Indicators

Technical Indicators are used to aid in manual trading as well as be included in automated strategies by indicating trends and providing signals for potential changes in trend. There have been a number of indicators developed over time by various statisticians and traders alike. Many of the most popular indicators are included in the platform as build in indicators. Custom indicators maybe created using the cAlgo editor. These indicators can be viewed in the cTrader as well as the cAlgo platforms. Moreover, you may download custom indicators build by other users from cTDN Indicators.

The New Indicator Template

Click on the Indicators tab

Click the new button from the side panel.

The editor will be loaded with the template for a new indicator:

```
1 // These namespaces are required for a cAlgo indicator.
2 // They contain basic classes of the C# language, and the cAlgo API.
3 // The last namespace is required for referencing build-in
4 // indicators such as SimpleMovingAverage, BollingerBands, etc.
```
using System;
using cAlgo.API;
using cAlgo.API.Internals;
using cAlgo.API.Indicators;

// Define indicators within the cAlgo.Indicators namespace
namespace cAlgo.Indicators
{
    // The Indicator class declaration must be preceded by the indicator
    // attribute,
    // [Indicator()], and it should be a public
    // Indicator derived class

    [Indicator(IsOverlay = false, TimeZone = TimeZones.UTC)]
    public class NewIndicator : Indicator
    {
        // The parameter attribute is necessary if the program will
        contain user input.
        [Parameter(DefaultValue = 0.0)]
        public double Parameter { get; set; }

        // The Output attribute is used to display the Output of
        // an Indicator series on the chart.
        [Output("Main")]
        public IndicatorDataSeries Result { get; set; }

        // The Initialize method is called upon loading of an indicator.
        // It is typically used to initialize variables and series such as
        // nested indicators.
        protected override void Initialize()
        {
            // Initialize and create nested indicators
        }

        // The calculate method cannot be ommited.
        // It is the main method of an indicator.
        // It is called on each historical bar defined by the index and
        in real time on each incoming tick.
        public override void Calculate(int index)
Initialize

The initialize method is used to initialize variables mainly such as indicators, data series, lists, etc. In general, any variables that need to be calculated only once on start up should be in the initialize method in order not to consume unnecessary resources.

Calculate

The calculate method is the main method of the indicator. It is called for each historic bar starting from the beginning of the series up to the current bar and then on each incoming tick. In the case of multi-symbol / multi-timeframe implementation, it will be called on each tick of each symbol that is used in the indicator.

It is typically used for the calculation of an indicator based on a formula. In this case, it is the difference of the High and the Low prices of each bar. The index parameter, represents each bar, ranging from 0 up to the current (last) bar. So, at each bar the Calculate method will be called and the Result will be assigned with the difference between the High and the Low of that bar.

A Simple Indicator

The High Minus Low Indicator calculates the difference between the current bar’s High and Low prices and displays this in an output series which is drawn on the chart as a line connecting the values at each bar.

```csharp
// Calculate value at specified index
// Result[index] = ...

using System;
using cAlgo.API;

namespace cAlgo.Indicators
{
    [Indicator("High Minus Low", IsOverlay = false, ScalePrecision = 5)]
    public class HighMinusLow : Indicator
    {
        [Output("Main", Color = Colors.Orange)]
    }
}
The `Indicator Attribute` is applied to the class to indicate certain characteristics such as:

- the indicators name: "High Minus Low"

- if it will be overlayed on the chart or on a separate panel: `IsOverlay = false`

- the scale precision on the chart: `ScalePrecision = 5`

See `Indicator Attribute` for a complete list.

The Indicator Attribute is required even if the properties in parenthesis are omitted, e.g.: `[Indicator()]`. As mentioned in the previous section the indicator must be an Indicator derived class:

```csharp
public class MyIndicator : Indicator
```

The `Output Attribute` is applied to mark an `IndicatorDataSeries` as output to the chart. The property that is marked with the output attribute should be public so that it can be referenced from other classes. As with the Indicator attribute, the Output attribute allows to specify characteristics such as:

- the name given to the output: "Main"
- the color: Color = Colors.Orange

See Output Attribute for a complete list.

The name will be displayed in the small panel that is displayed when you add an instance. From this panel you can change the other characteristics of the output, such as the color and line type.

IndicatorDataSeries is a list of doubles that can be indexed like an array. Hence, the value at each index in the list “Result” is assigned in the Calculate method as follows:

```csharp
public override void Calculate(int index)
{
}
```

To find the total number of elements in a series use the Count property: MarketSeries.High.Count.

As you might have noticed, MarketSeries.High and MarketSeries.Low are also indexed in the same fashion. They are of type DataSeries which is also a list of doubles but this type is “read only” (values can be retrieved but not assigned).

The MarketSeries interface contains the following series: Open, High, Low, Close, Median, Typical and Weighted price series as well as TickVolume, OpenTime, SymbolCode and TimeFrame of the relevant instance. In other words, it will retrieve the above values of whichever symbol and timeframe you choose.
Indicators with Parameters

Most of the time indicators can vary according to user input. The Simple Moving Average is designed to accept as input parameters the price source and the period for the calculation. Input must be preceded by the Parameter attribute like shown in the code below; the “Source” and the “Periods” properties are preceded by the Parameter attribute.

