process, or

scan, will run much more quickly. Also, for the fastest screens on large DataSets, it's best to use the Data Manager to update data first and to *disable* [File > Update Data on Demand](#).

Click the "Go" or the "Backtest" buttons to start the screening process. The results of the scan (if any) will appear in the Alerts view. You can also examine the individual charts by double clicking on a symbol. For a more automated daily scanning operation, add screening Strategies to the Strategy Monitor. See the User Guide for details.

Ranking the Results

To rank the results of the screen, specify a [numerical] *signalName* for the **BuyAtMarket** signal. In the previous example, make the change:

```
// WAS:
    BuyAtMarket(bar + 1);

// CHANGE TO:
    BuyAtMarket(bar + 1, RSI.Series(Close, 10)[bar].ToString("0.000"));
```

The *signalName* shows us the 10-period **RSI** of all of the Alerts generated. By clicking the "Signal Name" column header, it's easy to order the Alerts so that you can pick the trades with the highest or lowest value.

Related: [How to Convert Strategy to a Screen](#)

12.2 Sizing with 100% of Equity

See the Wealth-Lab Pro User Guide topic "100% of Equity Sizing" in the Strategy Window chapter.

12.3 Symbol Rotation

Symbol Rotation strategies are those that seek to remain highly exposed to the market (fully or near-fully invested) by choosing the best candidates from a given DataSet of symbols. Rotation strategies contain all code necessary to create indicators and trades for each symbol in the current DataSet. Run these strategies on a single symbol only - not in Multi-Backtest mode.

See the "RSI Rotation" and "Dogs of the Dow" Strategies included with the installation.

Note for Wealth-Lab Pro Version 4 legacy customers:

Multi-Symbol Backtest mode does not kick out of the implied symbol loop even if trades are created after a call to `SetContext()`.

12.4 Most Probable AtStop

To execute trading signals in backtesting in the most-probable sequence based on the relationship of the trade bar's opening price to the high and low of that bar, use the MostProbableAtStop function below in place of BuyAt/ShortAtStop trading signals.

Example ([How to run Example code?](#)^{3b})

■ C#

```
public Position MostProbableAtStop(int TradeBar, double BuyStopPrice,
double ShortStopPrice)
{
    if (TradeBar > Bars.Count - 1)           // Alert both, return null
    {
        ShortAtStop(TradeBar, ShortStopPrice);
        BuyAtStop(TradeBar, BuyStopPrice);
        return null;
    }
    if (Open[TradeBar] - Low[TradeBar] < High[TradeBar] - Open[TradeBar])
    {
        if (ShortAtStop(TradeBar, ShortStopPrice) != null)
            return LastPosition;
        else
            return BuyAtStop( TradeBar, BuyStopPrice);
    }
    else
    {
        if (BuyAtStop(TradeBar, BuyStopPrice) != null)
            return LastPosition;
        else
            return ShortAtStop(TradeBar, ShortStopPrice);
    }
}
```

12.5 Access ad-hoc Data from the Internet

A number of sites (like www.cftc.gov and www.cboe.com) provide all sorts of helpful data in plain ASCII files. To utilize it, you can go the obvious way: download, save the file on your local hard disk and create a static DataSet with the help of the ASCII adapter. But when new data becomes available, you have to download the file again.

This example illustrates how, using the power of .NET classes, you could access remote sources of information on-the-fly in your trading strategy. After the raw data for the total Put/Call Ratio is downloaded from the CBOE website and becomes available as a DataSeries, the strategy is able to make its trading decisions on Bollinger Bands derived from the Put/Call Ratio.

The code also demonstrates how to cache the resulting Bars object in the global memory (GOP) to speed up Multi-Symbol Backtests. Without caching, the strategy would have to generate redundant data requests to the server on each symbol in a DataSet. As soon the data is in place, access to the Internet is no longer required.

[Example \(How to run Example code?\)](#)^{3b}

See code in the Wealth-Lab Wiki: www2.wealth-lab.com/WL5Wiki/kbAccessInternet.ashx

12.6 Utility Scripts

Utility scripts are small programs that usually perform some supportive function for Strategy testing or trading - like checking reasonableness of historic data or perhaps exporting data to ASCII format for off-line analysis.

