Alert Object

The Alert object represents a trade that needs to be placed on the following bar.

Remarks

- To access the current symbol (or symbol for the current context after calling SetContext), use the Symbol property of the Bars Object.

Account Property

string Account

Returns an Account string, which contains the account of generated alert.

AlertDate Property

DateTime AlertDate

Returns an AlertDate structure, which contains the date of generated alert.

Example

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

namespace WealthLab.Strategies
{
    public class MyStrategy : WealthScript
    {
        // Writes trades into CSV file
        public void AlertsToFile()
        {
            StreamWriter atf;

            string path = Path.Combine( Environment.GetFolderPath(Environment.SpecialFolder.Personal),
                "Alerts(" + StrategyName + ").csv" );

            if( Alerts.Count > 0 )
            {
                string s = ",";
                atf = new StreamWriter( path, true );
                for( int n = 0; n < Alerts.Count; n++ )
                {
                    WealthLab.Alert a = Alerts[n];
                    atf.Write( a.Bars.Symbol + s
                        + a.AlertDate.ToShortDateString() + s
                        + a.AlertDate.ToShortTimeString() + s
                        + a.AlertType + s
                        + a.Price
                        + "\r\n" );  // etc.
                }
                atf.Close();
            }
        }

        protected override void Execute()
        {
            // Alert generating code - Example
            for(int bar = 3; bar < Bars.Count; bar++)
            {
                if (!IsLastPositionActive)
                {
                    // Two consecutive lower closes
                    if( ( Bars.Close[bar] < Bars.Close[bar-1] ) &
                        ( Bars.Close[bar-1] < Bars.Close[bar-2] ) )
                        BuyAtStop( bar+1, Close[bar]*1.03 );
                }
                if (IsLastPositionActive)
                {
                    SellAtLimit( bar+1, LastPosition, Close[bar]*1.10 );
                }
            }
            AlertsToFile();
        }
    }
}
```

AlertType Property

TradeType AlertType

Returns a TradeType enumerated list, which contains the type of generated alert. Possible values are:

- Buy
- Cover
- Sell
- Short

Example

```
protected override void Execute(){
    // Alert generating code
    for(int bar = 3; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
            SellAtLimit( bar+1, LastPosition, Close[bar]*1.01 );
        else
            BuyAtLimit( bar+1, Close[bar]*0.97 );
    }

    // Show the alert type
```

```
    if( Alerts.Count > 0 )
    {
        for( int i = 0; i < Alerts.Count; i++ )
        {
            WealthLab.Alert a = Alerts[i];
                PrintDebug( "Alert " + ( i+1 ) + " type: " + a.AlertType );
        }
    }
}
```

BarInterval Property

int BarInterval

Returns the intraday bar interval of the generated alert. For example, BarInterval will return 5 for 5-minute bars.

Remarks

- For non-intraday scales, BarInterval returns 0.

Example

```
protected override void Execute(){
    // Alert generating code
    for(int bar = 1; bar < Bars.Count; bar++)
    {
        if (!IsLastPositionActive)
        {
              BuyAtMarket( bar+1 );
        }
        if (IsLastPositionActive)
        {
            SellAtMarket( bar+1, LastPosition );
        }
    }

    // Returns the bar interval of an alert
    if( Alerts.Count > 0 )
    {
        for( int i = 0; i < Alerts.Count; i++ )
        {
            WealthLab.Alert a = Alerts[i];
            if( a.BarInterval > 0 )
            PrintDebug( "BarInterval: " + a.BarInterval + " tick, second or minute " ); else
            PrintDebug( "BarInterval is Daily or greater" );
        }
    }
}
```

BasisPrice Property

double BasisPrice

Returns a BasisPrice number, which contains the basis price of generated alert which is going to establish a position.

Example

```
protected override void Execute(){
    // Alert generating code
    for(int bar = 3; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
            SellAtLimit( bar+1, LastPosition, Close[bar]*1.01 );
         else
            BuyAtLimit( bar+1, Close[bar]*0.97 );
    }

    // Show the basis price of an alert
    if( Alerts.Count > 0 )
    {
        for( int i = 0; i < Alerts.Count; i++ )
        {
            WealthLab.Alert a = Alerts[i];
            if( ( a.AlertType != TradeType.Sell ) &
                ( a.AlertType != TradeType.Cover ) )
            PrintDebug( "Alert " + ( i+1 ) + " basis price: " + a.BasisPrice );
        }
    }
}
```

OrderType Property

OrderType OrderType

Returns an OrderType enumerated list, which contains the order type of generated alert. Possible values are:

- AtClose
- Limit
- Market
- Stop

Example

```
protected override void Execute(){
    // Alert generating code

    Random rnd = new Random();

    for(int bar = 1; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            // Exit trade after 3 days
            if ( bar+1 - LastPosition.EntryBar >= 3 )
                SellAtMarket( bar+1, LastPosition, "Time-based" );
        }
```

```
        else
        {
            // Random factor
            if ( rnd.Next(0,10) < 3 )
                BuyAtStop( bar+1, High[bar]*1.03, "Buy strength" ); else
                BuyAtLimit( bar+1, Low[bar]*0.97, "Buy weakness" );
        }
    }

    // Show the alert order type
    if( Alerts.Count > 0 )
    {
        for( int i = 0; i < Alerts.Count; i++ )
        {
            WealthLab.Alert a = Alerts[i];
            PrintDebug( "Alert " + ( i+1 ) + " order type:  " + a.OrderType );
        }
    }
}
```

Position Property

Position Position

Contains the Position object that corresponds to a Sell or Cover alert.

PositionType Property

PositionType PositionType

Returns a PositionType enumerated list, which contains the position type of generated alert. Possible values are:

- Long
- Short

Example

```
protected override void Execute(){
    // Alert generating code

    Random rnd = new Random();

    for(int bar = 1; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            // Exit trade after 3 days
            if ( bar+1 - LastPosition.EntryBar >= 3 )
                ExitAtMarket( bar+1, LastPosition, "3 days" );
        }
        else
        {
            if ( rnd.Next(0,1) == 1 )
                BuyAtMarket( bar+1 ); else
                ShortAtMarket( bar+1 );
        }
    }

    // Show the alert position type
    if( Alerts.Count > 0 )
    {
        for( int i = 0; i < Alerts.Count; i++ )
        {
            WealthLab.Alert a = Alerts[i];
                PrintDebug( "Alert " + ( i+1 ) + " is to go " + a.PositionType );
        }
    }
}
```

Price Property

double Price

Returns a Price number, which contains the price of generated alert, except for Market and AtClose orders which return 0.

Example

```
protected override void Execute(){
    // Alert generating code
    for(int bar = 4; bar < Bars.Count; bar++)
    {
        if (!IsLastPositionActive)
        {
            // Three consecutive lower closes
            if( ( Bars.Close[bar] < Bars.Close[bar-1] ) &
                ( Bars.Close[bar-1] < Bars.Close[bar-2] ) &
                ( Bars.Close[bar-2] < Bars.Close[bar-3] ) )
                BuyAtStop( bar+1, Close[bar]*1.03 );

            if( Close[bar] > Close[bar-3] )
                BuyAtMarket( bar+1 );
        }
        if (IsLastPositionActive)
        {
            SellAtLimit( bar+1, LastPosition, Close[bar]*1.01 );
        }
    }

    if( Alerts.Count > 0 )
    {
        for( int i = 0; i < Alerts.Count; i++ )
        {
            WealthLab.Alert a = Alerts[i];
            // Show the alert price if it's limit/stop order
                if( ( a.OrderType != OrderType.Market ) &
                    ( a.OrderType != OrderType.AtClose ) )
                PrintDebug( "Alert " + ( i+1 ) + " has a " + a.OrderType + " price of " + a.Price ); else
```

```
                PrintDebug( "Alert " + ( i+1 ) + " is a AtMarket/AtClose order; price N/A" );
        }
    }
}
```

Scale Property

BarScale Scale

Returns a BarScale enumerated list, which contains the bar scale of generated alert. Possible values are:

- Daily
- Minute
- Monthly
- Quarterly
- Second
- Tick
- Weekly
- Yearly

Example

```
protected override void Execute(){
    // Alert generating code
    for(int bar = 3; bar < Bars.Count; bar++)
    {
        if (!IsLastPositionActive)
        {
                BuyAtMarket( bar+1 );
        }
        if (IsLastPositionActive)
        {
            SellAtMarket( bar+1, LastPosition );
        }
    }

    // Returns alert bar scale
    if( Alerts.Count > 0 )
    {
        for( int i = 0; i < Alerts.Count; i++ )
        {
            WealthLab.Alert a = Alerts[i];
            PrintDebug( "Alert from strategy that executed on " + a.Scale + " scale ");
        }
    }
}
```

Shares Property

double Shares

Returns a Shares number, which contains the number of shares for generated alert.

Remarks

- In portfolio simulation mode, all trades are pre-executed using 1 share per Position, and then position sizing is applied after the fact. So the **Shares** property will always return 1 while the Strategy is executing.

Example

```
protected override void Execute(){
    // Alert generating code
    for(int bar = 3; bar < Bars.Count; bar++)
    {
        if (!IsLastPositionActive)
        {
            // Three consecutive lower closes
            if( ( Bars.Close[bar] < Bars.Close[bar-1] ) &
                ( Bars.Close[bar-1] < Bars.Close[bar-2] ) &
                ( Bars.Close[bar-2] < Bars.Close[bar-3] ) )
                BuyAtStop( bar+1, Close[bar]*1.03 );
        }
        if (IsLastPositionActive)
        {
            SellAtLimit( bar+1, LastPosition, Close[bar]*1.10 );
        }
    }

    // Show the number of shares in alert
    if( Alerts.Count > 0 )
    {
        for( int i = 0; i < Alerts.Count; i++ )
        {
            WealthLab.Alert a = Alerts[i];
                PrintDebug( "Alert " + ( i+1 ) + " is for " + a.Shares + " shares " );
        }
    }
}
```

SignalName Property

string SignalName

Returns a SignalName string, which contains the name of a signal which generated the alert.

Example

```
protected override void Execute(){
    // Alert generating code

    Random rnd = new Random();

    for(int bar = 1; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            // Exit trade after 3 days
```

```
            if ( bar+1 - LastPosition.EntryBar >= 3 )
                SellAtMarket( bar+1, LastPosition, "Time-based" );
        }
        else
        {
            // Random factor
            if ( rnd.Next(0,1) == 0 )
                BuyAtStop( bar+1, High[bar]*1.03, "Buy strength" ); else
                BuyAtLimit( bar+1, Low[bar]*0.97, "Buy weakness" );
        }
    }

    // Show the signal name of the alert
    if( Alerts.Count > 0 )
    {
        for( int i = 0; i < Alerts.Count; i++ )
        {
            WealthLab.Alert a = Alerts[i];
            PrintDebug( "Alert " + ( i+1 ) + " was generated by:  " + a.SignalName + " signal" );
        }
    }
}
```

Symbol Property

string Symbol

Returns a Symbol string, which contains the symbol name of generated alert.

Example

```
protected override void Execute(){
            // Alert generating code
            for(int bar = 3; bar < Bars.Count; bar++)
            {
                if (!IsLastPositionActive)
                {
                    // Three consecutive lower closes
                    if( ( Bars.Close[bar] < Bars.Close[bar-1] ) &
                        ( Bars.Close[bar-1] < Bars.Close[bar-2] ) &
                        ( Bars.Close[bar-2] < Bars.Close[bar-3] ) )
                        BuyAtStop( bar+1, Close[bar]*1.03 );
                }
                if (IsLastPositionActive)
                {
                    SellAtLimit( bar+1, LastPosition, Close[bar]*1.10 );
                }
            }

            // Show the symbol name for an alert
            if( Alerts.Count > 0 )
            {
                for( int i = 0; i < Alerts.Count; i++ )
                {
                    WealthLab.Alert a = Alerts[i];
                        PrintDebug( "Alert " + ( i+1 ) + " is for " + a.Symbol );
                }
            }
        }
```

Bars Object

The Bars object represents a collection of historical open, high, low, close and volume values.

BarInterval Property

int BarInterval

Returns the intraday bar interval of the Bars object. For example, if the Bars object contains 5-minute bars, BarInterval will return 5.

Remarks

- For non-intraday scales, BarInterval returns 0.

Example

```
protected override void Execute(){
    // Returns the chart bar interval
    if( Bars.BarInterval > 0 )
        PrintDebug( "BarInterval: " + Bars.BarInterval + " tick, second or minute " ); else
        PrintDebug( "BarInterval is Daily or greater" );
}
```

Cache Property

Dictionary<string, DataSeries> Cache

Each Bars object maintains an internal cache that stores the indicators (which are DataSeries objects) created based on itself, or its open, high, low, close, or volume. The Cache property provides access to these indicators. The Cache is a name=DataSeries Dictionary, and the DataSeries are stored using their Description as the Dictionary key. You will likely never need to use the Cache in Strategy code, but it can be useful when building custom Performance Visualizers, because it provides access to all of the indicators created by the Strategy.

Remarks

- When an indicator is created using its Series method, Wealth-Lab first looks for the indicator in the Cache and returns it if found. This prevents the same indicators from being created and calculated multiple times, increasing overall efficiency.
- You can explicitly clear the cache by calling Cache.Clear of the Bars object. Calling Bars.Cache.Clear at the end of the Strategy can help free resources for Strategies that process large amounts of data, especially intraday.

Example

```
protected override void Execute(){    // Creating a SMA the regular way:
    DataSeries sma = SMA.Series( Close,20 );

    // Creating a proxy data series:
    DataSeries sma_test  = new DataSeries(Bars,"test");

    // Find the 20-period SMA in the Bars.Cache property by its Description...
    if (Bars.Cache.ContainsKey("SMA(Close,20)"))
        //... and assign the result to the proxy series:
        sma_test = (DataSeries)Bars.Cache["SMA(Close,20)"];

    // Test by plotting the proxy series
    ChartPane test = CreatePane( 30, false, true );
    PlotSeries( test, sma_test, Color.Black, LineStyle.Solid, 1 );
}
```

Close Property

DataSeries Close

Returns a DataSeries object that represents the closing prices of the Bars object. Access individual closing prices via the square bracket syntax:

```
//Access closing price of the last bar
double lastClose = Bars.Close[Bars.Count - 1];
```

Remarks

- See the DataSeries object reference for a listing of available properties and methods on the DataSeries object.

Example

```
protected override void Execute(){
    //Access closing price of the last bar
    double lastClose = Bars.Close[Bars.Count-1];
    // The string is output with 2 digits
    DrawLabel( PricePane, "Last close: " + String.Format( "{0:f}", lastClose ), Color.Black );
}
```

ConvertDateToBar

int ConvertDateToBar(DateTime date, bool exactMatch);

Returns the bar number that matches the DateTime provided in the **date** parameter. If **exactMatch** is true, the precise DateTime value must be located in the Bars object. Otherwise, the first bar whose DateTime is greater than or equal to the specified **date** is returned.

Remarks

- (Doesn't affect WealthScript Strategy coding). In development of **PosSizers** and **Performance Visualizers**, accessing an EquityCurve or CashCurve value in multi-symbol portfolio simulations may produce unexpected results because the different historical DataSets aren't synchronized when backtesting. **Solution**: in a multi-symbol backtest, it's advised to use the **ConvertDateToBar** method of the *EquityCurve* or *CashCurve* DataSeries to get a correct bar. For example, here's how to determine the equity curve value at the beginning of a month using ConvertDateToBar:

```
double MonthStartEquity = EquityCurve[0];
DateTime b = bars.Date[bar];
DateTime tmp = new DateTime(b.Year, b.Month, 1);
MonthStartEquity = EquityCurve[EquityCurve.ConvertDateToBar(tmp, false)];
```

Example

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Drawing;
using System.Threading;
```

```
using System.Globalization;
using WealthLab;

namespace WealthLab.Strategies
{
    // Print how much days since year began

    public class MyStrategy : WealthScript
    {
        public int HowMuchDaysToBar( int bar, int year )
        {
            CultureInfo en = new CultureInfo("en-US");
            Thread.CurrentThread.CurrentCulture = en;
            String format = "yyyyMMdd";
            DateTime dtParsed = DateTime.ParseExact( Convert.ToString( year * 10000 + 101 ), format, en.DateTimeFormat );
            return bar - Bars.ConvertDateToBar( dtParsed, false ) + 1;
        }

        protected override void Execute()
        {
            int year = 2007;
            DrawLabel( PricePane, HowMuchDaysToBar( Bars.Count-1, year ) + " bars since year " + year + " started" );
        }
    }
}
```

Count Property

int Count

Returns the number of bars that are contained in the Bars object. The Bars object's Date, Open, High, Low, Close, Volume, and any "Named DataSeries" it contains, will always have the same number of values as the Bar's Count.

Example

```
protected override void Execute(){
    // Typical trading system main loop relies on Bars.Count property
    for(int bar = 20; bar < Bars.Count; bar++)
    {
    ///...
    }
}
```

DataScale Property

BarDataScale DataScale

Returns a BarDataScale struct, which contains the Bars object's Scale and BarInterval in one structure.

Example

```
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 );
}
```

Date Property

IList<DateTime> Date

Returns a list of DateTime values that represents the historical date/times of the Bars object. Access individual date values via the square bracket syntax:

```
//Access the last date being charted
DateTime lastDate = Bars.Date[Bars.Count - 1];
```

Example

```
protected override void Execute(){
    //Access the last date being charted
    DateTime lastDate = Bars.Date[Bars.Count - 1];
    DrawLabel( PricePane, "Last trading date: " + String.Format( "{0:d}", lastDate ), Color.Black );
}
```

FindNamedSeries

DataSeries FindNamedSeries(string name);

Locates a "Named DataSeries" that exists within the Bars object. Named DataSeries can be registered with a Bars object by specific Data Providers. A common example of a possible Named DataSeries is open interest for futures data. Another example are additional data fields that are imported in ASCII files.

Remarks

- If the specified Named DataSeries was not found, FindNamedSeries returns null.
- *Workaround:* Use *GetExternalSymbol* overload that accepts *dataSetName* as shown here. As an alternative, use GetAllDataForSymbol (example).

Example

```
protected override void Execute(){
    DataSeries MySeries;
    MySeries = Bars.FindNamedSeries( "SeriesName" );
}
```

FirstActualBar Property

int FirstActualBar

Returns an integer number of the bar that represents the first "real" bar of the secondary series. You can use this value to make sure that you don't enter trades on the symbol before its actual history began.

This function is useful in scripts that loop through and execute trades on all of the symbols in a DataSet. In these cases, Wealth-Lab's synchronization feature will transform secondary data series so that they synchronize with the Primary series, the one clicked to run the script. If a secondary data series has a shorter history than the Primary series, data bars are appended to the beginning of the secondary series so that it's BarCount equals that of the Primary series.

Example

```
using System;
using System.Collections;
using System.Text;
using WealthLab;

namespace WealthLab.Strategies
{
    public class MyStrategy : WealthScript
    {
        protected override void Execute()
        {
            const char tab = '\u0009';
            SortedList list = new SortedList( DataSetSymbols.Count );

            for(int ds = 0; ds < DataSetSymbols.Count; ds++)
            {
                SetContext( DataSetSymbols[ds], true );
                list.Add( ds, Bars.FirstActualBar );
                RestoreContext();
            }

            foreach( DictionaryEntry i in list )
                PrintDebug( "First bar of " + DataSetSymbols[(int)i.Key] + tab + ": -- #" + i.Value );
        }
    }
}
```

FormatValue

string FormatValue(double value);

Formats the specified **value** into a string, using the current number of Decimals.

Example

```
protected override void Execute(){
    // Output closing value to the chart
    DrawLabel( PricePane, Bars.FormatValue( Bars.Close[Bars.Count-1] ), Color.Red );
}
```

HasNamedDataSeries Property

bool HasNamedDataSeries

The HasNamedDataSeries property returns true if any "Named DataSeries" have been registered in the Bars object.

Example

```
protected override void Execute(){
{
    if( Bars.HasNamedDataSeries )
    {
        PrintDebug( Bars.Symbol + " contains " +
            Bars.NamedSeries.Count + " named series"  );
    }
    else
        PrintDebug( Bars.Symbol + " does not contain named series" );
}
```

High Property

DataSeries High

Returns a DataSeries object that represents the high prices of the Bars object. Access individual high prices via the square bracket syntax:

//Access high price of the last bar
double lastHigh = Bars.High[Bars.Count - 1];

Remarks

- See the DataSeries object reference for a listing of available properties and methods on the DataSeries object.

Example

```
protected override void Execute(){
    // Print high price of the last bar
    double high = Bars.High[Bars.Count-1];
    DrawLabel( PricePane, "High: " + String.Format( "{0:f}", high ), Color.Black );
}
```

IntradayBarNumber

int IntradayBarNumber(int bar)

Returns the intraday bar number of the day for intraday data. If the **Bars** object contains non-intraday data, **IntradayBarNumber** always returns -1. The first bar of a particular date returns 0, the next bar returns 1, and so on.

Example

```
protected override void Execute(){
    // Check for intraday data
    if ( Bars.IsIntraday )
    {
        // Color the middle of the trading day

        // First determine how many bars there are in one day
        int MaxBars = 0;
        double pct;

        for(int bar = Bars.Count-1; bar > -1; bar--)
            if ( Bars.IntradayBarNumber( bar ) == 0 )
        {
```

```
            MaxBars = Bars.IntradayBarNumber( bar-1 );
            break;
        }
        if ( MaxBars == 0 )
            return;

        // Now color the bars 40 - 60% within the day's range
        for(int bar = 0; bar < Bars.Count; bar++)
        {
            pct = (float) Bars.IntradayBarNumber( bar ) / MaxBars;
            if ( ( pct >= 0.4 ) & ( pct <= 0.6 ) )
                SetBarColor( bar, Color.Olive );

        }
    }
}
```

IsIntraday Property

bool IsIntraday

Returns whether the Bars object contains intraday data.

Example

```
protected override void Execute(){
    if ( Bars.IsIntraday != true )
        System.Windows.Forms.MessageBox.Show( "Not an intraday chart " );
    else
        System.Windows.Forms.MessageBox.Show( "The intraday bar interval is " + Bars.BarInterval );
}
```

IsLastBarOfDay

bool IsLastBarOfDay(int bar)

Returns true if this is the last bar of a particular day for intraday data. If the **Bars** object contains non-intraday data, **IsLastBarOfDay** always returns false. If **bar** equals the last bar of data in the chart, **IsLastBarOfDay** finds the previous bar that was the last bar of the day, and compares the time values to determine if the **bar** is in fact the last bar of the current day.

Example

```
protected override void Execute(){
    // Daytrading SMA crossover script (backtesting only)
    // that closes all positions at the end of the day.

    DataSeries hMAFast = SMA.Series( Close, 10 );
    DataSeries hMASlow = SMA.Series( Close, 30 );
    PlotSeries( PricePane, hMAFast, Color.Green, WealthLab.LineStyle.Solid, 1 );
    PlotSeries( PricePane, hMASlow, Color.Red, WealthLab.LineStyle.Solid, 1 );

    for(int bar = hMASlow.FirstValidValue; bar < Bars.Count; bar++)
    {
        if (!IsLastPositionActive)
        {
            if ( Bars.IsLastBarOfDay( bar ) == false )
                if ( CrossOver( bar, hMAFast, hMASlow ) )
                BuyAtMarket( bar+1, "XOver" );
        }
        else
        {
            Position p = LastPosition;
            if ( Bars.IsLastBarOfDay( bar ) == true )
                SellAtClose( bar, p, "EOD" );
            else
            {
                // normal intraday exit logic
                if ( CrossUnder( bar, hMAFast, hMASlow ) )
                    SellAtMarket( bar+1, p, "XUnder" );
                }
        }
    }
}
```

IsSynthetic

bool IsSynthetic(int bar);

Allows you to determine if individual bars in the Bars object are "synthetic". Synthetic bars are bars that are created as a result of the **AddCalendarDays** WealthScript method.

Remarks

- **Known issue**: *Bars.IsSynthetic* wrongly marks the first trading bar after a series of synthetic bars added by *AddCalendarDays*. It does not work as documented, i.e. synthetic bars are not marked.

Example

```
protected override void Execute(){
    // Highlight added bars

    if( Bars.Scale == 0 )
    {
        int added = AddCalendarDays( true );
        DrawLabel( PricePane, "Interpolated bars: "+ added.ToString(), Color.YellowGreen );

        for(int bar = 0; bar < Bars.Count; bar++)
        {
            if( Bars.IsSynthetic( bar ) )
                SetBarColor( bar, Color.YellowGreen );
        }
    } else
        System.Windows.Forms.MessageBox.Show( "Data must be Daily" );
}
```

LoadFromFile

void LoadFromFile(string fileName)
void LoadFromFile(string fileName, int maxBars)

void LoadFromFile(string fileName, System.DateTime startDate, System.DateTime endDate)
void LoadFromFile(string fileName, System.DateTime startDate, System.DateTime endDate, int maxBars)

Loads the Bars object from an existing file on disk. The binary file can be created by any Wealth-Lab data provider, or manually in Strategy code (see *SaveToFile*).

Optionally, it's possible to load a specific amount of most-recent bars no greater than the **maxBars** value (see example below). In addition, you can limit the time interval using the **startDate** and **endDate** parameters.

Example

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

namespace WealthLab.Strategies
{
    /*  Load bars from file */

    public class TestLoadBars : WealthScript
    {
        protected override void Execute()
        {
            Bars bars = new Bars("NewBars",BarScale.Daily,0);

            /* Pass "true" to GetDataPath() if using Wealth-Lab Pro,
            for Wealth-Lab Developer make it "false"

            Note: The data should NOT be relocated, otherwise you would have
            to correct the path inside the GetDataPath() function body */

            string sym = GetDataPath( false ) + @"Daily\A\A.WL";

            // Load just 100 recent bars of the stock called "A"
            bars.LoadFromFile( sym, 100 );

            // Plot the data
            bars = Synchronize( bars );
            ChartPane newBars = CreatePane( 50, true, true );
            PlotSymbol( newBars, bars, Color.Blue, Color.Red );
        }

        // Get the path to directory where data is stored
        public static string GetDataPath( bool pro )
        {
            string path = pro ?
                @"\Fidelity Investments\WealthLabPro\1.0.0.0\Data\FidelityStaticProvider\" :
                @"\Fidelity Investments\WealthLabDev\1.0.0.0\Data\YahooStaticProvider\";
            return
                Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData) + path;
        }
    }
}
```

Low Property

DataSeries Low

Returns a DataSeries object that represents the low prices of the Bars object. Access individual low prices via the square bracket syntax:

```
//Access low price of the last bar
double lastLow = Bars.Low[Bars.Count - 1];
```

Remarks

- See the DataSeries object reference for a listing of available properties and methods on the DataSeries object.

Example

```
protected override void Execute(){
    // Print low price of the last bar
    double low = Bars.Low[Bars.Count-1];
    DrawLabel( PricePane, "Low: " + String.Format( "{0:f}", low ), Color.Black );
}
```

NamedSeries Property

ICollection<DataSeries> NamedSeries

The NamedSeries property returns a list of all of the "Named DataSeries" that have been registered in the Bars object. Named DataSeries can be registered with a Bars object by specific Data Providers. A common example of a possible Named DataSeries is open interest for futures data. Another example are additional data fields that are imported in ASCII files.

Example

```
protected override void Execute(){
    // Run this on a DataSet with defined Custom series

    // For example:
    //DATE;TIME;VOLUME;OPEN;CLOSE;MIN;MAX
    //06/08/2007;22:36:41;0;21.83;21.83;21.83;21.83
    //06/08/2007;22:36:51;0;21.83;21.83;21.83;21.83
    //06/08/2007;22:37:01;0;21.83;21.83;21.83;21.83
    // Will output 'Min' and 'Max'

    if ( Bars.NamedSeries.Count > 0 )
    {
        foreach( DataSeries d in Bars.NamedSeries )
            PrintDebug( d.Description );
    }
}
```

Open Property

DataSeries Open

Returns a DataSeries object that represents the open prices of the Bars object. Access individual open prices via the square bracket syntax:

```
//Access open price of the last bar
double lastOpen = Bars.Open[Bars.Count - 1];
```

Remarks

- See the DataSeries object reference for a listing of available properties and methods on the DataSeries object.

Example

```
protected override void Execute(){
    // Print open price of the last bar
    double open = Bars.Open[Bars.Count-1];
    DrawLabel( PricePane, "Open price: " + String.Format( "{0:f}", open ), Color.Black );
}
```

SaveToFile

void SaveToFile(string fileName)

Saves the Bars object to a file on disk. The binary file can be recognized by Wealth-Lab natively (see *LoadFromFile*).

Example

```
protected override void Execute(){
    /* Create a Heikin-Ashi chart and save the resulting bars to file */

    HideVolume();
    Bars bars = new Bars( Bars.Symbol.ToString() + " (Heikin-Ashi)", BarScale.Daily, 0 );

    // Create Heikin-Ashi series
    DataSeries HO = Open + 0;
    DataSeries HH = High + 0;
    DataSeries HL = Low + 0;
    DataSeries HC = (Open + High + Low + Close) / 4;

    // Build the Bars object
    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] );

        bars.Add( Bars.Date[bar], HO[bar], HH[bar], HL[bar], HC[bar], Bars.Volume[bar]);
    }

    // Save the virtual Heikin-Ashi bars to disk
    string file = @"C:\Heikin-Ashi.WL";
    bars.SaveToFile( file );

    // Verify by loading from disk and plotting
    Bars haBars = new Bars("Saved Heikin-Ashi Bars",BarScale.Daily,0);
    haBars.LoadFromFile( file );
    haBars = Synchronize( haBars );
    ChartPane haPane = CreatePane(50, false, true);
    PlotSymbol(haPane, haBars, Color.DodgerBlue, Color.Red);
}
```

Scale Property

BarScale Scale

Returns the Scale of the data contained in the Bars object. Possible Scale values are:

- Daily
- Weekly
- Monthly
- Minute
- Second
- Tick
- Quarterly
- Yearly

Example

```
protected override void Execute(){
    System.Windows.Forms.MessageBox.Show( "Data scale is " + Bars.Scale );
}
```

SecurityName Property

string SecurityName

Returns the security name of the symbol contained in the Bars object. This will be the company name for stocks, and the name of the commodity or future for futures symbols.

Example

```
protected override void Execute(){
    System.Windows.Forms.MessageBox.Show( "We're now viewing " + Bars.SecurityName + " chart");
}
```

Symbol Property

string Symbol

Returns the symbol for the data that was loaded into the Bars object.

Example

```
protected override void Execute(){
    // Show the closing price with the symbol in a chart label }
    double x = Close[Bars.Count-1];
    DrawLabel( PricePane, "Closing price for " + Bars.Symbol + " is " + x, Color.DarkSlateGray );
}
```

SymbolInfo Property

The SymbolInfo object represents a number of symbol's properties: Decimals, Margin, Point Value, Security Type and Tick.

Example