```java
public class SimpleMovingAverage : Indicator
{
    [Parameter]
    public DataSeries Source { get; set; }

    [Parameter(DefaultValue = 14)]
    public int Periods { get; set; }

    [Output("Main", Color = Colors.Turquoise)]
    public IndicatorDataSeries Result { get; set; }

    public override void Calculate(int index)
    {
        double sum = 0;

        for (var i = index - Periods + 1; i <= index; i++)
        {
            sum += Source[i];
        }

        Result[index] = sum / Periods;
    }
}
```

Like the Indicator and Output attributes discussed earlier, the Parameter attribute can define certain characteristics to the input:
1 [Parameter("MA Periods", DefaultValue = 14, MinValue = 1, MaxValue = 20)]
   ● The name that is displayed in the input panel of an instance.
   ● The default value of an input which is the initial value before it can be changed by the user.
   ● The minimum and maximum values in case of numbers.

See Parameter attribute.

Currently, the supported types for input are: DataSeries, int, double, string, bool, MovingAverageType and TimeFrame.

If the input is a price source such as Open, High, Low, Close, then make the property a DataSeries.

If the input is a price source such as Open, High, Low, Close, then make the property a DataSeries type:

1 [Parameter("Price")]
2 public DataSeries Source { get; set; }

The default for DataSeries type cannot be set, the Close will be the default value.

If the input needs to be a number then int (integer) or double (decimal) can be used:

1 [Parameter("MA Periods", DefaultValue = 14, MinValue = 1, MaxValue = 20)]
2 public int Periods { get; set; }

Declare a bool for a yes/no type input. The value “Yes” corresponds to “true” and “No” to “false”:

1 [Parameter("Message", DefaultValue = true)]
public bool DisplayChartMessage { get; set; }

The above will display a “Yes/No” input option.

**Nested Indicators**

In certain occasions it is necessary to use nested indicators. That is the value of one indicator depends on the value of other indicators. Such is the case with the SampleDeMarker indicator found in the platform.

```csharp
public class SampleDeMarker : Indicator
{
    [Parameter(DefaultValue = 14)]
    public int Periods { get; set; }

    [Output("DMark", Color = Colors.Turquoise)]
    public IndicatorDataSeries DMark { get; set; }

    private IndicatorDataSeries deMin;
    private IndicatorDataSeries deMax;
    private MovingAverage deMinMA;
    private MovingAverage deMaxMA;

    protected override void Initialize()
    {
        deMin = CreateDataSeries();
        deMax = CreateDataSeries();
        deMinMA = Indicators.MovingAverage(deMin, Periods,
            MovingAverageType.Simple);
        deMaxMA = Indicators.MovingAverage(deMax, Periods,
            MovingAverageType.Simple);
    }

    public override void Calculate(int index)
    {
        deMin[index] = Math.Max(MarketSeries.Low[index - 1] -
            MarketSeries.Low[index], 0);
        deMax[index] = Math.Max(MarketSeries.High[index] -
```
In the example above deMinMA and deMaxMA are indicators which are used for the calculation of the demark indicator. Nested indicators need to be defined in the Initialize() method. For example deMinMA is defined as the simple moving average of the series deMin.

\[
deMinMA = Indicators.MovingAverage(deMin, Periods, MovingAverageType.Simple);
\]

deMin in turn is defined in the calculate method as the max of the last two Low price values.

\[
deMin[index] = Math.Max(MarketSeries.Low[index - 1] - MarketSeries.Low[index], 0);
\]

The intellisense will populate the list of all build-in indicators once you type “Indicators” followed by a period (dot).

The intellisense also shows the list of the input parameters to the referenced indicator once the indicator has been selected from the list.

**Referencing Custom Indicators**

A custom indicator is any indicator you create in cAlgo. The samples in the platform are custom indicators. In order to reference custom indicator you need to press “Manage References” button on the top left corner of the text editor.
In Indicators page of Reference Manager you need to check indicators which you want to be referenced and press Apply.

Referenced custom indicators must be defined in the Initialize method just like nested build in indicators but the syntax is slightly different. The name of the custom indicator must be enclosed in angle brackets following the "GetIndicator " directive. The parameters of the referenced indicator follow enclosed in parenthesis just like with build in indicators.

