Creating an Indicator Library in Wealth-Lab Pro®

Introduction

This article explains how to create technical indicator libraries for Wealth-Lab Pro®. These libraries contain compiled indicators that integrate into Wealth-Lab Pro and behave exactly like the indicators that ship with the product. They appear in their own folder in the Indicators list, and you can drag and drop them onto a chart and use them in General Indicator Wizard rules.

Building an Indicator Library requires a .NET development tool such as Microsoft Visual Studio or the free SharpDevelop, so this article is geared toward developers familiar with such a tool. If you do not want to get down to this level of complexity, you should know that you can also program and plot indicators directly in the Strategy code in Wealth-Lab Pro.

Indicator Library Basics

If you've stuck with the article this far, let's get into the nuts and bolts of what an Indicator Library is. In Wealth-Lab Pro, each individual indicator is implemented as a class that derives from the **DataSeries** base class. **DataSeries** manages a series of values (type **double**), and an associated list of **DateTimes** that represent a time-series of values.

Constructor

DataSeries can be created using one of 2 different constructors. The constructors represent the underlying data that the **DataSeries** is based on, which can be another **DataSeries**, or a **Bars** object (a Bars objects represents a complete OHLC/V data history for a security.)

When you create a new indicator class, create a constructor that matches one of the two **DataSeries** constructors, and calls the base constructor. If your indicator operates on a single **DataSeries** (for example, a Moving Average), select the constructor that takes a **DataSeries** parameter. If, however, your indicator requires open, high and low information (such as a Stochastic), or some other combination, use the constructor that take a **Bars** parameter.

Example using DataSeries

```
public SMA(DataSeries ds, int period, string description)
: base(ds, description)
```

Example using Bars

```
public StochD(Bars bars, int period, int smooth, string description)
: base(bars, description)
```

Note that in each case, the base constructor is called. This actually populates the DataSeries with zeroes, so you are all set to work with a fully populated series of values. It is now your job in the constructor to calculate the values of the indicator bar by bar, as described below.

Notice that the constructors also take a string parameter called description. This string represents a unique description of the indicator including its parameters. It is created in the "Series" static method, described below.

Your indicator can also take a number of optional parameters, like the "period" parameter of a moving average. Restrict these additional parameters to the following .NET types:

- DataSeries
- any enumerated type
- Bars (represents historical OHLC/V bars)
- Int32
- Double
- Boolean
- String
- DateTime

Assigning the Indicator Values

The indicator should assign its values directly in the constructor. Initially, the indicator is pre-populated with zeroes. Use the **Count** property to determine how many values to calculate and assign. Use the indexer access method (**this[]**) to assign values to the indicator as they are calculated.

To access individual values in a "source" **DataSeries**, use the built-in indexer (**source[]** notation). To access values in a source **Bars** object, use its Open, High, Low, Close and Volume properties, which are themselves **DataSeries** objects (**bars.Open[]**).

The sample code below populates the Simple Moving Average indicator.

```
//Calculate SMA values
for (int bar = period - 1; bar < ds.Count; bar++)
{
  double sum = 0;
  for (int innerBar = bar; innerBar > bar - period; innerBar--)
    sum += ds[innerBar];
  this[bar] = sum / period;
}
```

Controlling the First Plotted Bar

You can suppress plotting of initial values of your indicator. To accomplish this, set the **FirstValidValue** property to the value of the first bar that should be plotted. For our Simple Moving Average example, **FirstValidValue** should be set to the value of period-1. For example, if we have a 30 period SMA, the first 29 bars (bars 0 through 28) do not have valid values, and the first bar that has a valid value is the 30th bar, bar number 29.

Static Series Method

The static **Series** method is what Strategy code in Wealth-Lab 5.x will call to create instances of your indicator, and when it is dropped onto a chart. The Series method should have the same parameters as the constructor, minus the description. Inside the Series method, an instance of your indicator should be created, using the "new" keyword, and returned.