```
protected override void Execute(){
    SymbolInfo si = Bars.SymbolInfo;
    PrintDebug( Bars.Symbol );
    PrintDebug( "Symbol= " + si.Symbol );
    PrintDebug( "Point Value = " + si.PointValue );
    PrintDebug( "Tick = " + si.Tick );
    PrintDebug( "Margin = " + si.Margin );
    PrintDebug( "Decimals = " + si.Decimals);
    PrintDebug( "" );
}
```

Tag Property

object Tag

The **Tag** property allows you to store any object with a Bars object.

Example

```
protected override void Execute(){//
            //"currentPos" is null when sizing trading signals on bar+1 (Alert) in PosSizers.
            //This example illustrates how to send a double value to a PosSizer
            //from a Strategy to size an Alert.
            //

            for(int bar = 5; bar < Bars.Count; bar++)
            {
                if( IsLastPositionActive )
                {
                    SellAtMarket( bar+1, LastPosition );
                }
                else
                {
                    if( Close[bar] <= Lowest.Series( Close, 5 )[bar-1] )
                    {
                        double size = 100;
                        if( BuyAtMarket( bar+1 ) == null && (bar == Bars.Count-1) )
                        {
                            // Store some double value in the current Bars.Tag property
                            Bars.Tag = size;
                        }
                    }
                }
            }

            // Next, in your PosSizer's SizePosition method call:

            // double size = (double)bars.Tag;
        }
    }
}
```

Volume Property

DataSeries Volume

Returns a DataSeries object that represents the volume of the Bars object. Access individual bar volumes via the square bracket syntax:

```
//Access volume of the first bar
double firstVolume = Bars.Volume[0];
```

Remarks

- See the DataSeries object reference for a listing of available properties and methods on the DataSeries object.

Example

```
protected override void Execute(){
    //Access volume of the last bar
    double turnover = Bars.Close[Bars.Count-1] * Bars.Volume[Bars.Count-1];
    // Print stock turnover
    DrawLabel( PricePane, "Turnover: " + String.Format( "${0:0,0}", turnover ), Color.Black );
}
```

ChartPane Object

The ChartPane object represents one of the panes of the current chart.

ConvertValueToY

int ConvertValueToY(double value);

Converts the specified numeric **value** to the Y-pixel coordinate on the chart pane. This method is most valuable in custom Chart Style programming, but can also come in handy when developing custom Drawing Objects or using WealthScript's PaintHook functionality.

Decimals Property

int Decimals

Specifies the number of decimals places that will be used to format values in the chart pane's margins.

Remarks

- **Known issue**: ChartPane.Decimals doesn't work for all panes but the VolumePane

DisplayGrid Property

bool DisplayGrid

Determines whether the horizontal gridlines should be visible on this chart pane.

FormatChartValue

string FormatChartValue(double value);

Formats the specified numeric value to an abbreviated string, based on its value, and also taking into account the Decimals property.

Examples:

123.45 will format as 123.45
12,345 will format as 12.34K
1,234,567 will format as 1.23M

GetBackgroundColor

Color GetBackgroundColor(int bar);

Returns the background color at the specified **bar** in the pane.

Height Property

int Height

The current height of the pane, in pixels. This property has limited value at the time a Strategy executes, but can prove valuable when building custom ChartStyles, or using the PaintHook functionality of WealthScript.

IsPricePane Property

bool IsPricePane

Returns true if this chart pane is the one that the main price bars are being plotted on. This property has limited value in Strategy code, but can prove useful in programming custom Drawing Objects.

Example

```
protected override void Execute(){
if ( PricePane.IsPricePane )
        System.Windows.Forms.MessageBox.Show( "Operating in Strategy window" ); else
        System.Windows.Forms.MessageBox.Show( "Running in Strategy Monitor" );
}
```

LogScale Property

bool LogScale

Controls whether the chart pane will be plotted in semi-log scale.

A semi-log scale gives equal weight to percentage changes, rather then absolute value changes. For example, the distance from 1 to 10 will be the same size on the chart as the distance from 10 to 100. It's called "semi-log" because only the y-axis uses the log scale, whereas the x-axis [typically] remains evenly-spaced.

Example

```
protected override void Execute(){
    if( ( PricePane != null ) & ( VolumePane != null ) )
    {

        PricePane.LogScale = true;
        VolumePane.LogScale = true;

        DrawLabel( PricePane, "Price Pane is in Semi-Log Scale? " + PricePane.LogScale );
        DrawLabel( PricePane, "Volume Pane is in Semi-Log Scale? " + VolumePane.LogScale );
    }
}
```

SetBackgroundColor

void SetBackgroundColor(int bar, Color color);

Sets the background color at the specified **bar** in the pane to the specified **color**.

Top Property

int Top

Returns the location of the top of the pane from the top of the chart, in pixels. This property has no real value in Strategy code, but can be useful in custom Drawing Object development, or implementing PaintHooks in the WealthScript.

Common Signals

The Common Signal category contains methods that are commonly used to produce trading signals.

CrossOver

bool CrossOver(int bar, DataSeries ds1, DataSeries ds2);
bool CrossOver(int bar, DataSeries ds1, double value);

Returns true if the specified DataSeries (**ds1**) crosses over either another DataSeries (**ds2**), or a specific **value**, on the specified **bar**. Specifically, CrossOver returns true if the current value is above the target value at the specified **bar**, and the previous value was less than or equal to the target value at the previous bar.

Example

```
protected override void Execute(){
    DataSeries wma1 = WMA.Series( Close, 30 );
    DataSeries wma2 = WMA.Series( Close, 60 );
    PlotSeries( PricePane, wma1, Color.LightCoral, WealthLab.LineStyle.Solid, 1 );
    PlotSeries( PricePane, wma2, Color.LightBlue, WealthLab.LineStyle.Solid, 1 );

    // A simple Weighted Moving Average Crossover System

    for(int bar = wma2.FirstValidValue; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            Position p = LastPosition;
            SellAtStop( bar+1, p, p.EntryPrice * 0.96, "4% Stop" );
            SellAtLimit( bar+1, p, p.EntryPrice * 1.06, "6% Target" );
        }
        else
        {
            if ( CrossOver( bar, wma1, wma2 ) )
            BuyAtMarket( bar+1 );
        }
    }
}
```

CrossUnder

bool CrossUnder(int bar, DataSeries ds1, DataSeries ds2);
bool CrossUnder(int bar, DataSeries ds1, double value);

Returns true if the specified DataSeries (**ds1**) crosses under either another DataSeries (**ds2**), or a specific **value**, on the specified **bar**. Specifically, CorssUnder returns true if the current value is below the target value at the specified **bar**, and the previous value was greater than or equal to the target value at the previous bar.

Example

```
protected override void Execute(){
    ChartPane StochPane = CreatePane( 50, true, true );
    DataSeries D = StochD.Series( Bars, 3, 20 );
    DataSeries Signal = EMA.Series( D, 9, WealthLab.Indicators.EMACalculation.Modern );
    PlotSeries( StochPane, D, Color.Blue, WealthLab.LineStyle.Solid, 1 );
    PlotSeries( StochPane, Signal, Color.Gray, WealthLab.LineStyle.Solid, 1 );

    for(int bar = 30; bar < Bars.Count; bar++)
    {
            // It closes all positions when Stochastic
            // crosses below the signal line from above 80.

            if ( ( ActivePositions.Count > 0 ) &&
                CrossUnder( bar, D, Signal ) && ( D[bar-1] > 80 ) )
            {
                // Let's work directly with the list of active positions, introduced in WL5
                for( int p = ActivePositions.Count - 1; p > -1 ; p-- )
                    SellAtMarket( bar+1, ActivePositions[p] );
            }

            // This system opens a new position whenever Stochastic
            // crosses above its signal line from below 20.

            if ( CrossOver( bar, D, Signal ) )
            if ( D[bar-1] < 20 )
                BuyAtMarket( bar+1, "Stoch" );
    }
}
```

TurnDown

bool TurnDown(int bar, DataSeries series);

Returns true if the specified DataSeries has "turned down" as of the specified **bar**. The **series** has turned down if the value at **bar** is less than the value at **bar** - 1, and the next most recent change in value in the **series** was an increase.

Example

```
protected override void Execute(){
    // Buy when Williams %R turns down and is above 80
    DataSeries PctR = WilliamsR.Series( Bars, 30 );
    ChartPane PctRPane = CreatePane( 25, true, true );
    PlotSeriesOscillator( PctRPane, PctR, 90, 10, Color.LightCoral, Color.LightBlue, Color.Gray, WealthLab.LineStyle.Solid, 1 );

    // Time-based exit
    int days = 20;

    // Start trading loop with the first "valid" value of %R
    for(int bar = PctR.FirstValidValue; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            if ( bar+1 - LastPosition.EntryBar >= days )
                SellAtClose( bar, LastPosition );
        }
        else
        {
```

```
            // Color turndowns
            if ( TurnDown( bar, PctR ) )
            {
                SetSeriesBarColor( bar, PctR, Color.Red );
                if ( PctR[bar] > 80 )
                {
                    BuyAtMarket( bar+1, "WR" );
                }
            }
        }
    }
}
```

TurnUp

bool TurnUp(int bar, DataSeries series);

Returns true if the specified DataSeries has "turned up" as of the specified **bar**. The **series** has turned up if the value at **bar** is greater than the value at **bar** - 1, and the next most recent change in value in the **series** was a decrease.

Example

```
protected override void Execute(){
    // Enter the market when the slow stochastic turns up from below 15
    DataSeries stoch = StochD.Series( Bars, 5, 60 );
    ChartPane StochPane = CreatePane( 30, true, true );
    PlotSeries( StochPane, stoch, Color.Blue, WealthLab.LineStyle.Solid, 2 );

    // Start trading loop with the first "valid" value of StochD
    for(int bar = stoch.FirstValidValue; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            if(  CrossOver( bar, stoch, 80 ) )
            SellAtMarket( bar+1, LastPosition, "StochD Crosses 80" );
        }
        else
        {
            if ( stoch[bar-1] < 15 )
            {
            // Color TurnUps
            if ( TurnUp( bar, stoch ) )
                {
                SetSeriesBarColor( bar, stoch, Color.Red );
                BuyAtMarket( bar+1, "StochasticD Turns Up" );
                }
            }
        }
    }
}
```

Cosmetic Chart

The Cosmetic Chart category consists of methods you can use to plot shapes, images and various other annotations on the chart. It also contains methods to control the colors of the bars and chart background.

AnnotateBar

void AnnotateBar(string text, int bar, bool aboveBar, Color color, Color backgroundColor, Font font);
void AnnotateBar(string text, int bar, bool aboveBar, Color color, Color backgroundColor);
void AnnotateBar(string text, int bar, bool aboveBar, Color color);

Annotates the specified **bar** with the string passed in the **text** parameter, using the specified **color** for the font. Use the **aboveBar** parameter to control if the **text** is displayed above or below the **bar**. Using the version that accepts a **backgroundColor** parameter causes the **text** to be displayed over a filled background. Calling AnnotateBar multiple times causes the annotations to be stacked one on top of another either above or below the **bar**.

Use the version of the method that accept a **Font** parameter to draw the text using a custom font. If you use this version and do not want a colored background, specify Color.Empty for the **backgroundColor** parameter.

Example

```
protected override void Execute(){
    Font font = new Font("Arial", 12, FontStyle.Bold);
    // Demonstrates operation overload
    for(int bar = 200; bar < Bars.Count; bar++)
    {
        // Annotate a bar if it's a 200 day closing high
        if ( Bars.Close[bar] == Highest.Series( Close, 200 )[bar] )
            AnnotateBar( "High", bar, true, Color.DarkGreen );
        // Annotate a bar if it's a 200 day closing low
        if ( Bars.Close[bar] == Lowest.Series( Close, 200 )[bar] )
            AnnotateBar( "Low", bar, false, Color.DarkRed, Color.White );
        // Annotate the last bar
        if ( bar == Bars.Count-1 )
            AnnotateBar( "Last", bar-3, false, Color.DarkRed, Color.White, font );
    }
}
```

AnnotateChart

void AnnotateChart(ChartPane pane, string text, int bar, double value, Color color, Color backgroundColor, Font font, HorizontalAlignment alignment);
void AnnotateChart(ChartPane pane, string text, int bar, double value, Color color, Color backgroundColor, Font font);
void AnnotateChart(ChartPane pane, string text, int bar, double value, Color color, Color backgroundColor);
void AnnotateChart(ChartPane pane, string text, int bar, double value, Color color);

Annotates the chart with the specified **text** using the specified **color** at a location provided by the **bar** and **value** parameters. The **pane** parameter determines which chart pane is annotated. If you call the version of AnnotateChart that accepts a **backgroundColor** parameter, the text will be displayed over a filled background.

Use the version of the method that accept a **Font** parameter to draw the text using a custom font. If you use this version and do not want a colored background, specify Color.Empty for the **backgroundColor** parameter.

Use the first overloaded version of the method, with the **alignment** parameter, to control the alignment of the text, relative to the **bar**. Possible values are Left, Center or Right.

Remarks

- To annotate the price pane, use PricePane for the **pane** parameter.
- To annotate the volume pane, use VolumePane for the **pane** parameter.

Example

```
protected override void Execute(){
    // Define new font style
    Font font = new Font("Arial", 7, FontStyle.Regular);
    DataSeries smaVolume = SMA.Series( Volume, 50 );
    PlotSeries( VolumePane, smaVolume, Color.LightSalmon, WealthLab.LineStyle.Solid, 2 );
    // Demonstrates operator overload
    for(int bar = 50; bar < Bars.Count; bar++)
    {
        // Annotate the last bar if it demonstrates unusual volatility
        if ( bar == Bars.Count-1 )
        {
            if ( ATR.Series( Bars, 1 )[bar] >= 2 * ATR.Series( Bars, 14 )[bar] )
                AnnotateChart( PricePane, "Volatile!", bar-5, High[bar], Color.Green, Color.White, font, System.Windows.Forms.HorizontalAlignment.Left );
            if ( Volume[bar] >= 1.5 * smaVolume[bar] )
                AnnotateChart( VolumePane, "Volume is High", bar-10, Volume[bar], Color.Green, Color.White, font, System.Windows.Forms.HorizontalAlignment.Left );
        }
    }
}
```

ChartStyle Property

ChartStyle ChartStyle

Returns the instance of the ChartStyle object that is currently being used to render the chart. This is an object that derives from the base ChartStyle class, and is responsible for rendering the actual bars of the chart. Some ChartStyle objects contain additional data structures and information that can be used in your Strategy. Consult the specific ChartStyle documentation for any additional value that might be obtained.

Remarks

- To access any methods or properties that are specific to a ChartStyle derived object, you will need to cast the ChartStyle object returned here to the specific type you are expecting. For this to work, you will need to ensure that the desired chart style is actually selected in the toolbar.
- If the Strategy is run in a context where there is no chart (such as the Strategy Monitor or a Multi-Symbol Backtest), this property returns null.

Example

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

namespace WealthLab.Strategies
{
    public class MyStrategy : WealthScript
    {
        bool IsStrategyMonitor()
        {
            return ChartStyle == null;
        }
```

```
        protected override void Execute()
        {
            if ( !IsStrategyMonitor() )
                System.Windows.Forms.MessageBox.Show( "Operating in Strategy window" ); else
                System.Windows.Forms.MessageBox.Show( "Running in Strategy Monitor" );
        }
    }
}
```

CreatePane

ChartPane CreatePane(int height, bool abovePricePane, bool displayGrid);

Creates a new pane on the chart and returns the new pane as a ChartPane object. The **height** parameter controls the height of the pane, which fluctuates as the chart is resized. A value of 40 for **height** creates a pane with a standard height. You can create panes either above or below the price pane through the **abovePricePane** parameter. The **displayGrid** parameter controls whether the pane will display grid lines. You can plot indicators in the pane using PlotSeries and the various other plotting methods.

Remarks

- See the ChartPane object reference for more information about the properties and methods of ChartPanes.

Example

```
protected override void Execute(){
    //Create and plot Williams %R indicator
    WilliamsR wr = WilliamsR.Series( Bars, 20 );
    ChartPane wrPane = CreatePane( 40, true, true );
    PlotSeriesOscillator( wrPane, wr, 90, 10, Color.LightGreen, Color.LightCoral, Color.CadetBlue, LineStyle.Solid, 1);
}
```

DrawCircle

void DrawCircle(ChartPane pane, int radius, int bar, double value, Color color, Color fillColor, LineStyle style, int width, bool behindBars);
void DrawCircle(ChartPane pane, int radius, int bar, double value, Color color, LineStyle style, int width, bool behindBars);

protected void DrawCircle(ChartPane pane, int bar1, double value1, int bar2, double value2, Color color, Color fillColor, LineStyle style, int width, bool behindBars);
protected void DrawCircle(ChartPane pane, int bar1, double value1, int bar2, double value2, Color color, LineStyle style, int width, bool behindBars);

The DrawCircle method provides two ways to draw (and optionally fill) circles on a chart, in the specified **pane**. Each method accepts an optional **fillColor** parameter, that (if specified) causes the circle to be filled with a color. The final parameter, **behindBars**, determines whether the circle will be plotted behind or in front of the bars of the chart. The circle will be drawn using the specified **color**, **style**, and **width**.

The first method draws a circle with a **radius** specified in pixels, at the coordinates specified by the **bar** (X) and **value** (Y) parameters.

The second method draws a circle whose radius is a line specified by two points, **bar1**/**value1** and **bar2**/**value2**. In this way, your circles can be bound to actual bars/prices on the chart.

Remarks

- **Known issue**: Zooming the chart may fail if *DrawCircle()* is applied to the chart if the price range exceeds $2,000,000.00
- *Workaround:* Don't use DrawCircle() on bars where the price exceeds this figure:

```
for(int bar = 0; bar < Bars.Count; bar++)

{
if (Close[bar] < 2100000)
DrawCircle(PricePane, 9, bar, Close[bar]+ 0.05, Color.Empty, Color.Green, WealthLab.LineStyle.Solid, 1, true);
}
```

Example

```
protected override void Execute(){
    // Operator overload
    for(int bar = 200; bar < Bars.Count; bar++)
    {
        // Circle any 200 day High
        if ( High[bar] == Highest.Series( High, 200 )[bar] )
            DrawCircle( PricePane, 4, bar, High[bar], Color.Green, Color.DarkGreen, WealthLab.LineStyle.Solid, 1, true );
        // Circle any 200 day Low
        if ( Low[bar] == Lowest.Series( Low, 200 )[bar] )
            DrawCircle( PricePane, bar-1, Low[bar-1], bar, Low[bar-1], Color.Red, Color.DarkRed, WealthLab.LineStyle.Solid, 1, false );
    }
}
```

DrawEllipse

void DrawEllipse(ChartPane pane, int bar1, double value1, int bar2, double value2, Color color, Color fillColor, LineStyle style, int width, bool behindBars);
void DrawEllipse(ChartPane pane, int bar1, double value1, int bar2, double value2, Color color, LineStyle style, int width, bool behindBars);

Plots an ellipse on the **pane** using the specified **color**, **style** and **width**. If the version using **fillColor** is called, also fills the ellipse using the specified **fillColor**. The ellipse is bound by a rectangle defined by the points **bar1**, **value1** and **bar2**, **value2**. The **behindBars** parameter controls whether the ellipse is plotted behind, or in front of the bars of the chart.

Example

```
protected override void Execute(){
    int Bar = (int)TroughBar.Value( Bars.Count-1, Low, 5, WealthLab.Indicators.PeakTroughMode.Percent );
    double Price = Low[Bar];
    DrawEllipse( PricePane, Bar-4, Price*1.02, Bar+4, Price*0.98, Color.Red, Color.LightCoral, WealthLab.LineStyle.Solid, 1, false );
    Bar = (int)PeakBar.Value( Bars.Count-1, High, 5, WealthLab.Indicators.PeakTroughMode.Percent );
    Price = High[Bar];
    DrawEllipse( PricePane, Bar-4, Price*1.02, Bar+4, Price*0.98, Color.Green, Color.LightGreen, WealthLab.LineStyle.Solid, 1, true );
}
```

DrawHorzLine

void DrawHorzLine(ChartPane pane, double value, Color color, LineStyle style, int width);

Draws a horizontal line on a **pane**, and plots an accompanying label marking the value in the right margin of the chart. The line is drawn using the specified **color**, **style** and **width**.

Example

```
protected override void Execute(){
    // Plot RSI and draw horizontal lines at 30/70 levels
    ChartPane rsiPane = CreatePane( 50, true, false );
    PlotSeries( rsiPane, RSI.Series(Close,20), Color.Brown, WealthLab.LineStyle.Solid, 1 );
    DrawHorzLine( rsiPane, 30, Color.Green, WealthLab.LineStyle.Solid, 1 );
```

```
    DrawHorzLine( rsiPane, 70, Color.Red, WealthLab.LineStyle.Solid, 1 );
}
```

DrawImage

void DrawImage(ChartPane pane, Image image, int bar, double value, bool behindBars);

Draws the passed **image** object onto the specified **pane**. The **image** is centered on the specified **bar** and **value** in the **pane**. The **behindBars** parameter controls whether the image is rendered behind or in front of the chart bars and plotted indicators.

Example

```
protected override void Execute(){
    Image image = Image.FromFile("C:\\temp\\image.jpg");
    DrawImage( PricePane, image, Bars.Count-50, Close[Bars.Count-50], false );
}
```

DrawLabel

void DrawLabel(ChartPane pane, string text, Color color);
void DrawLabel(ChartPane pane, string text);

Draw a **text** label on the chart, on the specified **pane**. Optionally, you can provide a **color** for the font of the **text**. The label will be displayed in the upper left corner of the **pane** (below the security name if drawn on the price pane.) Calling DrawLabel multiple times will cause the labels to be stacked one on top of another.

Example

```
protected override void Execute(){
    for(int bar = 0; bar < Bars.Count; bar++)
    {
        // Check for negative price values
        if ( ( Close[bar] < 0 ) | ( Close[bar] < 0 ) |
        ( Close[bar] < 0 ) | ( Close[bar] < 0 ) )
            DrawLabel(PricePane, Bars.Date[bar].ToString());
    }
}
```

DrawLine

void DrawLine(ChartPane pane, int bar1, double value1, int bar2, double value2, Color color, LineStyle style, int width);

Draws a line on the specified **pane**, between the two points identified by **bar1**, **value1** and **bar2**, **value2**. The line is drawn using the specified **color**, **style** and **width**.

Example

```
protected override void Execute(){
    // Draw a line between the last 2 peaks
    int Bar = Bars.Count-1;
    double p1 = Peak.Value( Bar, High, 4, WealthLab.Indicators.PeakTroughMode.Percent );
    int pb1 = (int) PeakBar.Value( Bar, High, 4, WealthLab.Indicators.PeakTroughMode.Percent );
    double p2 = Peak.Value( pb1, High, 4, WealthLab.Indicators.PeakTroughMode.Percent );
    int pb2 = (int) PeakBar.Value( pb1, High, 4, WealthLab.Indicators.PeakTroughMode.Percent );
    DrawLine( PricePane, pb1, p1, pb2, p2, Color.Red, WealthLab.LineStyle.Dotted, 1 );
}
```

DrawPolygon

void DrawPolygon(ChartPane pane, Color color, Color fillColor, LineStyle style, int width, bool behindBars, params double[] coords);
void DrawPolygon(ChartPane pane, Color color, LineStyle style, int width, bool behindBars, params double[] coords);

The DrawPolygon allows you to draw a variety of shapes on the chart, optionally filled if you call the version that accepts a **fillColor** parameter. The shape will be drawn on the specified **pane**, using the **color**, **style**, and **width** that you pass as parameters. The **behindBars** parameter controls whether the shape will be drawn in front of or behind the bars of the chart.

The actual shape that will be drawn is defined in the **coords** parameter. You should pass a series of pairs of doubles, bar/value, that define the points of the polygon. For example, to draw a triangle, you would pass a total of 6 values, logically representing bar1, value1, bar2, value2, bar3, value3.

Example

```
protected override void Execute(){
    // Draw a rectangle outlining recent 10% peak and trough

    int Bar = Bars.Count-1;
    int b1 = (int)PeakBar.Value( Bar, Close, 10, WealthLab.Indicators.PeakTroughMode.Percent );
    int b2 = (int)TroughBar.Value( Bar, Close, 10, WealthLab.Indicators.PeakTroughMode.Percent );
    double p1 = Close[b1];
    double p2 = Close[b2];
    double[] rectangle = { b1, p1, b1, p2, b2, p2, b2, p1 }; // counter-clockwise

    DrawPolygon( PricePane, Color.Blue, Color.LightSteelBlue, WealthLab.LineStyle.Solid, 2, true, rectangle );
}
```

DrawText

void DrawText(ChartPane pane, string text, int x, int y, Color color, Color backgroundColor, Font font);
void DrawText(ChartPane pane, string text, int x, int y, Color color, Color backgroundColor);
void DrawText(ChartPane pane, string text, int x, int y, Color color);

Draws the specified text in the chart **pane**, at the **x**, **y** pixel coordinates. The coordinates are expressed as the number of pixels from the upper left corner of the **pane**. The **text** is drawn using the specified **color**. If you use the version that accepts a **backgroundColor** parameter, the **text** is drawn over a rectangle filled with that color.

Use the version of the method that accept a **Font** parameter to draw the **text** using a custom font. If you use this version and do not want a colored background, specify Color.Empty for the **backgroundColor** parameter.

Remarks

- To draw text on the price pane, use PricePane for the **pane** parameter.
- To draw text on the volume pane, use VolumePane for the **pane** parameter.

Example

```
protected override void Execute(){
    // Prints RSI value over the PricePane
    DrawText( PricePane, "14-period RSI is " + ( RSI.Series( Close, 14 )[Bars.Count-1] ), 0, 40, Color.Black, Color.Empty );
}
```

EnableTradeNotes

void EnableTradeNotes(bool Text, bool Arrow, bool Circle);

Controls the visibility of a trade tooltip, buy/sell arrows and intrabar entry/exit points on the chart.

The **Text** parameter controls whether or not a trade tooltip and a line to connect a trade's entry point to its associated exit point (if applicable) are drawn on the chart. **Arrow** controls whether or not buy and sell arrows appear above/below the bar where trades are opened and closed. **Circle** controls whether the circles are drawn at the exact spot where trades occur on the bar.

Remarks

- Disabling arrows will also make trade notes disappear even if **Text** is true.

Example

```
protected override void Execute(){
for(int bar = 1; bar < Bars.Count; bar++)
            {
                if (IsLastPositionActive)
                    SellAtMarket( bar+1, LastPosition );
                else
                    BuyAtMarket( bar+1 );
            }

            /* Turn off those pesky notes if there
            are many trades, show arrows only */

            if( Positions.Count > 20 )
                EnableTradeNotes( false, true, false );
        }
    }
}
```

HidePaneLines

void HidePaneLines();

Causes the lines separating the panes in a chart to not be displayed.

Example

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

HideVolume

void HideVolume();

Renders the volume pane invisible, providing more room to the Prices Pane in the chart.

Remarks

- Non-programmatically (and excluding the main chart) you can minimize and maximize panes by clicking the - and + next to the label in the pane's upper left corner. If the labels are not shown, enable them by clicking the "Show Indicators Labels on Chart" button in the chart toolbar.

Example

```
protected override void Execute(){
    // Some cosmetics
    HideVolume();
    HidePaneLines();
    //Plot Microsoft data on the same pane as the symbol being charted
    Bars msft = GetExternalSymbol("MSFT", true);
    ChartPane msftPane = CreatePane( 100, false, true);
    PlotSymbol( msftPane, msft, Color.Silver, Color.Silver);
}
```

PadBars

void PadBars(int numberOfBars);

Pads the right of the chart with empty space. The amount of space padded is based on the number specified in the **numberOfBars** parameter. New "pseudo-bars" are not created, but the current bar spacing selected and the **numberOfBars** determines how much empty space is padded to the right of the chart.

Example

```
protected override void Execute(){
    PadBars( 10 );
}
```

PlotFundamentalItems

void PlotFundamentalItems(ChartPane pane, string symbol, string itemName, Color color, LineStyle style, int width);
void PlotFundamentalItems(ChartPane pane, string itemName, Color color, LineStyle style, int width)

Plots historical fundamental data items onto the chart, on the specified **pane**. The desired items to plot are specified by the **itemName** parameter. If the desired items are symbol-specific, then use the version of the method that accepts a **symbol** parameter, and pass the stock symbol whose items you want to plot. The fundamental data is plotted in a special filled style, where the demarcation of each item is outlined in the specified **color**. When a new fundamental data item occurs, it can be clearly seen because this range is outlined using the **color**. The interior of the plotted area is filled with semi-transparent version of the **color** specified. Finally, plotted ranges are outlined using the indicated **width**.

Example

```
protected override void Execute(){
    ChartPane fPane1 = CreatePane( 25, true, false );
    ChartPane fPane2 = CreatePane( 25, true, false );

    // Plot IBM dividends on the chart of another stock

    if( (string)Bars.Symbol != "IBM" )
    {
        PlotFundamentalItems( fPane1, "IBM", "dividend", Color.Red, WealthLab.LineStyle.Solid, 1 );
    } else
    Abort();

    DataSeries divIBM = FundamentalDataSeries( "IBM", "dividend" );
```

```
    PlotSeries( fPane2, divIBM, Color.Red, WealthLab.LineStyle.Solid, 1 );
}
```

PlotSeries

void PlotSeries(ChartPane pane, DataSeries series, Color color, LineStyle style, int width);
void PlotSeries(ChartPane pane, DataSeries series, Color color, LineStyle style, int width, string label);

Plots the specified DataSeries (**series**) in the specified **pane** of the chart. The cosmetic appearance of the plotted DataSeries is controlled by the **color**, **style**, and **width** parameters.

Remarks

- When using the Histogram LineStyle, the **width** parameter determines the maximum width that each histogram bar is allowed to grow to. So, specify large values (such as 20) to allow the histogram bars to grow as you increase bar spacing.
- By default, the Description of the DataSeries is drawn as a label in the upper left corner of the pane. You can set the DataSeries' Description property to change this label, or use the overloaded version of the method.

Example

```
protected override void Execute(){
    // Plots KAMA series of Average prices
    PlotSeries( PricePane, KAMA.Series( ((High+Low)/2), 20 ), Color.Chocolate, WealthLab.LineStyle.Solid, 2 );
}
```

PlotSeriesDualFillBand

void PlotSeriesDualFillBand(ChartPane pane, DataSeries series1, DataSeries series2, Color fillColor1, Color fillColor2, Color color, LineStyle style, int width);
void PlotSeriesDualFillBand(ChartPane pane, DataSeries series1, DataSeries series2, Brush brush1, Brush brush2, Color color, LineStyle style, int width);

Plots and fills bands composed of two DataSeries (**series1** and **series2**) that periodically cross over each other, on the specified **pane**. Each individual DataSeries is plotted using the specified **color**, **style**, and **width**. The band between each DataSeries is filled with alternating colors (or brushes). **fillColor1** (or **brush1**) is used when **series1** is above **series2**, and **fillColor2** (or **brush2**) when **series2** is above **series1**.