```csharp
1  sma = Indicators.GetIndicator<SampleSMA>(Source, SmaPeriod);
```

Example:

```csharp
1  [Indicator(IsOverlay = true, TimeZone = TimeZones.UTC)]
2  public class SampleReferenceSMA : Indicator
```
Oscillators and the Levels Attribute

Indicators that oscillate around a constant value belong to the category of oscillators. When creating oscillators it is useful to draw a horizontal or "level" line at that constant value around which the indicator oscillates. In many cases that value is zero.

The momentum oscillator typically oscillates around 100, so we will add a level line at that value using the Levels attribute.

```csharp
namespace cAlgo.Indicators
{
    [Levels(100)]
    public DataSeries Source { get; set; }
    [Parameter(DefaultValue = 14)]
    public int SmaPeriod { get; set; }
    [Output("Referenced SMA Output")]
    public IndicatorDataSeries refSMA { get; set; }
    private SampleSMA sma;
    protected override void Initialize()
    {
        sma = Indicators.GetIndicator<SampleSMA>(Source, SmaPeriod);
    }
    public override void Calculate(int index)
    {
        refSMA[index] = sma.Result[index];
    }
}
```
[Indicator("Momentum Oscillator")]

public class MomentumOscillator : Indicator
{

    [Parameter]
    public DataSeries Source { get; set; }

    [Parameter(DefaultValue = 14, MinValue = 1)]
    public int Periods { get; set; }

    [Output("Main", Color = Colors.Green)]
    public IndicatorDataSeries Result { get; set; }

    public override void Calculate(int index)
    {
        Result[index] = 100 * Source[index] / Source[index - Periods];
    } 
}

Levels can only be used when the indicator is not overlayed on the chart. That is, the property IsOverlay is not set to true in the Indicator attribute like this: [Indicator("MyOscillator", IsOverlay = true)] This will not work. IsOverlay is false by default so if it is omitted like in the following example the indicator will be displayed below the chart and the Levels will be displayed properly.

If multiple lines are necessary, then add a comma separated list of the values within the parenthesis. The following are some examples:

[D]
[1] [Levels(0, 50, 100)]
[2] [Levels(50.5, 50.75)]
[3] [Levels(0.001, 0.002)]

IsLastBar and IsRealTime

In certain cases we may need an indicator that requires to be calculated at the last bar. Then, the IsLastBar property can be checked, to determine if as the name suggests, the index parameter of the Calculate method, is that of the last bar.
The following indicator displays the last open time in NY and Tokyo while the indicator is based on UTC time.

```java
[Indicator(IsOverlay = true, TimeZone = TimeZones.UTC)]
public class TimeInDifferentParts : Indicator
{
    public override void Calculate(int index)
    {
        if (IsRealTime)
            DisplayTime(index);
    }

    protected void DisplayTime(int index)
    {
        DateTime nyDateTime = MarketSeries.OpenTime[index].AddHours(-5);
        DateTime tokyoDateTime = MarketSeries.OpenTime[index].AddHours(7);

        string nyTime = nyDateTime.ToShortTimeString();
        string tokyoTime = tokyoDateTime.ToShortTimeString();

        ChartObjects.DrawText("Title", "Last Bar Open Time ",
                             StaticPosition.TopLeft, Colors.Lime);
        ChartObjects.DrawText("NY", " NY " + nyTime,
                             StaticPosition.TopLeft, Colors.Lime);
        ChartObjects.DrawText("Tokyo", " Tokyo " + tokyoTime,
                             StaticPosition.TopLeft, Colors.Lime);
    }
```
Combining two or more indicators in one

In case we want to have many indicators displayed on the same panel that are not overlayed on the chart we may combine them in one custom indicator.

**Example 1 - Aroon, RSI and Directional Movement System**

The following indicator combines the Aroon, RSI and Directional Movement System in one.

```csharp
using cAlgo.API;
using cAlgo.API.Indicators;

namespace cAlgo.Indicators
{
    [Indicator(IsOverlay = false, TimeZone = TimeZones.UTC, ScalePrecision = 5)]
    public class Aroon_RSI_DMS : Indicator
    {
        private Aroon aroon;
        private RelativeStrengthIndex rsi;
        private DirectionalMovementSystem dms;

        [Parameter]
        public DataSeries Source { get; set; }

        [Parameter(DefaultValue = 14)]
        public int Periods { get; set; }

        [Output("Aroon Up", Color = Colors.LightSkyBlue)]
        public IndicatorDataSeries AroonUp { get; set; }

        [Output("Aroon Down", Color = Colors.Red)]
        public IndicatorDataSeries AroonDn { get; set; }