Follow the steps below in the static **Series** method:

1. Build a string description of the indicator based on the parameters that were passed
2. Sees if an indicator with this description already exists in the indicator **Cache** (see below)

3. If so:
    1. Return the instance of the cached indicator
4. If not:
    1. Use the "new" keyword to create an instance of the indicator using the supplied parameters and the description string that was built in (1) above
    2. Save this indicator instance to the **Cache** (see below)
    3. return the instance

Below is the code for the **Series** method of the SMA (Simple Moving Average) indicator.

```
//Static Series method returns an instance of the indicator
public static SMA Series(DataSeries ds, int period)
{
  //Build description
  string description = "SMA(" + ds.Description + "," + period + ")";

  //See if it exists in the cache
  if (ds.Cache.ContainsKey(description))
    return (SMA)ds.Cache[description];

  //Create SMA, cache it, return it
  SMA sma = new SMA(ds, period, description);
  ds.Cache[description] = sma;
  return sma;
}
```

Building the Description

The first step in the implementation of the indicator's static **Series** method is creating a description string that uniquely describes the indicator. This is composed of the indicator name, then the parameter values (if any) enclosed in parenthesis. Note that the **Description** property of the source **DataSeries** is used when building the description. This ensures that the resulting description string will be unique.

If the indicator was applied to closing price with a period of 20, the resulting description would be:

```
SMA(Close,20)
```

Returning a Cached Copy

DataSeries and **Bars** objects have a property called **Cache**, which is a **Dictionary<string, DataSeries>**. While working with the **Cache** is not mandatory, you should cooperate with the caching mechanism to avoid multiple copies of your indicators with the same parameter values getting needlessly created.

The **Series** method sees if the indicator is already in the **Cache** by using the **ContainsKey** method of the **Dictionary** object. The indicator's description string is always the key.

If the **Cache** contains the indicator, **Series** retrieves it from the **Cache**, and then casts it back to the actual indicator type. The **Cache Dictionary** stores all of the indicators as **DataSeries** objects, so you need to cast the object back to your specific type before returning it.

Creating a New Instance

If the indicator is not found in the **Cache**, then the **Series** method creates an instance of it using the constructor, by using the C# "new" keyword. Your indicator should have a public constructor that accepts the same parameters as the **Series** method call, and an additional string parameter that represents the description that should be used for the indicator. Supplying the description as an additional parameter to the constructor serves two roles:

1. It enforces that the same string is used in the **Cache** that is used in the indicator's **Description** property.
2. It gives power users the flexibility to use the **new** operator to create indicator instances, and provide their own descriptions. Indicators created using the **new** operator do not participate in the caching mechanism.

Once the **Series** method creates the instance of the indicator, it adds it to the **Cache**, and returns it to the caller.

Creating the IndicatorHelper

In order to allow your indicator to be published and visible in Wealth-Lab Pro 5.x, you need to create a helper class that describes the indicator. The helper class is an object that derives from the **IndicatorHelper** base class, described below. You can create the helper class in the same source code file as the indicator.

IndicatorHelper Base Class Properties

public abstract Type IndicatorType

Return a reference to the Type object of the indicator that this helper is supporting.

public abstract IList<**string**> ParameterDescriptions

Return a string list, one string item for each parameter that your indicator accepts in its constructor. Provide a brief (one or two word) description of the parameters. These descriptions are used as labels in the Indicator Parameters form when the indicator is dropped onto a chart. Internally, you can use any class that exposes the IList interface. A typical implementation will use a static string array that is populated when it is declared (see SMA example).

public abstract IList<**object**> ParameterDefaultValues

Return a list of objects, one for each parameter that your indicator accepts in its constructor. Populate the list with the default values that the parameters should assume when an indicator is dropped onto a chart. You can use any class that exposes an IList interface to implement this in your helper class. A typical implementation will use a static array of objects that is populated when it is declared (see SMA example). For certain parameter types, you will use special values in the list:

Bars parameters
You should set the value of the object to the **BarsDataType**.Bars enum value. This lets Wealth-Lab know that the indicator accepts a Bars object for this parameter.

DataSeries parameters
You should set the value of the object to one of the **CoreDataSeries** enum values. These can be either open, high, low, close, or volume.

enum parameters
You can use any custom (or existing) enumerated type as a parameter. The indicator parameters form will display a drop down that is filled with the string values of the enumeration that the user can

then select from.

int parameters
You can assign an int value directly that corresponds to the default value that the parameter should assume. Or, you can assign an instance of the RangeBoundInt32 class, which takes a default value, minimum, and maximum values in its constructor. This will limit the range of values that users can enter in the Indicator Parameters form for this parameter.

double parameters
Like int, above, you can assign a double value directly, or use the RangeBoundDouble class to restrict the parameter's value.

string parameters
You can assign a string which will be the default value, and will allow the user to enter any other string in a text box in the Indicators Parameters form.