12.6.1 Data Checking

See the "Bad History Data Check" Strategy in the Utilities folder.

12.6.2 DataSet Symbol Details

While you can obtain the Symbol Details of a DataSet using the Data Manager, you may find it useful to access the information in a script. The script stores a Detail object in a List and uses a delegate sort method to sort the list by the date of the last bar; oldest dates are output first.

Example ([How to run Example code?](#)^{3b})

```
using System;
using System.Text;
using System.Collections.Generic;
using WealthLab;

namespace WealthLab.Strategies
{
    public class Detail
    {
        public string sym;
        public DateTime lastdt;
        public DateTime firstdt;
        public int nbars;
    }

    public class SymbolDetails : WealthScript
    {
        protected override void Execute()
        {
            ClearDebug();
            List<Detail> details = new List<Detail>();
            for(int n = 0; n < DataSetSymbols.Count; n++)
            {
                Detail detail = new Detail();
                detail.sym = DataSetSymbols[n];
                SetContext( detail.sym, false );
                detail.lastdt = Bars.Date[Bars.Count - 1];
                detail.firstdt = Bars.Date[0];
                detail.nbars = Bars.Count;
                details.Add(detail);
            }
            RestoreContext();

            // Sort by date (earliest dates printed first)
            details.Sort(delegate(Detail d1, Detail d2) { return d1.lastdt.CompareTo(d2.lastdt); });
            details.ForEach(delegate(Detail d) { PrintDebug(d.sym + "\t" + d.nbars + " "); });
        }
    }
}
```


13 APIs

Wealth-Lab Pro Version 6 makes extensive use of *components*. In fact, all installed data adapters, chart styles, performance visualizers, etc. are component-based. Likewise, developers can integrate their own components seamlessly. Documentation and examples for the APIs are available at Fidelity.com: http://personal.fidelity.com/products/trading/Trading_Platforms_Tools/wlp-under-the-hood.shtml

Index

- 1 -

100% of Equity Sizing 126

- A -

add (DataSeries) 12
 AllPositions 66
 Arms Index 103
 Average Price 12
 AveragePriceC 12

- B -

bar 4
 bar + n, n > 1 54
 Black-Scholes 71

- C -

Candlesticks
 Bearish 107
 Bullish 105
 Neutral and Reversal 108
 syntax 104
 Chart Style
 Kagi 96
 Line Break 98
 Point and Figure 98
 Renko 97
 Chart Styles
 Trending 94
 ChartPane
 create 23
 class-scope variables 45
 click to topic 76
 Compile Strategies 45
 Custom Indicators 38

- D -

Data

Internet 129
 Market Sentiment 79
 DataSeries
 Abs 13
 add, subtract, multiply, divide 12
 delay 12
 offset 12
 shift 12
 DateTime 8
 DayOfWeek 8
 divide (DataSeries) 12
 division-by-zero (DataSeries) 12

- E -

Example Code
 instructions 3
 External Symbols 18

- F -

Filter 124
 Floor Trader Pivots 87
 Fundamental Data
 Deletion 74
 Functions 74
 Refresh 74
 Retrieval 74

- G -

Good Til Canceled 53
 GTC 53

- H -

Heikin-Ashi 24
 Hour 8
 How to
 Run Example Code 3
 How to (Bars)
 access OHLC/V values 6
 access scale and interval 8
 access symbol, company name 10
 access tick, margin, point 10
 change OHLC/V series 6
 Count total ~ in chart 4

How to (Bars)

- date and time 8
- Find bar at a specified date 5
- Find first intraday bar number 4
- plot OHLC/V series 6

How to (ChartPane)

- create a custom pane 23
- hide Volume 23

How to (DataSeries)

- access secondary symbol data 18
- add, subtract DataSeries 12
- average price 12
- change data 16
- create series filled with zeroes 16
- delay a DataSeries 12
- multiply, divide DataSeries 12
- offset a DataSeries 12
- Return an absolute value ~ 13
- typical price 12
- value of data on a specific bar 15