Remarks

- Use **fillColors** and **brushes** that are semi-transparent, so that the chart bars can show up behind the filled bands.
- Histogram **style** does not make sense for filled bands, this **style** is treated as Solid by PlotSeriesDualFillBand.
- By default, the Description of the DataSeries is drawn as a label in the upper left corner of the pane. You can set the DataSeries' Description property to change this label.

Example

```
protected override void Execute(){
    // Plots and fills bands composed of Average Price and KAMA series that cross over each other.
    PlotSeriesDualFillBand( PricePane, KAMA.Series( Close, 10 ), ((High+Low)/2), Color.Red, Color.Blue, Color.Black, LineStyle.Solid, 1 );
}
```

PlotSeriesFillBand

void PlotSeriesFillBand(ChartPane pane, DataSeries upper, DataSeries lower, Color color, Color fillColor, LineStyle style, int width);
void PlotSeriesFillBand(ChartPane pane, DataSeries upper, DataSeries lower, Color color, Brush fillBrush, LineStyle style, int width);

Plots and fills an **upper** and **lower** band of two DataSeries, in the specified chart **pane**. The **upper** and **lower** bands are plotted using the specified **color**, **style** and **width**. The interior of the band is filled using the specified **fillColor**, or the specified **fillBrush**.

Remarks

- Use **fillColors** and **fillBrushes** that are semi-transparent, so that the chart bars can show up behind the filled band.
- Histogram **style** does not make sense for filled bands, this **style** is treated as Solid by PlotSeriesFillBand.
- By default, the Description of the DataSeries is drawn as a label in the upper left corner of the pane. You can set the DataSeries' Description property to change this label.

Example

```
protected override void Execute(){
    BBandLower bbL = BBandLower.Series( Close, 20, 2 );
    BBandUpper bbU = BBandUpper.Series( Close, 20, 2 );
    SolidBrush shadowBrush = new SolidBrush(Color.FromArgb(50, Color.Violet));
    PlotSeriesFillBand(PricePane, bbU, bbL, Color.Silver, shadowBrush, LineStyle.Solid, 2);
}
```

PlotSeriesOscillator

void PlotSeriesOscillator(ChartPane pane, DataSeries source, double overbought, double oversold, Color overboughtColor, Color oversoldColor, Color color, LineStyle style, int width);
void PlotSeriesOscillator(ChartPane pane, DataSeries source, double overbought, double oversold, Brush overboughtBrush, Brush oversoldBrush, Color color, LineStyle style, int width);

Plots the specified DataSeries (**source**) in the specified **pane**, using the provided **color**, **style** and **width**. Additionally, it allows you to define **overbought** and **oversold** levels. When the **source** moves below the **oversold** area, this area of the chart is filled using the **oversoldColor** (or **oversoldBrush**). Conversely, when the **source** moves above the **overbought** area, that area of the chart is filled with the **overboughtColor** (or **overboughtBrush**).

Remarks

- By default, the Description of the DataSeries is drawn as a label in the upper left corner of the pane. You can set the DataSeries' Description property to change this label.

Example

```
protected override void Execute(){
    //Create and plot RSI indicator
    RSI rsi = RSI.Series( Close, 14 );
    ChartPane rsiPane = CreatePane( 40, true, true );
    PlotSeriesOscillator( rsiPane, rsi, 70, 30, Color.Green, Color.Red, Color.CadetBlue, LineStyle.Solid, 1 );
}
```

PlotStops

void PlotStops();

Call **PlotStops** to cause stop and limit orders to be plotted on the chart. Stop and limit orders will appear as small colored dots on the chart, drawn at the appropriate bar and price levels. The stop/limit plots are color coded by order type:

- Buy = blue
- Sell = red
- Short = fuchsia
- Cover = green

Example

```
protected override void Execute(){
    // Displays Stop and Limit orders
    PlotStops();
    for(int bar = 20; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            if ( SellAtLimit( bar+1, LastPosition, Highest.Series( High, 20 )[bar] ) == null )
                SellAtStop( bar+1, LastPosition, Lowest.Series( Low, 20 )[bar] );
        }
        else
        {
            if ( BuyAtLimit( bar+1, Low[bar]-ATR.Series(Bars,5)[bar] ) == null )
                BuyAtStop( bar+1, High[bar]+ATR.Series(Bars,5)[bar] );
        }
    }
}
```

PlotSymbol

void PlotSymbol(ChartPane pane, Bars bars, Color upColor, Color downColor);

Plots the Bars object specified in the **bars** parameter onto the chart, in the indicated pane. To superimpose another symbol onto the price pane, use the **PricePane** property as the value of the **pane** parameter. The **pane** that is selected is automatically rescaled to support the range of the plotted data. The **upColor** and **downColor** parameters determine the colors that will be used to plot "up" bars (close greater than open) and "down" bars (close less than or equal to open).

Remarks

- The Bars object that is being plotted must be synchronized to the symbol being charted, or **PlotSymbol** will fail.
- Since it's not possible to align the x-axis for multiple charts, **PlotSymbol** will not work with the Trending Chart Styles.

Example

```
protected override void Execute(){    // Some cosmetics
    HideVolume();
    HidePaneLines();

    //Plot Microsoft data in a new pane
    Bars msft = GetExternalSymbol("MSFT", true);
    ChartPane msftPane = CreatePane( 100, false, true);
    PlotSymbol( msftPane, msft, Color.Silver, Color.Silver);

}
```

PlotSyntheticSymbol

void PlotSyntheticSymbol(ChartPane pane, string symbol, DataSeries open, DataSeries high, DataSeries low, DataSeries close, DataSeries volume, Color upBarColor, Color downBarColor);

Allows you to plot a synthetic symbol on the chart, in the specified **pane**. A synthetic symbol is composed of a group of DataSeries that represent the symbol's **open**, **high**, **low**, **close** and **volume**. The **symbol** parameter indicates a string that represents the name that should be applied to the synthetic symbol, this is plotted as a label on the chart. The **upBarColor** and **downBarColor** parameters determine the color to use when plotting up bars (close greater than open) and down bars (close less than or equal to open).

Remarks

- In order to plot a synthetic symbol, its constituent DataSeries must by synchronized to the main chart data that is already being plotted. If this is not the case, call Synchronize on each of the underlying DataSeries to synchronize them before plotting.
- Since it's not possible to align the x-axis for multiple charts, **PlotSyntheticSymbol** will not work with the Trending Chart Styles.

Example

```
protected override void Execute(){
    // Plot candle which consists of average Open/High/Low/Close values
    ChartPane SMAPane = CreatePane( 100, true, true );
    DataSeries O = SMA.Series( Open, 20 );
    DataSeries H = SMA.Series( High, 20 );
    DataSeries L = SMA.Series( Low, 20 );
    DataSeries C = SMA.Series( Close, 20 );
    PlotSyntheticSymbol( SMAPane, "SMACandle", O, H, L, C, null, Color.Blue, Color.Red );
}
```

PricePane Property

ChartPane PricePane

Returns the ChartPane where the Bars of the chart are plotted. You can use this pane to plot other indicators, such as moving averages and Bollinger Bands, or as a parameter to many other WealthScript cosmetic chart methods such as AnnotateBar and DrawPolygon.

Remarks

- Even if the Strategy is operating in a context that is not charted (such as the Strategy Monitor), this property will **not** return null.
- See the documentation on the ChartPane object for more information about its properties and methods.

SetBackgroundColor

SetBackgroundColor(int bar, Color color);

Sets the **color** that will be used to render the background of the chart at the individual **bar**. The background is colored from top to bottom, encompassing all panes on the chart.

Remarks

- Use SetPaneBackgroundColor to color the background of individual panes.

Example

```
protected override void Execute(){
    // Plot the weekly MACD in our daily chart
    SetScaleWeekly();
    DataSeries smaWeekly = SMA.Series( Close, 52 );
    RestoreScale();
    smaWeekly = Synchronize( smaWeekly );
    for(int bar = 52; bar < Bars.Count; bar++)
    {
        if ( Close[bar] > smaWeekly[bar] )
        {
            SetBackgroundColor( bar, Color.LightGreen );
        }
        else
```

```
        {
            SetBackgroundColor( bar, Color.LightPink );
        }
    }
}
```

SetBarColor

SetBarColor(int bar, Color color);

Sets the **color** that will be used to render the individual bar on the chart that is specified by the **bar** parameter.

Example

```
protected override void Execute(){
    // Color bars green when RSI < 20, otherwise
    // color up days blue and down days red }
    DataSeries hRSI = RSI.Series( Close, 14 );
    for(int bar = 50; bar < Bars.Count; bar++)
    {
        if ( hRSI[bar] < 60 )
            SetBarColor( bar, Color.Green );
        else
        if ( Close[bar] > Close[bar-1] )
        SetBarColor( bar, Color.Blue );
        else
        SetBarColor( bar, Color.Red );
    }
}
```

SetBarColors

void SetBarColors(Color colorUpBars, Color colorDownBars);

Changes that colors that will be used to plot the bars of the chart. "Up" bars are defined as close greater than open, and these bars will be colored using **colorUpBars**. Bars where close is less than or equal to open will be colored using **colorDownBars**.

Example

```
protected override void Execute(){
    SetBarColors( Color.Navy, Color.Maroon );
}
```

SetLogScale

void SetLogScale(ChartPane pane, bool logScale);

Turns semi-log scaling on or off for the specified **pane**. The **logScale** parameter indicates whether semi-logarithmic scaling should be applied to the **pane**.

Example

```
protected override void Execute(){
    SetLogScale( PricePane, true );
}
```

SetPaneBackgroundColor

void SetPaneBackgroundColor(ChartPane pane, int bar, Color color);

Changes the background color of the specified **pane**, for the specified **bar**, to the **color** indicated.

Example

```
protected override void Execute(){
    // Plot RSI and CMO, color backgrounds to show overbought/oversold levels
    ChartPane RSIPane = CreatePane( 30, true, true );
    ChartPane CMOPane = CreatePane( 30, true, true );
    PlotSeries( RSIPane, RSI.Series( Close, 14 ), Color.DarkBlue, WealthLab.LineStyle.Solid, 2 );
    PlotSeries( CMOPane, CMO.Series( Close, 14 ), Color.Blue, WealthLab.LineStyle.Solid, 2 );
    for(int bar = 20; bar < Bars.Count; bar++)
    {
        if ( RSI.Series( Close, 14 )[bar] < 30 )
            SetPaneBackgroundColor( RSIPane, bar, Color.LightGreen );
        else if ( RSI.Series( Close, 14 )[bar] > 70 )
            SetPaneBackgroundColor( RSIPane, bar, Color.LightPink );
        if ( CMO.Series( Close, 14 )[bar] < -50 )
            SetPaneBackgroundColor( CMOPane, bar, Color.LightGreen );
        else if ( CMO.Series( Close, 14 )[bar] > 50 )
            SetPaneBackgroundColor( CMOPane, bar, Color.LightPink );
    }
}
```

SetPaneMinMax

void SetPaneMinMax(ChartPane pane, double min, double max);

Allows you to set the scale of a particular **pane** manually. The minimum and maximum (**min** and **max**) values that you supply will be used to define the visible range of the pane. The actual visible scale of the pane will still dynamically adjust if the chart or plotted indicators extend beyond the range that you specify.

Example

```
protected override void Execute(){
    // Make sure a certain RSI range is always visible in the pane
    ChartPane RSIPane = CreatePane( 60, true, true );
    PlotSeries( RSIPane, RSI.Series( Close, 14 ), Color.Navy, WealthLab.LineStyle.Solid, 1 );
    SetPaneMinMax( RSIPane, 30, 70 );
}
```

SetSeriesBarColor

void SetSeriesBarColor(int bar, DataSeries ds, Color color);

Allows you to specify **colors** for individual **bars** of a specific DataSeries (**ds**) that is plotted on the chart.

Example

```
protected override void Execute(){
    // Color Bars of the indicator based on oversold/overbought levels
    DataSeries rsi = RSI.Series( Close, 14 );
    ChartPane rsiPane = CreatePane( 60, true, true );
    PlotSeries( rsiPane, rsi, Color.Gray, WealthLab.LineStyle.Solid, 2 );
    for(int bar = 50; bar < Bars.Count; bar++)
    {
        if ( rsi[bar] > 60 )
            SetSeriesBarColor( bar, rsi, Color.Red );
        else if ( rsi[bar] < 40 )
            SetSeriesBarColor( bar, rsi, Color.Blue );
    }
}
```

VolumePane Property

ChartPane VolumePane

Returns the ChartPane that the volume is being plotted in.

Remarks

- Even if the Strategy is operating in a context that is not charted (such as the Strategy Monitor), this property will **not** return null.
- See the documentation on the ChartPane object for more information about its properties and methods.

Data Access

The Data Access category contains methods you can use to access the data that the Strategy is currently operating on, and additional data that it might need.

Bars Property

Bars Bars

Returns the Bars object that the Strategy is currently operating on. Initially, this is the symbol being charted in the chart. But the Bars object can be changed via calls to SetContext, and the various SetScale methods to work in different time scales.

Remarks

- See the Bars object reference for a listing of available properties and methods on the Bars object.

ClearExternalSymbols

int ClearExternalSymbols();
int ClearExternalSymbols(string symbol);

Clears any external (secondary) symbol data. External symbols are obtained from calls to either GetExternalSymbol, SetContext, and from changing the data scale in the script. When external symbol data is requested in a script, Wealth-Lab caches the symbol data so that it does not have to be loaded again if the same symbol is requested later. This method allows you to clear this internal cache. The **symbol** parameter is optional. If it is specified, only that specific external symbol will be cleared. ClearExternalSymbols returns the number of symbols that were cleared from memory.

Example

```
protected override void Execute(){
    Bars msft = GetExternalSymbol( "MSFT", true );
    ChartPane msftPane = CreatePane( 50, true, true );
    PlotSymbol( msftPane, msft, Color.Blue, Color.Red );
    PrintDebug ( "Cleared ext. symbols: " + ClearExternalSymbols() );
}
```

Close Property

DataSeries Close

Returns a DataSeries object that represents the closing prices of the Bars object that the Strategy is currently operating on. You can also access the closing prices via the Bars.Close property. Access individual closing prices via the square bracket syntax:

```
//Access the close of the last bar
double lastClose = Close[Bars.Count - 1];
```

Remarks

- See the DataSeries object reference for a listing of available properties and methods on the DataSeries object.

Example

```
protected override void Execute(){    //Access closing price of the last bar
    double lastClose = Close[Bars.Count-1];
    // The string is output with 2 digits
    DrawLabel( PricePane, "Last close: " + String.Format( "{0:f}", lastClose ), Color.Black );
}
```

DataSetSymbols Property

IList<string> DataSetSymbols

Returns a list of strings that contain the symbols in the DataSet that contains the symbol currently being processed by the Strategy.

Remarks

- **Known issue**: When using *SetContext* in a DataSetSymbols loop, the dialog "Invalid Benchmark Buy and Hold Symbol:" is displayed when encountering a symbol with a *null* Bars object. This occurs even if Benchmark Buy & Hold is not enabled.
- **Known issue**: Synchronization issue with trading *DataSetSymbols* when the starting dates of the symbols' series are not the same, provided that some symbols are traded before the start dates of other symbols. For partial workarounds refer to this post.

Example

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

namespace WealthLab.Strategies
{
    public class AboveMA : WealthScript
    {
        private StrategyParameter paramPeriod;

        public AboveMA()
        {
            paramPeriod = CreateParameter("MA Period", 200, 10, 200, 5);
        }

        protected override void Execute()
        {
            int period = paramPeriod.ValueInt;
            DataSeries OverMA = new DataSeries( Bars, String.Concat("Number above MA (",period,")") );

            for(int ds = 0; ds < DataSetSymbols.Count; ds++)
            {
                SetContext( DataSetSymbols[ds], true );
                for(int bar = period; bar < Bars.Count; bar++)
                    if( Bars.Close[bar] > SMA.Series( Bars.Close,period )[bar] )
                        OverMA[bar]++;
                RestoreContext();
            }

            ChartPane omaPane = CreatePane(20,true,true);
            ChartPane omapctPane = CreatePane(20,true,true);
            DataSeries PctOverMA = OverMA/DataSetSymbols.Count*100;
```

```
            PctOverMA.Description = String.Concat( "Pct above MA (",period," )" );
            PlotSeries( omaPane, OverMA, Color.Blue, LineStyle.Histogram,2 );
            PlotSeries( omapctPane, PctOverMA, Color.DarkGreen, LineStyle.Solid,2 );
        }
    }
}
```

Date Property

IList<DateTime> Date

Returns a list of DateTime values that represents the historical date/times of the Bars object that the Strategy is currently operating on. You can also access the Dates via the Bars.Dates property. Access individual date values via the square bracket syntax:

```
//Access the last date being charted
DateTime lastDate = Date[Bars.Count - 1];
```

Example

```
protected override void Execute(){    //Access the last date being charted
    DateTime lastDate = Date[Bars.Count - 1];
    DrawLabel( PricePane, "Last trading date: " + String.Format( "{0:d}", lastDate ), Color.Black );
}
```

DateTimeToBar

int DateTimeToBar(DateTime date, bool exactMatch);

Returns the bar number (in the Bars object that the Strategy is currently operating on) that matches the DateTime provided in the **date** parameter. If **exactMatch** is true, the precise DateTime value must be located in the Bars object. Otherwise, the first bar whose DateTime is greater than or equal to the specified **date** is returned.

Example

```
protected override void Execute(){
    ClearDebug();
    DateTime d = new DateTime( 2008, 07, 15 );
    int Bar = DateTimeToBar( d, true );

    if( Bar == -1 )
        PrintDebug( "This bar does not exist in the chart" );
    else
    {
        for(int bar = 1; bar < Bars.Count; bar++)
            if ( bar == Bar )
                BuyAtClose( bar, "Buy" );
    }
}
```

GetExternalSymbol

Bars GetExternalSymbol(string symbol, bool synchronize);
Bars GetExternalSymbol(string dataSetName, string symbol, bool synchronize);

Returns a Bars object for the specified stock/futures **symbol**. The Bars object will be returned in the same data scale as the Bars object that the Strategy is currently operating on. If data for the specified symbol is not available, this method will raise an error. The synchronize method controls whether the returned Bars object will be automatically synchronized to the current Bars object. This is important if you want to plot the Bars, or indicators that are created from it. You can defer the synchronization by calling the Synchronize method at some later time.

Optional parameter **dataSetName** allows to specify the DataSet that will be searched when an external symbol is requested, which can differ from the DataSet of the symbol on which the Strategy is being executed.

Note: By specifying a DataSet with **dataSetName**, you are essentially specifying the associated Provider. If symbol is missing in the specified DataSet, but it's possible to find it in another DataSet of the same Provider, then it can be returned. This is in contrast to not specifying a DataSet in which case an alphabetical search through DataSets of any Provider is used to return the data.

Example

```
protected override void Execute(){
try
    {
        // Create a DataSeries with the relative strength of the current symbol vs. S&P 500
        Bars spy = GetExternalSymbol("SPY", true);
        DataSeries rs = DataSeries.Abs( Close / spy.Close );
        DataSeries rs3 = rs>>(21 * 3);
        rs.Description = "Relative strength " + Bars.Symbol + "/" + spy.Symbol;
        rs3.Description = "Relative strength " + Bars.Symbol + "/" + spy.Symbol + " (3 months ago)";

        HideVolume();
        ChartPane spyPane = CreatePane( 40, false, true );
        ChartPane rsPane = CreatePane( 40, false, true );
        PlotSymbol( spyPane, spy, Color.Blue,Color.Red);
        PlotSeries( rsPane, rs, Color.Black, LineStyle.Solid, 1 );
        PlotSeries( rsPane, rs3, Color.Blue, LineStyle.Solid, 1 );

        // Compare relative strength today and then 3 months ago
        for(int bar = 21 * 3; bar < Bars.Count; bar++)
        {
            // Highlight when the ratio today is greater than 3 months ago
            // and is also breaking a new 3 month relative high

            if( rs[bar] == Highest.Series( rs, 21 * 3 )[bar] )
                if( rs[bar] > rs3[bar] )
                {
                    SetBarColor( bar, Color.Blue );
                    rsPane.SetBackgroundColor( bar, Color.FromArgb(30,Color.Blue) );
                }
        }
    }
    catch
    {
        PrintDebug( "No data or could not find series" );
    }
}
```

GetSessionOpen Property

double GetSessionOpen(string symbol)

Returns current trading session's opening price for the specifed **symbol**. Returns 0 if the session's open price is not available or if the method is not supported by the DataSet's Data Provider.

GetSessionOpen() is designed primarily for EOD Strategies that require action based on the trading session's opening price (see Example). While it's still recommended to perform a Daily price update beforehand, you can schedule the Strategy Monitor to execute on the open of the market session. If you require a short delay to ensure all stocks have opened, you can right click the Strategy and choose "Run this Strategy now" or run the backtest in a Strategy Window.

Remarks

- GetSessionOpen() employs the Static Data Provider to return the session's opening price; implementations vary by Provider. For example, in response to the first call to GetSessionOpen(), the Fidelity Static Provider requests and caches the opening prices for all symbols in the DataSet. If the DataSet is large, processing the first symbol in the DataSet will appear slow while opening prices are collected.
- Intraday Strategies can directly obtain the opening price by finding the open price of the first bar of the current day, given by Open[bar - Bars.IntradayBarNumber(bar)] in a bar-indexed loop. Nonetheless, if the open price is required prior to the completion of the first intraday bar, consider using GetSessionOpen().

Example

This example shows how to employ GetSessionOpen() in a trading Strategy. For all bars prior to the last bar of the chart, the Strategy can access the opening price of the next bar. However, when processing the final bar of the chart, the open is obtained by calling GetSessionOpen(). If the value returned is not greater than 0, then the result is invalid and a trading Alert will not be processed. Otherwise, the Strategy will enable the BuyAtLimit order if the opening price of the "trade bar" is below the low of the last complete bar.

Example

```
protected override void Execute(){
double openTradeBar = 0;
            int lastCompleteBar = Bars.Count - 1;

            for (int bar = 0; bar < Bars.Count; bar++)
            {
                if (IsLastPositionActive)
                {
                    Position p = LastPosition;
                    if (bar - p.EntryBar > 1)
                        SellAtMarket(bar + 1, p);
                }
                else
                {
                    if (bar == lastCompleteBar)
                    {
                        openTradeBar = GetSessionOpen(Bars.Symbol);
                        DrawLabel(PricePane,"Today's Opening Price = " + openTradeBar.ToString("0.00"));
                    }
                    else
                        openTradeBar = Open[bar+1];

                    if (openTradeBar > 0 && openTradeBar < Low[bar])
                        BuyAtLimit(bar + 1, Low[bar] * 0.95);
                }
            }
        }
    }
}
```

High Property

DataSeries High

Returns a DataSeries object that represents the high prices of the Bars object that the Strategy is currently operating on. You can also access the high prices via the Bars.High property. Access individual high prices via the square bracket syntax:

```
//Access high price of the last bar
double lastHigh = High[Bars.Count - 1];
```

Remarks

- See the DataSeries object reference for a listing of available properties and methods on the DataSeries object.

Example

```
protected override void Execute(){    // Print high price of the last bar
    double high = High[Bars.Count-1];
    DrawLabel( PricePane, "High: " + String.Format( "{0:f}", high ), Color.Black );
}
```

Low Property

DataSeries Low

Returns a DataSeries object that represents the low prices of the Bars object that the Strategy is currently operating on. You can also access the low prices via the Bars.Low property. Access individual low prices via the square bracket syntax:

```
//Access low price of the last bar
double lastLow = Low[Bars.Count - 1];
```

Remarks

- See the DataSeries object reference for a listing of available properties and methods on the DataSeries object.

Example

```
protected override void Execute(){    // Print low price of the last bar
    double low = Low[Bars.Count-1];
    DrawLabel( PricePane, "Low: " + String.Format( "{0:f}", low ), Color.Black );
}
```

Open Property

DataSeries Open

Returns a DataSeries object that represents the open prices of the Bars object that the Strategy is currently operating on. You can also access the open prices via the Bars.Open property. Access individual open prices via the square bracket syntax:

```
//Access open price of the last bar
double lastOpen = Open[Bars.Count - 1];
```

Remarks

- See the DataSeries object reference for a listing of available properties and methods on the DataSeries object.

Example

```
protected override void Execute(){    // Print open price of the last bar
    double open = Open[Bars.Count-1];
    DrawLabel( PricePane, "Open price: " + String.Format( "{0:f}", open ), Color.Black );
}
```

RestoreContext

void RestoreContext();

Changes the context of the Strategy back to the Bars object that it was originally invoked on. In a Strategy window, this is the symbol that you are charting. The context can change to a different symbol by calling SetContext.

Remarks

- RestoreContext restores the context symbol only, but preserves changes in data scale that were made by calling the various SetScale methods.
- After you are through using an external symbol, you should call RestoreContext to restore the Strategy back to the original symbol.

Example

```
protected override void Execute(){
    // Compare the RSI of the stock with the RSI of QQQQ
    DataSeries QQQQ_RSI;
    SetContext( "QQQQ", true );
    QQQQ_RSI = RSI.Series( Close, 30 );
    RestoreContext();
    HideVolume();
    ChartPane qqqq = CreatePane( 50, true, true );
    ChartPane rsi = CreatePane( 50, false, true );
    PlotSeries( qqqq, QQQQ_RSI, Color.Red, WealthLab.LineStyle.Solid, 2 );
    PlotSeries( rsi, RSI.Series( Close, 30 ), Color.Blue, WealthLab.LineStyle.Solid, 2 );
}
```

SetContext

void SetContext(string symbol, bool synchronize);
void SetContext(Bars bars);

Sets the "context" of the Strategy to the **symbol** specified. This means that subsequent Positions (BuyAtMarket, ShortAtMarket, etc.) will be entered on the new **symbol**. Use this technique to backtest pairs trading Strategies, for example. The **synchronize** parameter specifies whether the data for the new **symbol** will be automatically date-synchronized with the primary symbol (the one that the Strategy was originally executed on, and charted.) If you defer synchronization, you can synchronize at a future point using the Synchronize function. This is required if you want to plot the **symbol**, or indicators produced from it. If data for the specified **symbol** is not available, SetContext will throw an error.

Remarks

- Call the method without the optional **synchronize** parameter for automatic synchronization.
- Call RestoreContext to restore the context to the original symbol.
- The SetContext(*Bars*) overload was introduced to allow trades on the synthetic option symbol (but is not limited to this scenario). It's always synchronized to the primary symbol.

Example

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

namespace WealthLab.Strategies
{
    public class MyStrategy : WealthScript
    {
        protected override void Execute()
        {
            SortedList Vlst = new SortedList( DataSetSymbols.Count );

            // Collect turnover values for the DataSet
            for(int ds = 0; ds < DataSetSymbols.Count; ds++)
            {
                SetContext( DataSetSymbols[ds], true );
                Vlst.Add( ds, Close[Bars.Count-1] * Volume[Bars.Count-1] );
                RestoreContext();
            }

            ICollection v = Vlst.Keys;
            foreach( int str in v )
                PrintDebug( DataSetSymbols[str] + " -- $" + Vlst[str] );
        }
    }
}
```

Synchronize

DataSeries Synchronize(DataSeries source);
Bars Synchronize(Bars source);

Returns a new DataSeries or Bars object, based on the **source** DataSeries or Bars object. The new DataSeries or Bars object is date-synchronized to the primary symbol that the Strategy was executed on. Use this method when you need to synchronize an external symbol acquired by calling SetContext or GetExternalSymbol, or any indicators created from them, for plotting. Also, use this method to expand a compressed DataSeries or Bars object, such as those obtained after calls to SetScaleWeekly or SetScaleMonthly. You must synchronize DataSeries and Bars objects if you want to plot them on the chart, or use them in operations with the primary Bars object, or indicators created from it.

Remarks

- If the primary symbol contains dates that occur before the first date in the **source** DataSeries or Bars, the new DataSeries or Bars will contain zero values for these dates.
- If the primary symbol contains dates that do not occur within the **source** DataSeries or Bars, they will be inserted, and the previous available value used for these dates.
- If the **source** contains dates that do not occur in the primary symbol, these values will be eliminated from the new DataSeries or Bars.

Example

```
protected override void Execute(){
    Bars msft = new Bars( "MSFT", BarScale.Daily, 1 );
    try
    {
        msft = GetExternalSymbol( "MSFT", false );
    }
    catch
    {
        PrintDebug( "No MSFT data?" );
```

```
    }

    msft = Synchronize( msft );
    ChartPane msftPane = CreatePane( 50, true, true );
    PlotSymbol( msftPane, msft, Color.Blue, Color.Red );
    PlotSeries( msftPane, SMA.Series( msft.Close,10 ), Color.Red, WealthLab.LineStyle.Solid, 1 );
}
```

TrendlineValue

double TrendlineValue(int bar, string trendlineName);

Provides access to the specified manually drawn trendline by name, and returns the value of that trendline at the specified bar on the chart. If the named trendline could not be found for the current symbol and time scale, the method returns 0.

Remarks

- In a Strategy that loops through DataSet symbols using **SetContext**, **TrendlineValue** is not working. If you click on the symbol with the trendline, TrendlineValue code is executed whereas when you click on another symbol to start the Strategy, **TrendlineValue** returns zero.

  This is by design, and the reason is that it's not possible to access a Trendline Value for other symbols through **SetContext** since the TrendLine(s) don't exist in the current chart.

Example

```
protected override void Execute(){
    double res;
    for(int bar = 1; bar < Bars.Count; bar++)
    {
        res = TrendlineValue( bar-1, "Resistance" );
        if ( Close[bar-1] < res )
        {
            res = TrendlineValue( bar, "Resistance" );
            if ( Close[bar] >= res )
            {
                SetBarColor( bar, Color.Green );
                DrawCircle( PricePane, 5, bar, res, Color.Red, Color.Red, WealthLab.LineStyle.Solid, 5, false );
            }
        }

    }
}
```

Volume Property

DataSeries Volume

Returns a DataSeries object that represents the volume of the Bars object that the Strategy is currently operating on. You can also access the volume via the Bars.Volume property. Access individual volume values via the square bracket syntax:

```
//Access the volume of the first bar
double firstBarVolume = Volume[0];
```

Remarks

- See the DataSeries object reference for a listing of available properties and methods on the DataSeries object.