DateTime parameters
If you want to DateTimePicker that is created in the Indicator Parameters form to be a time picker, supply a default **DateTime** value that contains zero for the year, month, and day portions.

public abstract Color DefaultColor

Return the color that should be used by default when plotting the indicator.

public virtual LineStyle DefaultStyle

Return the LineStyle (Solid, Dotted, Dashed or Histogram) that should be used when plotting the indicator. "Solid" is the default implementation, so if you want your indicator to render as a solid line by default you do not have to override this property.

public virtual int DefaultWidth

Return the default line width that should be used when plotting the indicator. The value returned in the default implementation is 1.

public abstract string Description

Return a description of the indicator, a few sentences is recommended. This description appears at the bottom of the Indicator List when you click on the indicator.

public virtual string URL

Return a URL that the user could navigate to to learn more about the indicator. Wealth-Lab will launch a web browser to the target URL if the user clicks the "More Info..." link in the Indicator List. If you do not have a URL, return a blank string.

public virtual string TargetPane

Determines where the indicator should be plotted when it is dropped onto a chart. If you want to indicator to always plot in the Price Pane, return "P", or for the Volume Pane return "V". Return a blank string if the indicator should just plot in the pane it was dropped in. Otherwise, pass a different string (typically the indicator's class name) to cause a new pane to be created for the indicator. When an indicator is dropped, Wealth-Lab will try to find a pane that already exists with your specified TargetPane string. If it finds one, the indicator will be plotted in this pane and a new pane not created. This feature can be used to cause "families" of related indicators to plot in the same pane (such as ADX, ADXR, DI+, DI-).

public virtual Type PartnerBandIndicatorType

If your indicator is a type that is always paired with another, such as BBandUpper or BBandLower, you can return the Type of the partner indicator by overriding this property. If you do this, Wealth-Lab will allow users to plot both bands when either one of the pair is dropped on a chart.
public virtual Color DefaultBandColor If your indicator is part of a pair, specify here the default color that should be suggested when the user selects to fill the band.
public virtual bool IsOscillator Override and return true if your indicator should be able to be plotted as an Oscillator, and overbought and oversold regions filled with different colors. If you do so, you can override the Oscillator methods below to customize the default behavior.
public virtual double OscillatorOverboughtValue Determines the overbought value of the Oscillator. Values plotted above this value will be filled with the color specified below. The default value is 0.
public virtual double OscillatorOversoldValue Determines the oversold value of the Oscillator. Values plotted below this value will be filled with the color specified below. The default value is 0.
public virtual Color OscillatorOverboughtColor Determines that color that will be used to fill areas of the Oscillator that extend into overbought territory. The default is Blue.
public virtual Color OscillatorOversoldColor Determines that color that will be used to fill areas of the Oscillator that decline into oversold territory. The default is Red.
public virtual Bitmap Glyph Override the Glyph property if you want a custom image to appear in the Indicators List in Wealth-Lab Pro for this indicator. The custom image should be 16x16 in size, and use **Fuchsia** as a background color.

Providing a Static Value Method

This is completely optional, and more of a convention than part of the formal indicator interface. A static **Value** method will allow your indicator class to calculate and return a value "on the fly" for a particular bar number, without having to go through the overhead of creating a complete instance of the indicator. This corresponds to the "Single Calc Mode" that was available in previous Wealth-Lab versions. If you decide to support this convention, implement a static **Value** method that accepts the same parameters as your indicator's constructor, with the addition of an **int** parameter that passes the bar number.

Below is an example of the static **Value** method for the Simple Moving Average indicator.

```
//This static method allows ad-hoc calculation of SMAs (single calc mode)
public static double Value(int bar, DataSeries ds, int period)
{
  if (ds.Count < period)
```

```
    return 0;
  else
  {
    double sum = 0;
    for(int i = bar; i > bar - period; i--)
      sum += ds[i];
    return sum / period;
  }
}
```

Using Indicators in WealthScript Code

WealthScript trading system will use indicators by using your static Series methods. Example:

```
//Create moving averages
DataSeries SMA20 = SMA.Series(Close, 20);
DataSeries SMA30 = SMA.Series(Close, 30);
```