How to (Draw Objects)

- draw Speed-Resistance Lines 31
- draw/extend a trendline 30
- highlight an area with a polygon 31

How to (Indicators)

- access an Indicator Series 36
- calculate a single value 37
- Create indicator from a secondary symbols' raw DataSeries 21
- Elder Force 40
- MACD with custom periods 39
- NRTR_WATR Indicator 40
- secondary symbol indicators 42
- Semi-formal custom indicator 38
- Spread 40
- Weighted Close 39

How to (Plot Text)

- control a DataSeries plot label 28
- write vertical text 28

How to (Plot)

- compare with Benchmark Symbol 24
- External Symbol 24
- Plot a DataSeries 26
- Plot Stops 53
- Secondary Symbol 24
- Strategy metrics 26
- Synthetic Symbol 24
- volume above price 26

How to (Strategy)

- Access ActivePositions 66
- Add Strategy Parameters 62
- AtClose Alerts 69
- Convert to Screen 50
- Create a trade 48
- Create a trade on specified date 49
- Create Screen 124
- Good Til Canceled 53
- Save Alerts to File 68
- Test for Open Position 51
- Time-based exit 54
- Trigger Limit/Stop Orders 52
- Use Position.AllPositions 66

How to (Time Frame)

- Access Higher Intraday Scale 85
- Access Monthly from Daily 92
- Access Weekly from Daily 91
- Daily from Intraday 87
- Determine Chart Scale 84
- Overlay Daily on Intraday 88

How to Use ActivePositions 65

<http://www2.wealth-lab.com/WL5Wiki/kbAccessIntern>
et.ashx 129

- | -

Indicators

- Custom 38
- Series method 36
- Stability 43
- Syntax 34
- Value method 37

Internet 129

interval 8

Intraday AtClose 70

- L -

Limitation

- Strategy Code 46

- M -

main loop 50

margin 10

Market Sentiment 79

Market Sentiment 79
 New Highs, Lows 120
 Market Sentiment Data 79
 Minute 8
 MP Strategies 65
 multiply (DataSeries) 12

- N -

negative prices (futures) 6

- O -

Option Expiry Date 119
 Options 71
 Overlay Daily Candlesticks 88

- P -

Parameters
 Strategy 62
 Peeking
 Don't Access Future Data 57
 Order of Trading Signals 57
 Position Active Status 58
 Trade on bar + 1 57
 Valid ~ 59
 What is ~? 57
 PlotSeries 26
 PlotSeriesDualFillBand 26
 PlotSeriesFillBand 26
 PlotSeriesOscillator. 26
 PlotSymbol 24
 PlotSyntheticSymbol 24
 PnF 98
 PnFTrendLine 101
 Point and Figure 98
 TPnF Class 99
 TrendLine Class 100
 point value 10
 Positions
 List 51
 Postdictive errors 57
 Programming Trading Strategies 45

- S -

scale 8
 Screen 124
 seed data 43
 Series method 36
 shift operator (DataSeries) 12
 SP Strategies 51
 spike detection 16
 Stability of Indicators 43
 Straps 62
 Strategies
 Multiple-Position (MP) 65
 Single-Position (SP) 51
 Symbol Rotation 127
 Strategy
 Parameters 62
 Strategy Class 45
 subtract (DataSeries) 12
 Survivorship bias 59
 Symbol
 External 18
 Plot a secondary symbol 24
 Secondary 18
 Symbol Rotation Strategies 127
 Synchronization
 Option 3 87
 Secondary Symbols 19
 Synthetic
 Option contracts 71

- T -

Template
 Multiple-Position (MP) Strategy 65
 Multi-Position (MP) Strategy 65
 Single-Position (SP) Strategy 51
 tick 10
 trading loop 50
 Trading Signals 48
 TrendLine
 PnF 101
 TrendLine Class 100
 Typical Price 12

- V -

Value method 37

Variables

class-scope 45

Volume

hide 23

- W -

Wealth-Lab.com Knowledge Base 132

- Y -

y-axis

scale control 27