Example

```
protected override void Execute(){    // Get turnover on the last bar ( close * volume )
    double turnover = Close[Bars.Count-1] * Volume[Bars.Count-1];
    // Print stock turnover
    DrawLabel( PricePane, "Turnover: " + String.Format( "${0:0,0}", turnover ), Color.Black );
}
```

DataSeries Object

DataSeries(string description)
DataSeries(Bars bars, string description)
DataSeries(DataSeries source, string description)

The DataSeries object represents a series of values (double type) with corresponding dates. You can perform mathematical operations on DataSeries as if they were primary types:

```
//Get average of high and low price
DataSeries avg = (High + Low) / 2;
```

You can also create offset copies of a DataSeries by using the shift operators, >> and <<:

```
//Shift a moving average 2 bars to the right
DataSeries shiftedMA = SMA.Series(Close, 14) >> 2;
```

The most practical use of the DataSeries function is to return a new zero-filled DataSeries that has the same number of elements as the Bars parameter. Use the result to fill the series bar by bar with calculated values:

```
protected override void Execute()
{
    // Create a zero-filled series
    DataSeries binSeries = new DataSeries(Bars, "Binary Series");
    // Fill the series with values based on some logic
    for(int bar = 1; bar < Bars.Count; bar++)
    {
        if( High[bar] > High[bar-1] )
            binSeries[bar] = 1;
        else if( Low[bar] < Low[bar - 1] )
            binSeries[bar] = -1;
    }
    ChartPane cp = CreatePane(40, true, false);
    PlotSeries(cp, binSeries, Color.Black, LineStyle.Histogram, 20);
}
```

[] indexer Property

double this[int i]

Allows you to access one of the values in the DataSeries. The DataSeries contains a number of values, indexed between 0 and Count - 1. Provide the index number of the value you want to access in the **i** parameter.

```
//Access the first and last values of a DataSeries (ds)
double firstValue = ds[0];
double lastValue = ds[ds.Count - 1];
```

Abs

static DataSeries Abs(DataSeries source);

This class level (static) method returns a new DataSeries that is the absolute value of the specified **source** DataSeries.

Example

```
protected override void Execute(){    //Return the absolute value of Rate of Change
    //DataSeries roc = ROC.Series( Close, 20 );
    DataSeries absRoc = DataSeries.Abs( ROC.Series( Close, 20 ) );
    DrawLabel( PricePane, "Abs(ROC) value on last bar: " + String.Format( "{0:f}", absRoc[absRoc.Count-1] ), Color.RoyalBlue );
}
```

Count Property

int Count

Returns the number of values contained in the DataSeries. The values are accessed by index, starting at index 0 and ending at index Count - 1.

Example

```
protected override void Execute(){
    for(int bar = 20; bar < Bars.Count; bar++)
    {
        // Your trading system rules
    }
}
```

Date Property

IList<DateTime> Date

Returns the list of DateTimes that are associated with the values contained in the DataSeries. The number of DateTimes in the Date list is always equal to the number of values contained in the DataSeries.

```
//Access the first and last dates contained in the DataSeries (ds)
DateTime dtFirst = ds.Date[0];
DateTime dsLast = ds.Date[ds.Count - 1];
```

Example

```
protected override void Execute(){
    //Access the first and last dates
    DateTime firstDate     = Date[0];
    DateTime lastDate      = Date[Bars.Count-1];
    DrawLabel( PricePane, "First trading date: " + String.Format( "{0:d}", firstDate ), Color.Black );
    DrawLabel( PricePane, "Last trading date: " + String.Format( "{0:d}", lastDate ), Color.Black );
}
```

Description Property

string Description

Represents the description associated with the DataSeries. You can change the description by assigning a different string value to this property. The Description is shown as a label in charts when the DataSeries is plotted, and appears in the tooltip that is visible when you move the mouse over an indicator on the chart.

Example

```
protected override void Execute(){
    // Average price series
    DataSeries average = (High + Low) / 2;
    average.Description = "Average Price" ;
    PrintDebug ( average.Description );
}
```

FirstValidValue Property

int FirstValidValue

Returns the bar number of the first "valid" value contained in the DataSeries. When an indicator (all indicators are DataSeries) is plotted, the plotting actually begins at the FirstValidValue. For previous bars, the indicator is not plotted. For example, the FirstValidValue of a 30 bar moving average would be bar number 29.

Example

```
protected override void Execute(){
    // Some price series
    DataSeries mySeries = SMA.Series( Close, 100 );
    PlotSeries( PricePane, mySeries, Color.Blue, WealthLab.LineStyle.Solid, 2 );
    ClearDebug();
    PrintDebug( mySeries.FirstValidValue );

    // Sets the trading loop to the first "valid" value of the DataSeries
    for(int bar = mySeries.FirstValidValue; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            if( CrossUnder( bar, High, mySeries[bar] ) )
                SellAtMarket( bar+1, LastPosition, "Exit Long" );
        }
        else
        {
            if( CrossOver( bar, Low, mySeries[bar] ) )
            BuyAtMarket( bar+1, "Enter Long" );
        }
    }
}
```

MaxValue Property

double MaxValue

Returns the maximum (highest) value that exists in the entire DataSeries.

Example

```
protected override void Execute(){
    // Show highest price
    DataSeries mySeries = Highest.Series( High, 1 );
    PrintDebug( "Highest price registered = " + mySeries.MaxValue );
}
```

MinValue Property

double MinValue

Returns the minimum (lowest) value that exists in the entire DataSeries.

Example

```
protected override void Execute(){
    // Show lowest price
    DataSeries mySeries = Lowest.Series( Low, 1 );
    PrintDebug( "Lowest price registered = " + mySeries.MinValue );
}
```

Partial Property

double PartialValue

Contains the value based on the partial bar in a streaming chart. In a streaming chart, a partial bar is visible at the far right end of the chart, containing the partial values for open, high, low, and close. Certain indicators can also update based on partial values.

Remarks

- If the DataSeries does not have a partial value available, the PartialValue property returns Double.NaN (not a number).

Example

```
protected override void Execute(){
// How to access the opening price of an incomplete (Ghost) bar
    // Helpful when trading gaps and in Opening Range Breakout strategies etc.

    // Run this Strategy on Daily scale shortly after market opens in Streaming mode
    // Your Streaming data provider should support partial values (e.g. Fidelity, Yahoo)

    if( Bars.Scale != BarScale.Daily )    {
        DrawLabel(PricePane, "To be used on Daily scale", Color.Red);
        return;
    }

    if (!IsStreaming)    {
        DrawLabel( PricePane, "Enable Streaming first", Color.Red);
    }
    else {
        for (int bar = 1; bar < Bars.Count; bar++)
        {
            double OpenPrice = (bar < Bars.Count - 1 ? Open[bar + 1] : Open.PartialValue);

            // Open.PartialValue equals double.NaN if streaming is not enabled
            if (OpenPrice == double.NaN)
                continue;

            if( bar == Bars.Count-1 )
                DrawLabel( PricePane, OpenPrice.ToString() );
        }
    }
}
```

Fundamental Data

The Fundamental Data category consists of methods you can use to access and manipulate fundamental data.

FundamentalDataItems

IList<FundamentalItem> FundamentalDataItems(string itemName);
IList<FundamentalItem> FundamentalDataItems(string symbol, string itemName);

The FundamentalDataItems object represents a collection of FundamentalItem objects.

Example

```
protected override void Execute(){
    // Show the list of the fundamental data item "assets"

    const char tab = '\u0009';
    string item = "assets";
    IList<FundamentalItem> fList = FundamentalDataItems(item);
    ClearDebug();
    PrintDebug(Bars.Symbol + tab + "Item Count: " + fList.Count);
    PrintDebug("FY" + tab + "FQ" + tab + "Bar" + tab + "Date      " + tab + "Value" + tab + "FormatValue");
    foreach (FundamentalItem fi in fList)
    {
        PrintDebug(fi.GetDetail("fiscal year") + tab
            + fi.GetDetail("current quarter") + tab
            + fi.Bar.ToString() + tab
            + fi.Date.ToShortDateString() + tab
            + fi.Value.ToString("$#,0.00")
            + tab + fi.FormatValue().Replace("\n", " ") );
    }
}
```

FundamentalDataSeries

DataSeries FundamentalDataSeries(string itemName);
DataSeries FundamentalDataSeries(string itemName, int offset);
DataSeries FundamentalDataSeries(string itemName, int aggregate, int offset);
DataSeries FundamentalDataSeries(string itemName, int aggregate, bool average, int offset);
DataSeries FundamentalDataSeries(string symbol, string itemName);
DataSeries FundamentalDataSeries(string symbol, string itemName, int aggregate, bool average, int offset);

Returns a DataSeries that represents the fundamental data for the specified "fundamental **itemName**". To access symbol-specific fundamental data without calling SetContext, use one of the last two overload signatures to pass the stock **symbol**. The individual fundamental data items are synchronized to the Bars object that the Strategy is currently operating on, and the fundamental data values become the values of the resulting DataSeries. Bars that do not contain any fundamental data items at their specific date will carry over the value of the previous, most recent fundamental data item. The resulting DataSeries can be plotted on the chart and manipulated as any other normal DataSeries.

Aggregate, average, and offset parameters perform those operations on the itemName Fundamental DataSeries in the order in which they appear in the parameter list. For example, a Fundamental DataSeries will first be aggregated by the number of integer aggregate periods specified. Then, if average is true, the average of the aggregate series is returned. Finally, the series is offset forward in time by the number of offset periods, where the period depends on the specified itemName. (Generally, the period is quarterly for corporate fundamental items.) By offsetting, it's easy to determine quarter-over-quarter or year-over-year changes.

Remarks

- See the documentation for the DataSeries object for more information about its properties and methods.
- See the Fundamental Data Guide under the Help menu for a list of fundamental data **itemNames** available.
- Use 0 or a positive number for aggregate and offset parameters.

Example

```
protected override void Execute(){
    //  55/34 Breakout strategy with an asset twist

    ChartPane fundPane = CreatePane(40, true, false);
    // Preferred plot method for Fundamental data items
    PlotFundamentalItems(fundPane, "assets", Color.Green, WealthLab.LineStyle.Invisible, 1);

    // Plot "assets" in a time series
    DataSeries assets = FundamentalDataSeries("assets");
    PlotSeries(fundPane, assets, Color.Black, WealthLab.LineStyle.Solid, 1);

    // Delay highest and lowest series by 1 bar to check Closing crossovers/unders
    DataSeries highest = Highest.Series(High, 55) >> 1;
    DataSeries lowest = Lowest.Series(Low, 34) >> 1;

    PlotSeries(PricePane, highest, Color.Green, WealthLab.LineStyle.Dashed, 1 );
    PlotSeries(PricePane, lowest, Color.Red, WealthLab.LineStyle.Dashed, 1 );

    for(int bar = 20; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive) {
            // exit if assets drop below $50M
            if (CrossUnder(bar, assets, 50000d))
                SellAtMarket(bar + 1, LastPosition, "assets < $50M");
            else if (CrossUnder(bar, Close, lowest))
                SellAtMarket(bar + 1, LastPosition, "lowest 34");
        }
        else {
            // Trade this instrument only if assets are over $50M
            if (assets[bar] > 50000d)
                if (CrossOver(bar, Close, highest))
                    BuyAtMarket(bar + 1);
        }
    }
}
```

FundamentalDataSeriesAnnual

DataSeries FundamentalDataSeriesAnnual (string itemName, int offset);
DataSeries FundamentalDataSeriesAnnual (string symbol, string itemName, int offset);

Returns a DataSeries that sums the 4 quarters of the Fiscal Year. This function is used in the ratio rules that are based on annual growth rates.

Remarks

- **Known issue**: *FundamentalDataSeriesAnnual* overload with symbol name ignores the symbol parameter, executing on the current (primary) Bars.
    - *Workaround:* Wrap the call to *FundamentalDataSeriesAnnual(symbol...)* in *SetContext/RestoreContext.*
- **Known issue**: *FundamentalDataSeriesAnnual* may return zero if the most recently reported fiscal quarter is FQ4.

- *Workarounds*: Call this function in place of *FundamentalDataSeriesAnnual*():

```
public DataSeries FDSeriesAnnual(string symbol, string item, int offset)
{
int bar = Bars.Count - 1;
DataSeries result = FundamentalDataSeriesAnnual(symbol, item, offset);
int fq = (int)GetFundamentalItem(bar, symbol, "fiscal quarter").Value;
while (fq == 4 &amp;amp;&amp;amp; bar > 0)
{
result[bar] = FundamentalDataSeries(symbol, item, 4, false, offset * 4)[bar];
bar--;
fq = (int)GetFundamentalItem(bar, symbol, "fiscal quarter").Value;
}
return result;
}
```

You can also use the aggregate overload for *FundamentalDataSeries* to add the last rolling 4 quarters of a fundamental item:

```
string eps = "earnings per share";
//Replace this:
//PlotSeries(FundPane,FundamentalDataSeriesAnnual(eps, 0), Color.DeepPink, LineStyle.Solid, 2);
//With this:
PlotSeries(FundPane,FundamentalDataSeries(eps, 4, false, 0), Color.DeepPink, LineStyle.Solid, 2);
```

Example

```
protected override void Execute(){
    //Calculate and plot the percentage annual earning growth
    DataSeries income = FundamentalDataSeriesAnnual( "net income", 0 );
    DataSeries income2 = FundamentalDataSeriesAnnual( "net income", 1 );
    DataSeries EG = 100 * (income - income2) / DataSeries.Abs(income2);

    // Plot the annualized income
    ChartPane annualIncomePane = CreatePane(40, true, true);
    PlotSeries( annualIncomePane, income, Color.Blue, WealthLab.LineStyle.Solid, 2);

    // Plot the earnings growth
    EG.Description = "Annual Earnings Growth %";
    ChartPane egPane = CreatePane(40, true, true );
    PlotSeries(egPane, EG, Color.Green, WealthLab.LineStyle.Histogram, 2);
}
```

GetFundamentalItem

FundamentalItem GetFundamentalItem (int bar, string symbol, string itemName);

Returns the FundamentalItem object for itemName that corresponds to the specified bar and symbol.

Remarks

- To avoid runtime errors, test for a null object before using the result.
- A FundamentalItem "off the chart" can be returned, and in this case the its *Bar* property will be set to -1.

Example

```
protected override void Execute(){
int bar = Bars.Count - 1;
    FundamentalItem fi = GetFundamentalItem(bar, Bars.Symbol, "assets");
    if (fi != null)
        DrawLabel(PricePane, "Current assets: " + fi.Value.ToString("$#,0") + " (millions)", Color.Blue);
    else
        DrawLabel(PricePane, "Current assets: not available", Color.Blue);
}
```

GetNextFundamentalItem

FundamentalItem GetNextFundamentalItem (int bar, string symbol, string itemName);

Returns the FundamentalItem object for itemName following the one that corresponds to the specified bar and symbol.

Remarks

- To avoid runtime errors, test for a null object before using the result.
- A FundamentalItem "off the chart" can be returned, and in this case the its *Bar* property will be set to -1.

Example

```
protected override void Execute(){
int bar = 0;
    FundamentalItem fi = GetNextFundamentalItem(bar, Bars.Symbol, "assets");
    if (fi != null)
        DrawLabel(PricePane, "first assets report in chart range " + fi.Date.ToShortDateString() + ": " + fi.Value.ToString("$#,0") + " (millions)", Color.Blue);
    else
        DrawLabel(PricePane, "assets: not available", Color.Blue);
}
```

FundamentalItem Object

The FundamentalItem object represents a single instance of a fundamental data point.

Bar Property

int Bar

Returns the bar number of the chart to which the FundamentalItem is synchronized.

Example (see FundamentalDataItems)

Date Property

DateTime Date

Returns the report Date corresponding to the FundamentalItem.

Example (see FundamentalDataItems)

FormatValue Method

string FormatValue

Returns a string containing a summary of the FundamentalItem's data.

Example (see FundamentalDataItems)

GetDetail Method

string GetDetail(string detailName);

Returns a string value of the specified detailName. Valid detailNames for the various fundamental types are as follows:

Analyst Ratings:
"firm name"; "normalized rating"; "action code"; "prev normalized rating"; "analyst name";

Economics
"period"; "observation date"; "ref id";
"period"; can be: "annual"; "yearly"; "monthly"; "quarterly"; "semi-annual"; "weekly";

Estimated Earnings
"period"; "current quarter"; "fiscal year"; "calendar year"; "current qtr month"; "sum TTM mean value"; "sum FTM mean value";
"period"; can be: "annual"; "yearly"; "monthly"; "quarterly"; "semi-annual"; "weekly";

Fundamentals
"period"; "current quarter"; "fiscal year";
"period"; can be: "annual"; "yearly"; "monthly"; "quarterly"; "semi-annual"; "weekly";

Earnings per Share
"period"; "current quarter"; "fiscal year";
"period"; can be: "annual"; "yearly"; "monthly"; "quarterly"; "semi-annual"; "weekly";

Insider Transactions
"transaction type"; "count"; "shares"; "insider"; "title";
"transaction type" can be: "B"; "S"; "N";
Since multiple transactions are stored per each item, if "transaction type" equals "B" or "S", then a count is appended to "shares", "insider", and "title"; e.g. "shares1", "insider1", and "title1"; up to "count". For "net insider transactions", "transaction type" has the same treatment.

Splits and Dividends:
"DPC"

Example (see FundamentalDataItems)

Name Property

string Name

Returns a string containing the itemName of the FundamentalItem.

Value Property

double Value

Returns a unit-less floating point value associated with the FundamentalItem. (Most FundamentalItem values are expressed in millions.)

Example (see FundamentalDataItems)

MarketInfo Class

The MarketInfo class represents a single market, including its open and closing times, the time zone it trades in, and its holidays and days that have special open and close times.

CloseTimeNative Property

DateTime CloseTimeNative

Return the time that the market typically closes, in its native time zone.

Example (see SpecialHours)

Description Property

string Description

Return a brief description of the market.

Holidays Property

List<DateTime> Holidays

Return a list of DateTime objects that represent the market's holidays; days that it does not trade.

Example

```
protected override void Execute(){
char tab = '\t';
            PrintDebug( "Total Holiday count: "+ Bars.MarketInfo.Holidays.Count );
            PrintDebug( "-----------------------" );
            PrintDebug( "Date" + tab + "Market Name" );
            foreach (DateTime dt in Bars.MarketInfo.Holidays)
                PrintDebug( dt.ToShortDateString() + tab + Bars.MarketInfo.Name.ToString() );

        }
    }
}
```

Name Property

string Name

Return the name of the market.

OpenTimeNative Property

DateTime OpenTimeNative

Return the standard time that the market opens for trading, in its native time zone.

Example (see SpecialHours)

SpecialHours Property

List<MarketSpecialHours> SpecialHours

Return a list of MarketSpecialHours objects that represent days where the market has special trading hours. The MarketSpecialHours class contains only three properties, all DateTimes: Date, OpenTimeNative, and CloseTimeNative.

Example

```
protected override void Execute(){
char tab = '\t';
            PrintDebug( "Total Shortened Session count: "+ Bars.MarketInfo.SpecialHours.Count );
            PrintDebug( "-----------------------" );
            PrintDebug( "Date" + tab + tab + "Open" + tab + "Close" + tab + "Market Name" );
            foreach (MarketSpecialHours sh in Bars.MarketInfo.SpecialHours)
                PrintDebug( sh.Date.ToShortDateString() + tab + sh.OpenTimeNative.ToString("t") + tab +
                    sh.CloseTimeNative.ToString("t") + tab + Bars.MarketInfo.Name.ToString() );

        }
    }
}
```

TimeZoneName Property

string TimeZoneName

Return the Windows string representing the time zone that the market trades in. For example, for EST, return "Eastern Standard Time".

LinearRegLine

double LinearRegLine(DataSeries series, int bar1, int bar2, double predict);

Allows you to perform ad-hoc linear regression analysis on the specified DataSeries. Specify the Start and End bars (**bar1** and **bar2**) for which to calculate the regression line. Then, specify the bar, **predict**, for which you want to predict a value. This could be a bar that extends into the future.

Example

```
protected override void Execute(){
int StBar = 0; int EndBar = 0;
            double Diff = 0; double MaxDiff = 0;

            for(int bar = 20; bar < Bars.Count; bar++)
            {
                if (IsLastPositionActive)
                {
                    /* Exit after N days */

                    Position p = LastPosition;
                    if ( bar+1 - p.EntryBar >= 20 )
                        SellAtMarket( bar+1, p, "Timed" );
                }
                else
                {
                    BuyAtLimit( bar+1, Close[bar]*0.95 );
                }
            }

            /* Highlight the regression channel of winning trades */

            foreach( Position p in Positions )
            {
                if( p.NetProfit > 0 )
                {
                    StBar = p.EntryBar;
                    EndBar = p.ExitBar;
                    if( EndBar == -1 )
                        EndBar = Bars.Count - 1;
                    double StPnt = LinearRegLine( Close, StBar, EndBar, (double)StBar );
                    double EnPnt = LinearRegLine( Close, StBar, EndBar, (double)EndBar );
                    MaxDiff = 0;

                    for(int bar = StBar; bar <= EndBar; bar++)
                    {
                        Diff = Math.Abs( Close[bar] - LinearRegLine( Close, StBar, EndBar, (double)bar ) );
                        MaxDiff = Math.Max( Diff, MaxDiff );
                    }

                    double[] rectangle = { StBar, StPnt - MaxDiff, StBar, StPnt + MaxDiff, EndBar, EnPnt + MaxDiff, EndBar, EnPnt - MaxDiff };

                    DrawPolygon( PricePane, Color.FromArgb( 80, Color.LightGreen ), Color.FromArgb( 80, Color.LightGreen ),
                        LineStyle.Solid, 1, true, rectangle );
                }
            }
        }
    }
}
```

LineExtendX

double LineExtendX(double x1, double y1, double x2, double y2, double y);

Extends the line specified by the x1, y1 and x2, y2 parameters, solving for x using the specified y parameter.

Remarks

- The equation used in the solution assumes a linear (not logarithmic) y-scale axis.

Example

```
protected override void Execute(){
    // Determine middle bar between last 2 peaks
    int bar = Bars.Count-1;
    double rev = 5;

    int bar1 = (int)PeakBar.Value( bar, High, rev, WealthLab.Indicators.PeakTroughMode.Percent );
    double price1 = Peak.Value( bar, High, rev, WealthLab.Indicators.PeakTroughMode.Percent );
    int bar2 = (int)PeakBar.Value( bar1, High, rev, WealthLab.Indicators.PeakTroughMode.Percent );
    double price2 = Peak.Value( bar1, High, rev, WealthLab.Indicators.PeakTroughMode.Percent );
    double price3 = ( price1 + price2 ) / 2;
    int bar3 = (int) LineExtendX( bar1, price1, bar2, price2, price3 );

    SetBarColor( bar3, Color.Red );
    if( ( bar2 > -1 ) & ( bar1 > -1 ) )
        DrawLine( PricePane, bar1, price1, bar2, price2, Color.Blue, WealthLab.LineStyle.Solid, 1 );
}
```

LineExtendY

double LineExtendY(double x1, double y1, double x2, double y2, double x);

Extends the line specified by the x1, y1 and x2, y2 parameters, solving for y using the specified x parameter.

Remarks

- The equation used in the solution assumes a linear (not logarithmic) y-scale axis.
- Use the **LineExtendYLog** method for logarithmic y-scale axis.

Example

```
protected override void Execute(){
    // Extend recent resistance line to most current bar

    int bar = Bars.Count-1;
    double rev = 5;

    int bar1 = (int)PeakBar.Value( bar, High, rev, WealthLab.Indicators.PeakTroughMode.Percent );
    double price1 = Peak.Value( bar, High, rev, WealthLab.Indicators.PeakTroughMode.Percent );
    int bar2 = (int)PeakBar.Value( bar1, High, rev, WealthLab.Indicators.PeakTroughMode.Percent );
    double price2 = Peak.Value( bar1, High, rev, WealthLab.Indicators.PeakTroughMode.Percent );
    double price3 = LineExtendY( bar1, price1, bar2, price2, bar );

    DrawLine( PricePane, bar1, price1, bar2, price2, Color.Blue, WealthLab.LineStyle.Solid, 1 );
    DrawLine( PricePane, bar2, price2, bar, price3, Color.Red, WealthLab.LineStyle.Solid, 1 );
}
```

LineExtendYLog

double LineExtendYLog(double x1, double y1, double x2, double y2, double x);

Extends the line specified by the x1, y1 and x2, y2 parameters, solving for y using the specified x parameter. A logarithmic y-scale axis is assumed.

Example

```
protected override void Execute(){
    // Extend recent resistance line to most current bar

    int bar = Bars.Count-1;
    double rev = 5;

    int bar1 = (int)PeakBar.Value( bar, High, rev, WealthLab.Indicators.PeakTroughMode.Percent );
    double price1 = Peak.Value( bar, High, rev, WealthLab.Indicators.PeakTroughMode.Percent );
    int bar2 = (int)PeakBar.Value( bar1, High, rev, WealthLab.Indicators.PeakTroughMode.Percent );
    double price2 = Peak.Value( bar1, High, rev, WealthLab.Indicators.PeakTroughMode.Percent );
    double price3 = LineExtendYLog( bar1, price1, bar2, price2, bar );

    DrawLine( PricePane, bar1, price1, bar2, price2, Color.Blue, WealthLab.LineStyle.Solid, 1 );
    DrawLine( PricePane, bar2, price2, bar, price3, Color.Red, WealthLab.LineStyle.Solid, 1 );
}
```

Options

The Options category contains helper methods you can use for trading options.

Remarks

- **Known issue**: The OHLC values of synthetic put contracts are incorrect (unlike calls)
- **Known issue**: If the price of the underlying is above the Put's strike prior to bar 30, then the Put is priced at zero. The opposite is true for calls.

CreateSyntheticOption

Bars CreateSyntheticOption(DateTime startDate, DateTime expiryDate, double strikePrice, bool isCallOption);
Bars CreateSyntheticOption(int asOfBar, int atLeastXDaysTilExpiration, int daysToPlotBeforeCreation, bool isCallOption);

Creates and returns a Bars object for a synthetic option contract of the symbol currently being processed with the specified parameters:

- start date, expiry date, strike price.
- as of bar (creation date), number of calendar days to allow until expiration, number of calendar days to plot before creation date

The boolean parameter **isCallOption** specifies whether the option will be a call (true) or a put (false) contract. The resulting Bars object's symbol has the following components: *!_symbol_strike_YYMMDD_optionType*, where YYMMDD is the monthly expiry and type is either *CALL* or *PUT*.

Remarks

- The first overload accepts any valid date for expiryDate
- Unless specified by the first overload, monthly contract expiry dates are assumed
- The Bars object will be returned in the same data scale as the Bars object that the Strategy is currently operating on.
- When using the second overload method, the strike for both puts and calls is implicitly derived from the integer part of the closing price on the bar passed to the asOfBar parameter.

Note: Live options trading is not supported by Wealth-Lab. This method returns hypothetical data.

Disclaimer: Backtesting provides a hypothetical calculation of how a security or portfolio of securities, subject to a trading strategy, would have performed over a historical time period. You should not assume that backtesting of a trading strategy will provide any indication of how your portfolio of securities, or a new portfolio of securities, might perform over time. You should choose your own trading strategies based on your particular objectives and risk tolerances. Be sure to review your decisions periodically to make sure they are still consistent with your goals. Past performance is no guarantee of future results.

Example

```
protected override void Execute(){
            ChartPane optionPane = CreatePane(50, true, true);

            for(int bar = 30; bar < Bars.Count; bar++)
            {
                if (IsLastPositionActive)
                {
                    if( CumUp.Value( bar, Close, 1 ) >= 2 )
                    {
                        // Sell the call
                        Bars contract = LastPosition.Bars;
                        SetContext( contract );
                        SellAtMarket( bar+1, LastPosition, "Sell call" );
                        RestoreContext();
                    }
                }
                else
                {
                    if( CumDown.Value( bar, Close, 1 ) >= 4 )
                    {
                        // Buy the call contract that has at least 30 days to expiry.
                        Bars contract = CreateSyntheticOption( bar, 30, 30, true );
                        SetContext( contract );
                        BuyAtMarket( bar+1, "Buy call" );
                        RestoreContext();

                        // Plot it if we haven't done so already
                        Color c = Color.FromArgb(255, 128, 128);
                        PlotSymbol(optionPane, contract, c, c);
                    }
                }
            }
        }
    }
}
```

IsOptionExpiryDate

bool IsOptionExpiryDate(int bar);
DateTime NextOptionExpiryDate(int bar);

Returns true if the specified **bar** falls on an options expiration date. Options expiration dates typically fall on the third Friday of every month. If that particular Friday falls on a holiday that the market is closed on, the following market day is the options expiration date.

Example

```
protected override void Execute(){
// Annotate Option Expiry Dates on the Chart

    for(int bar = 1; bar < Bars.Count; bar++)
    {
        if ( IsOptionExpiryDate( bar ) )
        {
            DrawCircle( PricePane, 4, bar, Open[bar], Color.Navy, Color.Navy, WealthLab.LineStyle.Solid, 1, false  );
            DrawCircle( PricePane, 4, bar, Close[bar], Color.Blue, Color.Blue, WealthLab.LineStyle.Solid, 1, true  );
        }

    }
}
```

NextOptionExpiryDate

DateTime NextOptionExpiryDate(int bar);
DateTime NextOptionExpiryDate(DateTime dt);

Returns DateTime of the closest options expiration date as of the specified **bar**. Options expiration dates typically fall on the third Friday of every month. If that particular Friday falls on a holiday that the market is closed on, the following market day is the options expiration date.

Example

```
protected override void Execute(){// Show next option expiration date
            DateTime nextExpiry = NextOptionExpiryDate(Bars.Count-1);
            DrawLabel(PricePane, "Next options expiration date falls on " + nextExpiry.Date.ToShortDateString());
        }
    }
}
```

Position Management

The Position Management category contains methods you can use to access and manipulate Positions that have been created by the Strategy.

ActivePositions Property

IList<Position> ActivePositions

Returns the number of Positions that are currently still active. Use the Positions property to access the collection of actual Positions, and check the Active property of each Position to determine if it is active or not.

Example

```
protected override void Execute(){
    ChartPane RSIPane = CreatePane(  50, true, true );
    PlotSeriesOscillator( RSIPane, RSI.Series( Close, 20 ), 70, 30,
        Color.Red, Color.Blue, Color.MidnightBlue, WealthLab.LineStyle.Dashed, 1 );

    for(int bar = 60; bar < Bars.Count; bar++)
    {
        if ( CrossUnder( bar, RSI.Series( Close, 20 ), 30 ) )
        {
            // Here we limit the system to 3 active Positions max
            if ( ActivePositions.Count < 3 )
                BuyAtMarket( bar+1 );
        }

        if ( ( ActivePositions.Count > 0 ) && CrossOver( bar, RSI.Series( Close, 20 ), 50 ) )
        {
            // Let's work directly with the list of active positions, introduced in WL5
            for( int p = ActivePositions.Count - 1; p > -1 ; p-- )
                SellAtMarket( bar+1, ActivePositions[p] );
        }
    }
}
```

Alerts Property

IList<Alert> Alerts

Returns a list of Alerts objects that represent all of the alerts that have been triggered by the Strategy so far. Use the Count proprety of the Alerts list to determine how many Alerts are in the list. Access the individual Alerts via the [] indexer.