Power users can use the **new** operator and create instances, providing whatever description they feel is appropriate. These indicators can be plotted, and the tooltip will use the description that they provided in the constructor call. Example:

```
DataSeries SMA20 = new SMA(Close, 20, "20 period SMA");
DataSeries SMA30 = new SMA(High, 10, "10 day SMA of Highs");
```

Providing a Custom Image for the Indicator Library

In Wealth-Lab Pro, your Indicator Library appears as a new folder in the Indicators List tool. The indicators in your Library appear as items under the folder node. By default, Wealth-Lab renders the folder node using a standard open/closed folder image. You can replace this standard folder image with a custom image. To do so, add a bitmap resource to your Indicator Library assembly, and make sure to set its build action to "Embedded Resource" in Visual Studio. The bitmap must be named **glyph.bmp**, should be 16x16 pixels in size, and should use **Fuchsia** as a background color.

Simple Moving Average Example

The following code contains the complete implementation of the Simple Moving Average (SMA) indicator, and its associated helper class.

```
/*
   SMA
   Simple Moving Average indicator
*/

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

namespace WealthLab.Indicators
{
   //SMA Indicator class
   public class SMA : DataSeries
   {
```

```
//---*** Public interface ***---

//Static Series method returns an instance of the indicator
public static SMA Series(DataSeries ds, int period)
{
    //Build description
    string description = "SMA(" + ds.Description + "," + period + ")";

    //See if it exists in the cache
    if (ds.Cache.ContainsKey(description))
        return (SMA)ds.Cache[description];

    //Create SMA, cache it, return it
    SMA sma = new SMA(ds, period, description);
    ds.Cache[description] = sma;
    return sma;
}

//This static method allows ad-hoc calculation of SMAs (single calc mode)
public static double Value(int bar, DataSeries ds, int period)
{
    if (ds.Count < period)
        return 0;
    else
    {
        double sum = 0;
        for(int i = bar; i > bar - period; i--)
            sum += ds[i];
        return sum / period;
    }
}

//Constructor
public SMA(DataSeries ds, int period, string description)
    : base(ds, description)
{
    //Remember parameters
    _sourceSeries = ds;
    _period = period;

    //Assign first bar that contains indicator data
    FirstValidValue = period - 1 + ds.FirstValidValue;

    //Calculate moving average values
    for (int bar = period - 1; bar < ds.Count; bar++)
    {
        double sum = 0;
        for (int innerBar = bar; innerBar > bar - period; innerBar--)
            sum += ds[innerBar];
        this[bar] = sum / period;
    }
}

//Calculate a value for a partial bar
public override void CalculatePartialValue()
{
    if (_sourceSeries.Count < _period - 1 || _sourceSeries.PartialValue == Double.NaN)
        PartialValue = Double.NaN;
    else
    {
        double sum = 0;
        for (int bar = _sourceSeries.Count - _period + 1; bar < _sourceSeries.Count; bar++)
            sum += _sourceSeries[bar];
        sum += _sourceSeries.PartialValue;
        PartialValue = sum / _period;
    }
    }

    //---*** Private members ***---
    DataSeries _sourceSeries;
    int _period;
}

//Helper class that allows SMA to work well in WL Pro
public class SMAHelper : IndicatorHelper
{
    //Return suggested default values
    public override IList<object> ParameterDefaultValues
    {
        get
        {
            return _paramDefaults;
        }
    }

    //Return parameter descriptions
    public override IList<string> ParameterDescriptions
    {
        get
        {
            return _paramNames;
        }
    }

    //Return default color
    public override Color DefaultColor
    {
        get
        {
            return Color.Red;
        }
    }

    //Return a reference to supported indicator type
    public override Type IndicatorType
    {
        get
        {
            return typeof(SMA);
        }
    }

    //Return a description of the indicator
    public override string Description
    {
        get
        {
            return @"The Simple Moving Average indicator calculates the average of a set of values over a specified period.";
        }
    }

    //Return a URL for more info
    public override string URL
    {
        get
        {
            return @"http://www.investopedia.com/terms/s/sma.asp";
        }
    }

    //Private members
    private static object[] _paramDefaults = { CoreDataSeries.Close, new RangeBoundInt32(20, 2, Int32.MaxValue) };
```

```
        private static string[] _paramNames = { "Source", "Period" };
    }
}
```