Remarks

- See the Alert object section for information on the Alert object's properties and methods.

Example

```
protected override void Execute(){
    // Alert generating code
    for(int bar = 3; bar < Bars.Count; bar++)
    {
        if (!IsLastPositionActive)
        {
            // Two consecutive lower closes
            if( CumDown.Series(Close, 1)[bar] >= 2 )
                BuyAtLimit( bar+1, Bars.High[bar], "Consecutive close lower" );
        } else
        if (IsLastPositionActive)
        {
            SellAtMarket( bar+1, LastPosition, "Exit next day" );
        }
    }

    // Alert properties
    if( Alerts.Count > 0 )
    {
        for( int i = 0; i < Alerts.Count; i++ )
        {
            WealthLab.Alert a = Alerts[i];

            PrintDebug( "Account: " + a.Account );  // blank string
            PrintDebug( "AlertDate: " + a.AlertDate );
            PrintDebug( "AlertType: " + a.AlertType );
            PrintDebug( "BarInterval: " + a.BarInterval );
            PrintDebug( "BasisPrice: " + a.BasisPrice );
            PrintDebug( "OrderType: " + a.OrderType );
            PrintDebug( "Last close: " + a.Bars.Close[Bars.Count-1] );  // Access Bars object
            PrintDebug( "PositionType: " + a.PositionType );
            PrintDebug( "Price: " + a.Price );
            PrintDebug( "RiskStopLevel: " + a.RiskStopLevel );
            PrintDebug( "Symbol: " + a.Symbol );
            PrintDebug( "Scale: " + a.Scale );
            PrintDebug( "Shares: " + a.Shares );
            PrintDebug( "SignalName: " + a.SignalName );

            try
            {
                PrintDebug( "Position: " + a.Position );
            }
            catch
            {
                PrintDebug( "Position: entry" );
            }
        }
    }
}
```

ClearPositions

void ClearPositions();

Clears all of the trading system Positions that have been generated so far by the Strategy.

Remarks

- In Multi-Symbol Backtest mode, the **ClearPositions** method clear trades for all symbols in a DataSet. On the Trades list, only the last symbol's trades are left.

  It is working by design of the new Wealth-Lab .NET, where all of the positions are stored in one list. That's why **ClearPositions** clears them all and does not work as expected in Multi-Symbol Backtest mode.

Example

```
protected override void Execute(){
    // Calculate the win/loss ratio of the system taking all trades.
    // Then re-run the system, but only take trades when the prior
    // win/loss ratio was above 50%. }
    int Winners, Trades;
    ChartPane WinLossPane = CreatePane( 50, false, true );
    ChartPane CMOPane = CreatePane( 40, true, true );
    PlotSeries( CMOPane, CMO.Series( Close, 20 ), Color.Blue, WealthLab.LineStyle.Solid, 2 );
    SetPaneMinMax( WinLossPane, 0, 100 );
    DataSeries WinLoss = new DataSeries( Bars, "WinLoss" );
    for(int bar = 60; bar < Bars.Count; bar++)
    {
        Winners = 0;
        Trades = 0;
        foreach( Position p in Positions )
            if ( !p.Active )
            {
                Trades++;
                if ( p.NetProfit > 0 )
                    Winners++;
            }
        if ( Trades > 0 )
        {
            WinLoss[bar] = Winners * 100 / Trades;
        }
        if ( CrossOver( bar, CMO.Series( Close, 20 ), -40 ) )
            BuyAtMarket( bar+1, "CMO" ); else
            if ( CrossUnder( bar, CMO.Series( Close, 20 ), 40 ) )
                SellAtMarket( bar+1, Position.AllPositions, "CMO" );
    }
    // Plot the Win/Loss Ratio
    PlotSeries( WinLossPane, WinLoss, Color.Green, WealthLab.LineStyle.Histogram, 5 );
    DrawLabel( WinLossPane, "Win/Loss Ratio", Color.Green );
    // Clear the trades
    ClearPositions();
    // Execute the system again, but only take the trade if the win/loss ratio was above 50
    for(int bar = 60; bar < Bars.Count; bar++)
    {
        if ( CrossOver( bar, CMO.Series( Close, 20 ), -40 ) )
        {
            if ( WinLoss[bar] > 50 )
                BuyAtMarket( bar+1, "CMO" );
        }
        else if ( CrossUnder( bar, CMO.Series( Close, 20 ), 40 ) )
            SellAtMarket( bar+1, Position.AllPositions, "CMO" );
    }
}
```

IsLastPositionActive Property

bool IsLastPositionActive

Indicates whether the most recently established Position (if any) is active or closed. If there were no Positions created yet the property returns false.

Example

```
protected override void Execute(){
    for(int bar = 20; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            double entryPrice = LastPosition.EntryPrice;
            //code your exit rules here
            SellAtStop( bar, LastActivePosition, entryPrice*0.8, "20% stop loss" );
            SellAtLimit( bar, LastActivePosition, entryPrice*1.07, "7% profit target" );
        }
        else
        {
            if ( CrossOver( bar, RSI.Series( ( (High+Low)/2), 40 ), 60 ) )
                BuyAtMarket( bar+1, "Buy Strength" );
        }
    }
}
```

LastActivePosition Property

Position LastActivePosition

Returns the most recently created Position object that is still active (has not yet been sold or covered). If there are no open Positions, **LastActivePosition** returns null.

Remarks

- See the documentation for the **Position** object for more information on its properties and methods.

Example

```
protected override void Execute(){
    int period = 14;
    int oversoldLevel = 30;
    int overboughtLevel = 70;
    DataSeries rsi = RSI.Series( Close, period );

    //Trading system loop
    for (int bar = rsi.FirstValidValue; bar < Bars.Count; bar++)
    {
        if ( LastActivePosition == null )
        {
            //Buy when RSI crosses above oversold level
            if( CrossOver( bar, rsi, oversoldLevel ) )
                BuyAtMarket( bar+1, "RSI Oversold");
```

```
        } else
        {
            // Sell when it crosses below oversold level
            if( CrossUnder( bar, rsi, overboughtLevel ) )
                SellAtMarket( bar+1, LastActivePosition );
        }
    }
}
```

LastPosition Property

Position LastPosition

Returns the most recently created Position object. If there were no Positions created yet, the property returns null.

Remarks

- See the Position object documentation for information about its properties and methods.

Example

```
protected override void Execute(){
    PlotSeries( PricePane, SMA.Series( Close, 20 ), Color.Red, WealthLab.LineStyle.Solid, 1 );
    PlotSeries( PricePane, SMA.Series( Close, 10 ), Color.Blue, WealthLab.LineStyle.Solid, 1 );
    for(int bar = 20; bar < Bars.Count; bar++)
    {
        if ( CrossOver( bar, SMA.Series( Close, 10 ), SMA.Series( Close, 20 ) ) )
            BuyAtMarket( bar+1 ); else
        if ( CrossOver( bar, SMA.Series( Close, 20 ), SMA.Series( Close, 10 ) ) )
            SellAtMarket( bar+1, LastPosition );
    }
}
```

MarketPosition Property

double MarketPosition

Returns the net number of shares that the Strategy has in open Positions. Long Positions add their shares to this value, and short Positions subtract their shares.

Remarks

- In portfolio testing mode, Strategies are pre-executed using a 1 share per Position sizing strategy, then position sizing is applied after the fact. For this reason, MarketPosition will always count a Position's shares as 1 when testing in this mode.

Example

```
protected override void Execute(){
    DataSeries hMA = WMA.Series( Close, 30 );
    DataSeries mp = new DataSeries( Bars, "Market Position" );

    for(int bar = hMA.FirstValidValue; bar < Bars.Count; bar++)
    {
        // Record the number of shares in open positions at each bar as a Data Series.
        mp[bar] = MarketPosition;

        {
            // Buy on a crossover of a 30-period weighted moving average
            // Hold up to 5 positions
            if( CrossOver( bar, Close, hMA[bar-1] ) )
            if( ActivePositions.Count < 5 )
                BuyAtMarket(bar + 1);
        }
        {
             // Sell all positions on a crossunder of a 30-period WMA
            if( CrossUnder( bar, Close, hMA[bar-1] ) )
                foreach ( Position p in Positions )
                    if ( p.Active )
                        SellAtMarket( bar+1, p );
        }
    }

    ChartPane ProfitPane = CreatePane( 30, true, true );
    PlotSeries( ProfitPane, mp, Color.DarkGreen, WealthLab.LineStyle.Histogram, 1 ); //'Open Profit'
    PlotSeries( PricePane, hMA, Color.Blue, WealthLab.LineStyle.Solid, 2 );
}
```

Positions Property

IList<Position> Positions

Returns the Positions that have been established to date by the Strategy (via BuyAtMarket, ShortAtMarket, etc.) Each item in this list is a Position object that represents either a long or a short position.

Remarks

- Use Positions.Count property to determine how many Positions are in the list.
- See the documentation for the Position object to learn about its properties and methods.

Example

```
protected override void Execute(){
    int lastBarBought = 0;

    //Trading system loop
    for (int bar = 41; bar < Bars.Count; bar++)
    {
        // Build up a position buying 40-period High breakouts
        if( ( LastActivePosition != null && LastActivePosition.EntryPrice > Bars.Close[bar] ) || LastActivePosition == null )
            if( bar >= lastBarBought + 9 )
            {
                if( BuyAtStop( bar+1, Highest.Series( High, 40 )[bar], "Breakout" ) != null )
                    lastBarBought = bar+1;
            }

        // Exit positions
        foreach( Position p in Positions )
            if ( p.Active )
```

```
                SellAtStop( bar+1, p, Lowest.Series( Low, 20 )[bar], "Breakdown" );
    }
}
```

SplitPosition

Position SplitPosition(Position position, double percentToRetain);

Splits a single Position into two Positions, allowing you to effectively scale out of a single Position using more than one exit. The **position** parameter contains the Position object that you wish to split. The **percentToRetain** parameter contains the percentage of shares that you wish to remain in the original Position object. SplitPosition returns a new Position object that contains the remaining shares. This new Position object is also added to the end of the Positions list.

Remarks

- **Problem:** Having split a Position into two with SplitPosition, the following properties incorrectly report 0 or NaN for the first part of the splitted Position if Strategy is run in a Portfolio Simulation mode: *MFEAsOfBarPercent, MFEAsOfBar, MAEAsOfBarPercent, MAEAsOfBar, NetProfitAsOfBarPercent, NetProfitAsOfBar.*
- *Partial workaround:* Switch to a Raw Profit position sizing mode.
- **Problem:** in Portfolio Simulation Mode, expect that either all or none of the Positions resulting from *SplitPosition* will be rejected according to the amount of cash available on the entry bar. While building the equity curve, Wealth-Lab treats each split Position as a separate Position competing for cash, and, if *Position.Priority* is random each Position has a random chance of being selected or rejected in a high-exposure MSB Portfolio Simulation.
- *Partial workaround:* To help reduce the frequency (but in no way guarantee) of taking different actions for split Positions, assign the same Priority to the new split Position as the original after calling *SplitPosition* (see code snippet below).

Example

```
protected override void Execute(){
bool soldForProfit = false;

    for(int bar = 50; bar < Bars.Count; bar++)
    {
        if ( ActivePositions.Count > 0 )
        {
            // Split the position to protect large gain
            if ( LastPosition.MFEAsOfBarPercent( bar ) > 20 )
            {
                if (!soldForProfit) {
                    Position p = LastPosition;                    // Position to split
                    Position s = SplitPosition( p, 49.99 );        // The new Position
                    s.Priority = p.Priority;                     // Assign the same Priority as the original Position
                    soldForProfit = SellAtMarket( bar+1, s, "Secure large profit" );
                }
                else {
                    // Exit the rest on a tight stop
                    SellAtStop( bar+1, LastActivePosition, Lowest.Series( Low,10 )[bar], "the rest" );
                }
            }
        }
        else
        {
            BuyAtStop( bar+1, Highest.Series( High, 50 )[bar], "no-frills breakout" );
            soldForProfit = false;
        }
    }
}
```

Position Object

Represents a single Position (trade) that was created by the Strategy. Use the Positions property to access all of the trades that have been created so far at any point in time.

Active Property

bool Active

Determines if the Position is still open or not. A position is closed when it is successfully sold (long positions) or covered (short positions).

Example

```
protected override void Execute(){
    DataSeries rsi = RSI.Series( ( (High+Low)/2 ), 40 );

    Position i;

    for(int bar = rsi.FirstValidValue; bar < Bars.Count; bar++)
    {
        if ( CrossOver( bar, rsi, 35 ) )
            BuyAtLimit( bar+1, High[bar] );

        if ( CrossUnder ( bar, rsi, 70 ) )
        {
            // Cycle through open positions
            foreach( Position p in Positions )
                if ( p.Active )
                    SellAtMarket( bar+1, p );
        }
    }
}
```

AutoProfitLevel Property

double AutoProfitLevel

Specifies the initial profit target level (price) of the Position. The value, analogous to **RiskStopLevel**, is the **price** at which the same-bar Limit order should be placed. It is valid for any BarScale.

Remarks

- **AutoProfitLevel** should be set if "same bar exits" wish to be used in **real-time trading**. It does not have any effect in backtesting.

Example

```
protected override void Execute(){
PlotStops();
    int bcm1 = Bars.Count - 1;
    DataSeries sma1 = SMA.Series(Close, 8);
    DataSeries sma2 = SMA.Series(Close, 20);
    PlotSeries(PricePane, sma1, Color.Green, LineStyle.Solid, 1);
    PlotSeries(PricePane, sma2, Color.Red, LineStyle.Solid, 1);

    for(int bar = Bars.FirstActualBar + 20; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            Position p = LastPosition;
            SellAtLimit( bar+1, p, p.AutoProfitLevel * 1.01 );
        }
        else if ( CrossOver(bar, sma1, sma2) )
        {
            AutoProfitLevel = Bars.High[bar];
            // also use same-bar exit for backtesting
            if (BuyAtMarket( bar+1 ) != null && bar < bcm1)
                SellAtLimit( bar + 1, LastPosition, LastPosition.AutoProfitLevel, "same-bar exit" );
        }
    }
}
```

Bars Property

Bars Bars

Returns the Bars object that the Position was traded against. Certain Strategies (such as pairs trading or symbol rotation) can trade on multiple symbols. The Bars property allows you to determine which symbol a particular Position was established on.

Remarks

- See the **Bars** object reference for more information about its properties and events.

BarsHeld Property

int BarsHeld

Returns the number of bars that the Position was held. If the Position is still active, **BarsHeld** returns the total number of bars held as of the last bar of the chart. The **BarsHeld** property is primarily intended for use by Performance Visualizers, not Strategies.

Example

```
protected override void Execute(){
    // Return the total number of bars held as of the last bar of the chart

    for(int bar = 20; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            if( bar == LastPosition.Bars.Count-1 )
            DrawLabel( PricePane, "Bars held: " + LastPosition.BarsHeld , Color.Blue );
        }
        else
        {
            BuyAtMarket( bar+1 );
        }
    }
}
```

BasisPrice Property

double BasisPrice

Returns the Position's "basis price". This is the price that was used to establish how many shares the Position should be sized to. For market orders, the basis price is typically the closing price of the previous bar. The actual entry price can of course differ because the market may open above or below the previous close. In certain situation (unless a margin factor is applied to simulations), this difference can cause a trade to not be executed (even a market order) due to insufficient capital. For limit orders, the basis price is always the limit price of the order. For stop orders, the basis price is always the stop price specified.

Example

```
protected override void Execute(){
    // Display differences between Basis Price and Entry Price

    for(int bar = 4; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
```

```
        {
            if( bar == LastPosition.Bars.Count-1 )
                DrawLabel( PricePane, (LastPosition.EntryPrice - LastPosition.BasisPrice).ToString(), Color.Black ) ;
            SellAtStop( bar+1, LastPosition, Lowest.Series( Low, 3 )[bar] );
        }
        else
        {
            BuyAtStop( bar+1, Highest.Series( High, 3 )[bar] );
        }
    }
}
```

EntryBar Property

int EntryBar

Returns the bar number that the Position was entered on.

Remarks

- (Doesn't affect WealthScript Strategy coding). In development of **PosSizers** and **Performance Visualizers**, checking for **EntryBar** or **ExitBar** in portfolio simulations may produce unexpected results because the different historical DataSets aren't synchronized when backtesting.

  Solution: check for the date with EntryDate/ExitDate rather than the bar number:

  ```
  // Fails:
  //if (Positions[n].EntryBar == bar + 1)
  // Workaround:
  if(Positions[n].EntryDate == bars.Date[bar + 1].Date)
  ```

Example

```
protected override void Execute(){
    for(int bar = 20; bar < Bars.Count; bar++)
    {
        if (!IsLastPositionActive)
        {
            if ( StochK.Series( Bars, 20 )[bar] > 70 )
            BuyAtMarket( bar+1, "Stoch" );
        }
        else
        {
            // Sell on 10th day
            if ( bar+1 - LastPosition.EntryBar == 10 )
            SellAtClose( bar, LastPosition, "10 day" );
        }
    }
}
```

EntryCommission Property

double EntryCommission

Returns the commission value that was applied to the entry trade for the Position.

Remarks

- **EntryCommission** is not available during Strategy execution, and is only available to Performance Visualizers, Commission structures and PosSizers that execute after position sizing has been applied.

EntryDate Property

DateTime EntryDate

Returns the date/time that the Position was entered on.

Example

```
protected override void Execute(){
    // Dumps entry dates to Debug window
    // Run this on Daily
    if ( Bars.IsIntraday != true )
    {
        for(int bar = 50; bar < Bars.Count; bar++)
        {
            if (IsLastPositionActive)
            {
                PrintDebug( LastPosition.EntryDate );
                // Sell after 10 days
                if ( bar+1 == Bars.ConvertDateToBar( LastPosition.EntryDate, false ) + 10  )
                    SellAtMarket( bar+1, LastPosition, "10 day" );
            }
            else
            {
                if ( StochK.Series( Bars, 20 )[bar] > 70 )
                    BuyAtMarket( bar+1, "Stoch" );
            }
        }
    } else
        Abort();
}
```

EntryOrderType Property

OrderType EntryOrderType

Returns the type of order that was used to establish the Position. Possible values are:

- OrderType.Market
- OrderType.Limit
- OrderType.Stop
- OrderType.AtClose

Example

```
protected override void Execute(){
    for(int bar = 20; bar < Bars.Count; bar++)
    {
        double atr = ATR.Series( Bars,10 )[bar];

        if (IsLastPositionActive)
        {
            string signal = LastPosition.EntrySignal;

            // Simple switching of exits depending on entry order type
            if ( LastPosition.EntryOrderType == OrderType.Limit )
                SellAtStop( bar+1, LastPosition, Lowest.Series( Low,40 )[bar], "Breakdown" );
            else
                if( LastPosition.EntryOrderType == OrderType.Stop )
                SellAtLimit( bar+1, LastPosition, Close[bar]+2*atr, "Target" );
        }
        else
        {
```

```
            if ( BuyAtLimit( bar+1, Lowest.Series( Low,40 )[bar], "Deep down" ) == null )
                BuyAtStop( bar+1, Bars.Close[bar]+atr, "Range" );
        }
    }
}
```

EntryPrice Property

double EntryPrice

Returns the entry price of the Position.

Example

```
protected override void Execute(){
    // Use an ATR stop based on the entry price

    DataSeries sma1 = SMA.Series( Close, 10 );
    DataSeries sma2 = SMA.Series( Close, 40 );
    PlotSeries( PricePane, sma1, Color.LightCoral, WealthLab.LineStyle.Solid, 1 );
    PlotSeries( PricePane, sma2, Color.DarkGreen, WealthLab.LineStyle.Solid, 1 );

    for(int bar = sma2.FirstValidValue; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            if ( CrossUnder( bar, sma1, sma2 ) )
                SellAtMarket( bar+1, LastPosition, "SMA" );
            else
            if ( Close[bar] < ( LastPosition.EntryPrice - ATR.Series( Bars, 10 )[bar] * 4 ) )
                SellAtMarket( bar+1, LastPosition, "ATR" );
        }
        else
        {
            if ( CrossOver( bar, sma1, sma2 ) )
                BuyAtMarket( bar+1, "SMA" );
        }
    }
}
```

EntrySignal Property

string EntrySignal

Returns the "signal name" that was supplied in the "BuyAtXXX" or "ShortAtXXX" method that was used to establish the Position. All "BuyAtXXX" and "ShortAtXXX" methods allow you to specify an optional signalName parameter. The value that you specify there is visible in the trade list, and is also accessible via the **EntrySignal** property.

Example

```
protected override void Execute(){
    for(int bar = 40; bar < Bars.Count; bar++)
    {
        double atr = ATR.Series( Bars,10 )[bar];

        if (IsLastPositionActive)
        {
            string signal = LastPosition.EntrySignal;

            // Simple switching of exits depending on entry signal
            if ( signal == "Highest" )
                SellAtStop( bar+1, LastPosition, Lowest.Series( Low,40 )[bar], "Breakdown" );
            else
                if( signal == "Range" )
                SellAtLimit( bar+1, LastPosition, Close[bar]+2*atr, "Target" );
        }
            else
            {
                if ( BuyAtStop( bar+1, Highest.Series( High,40 )[bar], "Highest" ) == null )
                    BuyAtStop( bar+1, Bars.Close[bar]+atr, "Range" );
            }
    }
}
```

ExitBar Property

int ExitBar

Returns the bar number that the Position was exited (closed) on. If the Position is still active, **ExitBar** returns -1.

Remarks

- (Doesn't affect WealthScript Strategy coding). In development of **PosSizers** and **Performance Visualizers**, checking for **EntryBar** or **ExitBar** in portfolio simulations may produce unexpected results because the different historical DataSets aren't synchronized when backtesting.

  Solution: check for the date with EntryDate/ExitDate rather than the bar number:

  ```
  // Fails:
  //if (Positions[n].ExitBar == bar + 1)
  // Workaround:
  if(Positions[n].ExitDate == bars.Date[bar + 1].Date)
  ```

Example

```
using System;
using System.Text;
using System.Drawing;
using WealthLab;

namespace WealthLab.Strategies
{
    public class ExitBar : WealthScript
    {
        // Display the shortest and the longest holding time of closed positions

        protected override void Execute()
        {
            for(int bar = 30; bar < Bars.Count; bar++)
            {
                if (!IsLastPositionActive )
                {
                    if ( CrossUnder( bar, Indicators.RSI.Series( Close, 10 ), 20 ) )
                        BuyAtMarket( bar+1 );
                }
                else
                {
                    if ( CrossOver( bar, Indicators.RSI.Series( Close, 10 ), 60 ) )
                        SellAtMarket( bar+1, LastPosition );
                }
            }

            int LowBar = 0;
            int HighBar = 0;
```

```
            int BarsHeld;

            foreach ( Position p in Positions )
            {
                if( !p.Active )
                {
                    BarsHeld = p.ExitBar - p.EntryBar;
                    if( BarsHeld > HighBar )
                        HighBar = BarsHeld;
                    if ( ( BarsHeld < LowBar ) | ( LowBar <= 0 ) )
                        LowBar = BarsHeld;
                }
            }

            DrawLabel( PricePane, "Longest Holding Time: " + HighBar, Color.Black );
            DrawLabel( PricePane, "Shortest Holding Time: " + LowBar, Color.Black );
        }
    }
}
```

ExitCommission Property

double ExitCommission

Returns the commission value that was applied to the exit trade for the Position. If the Position is still active, ExitCommission returns 0.

Remarks

- **ExitCommission** is not available during Strategy execution, and is only available to Performance Visualizers, Commission structures and PosSizers that execute after position sizing has been applied.

ExitDate Property

DateTime ExitDate

Returns the date/time that the Position was exited (closed) on. If the Position is still active, **ExitDate** returns DateTime.MinValue.

Example

```
protected override void Execute(){
    for (int bar = 41; bar < Bars.Count; bar++)
    {
        if( IsLastPositionActive )
        {
            if( bar == Bars.Count-1 )
                DrawLabel( PricePane, "Holding a position...", Color.LightBlue );
            SellAtStop( bar+1, LastPosition, Lowest.Series( Low, 20 )[bar], "Breakdown" );
        }
        else
        {
            // Prints how much time passed since last exit from LastPosition
            if( ( bar == Bars.Count-1 ) & ( Positions.Count > 0 ) )
            {
                int x = DateTime.Compare(Bars.Date[bar], LastPosition.ExitDate);
                if( x > 0 )
                {
                    DateTime exitDate = LastPosition.ExitDate;
                    DateTime today = Bars.Date[bar];
                    TimeSpan sinceExit = today.Subtract( exitDate );
                    DrawLabel( PricePane, "Time since last exit: " + sinceExit.Days + " days, " + sinceExit.Hours + " hours, " + sinceExit.Minutes + " minutes, " + sinceExit.Seconds + " seconds", Color.Blue );
                }
            }
            BuyAtStop( bar+1, Highest.Series( High, 40 )[bar], "Breakout" );
        }
    }
}
```

ExitOrderType Property

OrderType ExitOrderType

Returns the type of order that was used to exit (close) the Position. Possible values are:

- OrderType.Market
- OrderType.Limit
- OrderType.Stop
- OrderType.AtClose

Example

```
protected override void Execute(){
    ClearDebug();

    for(int bar = 20; bar < Bars.Count; bar++)
    {
        double atr = ATR.Series( Bars,10 )[bar];
        if (IsLastPositionActive)
        {
            if(    SellAtStop( bar+1, LastPosition, Lowest.Series( Low,40 )[bar] ) == false )
                SellAtLimit( bar+1, LastPosition, Close[bar]+2*atr );
        }
        else
        {
            BuyAtStop( bar+1, Highest.Series( High,20 )[bar] );
        }
    }

    // Print exit order type statistics

    int stop = 0;
    int limit = 0;

    foreach( Position p in Positions )
        if( !p.Active )
        {
            if( p.ExitOrderType == OrderType.Stop )
                stop++;
            if( p.ExitOrderType == OrderType.Limit )
                limit++;
        }
    PrintDebug( "Exits on stop: " + stop, "Exits at limits: " + limit );
}
```

ExitPrice Property

double ExitPrice

Returns the exit price of the Position. If the Position is still active, **ExitPrice** returns 0.

Example

```
using System;
using System.Collections.Generic;
using System.Text;
```

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

namespace WealthLab.Strategies
{
    public class MyStrategy : WealthScript
    {
        // This procedure reports the entry and exit levels of all trades
        void TradeReport()
        {
            foreach( Position p in Positions )
            {
                PrintDebug( "Entry:" + p.EntryDate + " at " + p.EntryPrice );
                PrintDebug( "Exit:" + p.ExitDate + " at " + p.ExitPrice );
            }
        }

        protected override void Execute()
        {
            for(int bar = 20; bar < Bars.Count; bar++)
            {
                if (IsLastPositionActive)
                    SellAtStop( bar+1, LastPosition, Lowest.Series(Close,20)[bar], "Exit" );
                else
                    BuyAtStop( bar+1, Close[bar]+2*ATR.Series(Bars,10)[bar], "Long" );
            }
            TradeReport();
        }
    }
}
```

ExitSignal Property

string ExitSignal

Returns the "signal name" that was supplied in the "SellAtXXX" or "CoverAtXXX" method that was used to close the Position. All "SellAtXXX" and "CoverAtXXX" methods allow you to specify an optional signalName parameter. The value that you specify there is visible in the trade list, and is also accessible via the **ExitSignal** property. If the Position is still active, **ExitSignal** returns a blank string.

Example

```
protected override void Execute(){
    // Marks the position exit bar with the exit signal name

    for(int bar = 20; bar < Bars.Count; bar++)
    {
        double atr = ATR.Series( Bars,10 )[bar];

        if (IsLastPositionActive)
        {
            Position p = LastActivePosition;
            if ( SellAtStop( bar+1, LastPosition, Lowest.Series( Low,10 )[bar], "Breakdown" ) )
                AnnotateBar( p.ExitSignal.ToString(), bar, false, Color.Red ); else
            if( SellAtLimit( bar+1, LastPosition, Close[bar]+atr, "Target" ) )
                AnnotateBar( p.ExitSignal.ToString(), bar, true, Color.Blue );

        }
        else
            BuyAtStop( bar+1, Highest.Series( High, 20 )[bar] );
    }
}
```

HighestHighAsOfBar

double HighestHighAsOfBar(int bar);

Returns the highest price registered in the Position, as of the specified **bar** number.

Example

```
protected override void Execute(){
    for(int bar = 50; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            Position p = LastPosition;
            SellAtTrailingStop( bar+1, p, p.HighestHighAsOfBar(bar) - 3*ATR.Value( bar, Bars, 14 ), "3x ATR Stop" );
        }
        else
            BuyAtStop( bar+1, Highest.Series( High, 20 )[bar] );
    }
}
```

LowestLowAsOfBar

double LowestLowAsOfBar(int bar);

Returns the lowest price registered in the Position, as of the specified **bar** number.

Example

```
protected override void Execute(){
    for(int bar = 50; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            Position p = LastPosition;
            CoverAtTrailingStop( bar+1, p, p.LowestLowAsOfBar(bar) + 3*ATR.Value( bar, Bars, 14 ), "3x ATR Stop" );
        }
        else
            ShortAtStop( bar+1, Lowest.Series( Low, 20 )[bar] );
    }
}
```

MAE Property

double MAE

Returns the Maximum Adverse Excursion (MAE) that was generated by the Position, with commissions applied. **MAE** represents the largest intraday loss that the trade experienced during its lifetime. This property is intended for use by Performance Visualizers, and not in Strategies.

Remarks

- **MAE** is not available during Strategy execution, and is only available to Performance Visualizers that execute after position sizing has been applied.

MAEAsOfBar

double MAEAsOfBar(int bar);

Returns the Maximum Adverse Excursion (MAE) that was generated by the Position, with commissions applied, as of the specified **bar** number. **MAEAsOfBar** represents the largest intraday loss that the trade experienced up to the specified **bar**.

Remarks

- In portfolio simulation mode, all trades are pre-executed using 1 share per Position, and then position sizing is applied after the fact. So the **MAEAsOfBar** property will always be based on 1 share while the Strategy is executing.
- The **MAEAsOfBar** property is always available to Performance Visualizers, which execute after the position sizing has been applied.
- **Problem:** Having split a Position into two with SplitPosition, the following properties incorrectly report 0 or NaN for the first part of the splitted Position if Strategy is run in a Portfolio Simulation mode: *MFEAsOfBarPercent, MFEAsOfBar, MAEAsOfBarPercent, MAEAsOfBar, NetProfitAsOfBarPercent, NetProfitAsOfBar.*
    - *Partial workaround:* Switch to a Raw Profit position sizing mode.

Example

```
protected override void Execute(){
    // Record a position's Maximum Adverse Excursion (MAE) at each bar as a Data Series.
    // This system buys on a crossover of a 30-period weighted
    // moving average and sells after 20 bars.

    int timedExit = 20; // just exit after 20 days
    DataSeries hMA = WMA.Series( Close, 30 );
    DataSeries hMAESer = new DataSeries( Bars, "MAE" );

    for(int bar = hMA.FirstValidValue; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            hMAESer[bar] = LastActivePosition.MAEAsOfBar( bar );
            if ( bar+1 - LastPosition.EntryBar >= timedExit )
                SellAtMarket( bar+1, LastPosition );
        }
        else
        {
            if( CrossOver( bar, Close, hMA[bar-1] ) )
                BuyAtMarket( bar+1, "Xover" );

        }
    }

    ChartPane ProfitPane = CreatePane( 50, true, true );
    PlotSeries( ProfitPane, hMAESer, Color.DarkGreen, WealthLab.LineStyle.Histogram, 1 );
    PlotSeries( PricePane, hMA, Color.Blue, WealthLab.LineStyle.Solid, 2 );
}
```

MAEAsOfBarPercent

double MAEAsOfBarPercent(int bar);

Returns the Maximum Adverse Excursion (MAE) that was generated by the Position, with commissions applied, as a percentage, as of the specified **bar** number. **MAEAsOfBarPercent** represents the largest intraday percentage loss that the trade experienced up to the specified **bar**.

Remarks

- In portfolio simulation mode, all trades are pre-executed using 1 share per Position, and then position sizing is applied after the fact. So the **MAEAsOfBarPercent** property will always be based on 1 share while the Strategy is executing.
- The **MAEAsOfBarPercent** property is always available to Performance Visualizers, which execute after the position sizing has been applied.
- **Problem:** Having split a Position into two with SplitPosition, the following properties incorrectly report 0 or NaN for the first part of the splitted Position if Strategy is run in a Portfolio Simulation mode: *MFEAsOfBarPercent, MFEAsOfBar, MAEAsOfBarPercent, MAEAsOfBar, NetProfitAsOfBarPercent, NetProfitAsOfBar.*
    - *Partial workaround:* Switch to a Raw Profit position sizing mode.

Example

```
protected override void Execute(){
    // Record a position's Maximum Adverse Excursion (MAE) percentage
    // at each bar as a Data Series.
    // This system buys on a crossover of a 30-period weighted
    // moving average and sells after 20 bars.

    int timedExit = 20; // just exit after 20 days
    DataSeries hMA = WMA.Series( Close, 30 );
    DataSeries hMAEPctSer = new DataSeries( Bars, "MAE (Percentage)" );

    for(int bar = hMA.FirstValidValue; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            hMAEPctSer[bar] = LastActivePosition.MAEAsOfBarPercent( bar );
            if ( bar+1 - LastPosition.EntryBar >= timedExit )
                SellAtMarket( bar+1, LastPosition );
        }
        else
        {
            if( CrossOver( bar, Close, hMA[bar-1] ) )
                BuyAtMarket( bar+1, "Xover" );

        }
    }

    ChartPane ProfitPane = CreatePane( 50, true, true );
    PlotSeries( ProfitPane, hMAEPctSer, Color.DarkGreen, WealthLab.LineStyle.Histogram, 1 );
    PlotSeries( PricePane, hMA, Color.Blue, WealthLab.LineStyle.Solid, 2 );
}
```

MAEPercent Property

double MAEPercent

Returns the Maximum Adverse Excursion (MAE) that was generated by the Position, with commissions applied, as a percentage. **MAEPercent** represents the largest intraday percentage loss that the trade experienced during its lifetime. This property is intended for use by Performance Visualizers, and not in Strategies.

Remarks

- **MAEPercent** is not available during Strategy execution, and is only available to Performance Visualizers that execute after position sizing has been applied.

MFE Property

double MFE

Returns the Maximum Favorable Excursion (MFE) that was generated by the Position, with commissions applied. **MFE** represents the highest intraday profit that the trade experienced during its lifetime. This property is intended for use by Performance Visualizers, and not in Strategies.

Remarks

- **MFE** is not available during Strategy execution, and is only available to Performance Visualizers that execute after position sizing has been applied.

MFEAsOfBar

double MFEAsOfBar(int bar);

Returns the Maximum Favorable Excursion (MFE) that was generated by the Position, with commissions applied, as of the specified **bar** number. **MFEAsOfBar** represents the highest intraday profit that the trade experienced up to the specified **bar**.

Remarks

- In portfolio simulation mode, all trades are pre-executed using 1 share per Position, and then position sizing is applied after the fact. So the **MFEAsOfBar** property will always be based on 1 share while the Strategy is executing.
- The **MFEAsOfBar** property is always available to Performance Visualizers, which execute after the position sizing has been applied.
- **Problem:** Having split a Position into two with SplitPosition, the following properties incorrectly report 0 or NaN for the first part of the splitted Position if Strategy is run in a Portfolio Simulation mode: *MFEAsOfBarPercent, MFEAsOfBar, MAEAsOfBarPercent, MAEAsOfBar, NetProfitAsOfBarPercent, NetProfitAsOfBar.*
    - *Partial workaround:* Switch to a Raw Profit position sizing mode.

Example

```
protected override void Execute(){
    // Record a position's Maximum Favorite Excursion (MFE) at each bar as a Data Series.
    // This system buys on a crossover of a 30-period weighted
    // moving average and sells after 20 bars.

    int timedExit = 20; // just exit after 20 days
    DataSeries hMA = WMA.Series( Close, 30 );
    DataSeries hMFESer = new DataSeries( Bars, "MFE" );

    for(int bar = hMA.FirstValidValue; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            hMFESer[bar] = LastActivePosition.MFEAsOfBar( bar );
            if ( bar+1 - LastPosition.EntryBar >= timedExit )
                SellAtMarket( bar+1, LastPosition );
        }
        else
        {
            if( CrossOver( bar, Close, hMA[bar-1] ) )
                BuyAtMarket( bar+1, "Xover" );

        }
    }

    ChartPane ProfitPane = CreatePane( 50, true, true );
    PlotSeries( ProfitPane, hMFESer, Color.DarkGreen, WealthLab.LineStyle.Histogram, 1 );
    PlotSeries( PricePane, hMA, Color.Blue, WealthLab.LineStyle.Solid, 2 );
}
```

MFEAsOfBarPercent

double MFEAsOfBarPercent(int bar);

Returns the Maximum Favorable Excursion (MFE) that was generated by the Position, with commissions applied, as a percentage, as of the specified **bar** number. **MFEAsOfBarPercent** represents the highest intraday percentage profit that the trade experienced up to the specified **bar**.

Remarks

- In portfolio simulation mode, all trades are pre-executed using 1 share per Position, and then position sizing is applied after the fact. So the **MFEAsOfBarPercent** property will always be based on 1 share while the Strategy is executing.
- The **MFEAsOfBarPercent** property is always available to Performance Visualizers, which execute after the position sizing has been applied.
- **Problem:** Having split a Position into two with SplitPosition, the following properties incorrectly report 0 or NaN for the first part of the splitted Position if Strategy is run in a Portfolio Simulation mode: *MFEAsOfBarPercent, MFEAsOfBar, MAEAsOfBarPercent, MAEAsOfBar, NetProfitAsOfBarPercent, NetProfitAsOfBar.*
    - *Partial workaround:* Switch to a Raw Profit position sizing mode.

Example

```
protected override void Execute(){
    // Record a position's Maximum Favorable Excursion (MFE) percentage
    // at each bar as a Data Series.
    // This system buys on a crossover of a 30-period weighted
    // moving average and sells after 20 bars.

    int timedExit = 20; // just exit after 20 days
    DataSeries hMA = WMA.Series( Close, 30 );
    DataSeries hMFEPctSer = new DataSeries( Bars, "MFE (Percentage)" );

    for(int bar = hMA.FirstValidValue; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            hMFEPctSer[bar] = LastActivePosition.MFEAsOfBarPercent( bar );
            if ( bar+1 - LastPosition.EntryBar >= timedExit )
                SellAtMarket( bar+1, LastPosition );
        }
        else
        {
            if( CrossOver( bar, Close, hMA[bar-1] ) )
                BuyAtMarket( bar+1, "Xover" );

        }
    }

    ChartPane ProfitPane = CreatePane( 50, true, true );
    PlotSeries( ProfitPane, hMFEPctSer, Color.DarkGreen, WealthLab.LineStyle.Histogram, 1 );
    PlotSeries( PricePane, hMA, Color.Blue, WealthLab.LineStyle.Solid, 2 );
}
```

MFEPercent Property

double MFEPercent

Returns the Maximum Favorable Excursion (MFE) that was generated by the Position, with commissions applied, as a percentage. **MFEPercent** represents the highest intraday percentage profit that the trade experienced during its lifetime. This property is intended for use by Performance Visualizers, and not in Strategies.

Remarks

- **MFEPercent** is not available during Strategy execution, and is only available to Performance Visualizers that execute after position sizing has been applied.

NetProfit Property

double NetProfit

Returns the profit that was generated by the Position, excluding commissions. This property is intended for use by Performance Visualizers, and not in Strategies.

Remarks

- In portfolio simulation mode, all trades are pre-executed using 1 share per Position, and then position sizing is applied after the fact. So the **NetProfit** property will always be based on 1 share while the Strategy is executing.
- The **NetProfit** property is always available to Performance Visualizers, which execute after the position sizing has been applied.

NetProfitAsOfBar

double NetProfitAsOfBar(int bar);

Returns the profit that was generated by the Position, excluding commissions, as of the specified **bar** number.

Remarks

- In portfolio simulation mode, all trades are pre-executed using 1 share per Position, and then position sizing is applied after the fact. So **NetProfitAsOfBar** will always be based on 1 share while the Strategy is executing.
- **NetProfitAsOfBar** is always available to Performance Visualizers, which execute after the position sizing has been applied.
- **Problem:** Having split a Position into two with SplitPosition, the following properties incorrectly report 0 or NaN for the first part of the splitted Position if Strategy is run in a Portfolio Simulation mode: *MFEAsOfBarPercent, MFEAsOfBar, MAEAsOfBarPercent, MAEAsOfBar, NetProfitAsOfBarPercent, NetProfitAsOfBar.*
    - *Partial workaround:* Switch to a Raw Profit position sizing mode.

Example

```
protected override void Execute(){
    // Record a position's net profit at each bar as a Data Series.
    // This system buys on a crossover of a 30-period weighted
    // moving average and sells after 20 bars.

    int timedExit = 20; // just exit after 20 days
    DataSeries hMA = WMA.Series( Close, 30 );
    DataSeries hPftSer = new DataSeries( Bars, "Net Profit" );
```

```
    for(int bar = hMA.FirstValidValue; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            hPftSer[bar] = LastActivePosition.NetProfitAsOfBar( bar );
            if ( bar+1 - LastPosition.EntryBar >= timedExit )
                SellAtMarket( bar+1, LastPosition );
        }
        else
        {
            if( CrossOver( bar, Close, hMA[bar-1] ) )
                BuyAtMarket( bar+1, "Xover" );

        }
    }

    ChartPane ProfitPane = CreatePane( 50, true, true );
    PlotSeries( ProfitPane, hPftSer, Color.DarkGreen, WealthLab.LineStyle.Histogram, 1 ); //'Open Profit'
    PlotSeries( PricePane, hMA, Color.Blue, WealthLab.LineStyle.Solid, 2 );
}
```

NetProfitAsOfBarPercent

double NetProfitAsOfBarPercent(int bar);

Returns the percentage profit that was generated by the Position, excluding commissions, as of the specified **bar** number.

Remarks

- In portfolio simulation mode, all trades are pre-executed using 1 share per Position, and then position sizing is applied after the fact. So **NetProfitAsOfBarPercent** will always be based on 1 share while the Strategy is executing.
- **NetProfitAsOfBarPercent** is always available to Performance Visualizers, which execute after the position sizing has been applied.
- **Problem:** Having split a Position into two with SplitPosition, the following properties incorrectly report 0 or NaN for the first part of the splitted Position if Strategy is run in a Portfolio Simulation mode: *MFEAsOfBarPercent, MFEAsOfBar, MAEAsOfBarPercent, MAEAsOfBar, NetProfitAsOfBarPercent, NetProfitAsOfBar.*
    - *Partial workaround:* Switch to a Raw Profit position sizing mode.

Example

```
protected override void Execute(){
    // Record a position's percentage profit at each bar as a Data Series.
    // This system buys on a crossover of a 30-period weighted
    // moving average and sells after 20 bars.

    int timedExit = 20; // just exit after 20 days
    DataSeries hMA = WMA.Series( Close, 30 );
    DataSeries hPftPctSer = new DataSeries( Bars, "Net Profit (Percent)" );

    for(int bar = hMA.FirstValidValue; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            hPftPctSer[bar] = LastActivePosition.NetProfitAsOfBarPercent( bar );
            if ( bar+1 - LastPosition.EntryBar >= timedExit )
                SellAtMarket( bar+1, LastPosition );
        }
        else
        {
            if( CrossOver( bar, Close, hMA[bar-1] ) )
                BuyAtMarket( bar+1, "Xover" );

        }
    }

    ChartPane ProfitPane = CreatePane( 50, true, true );
    PlotSeries( ProfitPane, hPftPctSer, Color.DarkGreen, WealthLab.LineStyle.Histogram, 1 ); //'Open Profit'
    PlotSeries( PricePane, hMA, Color.Blue, WealthLab.LineStyle.Solid, 2 );
}
```

NetProfitPercent Property

double NetProfitPercent

Returns the profit that was generated by the Position, excluding commissions, as a percentage. This property is intended for use by Performance Visualizers, and not in Strategies.

Remarks

- In portfolio simulation mode, all trades are pre-executed using 1 share per Position, and then position sizing is applied after the fact. So the **NetProfitPercent** property will always be based on 1 share while the Strategy is executing.
- The **NetProfitPercent** property is always available to Performance Visualizers, which execute after the position sizing has been applied.

PositionType Property

PositionType PositionType

Returns the type of Position, either long or short. Possible values are:

- PositionType.Long
- PositionType.Short

Example

```
protected override void Execute(){
    for(int bar = 5; bar < Bars.Count; bar++)
    {
        if( IsLastPositionActive )
        {
            if( LastPosition.PositionType == PositionType.Long )
                SellAtStop( bar+1, LastPosition, Lowest.Series( Low, 4 )[bar-1] ); else
                CoverAtStop( bar+1, LastPosition, Highest.Series( High, 4 )[bar-1] );
        } else
        {
            if( BuyAtStop( bar+1, Highest.Series( High, 4 )[bar-1] ) == null )
                ShortAtStop( bar+1, Lowest.Series( Low, 4 )[bar-1] );
        }
    }
}
```

Priority Property

double Priority

The **Priority** property comes in to play if there is a situation where there are several trade alerts generated in a simulation, but there is only enough capital to take some of the alerts. In this case, the trades are executed in order of the Positions' **Priority** value, with the higher numeric values taking precedence.

Priority is generally used for Strategies that use **Buy/ShortAtMarket** (or **AtClose**) entries. For example, assume that your trading system generates 10 orders to place on the next bar, but you have cash enough for 4 orders only. Prior to placing orders, it's possible to determine which of the orders to place based on some indicator or price.

AtLimit/AtStop Entry Orders

Generally speaking, you should not assign **Priority** for Strategies that use AtLimit/AtStop entries. Doing so may create a peeking effect since it's often not possible to know which limit (or stop) orders will execute first when orders are placed for multiple instruments. You can, however, realistically use the inverse of the HHmm time-of-day as the correct Priority value. In other words, trades that occur earlier in the day should be assigned higher priority.

Exceptions:

1. If the script employs a "multi-dip buyer" strategy, assign a higher Priority value to AtLimit orders with higher limit prices. If you don't, the possibility exists to execute orders with lower limit prices first (and vice-versa for ShortAtLimit).
2. You can intentionally peek to determine if an AtLimit/AtStop order occurred at the opening price, and in this case you could assign an equally-high priority to these Positions. This is a valid use of peeking in backtesting.

Warning!

You *should* employ **Priority** in Strategies that use multiple order-entry types, such as AtMarket and AtLimit orders. Since the Strategy window does not distinguish between the types, set a higher priority for AtMarket entries so that they are processed before AtLimit/AtStop orders on the same bar.

Note

- Positions are initialized a random **Priority** value. This means that simulations run consecutively could generate different results if there is not enough capital to take all of the trades generated.

Example

```
protected override void Execute(){

    // Commodity Selection Index by Welles Wilder Jr. (c) 1979
    // Run this strategy in Futures mode on a symbol which has defined margin/point value

    int Commission = 8;
    int adxPeriod = 14;
    DataSeries CSI = new DataSeries( Bars, "Commodity Selection Index (CSI)" );
    SymbolInfo si = Bars.SymbolInfo;

    if( si.Margin > 0 )
    {
        for(int bar = adxPeriod; bar < Bars.Count; bar++)
        {
            CSI[bar] = ADXR.Series( Bars, adxPeriod )[bar] * ATR.Series( Bars, adxPeriod )[bar] * ( ( si.PointValue / Math.Sqrt( si.Margin ) ) * (float) 1 / ( 150 + Commission )  ) * 100;
        }
    } else
        // Will not execute if margin is not specified in Symbol Info Manager
        Abort();

    // Plot Commodity Selection Index on chart
    ChartPane CSIPane = CreatePane( 40, true, true );
    PlotSeries( CSIPane, CSI, Color.Blue, WealthLab.LineStyle.Solid, 2 );

    PlotSeries( PricePane, Lowest.Series( Close, 10 ), Color.Blue, WealthLab.LineStyle.Solid, 1 );
    PlotSeries( PricePane, Highest.Series( Close, 10 ), Color.Red, WealthLab.LineStyle.Solid, 1 );

    for(int bar = 20; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            Position p = LastPosition;
            if ( p.PositionType == PositionType.Long )
            SellAtStop( bar+1, p, Lowest.Series( Close, 10 )[bar], "Exit" );
            if ( p.PositionType == PositionType.Short )
            CoverAtStop( bar+1, p, Highest.Series( Close, 10 )[bar], "Cover" );
        }
        else
        {
            if ( Close[bar] > Highest.Series( Close, 20 )[bar-1] )
                if ( BuyAtMarket( bar+1, Convert.ToString( CSI[bar] ) ) != null )
                    LastActivePosition.Priority = CSI[bar];

            if ( Close[bar] < Lowest.Series( Close, 20 )[bar-1] )
                if ( ShortAtMarket( bar+1, Convert.ToString( CSI[bar] ) ) != null )
                    LastActivePosition.Priority = CSI[bar];
        }
    }
}
```

ProfitPerBar Property

double ProfitPerBar

Returns the net profit of the Position divided by the number of bars held. If the Position is still active, the number of bars is based onthe total number of bars held as of the last bar of the chart. The **ProfitPerBar** property is primarily intended for use by Performance Visualizers, not Strategies.

Remarks

- In portfolio simulation mode, all trades are pre-executed using 1 share per Position, and then position sizing is applied after the fact. So the **ProfitPerBar** property will always be based on 1 share while the Strategy is executing.
- The **ProfitPerBar** property is always available to Performance Visualizers, which execute after the position sizing has been applied.

RiskStopLevel Property

double RiskStopLevel

Specifies the initial stop level (price) of the Position. This stop level is used when you select the Maximum Risk Pct Position Sizing option. This option specifies the maximum amount of capital you are willing to risk on each trade. When this option is selected, you must set the value of **RiskStopLevel** in your Strategy code to indicate the initial stop loss value for a newly created Position.

For example, a simple channel breakout system might enter at the highest 20 bar high, and exit at the lowest 20 bar low. Prior to issuing the **BuyAtMarket**, or **BuyAtStop**, you should set **RiskStopLevel** to the lowest Low value of the past 20 bars as the initial stop level for the long Position.

Remarks

- If you select the Maximum Risk Pct position sizing option and do not set **RiskStopLevel** in your Strategy code, you will receive an error message when attempting to run the Strategy.
- You must also be diligent in your Strategy to actually use the established stop level as an exit. If you do not, the Strategy could lose considerably more than the Maximum Risk that you established in the Position Size setting.
- Once a RiskStopLevel is established for a Position, do not change it. The [last] value assigned to a Position's RiskStopLevel is used to determine % Risk sizing, consequently reassigning its value after the Position is established is effectively a peeking error.

Example

```
protected override void Execute(){    ClearDebug();
    for(int bar = 3; bar < Bars.Count; bar++)
    {
        if( CumUp.Value(bar, Close, 1) >= 2 )
        {
            for(int pos = ActivePositions.Count - 1; pos >= 0; pos--)
            {
                Position p = ActivePositions[pos];
                PrintDebug(bar + "\t" + p.EntryBar  + "\t" + p.RiskStopLevel.ToString("0.00") + "\t" + p.EntrySignal );
                SellAtMarket(bar + 1, p);
            }
            PrintDebug("");
        }

        RiskStopLevel = Close[bar] - 1.0;
        if( CumDown.Value(bar, Close, 1) >= 2 )
            BuyAtMarket(bar + 1, Convert.ToString(RiskStopLevel) );
    }
}
```

Shares Property

double Shares

Returns the number of shares (or contracts) that the Position contains.

Remarks

- In portfolio simulation mode, all trades are pre-executed using 1 share per Position, and then position sizing is applied after the fact. So the **Shares** property will always return 1 while the Strategy is executing.
- The **Shares** property is always available to Performance Visualizers, which execute after the position sizing has been applied.

Example

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

namespace WealthLab.Strategies
{
    public class MyStrategy : WealthScript
    {
        // Write number of shares of open Positions to debug window
        protected void WriteOpenTrades()
        {
            ClearDebug();
            // Cycle through open positions
            foreach( Position p in Positions )
                if ( p.Active )
                PrintDebug( p.Shares + " Shares " + p.Bars.Symbol );
        }

        protected override void Execute()
        {
            for(int bar = 20; bar < Bars.Count; bar++)
            {
                if (IsLastPositionActive)
                {
                    SellAtMarket( bar+1, LastPosition );
                }
                else
                {
                    BuyAtMarket( bar+1 );
                }

                // Try this in Raw Profit, with Fixed dollar
                WriteOpenTrades();
            }
        }
    }
}
```

Size Property

double Size

Returns the dollar size of the Position. For equities and mutual funds this is the shares multiplied by the entry price. For futures, this is the contracts (Shares property) multiplied by the margin of the contract (Bars.Margin).

Remarks

- In portfolio simulation mode, all trades are pre-executed using 1 share per Position, and then position sizing is applied after the fact. So the **Size** property will always be based on 1 share while the Strategy is executing.
- The **Size** property is always available to Performance Visualizers, which execute after the position sizing has been applied.

Example

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

namespace WealthLab.Strategies
{
    public class MyStrategy : WealthScript
    {
        // Write number of shares of open Positions to debug window
        protected void WriteOpenTrades()
        {
            ClearDebug();
            // Cycle through open positions
            foreach( Position p in Positions )
                if ( p.Active )
                    PrintDebug( "Position value in " + p.Bars.Symbol + ": " + String.Format( "{0:c}", p.Size ) );
        }

        protected override void Execute()
        {
            for(int bar = 20; bar < Bars.Count; bar++)
            {
                if (IsLastPositionActive)
                {
                    SellAtMarket( bar+1, LastPosition );
                }
                else
                {
                    BuyAtMarket( bar+1 );
                }

                // Run this in Raw Profit mode
                WriteOpenTrades();
            }
        }
    }
}
```

Tag Property

object Tag

The **Tag** property allows you to store any object with a Position.

Example

```
protected override void Execute(){
    for(int bar = 10; bar < Bars.Count; bar++)
    {
        if( IsLastPositionActive )
        {
            SellAtLimit( bar+1, LastPosition, (double) LastPosition.Tag );
        } else
        {
            if( ( Close[bar] > Close[bar-3] ) & ( Close[bar-3] > Close[bar-5] ) )
                if( BuyAtMarket( bar+1 ) != null )
                    // Store target price in the position's tag property
                    LastPosition.Tag = (Close[bar]*1.05 );
        }
    }
}
```

TrailingStop Property

double TrailingStop

Provides access to the most recent trailing stop value for the Position. Trailing stop levels come from calling the **SellAtTrailingStop** or **CoverAtTrailingStop** WealthScript methods. The trailing stop is adjusted upward if the most recently passed value is higher than the current stop level.

Example

```
protected override void Execute(){
    PlotStops();
    int period = 20;
    SMA sma = SMA.Series( Close, period );
    PlotSeries( PricePane, sma, Color.BurlyWood, WealthLab.LineStyle.Solid, 1 );

    for(int bar = 3*period; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            Position p = LastActivePosition;

            // Initiate a trailing stop after a 5% gain
            if ( p.MFEAsOfBarPercent( bar ) > 5 )
            {
                CoverAtTrailingStop( bar+1, p, sma[bar], "Trailing Stop" );
            }
            else
                CoverAtStop( bar+1, p, p.EntryPrice * 1.10, "10% Stop Loss" );

            if( ( bar == Bars.Count-1 ) & ( p.TrailingStop > 0 ) )
                DrawLabel( PricePane, "Current trailing stop value = " + p.TrailingStop.ToString(), Color.Indigo );

        }
        else
        {
            // sample entry rule
            ShortAtStop( bar+1, sma[bar]*0.97, "3% band around SMA" );
        }
    }
}
```

SymbolInfo Object

The SymbolInfo object represents a number of symbol's properties: Decimals, Margin, Point Value, Security Type and Tick.

Remarks

- The SymbolInfo object's properties should not be altered dynamically in a script, and if altered, the final value assigned to a property will change the Property value stored in the SymbolInfo Manager.

Decimals Property

int Decimals

Specifies the number of decimals that should be used when displaying the price values in the Bars object.

Example

```
protected override void Execute(){
    // Number of decimals
    SymbolInfo si = Bars.SymbolInfo;
    PrintDebug( "Decimals = " + si.Decimals);
}
```

Margin Property

double Margin

Returns the margin value if the Bars object contains data for a futures contract. The margin value is the amount deducted in backtesting for buying or shorting a single contract.

Example

```
protected override void Execute(){
    // Commodity Selection Index by Welles Wilder Jr. (c) 1979
    int Commission = 8;
    int adxPeriod = 14;
    DataSeries CSI = new DataSeries( Bars, "Commodity Selection Index (CSI)" );
    SymbolInfo si = Bars.SymbolInfo;

    if( si.Margin > 0 )
    {
        for(int bar = adxPeriod; bar < Bars.Count; bar++)
        {
            CSI[bar] = ADXR.Series( Bars, adxPeriod )[bar] * ATR.Series( Bars, adxPeriod )[bar] * ( ( si.PointValue / Math.Sqrt( si.Margin ) ) * (float) 1 / ( 150 + Commission )  ) * 100;
        }
    } else
        // Will not execute if margin is not specified in Symbol Info Manager
    Abort();

    // Plot Commodity Selection Index on chart
    ChartPane CSIPane = CreatePane( 75, true, true );
    PlotSeries( CSIPane, CSI, Color.Blue, WealthLab.LineStyle.Solid, 2 );

}
```

PointValue Property

double PointValue

Returns the point value if the Bars object contains data for a futures contract. The point value represents how much profit is gained when a single contract moves up one full point.

Remarks

- The default point value for stocks is 1, but can be adjusted in the Symbol Info Manager.
- See the "Futures Mode" topic in the Reference chapter of the User Guide for more information.

Example

```
protected override void Execute(){
    // "The Price Movement Index", as found in the book by Nauzer Balsara,
    // "Money Management Strategies for Futures Traders"

    int Sessions = 10; // no. of trading sessions to measure dollar value of price move
    double DollarValueInTick, TicksInPriceMove, DollarValue, PriceMovementIndex;
    DataSeries PMI = new DataSeries( Bars, "Price Movement Index (PMI)" );

    SymbolInfo si = Bars.SymbolInfo;
    DollarValueInTick = si.Tick * si.PointValue;

    if( si.Margin > 0 )
    {
        for(int bar = Sessions; bar < Bars.Count; bar++)
        {
            TicksInPriceMove = ( Highest.Value( bar, High, Sessions ) - Lowest.Value( bar, Low, Sessions ) ) / si.Tick;
            DollarValue = DollarValueInTick * TicksInPriceMove;
            PMI[bar] = DollarValue / si.Margin * 100;
        }
    } else
        // Will not execute if margin was not found in Symbol Info Manager
        Abort();

    // Plot Price Movement Index on chart
    ChartPane PMIPane = CreatePane( 75, true, true );
    PlotSeries( PMIPane, PMI, Color.Blue, WealthLab.LineStyle.Solid, 2 );

}
```

SecurityType Property

SecurityType SecurityType

Returns the type of data contained in the Bars object. Possible values are:

- Equity
- Future
- MutualFund

Example

```
protected override void Execute(){
    SymbolInfo si = Bars.SymbolInfo;
    // Sense equity/future to switch trading logic
    if ( si.SecurityType == WealthLab.SecurityType.Future )
    {
        // Commodities trading logic
    }
    else
    {
        // Stocks trading logic
    }
}
```

Tick Property

double Tick

Returns the tick value if the Bars object contains data for a futures contract. The tick value represents the granularity of the futures contract. Wealth-Lab will adjust limit and stop order prices so that they conform to the tick level of the contract. For example, if the contract tick value is 0.25, a BuyAtLimit or ShortAtStop order generated at 12.34 will be rounded to 12.25.

Remarks

- The default tick value for stocks is 0.01, but can be adjusted in the Symbol Info Manager.
- See the "Futures Mode" topic in the Reference chapter of the User Guide for more information.

Example

```
protected override void Execute(){
    double stop;

    // Calculate stop value using the Symbol Info Manager data (must be entered)
    switch (Bars.SymbolInfo.Symbol)
    {
        case "CL_RAD":
            stop = 1000 * Bars.SymbolInfo.Tick;
            break;
        case "NG_RAD":
            stop = 500 * Bars.SymbolInfo.Tick;
            break;
        default:
            stop = 700 * Bars.SymbolInfo.Tick;
            break;
    }

    PrintDebug( "Stop value is " + stop + " ticks" );
}
```

System

The System category contains various miscellaneous methods that apply to the overall Wealth-Lab system.

Abort

void Abort();

Causes the Strategy to immediately cease execution.

Example

```
protected override void Execute(){
    if ( Bars.Count < 1000 )
    {
        Abort();
    }
}
```

ClearDebug

void ClearDebug();

Clears all Debug Window messages.

Remarks

- Messages accumulate in the Debug Window during Strategy execution and Multi-Symbol Backtest until they are cleared by calling ClearDebug or by clicking the Clear button in the Debug Window's toolbar.
- When actively debugging a script, call ClearDebug at the beginning to refresh the Debug Window for new messages.

Example

```
protected override void Execute(){
    for(int i = 0; i < 1; i++)
    {
        PrintDebug( "Test string" );
    }
    ClearDebug();
}
```

ClearGloblals

void ClearGlobals();

Completely clears any objects that were stored in the Global Object Pool (GOP) via calls to SetGloblal.

Remarks

- See GetGlobal and SetGlobal for more information on the Global Object Pool.

Example

```
protected override void Execute(){
    // Run example for SetGlobal first
    DataSeries average = (DataSeries) GetGlobal("average");
    if( average.Count > 0 )
        PrintDebug( GetGlobal("average").ToString() + " found in GOP; BarCount = " + average.Count.ToString() );
    ClearGlobals();
    if ( GetGlobal("average").ToString() == "" )
        PrintDebug ( "GOP was cleared" ); // null
}
```

CreateParameter Method

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

Used in a Strategy class constructor to create a **StrategyParameter** type. 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.

Remarks

- Strategy Parameters are optional.
- Details about incorporating Strategy Parameters can be found in the WealthScript Language Guide.
- **Known issue**: Non-white space character cannot be typed if included in *CreateParameter* after ampersand. After including an ampersand as part of *CreateParameter* description (e.g. "L&S"), you will not be able to type the character after the ampserand (i.e. "S"), whitespace excluded, in that Strategy Window after compiling.
    - *Workaround:* Don't use an ampersand for the parameter's string name. If you must, just leave a space after it.

Example

```
protected override void Execute(){
    /*     See pre-built Strategies such as the "Glitch Index"
        and "Moving Average Crossover", or the ShortAtClose example
        in the QuickRef.*/
}
```

FlushDebug

void FlushDebug(string message);

Forces any debug messages that have been generated during the Strategy execution (by calling **PrintDebug**) to be displayed in the Debug Window immediately. Normally, all debug messages are displayed after the Strategy completes its execution.

Example

```
protected override void Execute(){
    PrintDebug( "Now You See Him" );
    FlushDebug( );
    System.Windows.Forms.MessageBox.Show( "Try commenting FlushDebug", "Message from WL5");
    // The debug string will not be seen until the messagebox is closed
}
```

GetChartBitmap

Bitmap GetChartBitmap(int width, int height);

Renders an image of the chart, including plotted indicators and manually drawn objects, as a Bitmap of the specified **width** and **height**. Use Bitmap.Save method to save the image to a file of a particular image type.

Example

```
protected override void Execute(){
    // Captures screen image in a PNG file under the WL installation directory
    Bitmap bm = GetChartBitmap( 500, 300 );
    bm.Save( Bars.Symbol + ".png", System.Drawing.Imaging.ImageFormat.Png );
    //For System.Drawing.Bitmap details, refer to MSDN:
    //http://msdn2.microsoft.com/en-us/library/system.drawing.bitmap_members.aspx
}
```

GetGlobal

object GetGlobal(string key);

The Global Object Pool (GOP) is a global storage area that Strategies can place objects into (SetGlobal) and at some point in the future read objects from (GetGlobal). Objects remain in the GOP throughout the lifetime of the Wealth-Lab application, and can be shared among Strategies that operate in any context (Strategy Window, Strategy Explorer, etc.)

Each object in the GOP has a unique string **key** associated with it. GetGlobal returns the object in the GOP with the specified **key**. If the object was not found, the method returns null.

Remarks

- You will need to cast the resulting object to the type you are expecting before being able to work with it.

Example

```
protected override void Execute(){
    // You should run the SetGlobal example before executing this
    // Getting entire series from the global storage is also convenient
    // But first we cast the object into DataSeries
    DataSeries average = (DataSeries) GetGlobal("average");
    ChartPane averagePane = CreatePane( 75, true, false );
    PlotSeries( averagePane, average, Color.Black, WealthLab.LineStyle.Solid, 1 );
}
```

GetTradingLoopStartBar Property

int GetTradingLoopStartBar(int startBar)

Returns the larger of two parameters:

- the passed startBar value, or
- the largest value of the StrategyParameters that have "period" in their Name property.

Especially when optimizing Strategies that use indicators with multiple periods, employ GetTradingLoopStartBar as the initial bar index for the trading loop to prevent runtime errors in the script or creating trades before all indicators are valid.

Example

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

namespace WealthLab.Strategies
{
    public class ChannelBreakoutExample : WealthScript
    {
        private StrategyParameter p1;
        private StrategyParameter p2;

        public ChannelBreakoutExample()
        {
            p1 = CreateParameter("Period High",20,2,200,20);
            p2 = CreateParameter("Period Low",40,2,200,20);
        }

        protected override void Execute()
        {
            Highest h = Highest.Series(High, p1.ValueInt);
            Lowest l = Lowest.Series(Low, p2.ValueInt);

            PlotSeries(PricePane, h >> 1, Color.Red, LineStyle.Solid, 1);
            PlotSeries(PricePane, l >> 1, Color.Green, LineStyle.Solid, 1);

            for(int bar = GetTradingLoopStartBar(1); bar < Bars.Count; bar++)
            {
                if (IsLastPositionActive)
                {
                    SellAtStop(bar + 1, LastPosition, Lowest.Series(Low, p2.ValueInt)[bar]);
                }
                else
                {
                    BuyAtStop(bar + 1, Highest.Series(High, p1.ValueInt)[bar]);
                }
            }
        }
    }
}
```

IsStreaming Property

bool IsStreaming

Returns a bool value indicating whether the Strategy is executing on a streaming data source or a static data source. Wealth-Lab executes Strategies on streaming data sources each time a new bar of data is completely formed for the current chart time scale.

Example

```
protected override void Execute(){
    // For example, use IsStreaming to disable IsLastBarOfDay logic
```

```
    for(int bar = 20; bar < Bars.Count; bar++)
    {
        bool LastBar = Bars.IsLastBarOfDay(bar);

        if (IsLastPositionActive)
        {
            if ( LastBar & !IsStreaming )
            {
                SellAtClose( bar, LastPosition, "EOD" );
            }
        }
        else
        {
            // plain vanilla entry rule
            BuyAtStop( bar+1, Highest.Series( High, 20 )[bar] );
        }
    }
}
```

PrintDebug

void PrintDebug(string message);
void PrintDebug(object message);
void PrintDebug(Object[] messages);

Prints the string specified by **message** to the application Debug Window, and displays the Debug Window if it is currently not visible. For performance reasons, Wealth-Lab caches all of the printed debug strings internally and finally displays them in the Debug Window after the Strategy finishes executing. To force the debug messages to appear during a Strategy execution, call **FlushDebug**.

Example

```
protected override void Execute(){
    for(int bar = 60; bar < Bars.Count; bar++)
    {
        // Print the bars where there were SMA crossovers
        if ( CrossOver( bar, SMA.Series( Close, 20 ), SMA.Series( Close, 60 ) ) )
        {
            PrintDebug( bar );
        }
    }
}
```

PrintStatusBar

void PrintStatusBar(string message);

Displays the string specified in **message** to the main status bar. Caution: printing too many times to the status bar, for example printing during each bar of data in the Strategy main loop, can result in a significant slow down of your Strategy execution speed.

Example

```
protected override void Execute(){
//Execution progress in status bar

    for(int bar = 20; bar < Bars.Count; bar++)
        PrintStatusBar("Processing " + ( bar * 100 / Bars.Count ) + "% complete");
}
```

RemoveGlobal

void RemoveGlobal(string key);
void RemoveGlobal(object value);

Removes an object from the Global Object Pool (GOP) by either **key** or **value**.

Remarks

- See GetGlobal and SetGlobal for a description of the Global Object Pool.

Example

```
protected override void Execute(){
    // Run example for SetGlobal first
    PrintDebug ( GetGlobal("average").ToString() );
    RemoveGlobal( "average" );
    PrintDebug ( GetGlobal("average").ToString() ); // null
}
```

SetGlobal

void SetGlobal(string key, object value);

The Global Object Pool (GOP) is a global storage area that Strategies can place objects into (SetGlobal) and at some point in the future read object from (GetGlobal). Objects remain in the GOP throughout the lifetime of the Wealth-Lab application, and can be shared among Strategies that operate in any context (Strategy Window, Strategy Explorer, etc.)

Each object in the GOP has a unique string **key** associated with it. SetGlobal places an object (**value**) into the GOP, using the specified **key**. This will overwrite any existing object that was placed using the same **key**.

Example

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

namespace WealthLab.Strategies
{
    public class MyStrategy : WealthScript
    {
        // Put series in the global storage
        void SetGlobalSeries( string sName, DataSeries series )
        {
            SetGlobal( sName, series );
        }

        protected override void Execute()
```

```
        {
            SetGlobalSeries( "average", ((High + Low)/2) );
        }
    }
}
```

StrategyName Property

string StrategyName

Returns the name of the Strategy that is currently being executed.

Example

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

namespace WealthLab.Strategies
{
    public class AlertsToFile : WealthScript
    {
        // Collects generated alerts and writes them into file
        void WriteAlerts()
        {
            StreamWriter alertFile;
            string str;

            if( Alerts.Count > 0 )
            {
                // Open output file
                alertFile = new StreamWriter( "Alerts.txt", true );
                // Strategy Name
                alertFile.Write( "Alert for strategy name: " + StrategyName + "\r\n" );

                for( int i = 0; i < Alerts.Count; i++ )
                {
                    WealthLab.Alert a = Alerts[i];

                    str = ( "AlertDate: " + a.AlertDate ) + "\r\n" +
                        ( "AlertType: " + a.AlertType ) + "\r\n" +
                        ( "OrderType: " + a.OrderType ) + "\r\n" +
                        ( "PositionType: " + a.PositionType ) + "\r\n" +
                        ( "Price: " + a.Price ) + "\r\n" +
                        ( "Symbol: " + a.Symbol ) + "\r\n" +
                        ( "Shares: " + a.Shares ) + "\r\n" +
                        ( "SignalName: " + a.SignalName ) + "\r\n" + "\r\n";

                    // Creates the file containing alerts under WLP installation folder
                    alertFile.Write( str );
                }
                alertFile.Close();
            }
        }

        protected override void Execute()
        {
            for(int bar = 40; bar < Bars.Count; bar++)
            {
                if (IsLastPositionActive)
                    SellAtStop( bar+1, LastPosition, Lowest.Series( Bars.Low, 20 )[bar], "Exit" );
                else
                    BuyAtStop( bar+1, Highest.Series( Bars.High, 40 )[bar], "Entry" );
            }

            WriteAlerts();
        }
    }
}
```

Technical Indicators

The Technical Indicators category contains information on all of the indicators that are available in the WealthLab.Indicators standard indicators library. Indicators are all DataSeries objects, and are created in scripts by using the Series method, as shown below:

```
DataSeries sma20 = SMA.Series(Close, 20);
```

Additionally, some indicators support a Value method that you can use to calculate and return the value of an indicator on a specific bar. Value methods always recalculate their value each time they are called.

```
double smaValue = SMA.Value(bar, Close, 20);
```

Time Frames

The Time Frames category contains methods you can use to access different time frames (such as weekly or monthly) within your Strategy.

AddCalendarDays

int AddCalendarDays(bool interpolate)

Adds all missing calendar days to the chart data, including weekends, holidays, and any other non-trading day. Newly added bars are considered "synthetic", and these bar numbers return true when **Bars.InSynthetic** is called. AddCalendarDays returns the number of new bars that were added.

The value of the inserted bars depends on the **interpolate** parameter. If **interpolate** is false, the new bars assume the OHLC values of the next actual bar. If **interpolate** is true, the OHLC values of the new bars are calculated using linear interpolation between the previous bar and the next actual bar. Note that interpolating values will result in the bars being created based on future information (next bar's value) so be careful if using these bars in trading system development.

Remarks

- AddCalendarDays is available in Daily scale only.
- AddCalendarDays is not compatible with streaming Strategies.
- Be careful to avoid peeking as AddCalendarDays uses future data.
- **Known issue**: *AddCalendarDays* breaks chart scroll
- **Known issue**: *Bars.IsSynthetic* wrongly marks the first trading bar after a series of synthetic bars added by *AddCalendarDays*. It does not work as documented, i.e. synthetic bars are not marked.
- **Known issue**: *AddCalendarDays* always peeks, even when the interpolate parameter is set to false. A work-around to that behavior can be found in this forum thread.

Example

```
protected override void Execute(){

    if( Bars.Scale == 0 )
    {
        DrawLabel( PricePane, Bars.Count + " bars before", Color.Black );
        int added = AddCalendarDays( true );
        DrawLabel( PricePane, Bars.Count + " bars after", Color.Black );
        DrawLabel( PricePane, "Added " + added + " bars", Color.Blue );
    } else
    DrawLabel( PricePane, "Daily data required...", Color.LightCoral );
}
```

RestoreScale

void RestoreScale();

Restores the data scale that the Strategy is currently operating on back to the original scale that it was invoked on. The internal data scale can be changed by calling the various SetScale methods.

Remarks

- RestoreScale restores the data scale, but preserves the current context symbol (which may have been changed via SetContext).

Example

```
protected override void Execute(){
// Chart SMA from 30 minute compressed data on a lower scale
    SetScaleCompressed( 30 );
    DataSeries SMA10_60 = SMA.Series( Close, 10 );
    RestoreScale();
    SMA10_60 = Synchronize( SMA10_60 );

    PlotSeries( PricePane, SMA10_60, Color.Blue, WealthLab.LineStyle.Solid, 1 );
}
```

SetScaleCompressed

void SetScaleCompressed(int barInterval);

Changes the base time scale of the Strategy to a more highly compressed intraday scale. The context Bars is replaced with a new Bars object that is compressed to the specified **barInterval**. For example, if the source data is a 5 minute chart, you can compress the data to 10, 15, or 30 minute scale (any multiple of 5). Any indicators, and external symbols produced will also be in the compressed scale. If you need to plot any of the compressed DataSeries or Bars, you must first expand them to the original intraday scale using the Synchronize method. Call RestoreScale to revert the Strategy back to the original intraday time scale.

Important!

You must call RestoreScale() to return to the original time scale particularly for plotting and executing trading signals. In general, only remain in a compressed scale to create indicators and immediately revert to the base scale by calling RestoreScale().

Remarks

- **SetScaleCompressed** only works on charts using intraday scaled data.
- You can compress data in Minute, Second, or Tick scales. The resulting compressed data retains the corresponding source base scale. It is currently not possible through WealthScript methods to compress second or tick based data to minute, for example, but this is possible by directly using the **BarScaleConvertor** utility class.
- **SetScaleCompressed** operates only on the standard OHLC/V DataSeries of the Bars object and does not apply to manually created and/or Named DataSeries.

Example

```
protected override void Execute(){
    // The chart will depict 20-minute SMA and RSI
    // on compressed and original scales

    DataSeries SMA20 = SMA.Series( Close, 20 );
    SetScaleCompressed( 15 );
    DataSeries SMA20_15 = SMA.Series( Close, 20 );
    RestoreScale();
    SMA20_15 = Synchronize( SMA20_15 );

    PlotSeries( PricePane, SMA20, Color.Red, WealthLab.LineStyle.Solid, 1 );
    PlotSeries( PricePane, SMA20_15, Color.Blue, WealthLab.LineStyle.Solid, 1 );

    ChartPane RSIPane = CreatePane( 50, true, true );
    SetScaleCompressed( 15 );
    DataSeries RSI20_15 = RSI.Series( Close, 20 );
    RestoreScale();
    RSI20_15 = Synchronize( RSI20_15 );

    PlotSeries( RSIPane, RSI.Series( Close, 20 ), Color.Red, WealthLab.LineStyle.Solid, 1 );
    PlotSeries( RSIPane, RSI20_15, Color.Blue, WealthLab.LineStyle.Solid, 1 );
}
```

SetScaleDaily

void SetScaleDaily();

Changes the base time scale of the Strategy to daily, from intraday. The context Bars is replaced with a new Bars object compressed to the daily scale. Any indicators, and external symbols produced will also be in daily scale. If you need to plot any of the compressed daily DataSeries or Bars, you must first expand them to the original intraday scale using the Synchronize method. Call RestoreScale to revert the Strategy back to the original time scale.

Important!

You must call RestoreScale() to return to the original time scale particularly for plotting and executing trading signals. In general, only remain in a compressed scale to create indicators and immediately revert to the base scale by calling RestoreScale().

Remarks

- **SetScaleDaily** only works on charts using intraday scaled data.
- **SetScaleDaily** operates only on the standard OHLC/V DataSeries of the Bars object and does not apply to manually created and/or Named DataSeries.

Example

```
protected override void Execute(){
//Look for a Daily SMA CrossOver in our intraday chart
    if ( Bars.IsIntraday )
    {
        SetScaleDaily();
        DataSeries SMA1 = SMA.Series( Close, 10 );
        DataSeries SMA2 = SMA.Series( Close, 40 );
        RestoreScale();
        SMA1 = Synchronize( SMA1 );
        SMA2 = Synchronize( SMA2 );
        PlotSeries( PricePane, SMA1, Color.Red, WealthLab.LineStyle.Solid, 1 );
        PlotSeries( PricePane, SMA2, Color.Blue, WealthLab.LineStyle.Solid, 1 );
        for(int bar = 20; bar < Bars.Count; bar++)
        {
            if ( CrossOver( bar, SMA1, SMA2 ) )
                SetBackgroundColor( bar, Color.Blue );
        }
    }
}
```

SetScaleMonthly

void SetScaleMonthly();

Changes the base time scale of the Strategy to monthly. The context Bars is replaced with a new Bars object compressed to the monthly scale. Any indicators, and external symbols produced will also be in monthly scale. If you need to plot any of the compressed DataSeries or Bars, you must first expand them to the original scale using the Synchronize method. Call RestoreScale to revert the Strategy back to the original time scale.

Important!

You must call RestoreScale() to return to the original time scale particularly for plotting and executing trading signals. In general, only remain in a compressed scale to create indicators and immediately revert to the base scale by calling RestoreScale().

Remarks

- **SetScaleMonthly** operates only on the standard OHLC/V DataSeries of the Bars object and does not apply to manually created and/or Named DataSeries.
- **Known issue**: Applying *SetScaleMonthly* to a compressed Weekly chart (source data is Daily) of an external symbol may result in an incorrect compressed data (month has more than 4 weeks). Use a workaround from this forum thread.

Example

```
protected override void Execute(){
    // Plot the 5 month RSI in our daily chart
    SetScaleMonthly();
    DataSeries MonthlyRSI = RSI.Series( Close, 5 );
    RestoreScale();
    MonthlyRSI = Synchronize( MonthlyRSI );
    ChartPane RSIPane = CreatePane( 50, true, true );
    PlotSeries( RSIPane, MonthlyRSI, Color.Navy, WealthLab.LineStyle.Solid, 2 );
}
```

SetScaleWeekly

void SetScaleWeekly();

Changes the base time scale of the Strategy to weekly. The context Bars is replaced with a new Bars object compressed to the weekly scale. Any indicators, and external symbols produced will also be in weekly scale. If you need to plot any of the compressed DataSeries or Bars, you must first expand them to the original scale using the Synchronize method. Call RestoreScale to revert the Strategy back to the original time scale.

Important!

You must call RestoreScale() to return to the original time scale particularly for plotting and executing trading signals. In general, only remain in a compressed scale to create indicators and immediately revert to the base scale by calling RestoreScale().

Remarks

- **SetScaleWeekly** operates only on the standard OHLC/V DataSeries of the Bars object and does not apply to manually created and/or Named DataSeries.

Example

```
protected override void Execute(){
    // Plot the weekly MACD in our daily chart
    SetScaleWeekly();
    DataSeries WeeklyMACD = MACD.Series( Close );
    RestoreScale();
    WeeklyMACD = Synchronize( WeeklyMACD );
    ChartPane MACDPane = CreatePane( 50, true, true );
    PlotSeries( MACDPane, WeeklyMACD, Color.Maroon, WealthLab.LineStyle.Histogram, 2 );
}
```

Trading

The Trading category contains methods used to enter and exit long and short Positions.

AutoProfitLevel Property

double AutoProfitLevel

Specifies the initial profit target level for the next Position to be created. The value, analogous to **RiskStopLevel**, is the **price** at which the same-bar Limit order should be placed. It is valid for any BarScale.

Remarks

- **AutoProfitLevel** should be set if "same bar exits" wish to be used in **real-time trading**. It does not have any effect in backtesting.

Example

```
protected override void Execute(){
PlotStops();
    int bcm1 = Bars.Count - 1;
    DataSeries sma1 = SMA.Series(Close, 8);
    DataSeries sma2 = SMA.Series(Close, 20);
    PlotSeries(PricePane, sma1, Color.Green, LineStyle.Solid, 1);
    PlotSeries(PricePane, sma2, Color.Red, LineStyle.Solid, 1);

    for(int bar = Bars.FirstActualBar + 20; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            Position p = LastPosition;
            SellAtLimit( bar+1, p, p.AutoProfitLevel * 1.01 );
        }
        else if ( CrossOver(bar, sma1, sma2) )
        {
            AutoProfitLevel = Bars.High[bar];
            // also use same-bar exit for backtesting
            if (BuyAtMarket( bar+1 ) != null && bar < bcm1)
                SellAtLimit( bar + 1, LastPosition, LastPosition.AutoProfitLevel, "same-bar exit" );
        }
    }
}
```

BuyAtClose

Position BuyAtClose(int bar, string signalName);
Position BuyAtClose(int bar);

Buys a new long position at the specified **bar**, using the closing price of the bar as the entry price. The position size will be calculated based on the closing price of the previous bar. Returns a new **Position** object that represents the newly established position.

Remarks

- Slippage, when activated, can affect the trade's execution price.
- The optional **signalName** parameter will appear in the Strategy window trade list report.

Example

```
protected override void Execute(){
    for(int bar = 3; bar < Bars.Count; bar++)
    {
        if (!IsLastPositionActive)
        {
            // Three consecutive lower closes
            if( ( Bars.Close[bar] < Bars.Close[bar-1] ) &
                ( Bars.Close[bar-1] < Bars.Close[bar-2] ) &
                ( Bars.Close[bar-2] < Bars.Close[bar-3] ) )
                BuyAtClose( bar );
        }
        if (IsLastPositionActive)
        {
            SellAtMarket( bar+1, LastPosition );
        }
    }
}
```

BuyAtLimit

Position BuyAtLimit(int bar, double limitPrice, string signalName);
Position BuyAtLimit(int bar, double limitPrice);

Buys a new long position at the specified **bar**, using a limit order at the specified **limitPrice**. The position size will be calculated based on the **limitPrice**. If the price of the bar reaches the **limitPrice** or lower, BuyAtLimit returns a new **Position** object that represents the newly established position. If the **limitPrice** was not reached, BuyAtLimit returns **null**.

Remarks

- Slippage, when activated, can cause limit orders to fail, even if the price of the bar reaches the **limitPrice**.
- If the market open below the **limitPrice**, the entry price of the position will be set to the market open price of the **bar**.
- The optional **signalName** parameter will appear in the Strategy window trade list report.

Example

```
protected override void Execute(){
    for(int bar = 20; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            //code your exit rules here
        }
        else
        {
            // Buy at limit at last bar's high minus 1.5 * 5-period ATR
            BuyAtLimit( bar+1, High[bar]-1.5*ATR.Series( Bars, 5 )[bar] );
        }
    }
}
```

BuyAtMarket

Position BuyAtMarket(int bar, string signalName);
Position BuyAtMarket(int bar);

Buys a new long position at the specified **bar**, using the open price of the bar as the entry price. The position size will be calculated based on the closing price of the previous bar.

Remarks

- Slippage, when activated, can affect the trade's execution price.
- The optional **signalName** parameter will appear in the Strategy window trade list report.

Example

```
protected override void Execute(){
    // Open long position on the following bar based on this bar's indicator values
    DataSeries sma = SMA.Series( Close, 20 );
    for(int bar = 20; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            // Exit trade after 5 days
            if ( bar+1 - LastPosition.EntryBar >= 5 )
            SellAtMarket( bar+1, LastPosition, "Time-Based" );
        }
        else
        {
            // Buy at market next bar when the recent closing price crosses over the 20-period SMA
            if ( CrossOver( bar, Close, sma ) )
                BuyAtMarket( bar+1, "SMA CrossOver" );
        }
    }
}
```

BuyAtStop

Position BuyAtStop(int bar, double stopPrice, string signalName);
Position BuyAtStop(int bar, double stopPrice);

Buys a new long position at the specified **bar**, using a stop order at the specified **stopPrice**. The position size will be calculated based on the **stopPrice**. If the price of the bar reaches the **stopPrice** or higher, BuyAtStop returns a new **Position** object that represents the newly established position. If the **stopPrice** was not reached, BuyAtStop returns **null**.

Remarks

- Slippage, when activated, can affect the trade's execution price.
- If the market open above the **stopPrice**, the entry price of the position will be set to the market open price of the **bar**.
- The optional **signalName** parameter will appear in the Strategy window trade list report.

Example

```
protected override void Execute(){
    DataSeries peak = Highest.Series( High, 20 );
    for(int bar = 20; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            //code your exit rules here
            //...
        }
        else
        {
            // Enter when the 20-period high is touched
            BuyAtStop( bar+1, peak[bar], "Breakout" );
        }
    }
}
```

CoverAtAutoTrailingStop

bool CoverAtAutoTrailingStop(int bar, Position pos, double triggerPct, double profitReversalPct, string signalName);
bool CoverAtAutoTrailingStop(int bar, Position pos, double triggerPct, double profitReversalPct);

Covers the short Position specified in the **pos** parameter at the specified **bar**, using a trailing stop order. The trailing stop is initiated only after the position reaches the profit level specified in the **triggerPct** parameter. The stop price is calculated based on the **profitReversalPct** parameter. This value indicates the percentage reversal in the Position's profit that should be used as a stop level.

For example, assume we specify 30 for **profitReversalPct**, and our short Position had an entry price of $12 and is currently at $10 (a 20% profit so far).

The total profit so far is $12 - $10 = $2

30% of $2 is $0.60

The stop order will be placed at $10 + $0.60 = $10.60

The trailing stop price is maintained with the Position, and it is modified only when the calculated stop price is below the current trailing stop price. **CoverAtAutoTrailingStop** returns a bool value indicating whether the price hit the current trailing stop level or above, and the Position was covered. **CoverAtAutoTrailingStop** will also return false if the **bar** specified is greater than the number of bars on the chart. In this case, a cover Alert will be generated instead.

Remarks

- Slippage, when activated, can affect a trade's execution price.
- If the market open above the current trailing stop price, the position will be covered at the market open price of the **bar**.
- The optional **signalName** parameter will appear in the Strategy window trade list report.
- To cover all active short Positions, specify **Position.AllPositions** in the **pos** parameter.
- The current trailing stop price level is available by accessing the **TrailingStop** Position property.

Example

```
protected override void Execute(){
    int period = 20;
    PlotStops();
    for(int bar = period; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            Position p = LastActivePosition;
            // Protect a 10% gain after giving back 25% to market
            if ( !CoverAtAutoTrailingStop( bar+1, p, 10, 25, "AutoStop" ) )
                // Stop loss at 10%
                CoverAtStop( bar+1, p, p.EntryPrice * 1.10, "Stop Loss" );
        }
        else
```

```
            // Enter on channel breakdown
            ShortAtStop( bar+1, Lowest.Series( Low, period )[bar] );
        }
    }
}
```

CoverAtClose

bool CoverAtClose(int bar, Position pos, string signalName);
bool CoverAtClose(int bar, Position pos);

Covers the short Position specified in the **pos** parameter at the specified **bar**, using the closing price of the bar as the exit price. CoverAtClose will return false if the **bar** specified is greater than the number of bars on the chart. In this case, a cover Alert will be generated instead.

Remarks

- Slippage, when activated, can affect the trade's execution price.
- The optional **signalName** parameter will appear in the Strategy window trade list report.
- To cover all active short Positions, specify **Position.AllPositions** in the **pos** parameter.

Example

```
protected override void Execute(){
    for(int bar = 2; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            // Exit short after 20 days on close
            if ( bar+1 - LastPosition.EntryBar >= 20 )
                CoverAtClose( bar, LastPosition, "20 day exit" );
        }
        else
        {
            //code your entry rules here
            ShortAtStop( bar+1, Low[bar] );
        }
    }
}
```

CoverAtLimit

bool CoverAtLimit(int bar, Position pos, double limitPrice, string signalName);
bool CoverAtLimit(int bar, Position pos, double limitPrice);

Covers the short Position specified in the **pos** parameter at the specified **bar**, using a limit order at the specified **limitPrice**. CoverAtLimit returns a bool value indicating whether the price reached the **limitPrice** or below, and the Position was sold. CoverAtLimit will also return false if the **bar** specified is greater than the number of bars on the chart. In this case, a cover Alert will be generated instead.

Remarks

- Slippage, when activated, can cause a limit order to fail, even if the price reaches the **limitPrice**.
- If the market opens below the **limitPrice**, the position will be covered at the market open of the **bar**.
- The optional **signalName** parameter will appear in the Strategy window trade list report.
- To cover all active short Positions, specify **Position.AllPositions** in the **pos** parameter.

Example

```
protected override void Execute(){
    for(int bar = 50; bar < Bars.Count; bar++)
    {
        double adaptiveTarget;
        double big = 4;
        double regular = 1.5;
        if (IsLastPositionActive)
        {
            Position p = LastPosition;
            if ( ROC.Value( bar, ADX.Series( Bars, 14 ), 2 ) > 0 )
            {
                adaptiveTarget = p.EntryPrice - big * ATR.Value( bar, Bars, 20 );
            }
            else
                adaptiveTarget = p.EntryPrice  - regular * ATR.Value( bar, Bars, 20 );
        //  Doubly Adaptive Profit Objective by Chuck LeBeau
            CoverAtLimit( bar + 1, LastPosition, adaptiveTarget, "Adaptive Profit Target" );
        }
        else
        {
            //code your entry rules here
            ShortAtStop( bar+1, Lowest.Value( bar, Low, 20 ) );
        }
    }
}
```

CoverAtMarket

bool CoverAtMarket(int bar, Position pos, string signalName);
bool CoverAtMarket(int bar, Position pos);

Covers the short Position specified in the **pos** parameter at the specified **bar**, using the open price of the bar as the exit price. CoverAtMarket will return false if the **bar** specified is greater than the number of bars on the chart. In this case, a cover Alert will be generated instead.

Remarks

- Slippage, when activated, can affect the trade's execution price.
- The optional **signalName** parameter will appear in the Strategy window trade list report.
- To cover all active short Positions, specify **Position.AllPositions** in the **pos** parameter.

Example

```
protected override void Execute(){
    DataSeries kl = KeltnerLower.Series( Bars, 10, 10 );
    DataSeries kh = KeltnerUpper.Series( Bars, 10, 10 );
    PlotSeries( PricePane, kh, Color.Blue, LineStyle.Solid, 1 );
    PlotSeries( PricePane, kl, Color.Red, LineStyle.Solid, 1 );
    for(int bar = 50; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            // Cover at market when price crosses over the Keltner UpperBand
```

```
            if ( CrossOver( bar, Close, kh ) )
                CoverAtMarket( bar + 1, LastPosition, "Keltner CrossOver" );
        }
        else
        {
            // Short if close crosses under the Keltner LowerBand
            if ( CrossUnder( bar, Close, kl ) )
                ShortAtMarket( bar + 1, "Keltner CrossUnder" );
        }
    }
}
```

CoverAtStop

bool CoverAtStop(int bar, Position pos, double stopPrice, string signalName);
bool CoverAtStop(int bar, Position pos, double stopPrice);

Covers the short Position specified in the **pos** parameter at the specified **bar**, using a stop order at the specified **stopPrice**. CoverAtStop returns a bool value indicating whether the price hit the **stopPrice** or above, and the Position was covered. CoverAtStop will also return false if the **bar** specified is greater than the number of bars on the chart. In this case, a cover Alert will be generated instead.

Remarks

- Slippage, when activated, can affect a trade's execution price.
- If the market open above the **stopPrice**, the position will be sold at the market open price of the **bar**.
- The optional **signalName** parameter will appear in the Strategy window trade list report.
- To cover all active short Positions, specify **Position.AllPositions** in the **pos** parameter.

Example

```
protected override void Execute(){
    for(int bar = 20; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            // Cover the short position if prices move against us by 10%
            CoverAtStop( bar+1, LastPosition, LastPosition.EntryPrice * 1.1, "10% Stop" );
        }
        else
        {
            //code your entry rules here
            ShortAtMarket( bar+1 );
        }
    }
}
```

CoverAtTrailingStop

bool CoverAtTrailingStop(int bar, Position pos, double stopPrice, string signalName);
bool CoverAtTrailingStop(int bar, Position pos, double stopPrice);

Covers the short Position specified in the **pos** parameter at the specified **bar**, using a trailing stop order. The trailing stop price is maintained with the Position, and it is modified only when the specified **stopPrice** is below the current trailing stop price. **CoverAtTrailingStop** returns a bool value indicating whether the price hit the current trailing stop level or above, and the Position was covered. **CoverAtTrailingStop** will also return false if the **bar** specified is greater than the number of bars on the chart. In this case, a cover Alert will be generated instead.

Remarks

- Slippage, when activated, can affect a trade's execution price.
- If the market open above the current trailing stop price, the position will be covered at the market open price of the **bar**.
- The optional **signalName** parameter will appear in the Strategy window trade list report.
- To cover all active short Positions, specify **Position.AllPositions** in the **pos** parameter.
- The current trailing stop price level is available by accessing the **TrailingStop** Position property.

Example

```
protected override void Execute(){
    PlotStops();
    int period = 20;
    SMA sma = SMA.Series( Close, period );
    PlotSeries( PricePane, sma, Color.BurlyWood, WealthLab.LineStyle.Solid, 1 );
    for(int bar = 3*period; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            Position p = LastActivePosition;
            // Initiate a trailing stop after a 5% gain
            if ( p.MFEAsOfBarPercent( bar ) > 5 )
            {
                CoverAtTrailingStop( bar+1, p, sma[bar], "Trailing Stop" );
            }
            else
                CoverAtStop( bar+1, p, p.EntryPrice * 1.10, "10% Stop Loss" );
        }
        else
        {
            // sample entry rule
            ShortAtStop( bar+1, sma[bar]*0.97, "3% band around SMA" );
        }
    }
}
```

ExitAtAutoTrailingStop

bool ExitAtAutoTrailingStop(int bar, Position pos, double triggerPct, double profitReversalPct, string signalName);
bool ExitAtTrailingStop(int bar, Position pos, double triggerPct, double profitReversalPct);

Provides a shortcut that allows you to use a common exit method for both long and short positions. Internally, **ExitAtAutoTrailingStop** routes to either **SellAtAutoTrailingStop** or **CoverAtAutoTrailingStop**, depending on the **PositionType** of the **Position** that was passed to it.

Example

```
protected override void Execute(){
    double atr, entry, stop;
    int period = 14;     // ATR period
    double mult = 2;    // ATR multiplier
    for(int bar = period * 2; bar < Bars.Count; bar++)
    {
        atr = ATR.Value( bar, Bars, period ) * mult;
        entry = Bars.Close[bar] + atr;
```

```
        if (IsLastPositionActive)
        {
            Position p = LastActivePosition;
            stop = p.EntryPrice;
            // Protect a 10% gain after giving back 25% to market
            if ( !ExitAtAutoTrailingStop( bar+1, p, 10, 25, "AutoStop" ) )
                // Auto-sensing stop loss at 10%
                if ( p.PositionType == PositionType.Long  )
                {
                    stop = p.EntryPrice * 0.9;
                }
                else
                    stop = p.EntryPrice * 1.1;
                ExitAtStop( bar+1, p, stop, "Stop Loss" );
        }
        else
        {
            // "Serendipity entry" (c) Chuck LeBeau
            if ( BuyAtStop( bar+1, entry ) == null )
            {
                entry = Bars.Close[bar] - atr*mult;
                ShortAtStop( bar+1, entry );
            }
        }
    }
}
```

ExitAtClose

bool ExitAtClose(int bar, Position pos, string signalName);
bool ExitAtClose(int bar, Position pos);

Provides a shortcut that allows you to use a common exit method for both long and short positions. Internally, **ExitAtClose** routes to either **SellAtClose** or **CoverAtClose**, depending on the **PositionType** of the **Position** that was passed to it.

Example

```
protected override void Execute(){
    // Gap Closer II ( Consecutive Gap Closer )
    bool gapDown, gapUp;
    for(int bar = 4; bar < Bars.Count-1; bar++)
    {
        if (IsLastPositionActive)
        {
            Position p = LastPosition;
            // Take no risk overnight
            ExitAtClose( bar, p );
        }
        else
        {
            // consecutive gap up
            gapUp = ( Bars.Open[bar+1] > Bars.High[bar] ) &
                ( Bars.Open[bar] > Bars.High[bar-1] ) &
                // 1st gap not filled
                ( Bars.Low[bar] > Bars.High[bar-1] ) &
                // 1st gap was larger than 2nd gap
                ( ( Bars.Open[bar]-Bars.High[bar-1] ) > ( Bars.Open[bar+1]-Bars.High[bar] ) );
            // consecutive gap down
            gapDown = ( Bars.Open[bar+1] < Bars.Low[bar] ) &
                ( Bars.Open[bar] < Bars.Low[bar-1] ) &
                // 1st gap not filled
                ( Bars.Low[bar-1] > Bars.High[bar] ) &
                // 1st gap was larger than 2nd gap
                ( ( Bars.Low[bar-1]-Bars.Open[bar] ) > ( Bars.Low[bar]-Bars.Open[bar+1] ) );
            // Buy/short the gap
            if ( gapDown )
                BuyAtMarket( bar+1, "Gap Down" );
            else if ( gapUp )
                ShortAtMarket( bar+1, "Gap Up" );
        }
    }
}
```

ExitAtLimit

bool ExitAtLimit(int bar, Position pos, double limitPrice, string signalName);
bool ExitAtLimit(int bar, Position pos, double limitPrice);

Provides a shortcut that allows you to use a common exit method for both long and short positions. Internally, **ExitAtLimit** routes to either **SellAtLimit** or **CoverAtLimit**, depending on the **PositionType** of the **Position** that was passed to it.

Example

```
protected override void Execute(){
    double entry, profit;
    int period = 10;     // period
    double mult = 2;    // ATR multiplier
    DataSeries hi = Highest.Series(High, period);
    DataSeries lo = Lowest.Series(Low, period);
    for(int bar = 20; bar < Bars.Count; bar++)
    {
        // Exit either position at limit @ entry price +/- 2 times the 20-period ATR
        if (IsLastPositionActive)
        {
            Position p = LastPosition;
            // The power of "?" ternary operator:
            profit = (LastPosition.PositionType == PositionType.Long) ? p.EntryPrice + ATR.Value(bar, Bars, period) * mult : profit = p.EntryPrice - ATR.Value(bar, Bars, period) * mult;
            ExitAtLimit( bar, LastPosition, profit );
        }
        else
        {
            // plain vanilla channel breakout
            if ( BuyAtStop( bar+1, hi[bar], "Long" ) == null )
            {
                ShortAtStop( bar+1, lo[bar], "Short" );
            }
        }
    }
}
```

ExitAtMarket

bool ExitAtMarket(int bar, Position pos, string signalName);
bool ExitAtMarket(int bar, Position pos);

Provides a shortcut that allows you to use a common exit method for both long and short positions. Internally, **ExitAtMarket** routes to either **SellAtMarket** or **CoverAtMarket**, depending on the **PositionType** of the **Position** that was passed to it.

Example

```
protected override void Execute(){
    int period = 20;    // SMA period
    bool Event;
    DataSeries sma = SMA.Series( Close, period );
    DataSeries hi = Highest.Series(High, period);
    DataSeries lo = Lowest.Series(Low, period);
    PlotSeries( PricePane, hi, Color.Blue, WealthLab.LineStyle.Dotted, 2 );
    PlotSeries( PricePane, lo, Color.Red, WealthLab.LineStyle.Dotted, 2 );
    PlotSeries( PricePane, sma, Color.DarkGreen, WealthLab.LineStyle.Solid, 2 );
    for(int bar = 20; bar < Bars.Count; bar++)
    {
        // Exit position at market on a SMA CrossUnder/CrossOver
        if (IsLastPositionActive)
        {
            Position p = LastPosition;
            // Ternary operator "?" fits with the Exit* operator syntax
            Event = ( LastPosition.PositionType == PositionType.Long ) ? ( CrossUnder( bar, Close, sma ) ) : ( CrossOver( bar, Close, sma ) );
            if ( Event )
            {
                ExitAtMarket( bar+1, p );
            }
        }
        else
        {
            // plain vanilla channel breakout
            if ( BuyAtStop( bar+1, hi[bar] ) == null )
            {
                ShortAtStop( bar+1, lo[bar] );
            }
        }
    }
}
```

ExitAtStop

bool ExitAtStop(int bar, Position pos, double stopPrice, string signalName);
bool ExitAtStop(int bar, Position pos, double stopPrice);

Provides a shortcut that allows you to use a common exit method for both long and short positions. Internally, **ExitAtStop** routes to either **SellAtStop** or **CoverAtStop**, depending on the **PositionType** of the **Position** that was passed to it.

Example

```
protected override void Execute(){
    double entry, level;
    int period = 20;     // period
    double mult = 3;    // ATR multiplier
    DataSeries hi = Highest.Series(High, period );
    DataSeries lo = Lowest.Series(Low, period );
    DataSeries atr = ATR.Series( Bars, period );
    for(int bar = 20; bar < Bars.Count; bar++)
    {
        // Exit position at a respective Chandelier stop
        if (IsLastPositionActive)
        {
            Position p = LastPosition;
            level = (LastPosition.PositionType == PositionType.Long) ? hi[bar] - (atr[bar] * mult ) : lo[bar] + (atr[bar] * mult );
            ExitAtStop( bar+1, p, level, "Chandelier exit" );
        }
        else
        {
            // plain vanilla channel breakout
            if ( BuyAtStop( bar+1, hi[bar], "Long" ) == null )
            {
                ShortAtStop( bar+1, lo[bar], "Short" );
            }
        }
    }
}
```

ExitAtTrailingStop

bool ExitAtTrailingStop(int bar, Position pos, double stopPrice, string signalName);
bool ExitAtTrailingStop(int bar, Position pos, double stopPrice);

Provides a shortcut that allows you to use a common exit method for both long and short positions. Internally, **ExitAtTrailingStop** routes to either **SellAtTrailingStop** or **CoverAtTrailingStop**, depending on the **PositionType** of the **Position** that was passed to it.

Example

```
protected override void Execute(){
    double entry, level;
    int period = 20;     // period
    double mult = 3;    // ATR multiplier
    DataSeries hi = Highest.Series( High, period );
    DataSeries lo = Lowest.Series(Low, period );
    DataSeries atr = ATR.Series( Bars, period );
    PlotStops();
    for(int bar = 20; bar < Bars.Count; bar++)
    {
        // Exit position at respective Chandelier stop
        if (IsLastPositionActive)
        {
            Position p = LastPosition;
            level = ( LastPosition.PositionType == PositionType.Long ) ? hi[bar] - ( atr[bar] * mult ) : lo[bar] + (atr[bar] * mult );
            ExitAtTrailingStop( bar+1, p, level, "Trailing Chandelier" );
        }
        else
```

```
        {
            // plain vanilla channel breakout
            if ( BuyAtStop( bar+1, hi[bar] ) == null )
            {
                ShortAtStop( bar+1, lo[bar] );
            }
        }
    }
}
```

RiskStopLevel Property

double RiskStopLevel

Specifies the initial stop level (price) for the next Position to be created. This stop level is used when you select the Maximum Risk Pct Position Sizing option. This option specifies the maximum amount of capital you are willing to risk on each trade. When this option is selected, you must set the value of **RiskStopLevel** in your Strategy code to indicate the initial stop loss value for a newly created Position.

For example, a simple channel breakout system might enter at the highest 20 bar high, and exit at the lowest 20 bar low. Prior to issuing the **BuyAtMarket**, or **BuyAtStop**, you should set **RiskStopLevel** to the lowest Low value of the past 20 bars as the initial stop level for the long Position.

Remarks

- If you select the Maximum Risk Pct position sizing option and do not set **RiskStopLevel** in your Strategy code, you will receive an error message when attempting to run the Strategy.
- You must also be diligent in your Strategy to actually use the established stop level as an exit. If you do not, the Strategy could lose considerably more than the Maximum Risk that you established in the Position Size setting.
- Once a RiskStopLevel is established for a Position, do not change it. The [last] value assigned to a Position's RiskStopLevel is used to determine % Risk sizing, consequently reassigning its value after the Position is established is effectively a peeking error.

Example

```
protected override void Execute(){
    PlotStops();
    for(int bar = 30; bar < Bars.Count; bar++)
    {
        ATR atr = ATR.Series( Bars, 14 );
        if (IsLastPositionActive)
        {
            Position p = LastPosition;
            // Protective stop: "Yo-Yo Exit" by Chuck LeBeau
            SellAtStop( bar+1, p, p.RiskStopLevel, "Yo-Yo Stop" );
            SellAtLimit( bar+1, p, p.EntryPrice*1.1, "10% profit" );
        }
        else
        {
            //Set our risk stop at an ATR unit below the last day's low
            RiskStopLevel = ( Bars.Low[bar] - 1*atr[bar] );
            BuyAtStop( bar+1, Bars.Close[bar]+2*atr[bar], "Volatility Breakout" );
        }
    }
}
```

SellAtAutoTrailingStop

bool SellAtAutoTrailingStop(int bar, Position pos, double triggerPct, double profitReversalPct, string signalName);
bool SellAtAutoTrailingStop(int bar, Position pos, double triggerPct, double profitReversalPct);

Sells the Position specified in the **pos** parameter at the specified **bar**, using a trailing stop order. The trailing stop is initiated only after the position reaches the profit level specified in the **triggerPct** parameter. The stop price is calculated based on the **profitReversalPct** parameter. This value indicates the percentage reversal in the Position's profit that should be used as a stop level.

For example, assume we specify 30 for profitReversalPct, and our Position had an entry price of $10 and is currently at $12 (a 20% profit so far).

The total profit so far is $12 - $10 = $2

30% of $2 is $0.60

The stop order will be placed at $12 - $0.60 = $11.40

The trailing stop price is maintained with the Position, and it is modified only when the calculated stop price is above the current trailing stop price. **SellAtAutoTrailingStop** returns a bool value indicating whether the price hit the current trailing stop level or below, and the Position was sold. **SellAtAutoTrailingStop** will also return false if the **bar** specified is greater than the number of bars on the chart. In this case, a sell Alert will be generated instead.

Remarks

- Slippage, when activated, can affect a trade's execution price.
- If the market open below the current trailing stop price, the position will be sold at the market open price of the **bar**.
- The optional **signalName** parameter will appear in the Strategy window trade list report.
- To sell all active long Positions, specify **Position.AllPositions** in the **pos** parameter.
- The current trailing stop price level is available by accessing the **TrailingStop** Position property.

Example

```
protected override void Execute(){
    int period = 20;
    DataSeries kama = KAMA.Series( Bars.Close, 20 );
    PlotStops();
    for(int bar = period*3; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            Position p = LastActivePosition;
            // Protect a 10% gain after giving back 25% to market
            if ( !SellAtAutoTrailingStop( bar+1, p, 10, 25, "10% AutoStop" ) )
                SellAtStop( bar+1, p, p.EntryPrice * 0.90, "10% stop loss" );
        }
        else
            // Enter when closing cross over KAMA
            if ( CrossOver( bar, Close, kama ) )
                BuyAtStop( bar+1, Close[bar] );
    }
}
```

SellAtClose

bool SellAtClose(int bar, Position pos, string signalName);
bool SellAtClose(int bar, Position pos);

Sells the Position specified in the **pos** parameter at the specified **bar**, using the closing price of the bar as the exit price. SellAtClose will return false if the **bar** specified is greater than the number of bars on the chart. In this case, a sell Alert will be generated instead.

Remarks

- Slippage, when activated, can affect the trade's execution price.
- The optional **signalName** parameter will appear in the Strategy window trade list report.
- To sell all active long Positions, specify **Position.AllPositions** in the **pos** parameter.

Example

```
protected override void Execute(){
    for(int bar = 2; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            // Exit long after 20 days
            if ( bar+1 - LastPosition.EntryBar >= 20 )
                SellAtClose( bar, LastPosition, "20 day exit" );
        }
        else
        {
            //code your entry rules here
            BuyAtStop( bar+1, High[bar] );
        }
    }
}
```

SellAtLimit

bool SellAtLimit(int bar, Position pos, double limitPrice, string signalName);
bool SellAtLimit(int bar, Position pos, double limitPrice);

Sells the Position specified in the **pos** parameter at the specified **bar**, using a limit order at the specified **limitPrice**. SellAtLimit returns a bool value indicating whether the price reached the **limitPrice** or above, and the Position was sold. SellAtLimit will also return false if the **bar** specified is greater than the number of bars on the chart. In this case, a sell Alert will be generated instead.

Remarks

- Slippage, when activated, can cause a limit order to fail, even if the price reaches the **limitPrice**.
- If the market opens above the **limitPrice**, the position will be sold at the market open of the **bar**.
- The optional **signalName** parameter will appear in the Strategy window trade list report.
- To sell all active long Positions, specify **Position.AllPositions** in the **pos** parameter.

Example

```
protected override void Execute(){
    DataSeries high = Highest.Series( High, 10 );
    DataSeries ema1 = EMA.Series( Close, 2, WealthLab.Indicators.EMACalculation.Modern );
    DataSeries ema2 = EMA.Series( Close, 10, WealthLab.Indicators.EMACalculation.Modern );
    PlotSeries( PricePane, ema1, Color.LightBlue, WealthLab.LineStyle.Solid, 2 );
    PlotSeries( PricePane, ema2, Color.ForestGreen, WealthLab.LineStyle.Solid, 2 );
    for(int bar = 30; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            // Try to get out at a recent high + some percent
            if ( SellAtLimit( bar+1, LastPosition, high[bar]*1.02, "Limit Sell" ) )
                PrintDebug( "Sold" );
        }
        else
        {
            // "Anti-Trend EMA" entry
            if ( CrossUnder( bar, ema1, ema2 ) )
                BuyAtMarket( bar+1 );
        }
    }
}
```

SellAtMarket

bool SellAtMarket(int bar, Position pos, string signalName);
bool SellAtMarket(int bar, Position pos);

Sells the Position specified in the **pos** parameter at the specified **bar**, using the open price of the bar as the exit price. SellAtMarket will return false if the **bar** specified is greater than the number of bars on the chart. In this case, a sell Alert will be generated instead.

Remarks

- Slippage, when activated, can affect the trade's execution price.
- The optional **signalName** parameter will appear in the Strategy window trade list report.
- To sell all active long Positions, specify **Position.AllPositions** in the **pos** parameter.

Example

```
protected override void Execute(){
    int shift = 4;
    DataSeries dma1 = SMA.Series( Close, 7 ) >> shift;
    DataSeries dma2 = SMA.Series( Close, 25 ) >> shift;
    PlotSeries( PricePane, dma1, Color.Green, LineStyle.Solid, 1);
    PlotSeries( PricePane, dma2, Color.Blue, LineStyle.Solid, 1);
    for(int bar = 30; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            // Displaced Moving Average (DMA) strategy exit
            if ( CrossUnder( bar, dma1, dma2 ) )
                SellAtMarket( bar+1, LastPosition, "DMA CrossUnder" );
        }
        else
        {
            // Displaced Moving Average (DMA) strategy entry
            if ( CrossOver( bar, dma1, dma2 ) )
                BuyAtMarket( bar+1, "DMA CrossOver" );
        }
    }
}
```

SellAtStop

bool SellAtStop(int bar, Position pos, double stopPrice, string signalName);
bool SellAtStop(int bar, Position pos, double stopPrice);

Sells the Position specified in the **pos** parameter at the specified **bar**, using a stop order at the specified **stopPrice**. SellAtStop returns a bool value indicating whether the price hit the **stopPrice** or below, and the Position was sold. SellAtStop will also return false if the **bar** specified is greater than the number of bars on the chart. In this case, a sell Alert will be generated instead.

Remarks

- Slippage, when activated, can affect a trade's execution price.
- If the market open below the **stopPrice**, the position will be sold at the market open price of the **bar**.
- The optional **signalName** parameter will appear in the Strategy window trade list report.
- To sell all active long Positions, specify **Position.AllPositions** in the **pos** parameter.

Example

```
protected override void Execute(){
    for(int bar = 20; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            Position p = LastPosition;
            // Cover the long position if prices move against us by 7%
            SellAtStop( bar+1, p, p.EntryPrice * 0.93, "7% Stop" );
        }
        else
        {
            //code your entry rules here
            BuyAtMarket( bar+1 );
        }
    }
}
```

SellAtTrailingStop

bool SellAtTrailingStop(int bar, Position pos, double stopPrice, string signalName);
bool SellAtTrailingStop(int bar, Position pos, double stopPrice);

Sells the Position specified in the **pos** parameter at the specified **bar**, using a trailing stop order. The trailing stop price is maintained with the Position, and it is modified only when the specified **stopPrice** is above the current trailing stop price. **SellAtTrailingStop** returns a bool value indicating whether the price hit the current trailing stop level or below, and the Position was sold. **SellAtTrailingStop** will also return false if the **bar** specified is greater than the number of bars on the chart. In this case, a sell Alert will be generated instead.

Remarks

- Slippage, when activated, can affect a trade's execution price.
- If the market open below the current trailing stop price, the position will be sold at the market open price of the **bar**.
- The optional **signalName** parameter will appear in the Strategy window trade list report.
- To sell all active long Positions, specify **Position.AllPositions** in the **pos** parameter.
- The current trailing stop price level is available by accessing the **TrailingStop** Position property.

Example

```
protected override void Execute(){
    PlotStops();
    int period = 20;
    DataSeries atr = ATR.Series( Bars, period );
    for(int bar = period; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            Position p = LastActivePosition;
            SellAtStop( bar+1, p, p.EntryPrice * 0.95, "5% Stop Loss" );
            // Trailing Chandelier exit (Chuck LeBeau) at 40-bar highest high minus 3 ATR units
            SellAtTrailingStop( bar+1, p, Highest.Series( High, period )[bar]-3*atr[bar], "Trailing Stop" );
        }
        else
        {
            // sample entry rule
            BuyAtStop( bar+1, Highest.Series( High, period )[bar] );
        }
    }
}
```

SetShareSize

double SetShareSize(double shares)

Use **SetShareSize** to assign a fixed number of Shares (or contracts) per Position in your Strategy. Subsequent trades will use the number of Shares or contracts that you specified.

Remarks

- In Raw Profit modes, SetShareSize does not have effect. It applies to Portfolio Simulation mode only.
- When using **SetShareSize**, you must choose the radio button for **WealthScript Override (SetShareSize)** in the Position Sizing control to enable **SetShareSize** to influence position sizing.

Example

```
protected override void Execute(){
SMA sma = SMA.Series( Close, 50 );

            for(int bar = sma.FirstValidValue; bar < Bars.Count; bar++)
            {
                if (IsLastPositionActive)
                {
                    /* Exit after N days */

                    Position p = LastPosition;
                    if ( bar+1 - p.EntryBar >= 3 )
                        SellAtMarket( bar+1, p, "Timed" );
                }
                else
                {
                    /* When the Close is below the SMA,
                    size the new position twice the normal */

                    if( Close[bar] > sma[bar] )
                        SetShareSize( 1000 );
                    else
                        SetShareSize( 2000 );

                    BuyAtLimit( bar+1, Close[bar]*0.93 );
                }
            }
```

```
        }
    }
}
```

ShortAtClose

Position ShortAtClose(int bar, string signalName);
Position ShortAtClose(int bar);

Enters a new short position at the specified **bar**, using the closing price of the bar as the entry price. The position size will be calculated based on the closing price of the previous bar. Returns a new **Position** object that represents the newly established short position.

Remarks

- Slippage, when activated, can affect the trade's execution price.
- The optional **signalName** parameter will appear in the Strategy window trade list report.

Example

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

namespace WealthLab.Strategies
{
    public class ShortAtCloseDemo : WealthScript
    {
        //Lowest Low parameter
        private StrategyParameter per;

        public ShortAtCloseDemo()
        {
            per = CreateParameter("CumUp period, days", 10, 10, 30, 5);
        }

        protected override void Execute()
        {
            //Obtain period from parameter
            int period = per.ValueInt;

            CumUp cumUp = CumUp.Series( Bars.Close, period );
            ChartPane cu = CreatePane( 50, true, false );
            PlotSeries( cu, cumUp, Color.Red, WealthLab.LineStyle.Dotted, 2 );

            for(int bar = period; bar < Bars.Count; bar++)
            {
                if (!IsLastPositionActive)
                {
                    if( Close[bar] > SMA.Series( Close,2 )[bar] )
                        // Short at Close on CumUp >= 5 in 10 bars
                        if( cumUp[bar] >= 5 )
                            ShortAtClose( bar );
                }
                if (IsLastPositionActive)
                {
                    // .. Exit Rules ...
                    CoverAtMarket( bar+1, LastPosition );
                }
            }
        }
    }
}
```

ShortAtLimit

Position ShortAtLimit(int bar, double limitPrice, string signalName);
Position ShortAtLimit(int bar, double limitPrice);

Enters a new short position at the specified **bar**, using a limit order at the specified **limitPrice**. The position size will be calculated based on the **limitPrice**. If the price of the bar reaches the **limitPrice** or higher, ShortAtLimit returns a new **Position** object that represents the newly established short position. If the **limitPrice** was not reached, ShortAtLimit returns **null**.

Remarks

- Slippage, when activated, can cause limit orders to fail, even if the price of the bar reaches the **limitPrice**.
- If the market open above the **limitPrice**, the entry price of the position will be set to the market open price of the **bar**.
- The optional **signalName** parameter will appear in the Strategy window trade list report.

Example

```
protected override void Execute(){
    Bars benchmark = GetExternalSymbol( "QQQQ", true );
    ChartPane mkt = CreatePane( 35, true, false );
    DataSeries adx = ADX.Series( Bars, 14 );
    DataSeries adx_market = ADX.Series( benchmark, 14 );
    DataSeries diP = DIPlus.Series( Bars, 14 );
    DataSeries diM = DIMinus.Series( Bars, 14 );
    PlotSeries( mkt, adx, Color.Red, WealthLab.LineStyle.Solid, 1 );
    PlotSeries( mkt, adx_market, Color.DarkRed, WealthLab.LineStyle.Solid, 2 );
    for(int bar = 30; bar < Bars.Count; bar++)
    {
        if (!IsLastPositionActive)
        {
            // Short symbol when it lags some general market benchmark
            if( adx[bar] < adx_market[bar] )
                if( diP[bar] < diM[bar] )
                    // Short next bar at the limit of 20-period high
                    ShortAtLimit( bar+1, Lowest.Value( bar, Bars.High, 20 ), "Short @ Limit" );
        }
        if (IsLastPositionActive)
        {
            //code your exit rules here
            //...
        }
    }
}
```

ShortAtMarket

Position ShortAtMarket(int bar, string signalName);
Position ShortAtMarket(int bar);

Enters a new short position at the specified **bar**, using the open price of the bar as the entry price. The position size will be calculated based on the closing price of the previous bar. Returns a new **Position** object that represents the newly established short position.

Remarks

- Slippage, when activated, can affect the trade's execution price.
- The optional **signalName** parameter will appear in the Strategy window trade list report.

Example

```
protected override void Execute(){
    DataSeries rsi = RSI.Series( Close, 14 );
    ChartPane rsiPane = CreatePane( 40, true, false );
    PlotSeries( rsiPane, rsi, Color.Chartreuse, WealthLab.LineStyle.Dotted, 2 );
    DrawHorzLine( rsiPane, 70, Color.Red, WealthLab.LineStyle.Dashed, 1 );
    for(int bar = 30; bar < Bars.Count; bar++)
    {
        if (IsLastPositionActive)
        {
            // Exit after 10 days
            if ( bar+1 - LastPosition.EntryBar == 10 )
                CoverAtMarket( bar+1, LastPosition, "Time-Based" );
        }
        else
        {
            // Establish a short position if RSI gets overbought
            if ( RSI.Series( Close, 14 )[bar] > 70 )
                ShortAtMarket( bar+1, "RSI Short Signal" );
        }
    }
}
```

ShortAtStop

Position ShortAtStop(int bar, double stopPrice, string signalName);
Position ShortAtStop(int bar, double stopPrice);

Enters a new short position at the specified **bar**, using a stop order at the specified **stopPrice**. The position size will be calculated based on the **stopPrice**. If the price of the bar reaches the **stopPrice** or lower, ShortAtStop returns a new **Position** object that represents the newly established short position. If the **stopPrice** was not reached, ShortAtStop returns **null**.

Remarks

- Slippage, when activated, can affect the trade's execution price.
- If the market open below the **stopPrice**, the entry price of the position will be set to the market open price of the **bar**.
- The optional **signalName** parameter will appear in the Strategy window trade list report.

Example

```
protected override void Execute(){
    for(int bar = 5; bar < Bars.Count; bar++)
    {
        if (!IsLastPositionActive)
        {
            // "Oops" ( Larry Williams )
            if ( Bars.Open[bar] > Bars.High[bar-1] )
                ShortAtStop( bar, Bars.Low[bar-1], "Oops" );
        }
        if (IsLastPositionActive)
            ExitAtClose( bar, LastPosition );
    }
}
```