        [Output("Rsi", Color = Colors.Green)]
        public IndicatorDataSeries Rsi { get; set; }
    }
```
public IndicatorDataSeries DmsDIPlus { get; set; }

[Output("DI Minus", Color = Colors.DarkRed)]
public IndicatorDataSeries DmsDIMinus { get; set; }

[Output("ADX", Color = Colors.Blue)]
public IndicatorDataSeries DmsADX { get; set; }

protected override void Initialize()
{
    // Initialize and create nested indicators
    aroon = Indicators.Aroon(Periods);
    rsi = Indicators.RelativeStrengthIndex(Source, Periods);
    dms = Indicators.DirectionaMovementSystem(Periods);
}

public override void Calculate(int index)
{
    AroonUp[index] = aroon.Up[index];
    AroonDn[index] = aroon.Down[index];

    Rsi[index] = rsi.Result[index];

    DmsADX[index] = dms.ADX[index];
    DmsDIMinus[index] = dms.DIMinus[index];
    DmsDIPlus[index] = dms.DIPlus[index];
}
Multiple timeframes

Example 1- Using multiple timeframes

The following example displays a moving average on different timeframes.

```csharp
using cAlgo.API;
using cAlgo.API.Internals;
using cAlgo.API.Indicators;

namespace cAlgo.Indicators
{
    [Indicator(IsOverlay = true, TimeZone = TimeZones.UTC)]
    public class MultiTF_MA : Indicator
    {
        [Parameter(DefaultValue = 50)]
        public int Period { get; set; }

        [Output("MA", Color = Colors.Yellow)]
        public IndicatorDataSeries MA { get; set; }

        [Output("MA5", Color = Colors.Orange)]
        public IndicatorDataSeries MA5 { get; set; }

        [Output("MA10", Color = Colors.Red)]
        public IndicatorDataSeries MA10 { get; set; }

        private MarketSeries series5;
        private MarketSeries series10;

        private MovingAverage ma;
        private MovingAverage ma5;
        private MovingAverage ma10;

        protected override void Initialize()
        {
            series5 = MarketData.GetSeries(TimeFrame.Minute5);
            series10 = MarketData.GetSeries(TimeFrame.Minute10);
        }
    }
```
ma = Indicators.MovingAverage(
    MarketSeries.Close, Period,
    MovingAverageType.Triangular);
ma5 = Indicators.MovingAverage(
    series5.Close, Period,
    MovingAverageType.Triangular);
ma10 = Indicators.MovingAverage(
    series10.Close, Period,
    MovingAverageType.Triangular);
}

public override void Calculate(int index)
{
    MA[index] = ma.Result[index];

    var index5 = series5.OpenTime.GetIndexByTime(  
        MarketSeries.OpenTime[index]);
    if (index5 != -1)
        MA5[index] = ma5.Result[index5];

    var index10 = series10.OpenTime.GetIndexByTime(  
        MarketSeries.OpenTime[index]);
    if (index10 != -1)
        MA10[index] = ma10.Result[index10];
}

Example 2- Using multiple timeframes and symbols
public class MultiSymbolMA : Indicator
{
    private MovingAverage ma1, ma2, ma3;
    private MarketSeries series2, series3;
    private Symbol symbol2, symbol3;

    [Parameter(DefaultValue = "EURCHF")]
    public string Symbol2 { get; set; }

    [Parameter(DefaultValue = "EURCAD")]
    public string Symbol3 { get; set; }

    [Parameter(DefaultValue = 14)]
    public int Period { get; set; }

    [Parameter(DefaultValue = MovingAverageType.Simple)]
    public MovingAverageType MaType { get; set; }

    [Output("MA Symbol 1", Color = Colors.Magenta)]
    public IndicatorDataSeries Result1 { get; set; }

    [Output("MA Symbol 2", Color = Colors.Magenta)]
    public IndicatorDataSeries Result2 { get; set; }

    [Output("MA Symbol 3", Color = Colors.Magenta)]
    public IndicatorDataSeries Result3 { get; set; }

    protected override void Initialize()
    {
        symbol2 = MarketData.GetSymbol(Symbol2);
        symbol3 = MarketData.GetSymbol(Symbol3);

        series2 = MarketData.GetSeries(symbol2, TimeFrame);
        series3 = MarketData.GetSeries(symbol3, TimeFrame);

        ma1 = Indicators.MovingAverage(MarketSeries.Close, Period, MaType);
    }
        ma2 = Indicators.MovingAverage(series2.Close, Period,
        MaType);
        ma3 = Indicators.MovingAverage(series3.Close, Period,
        MaType);
    }
    
    public override void Calculate(int index)
    {
        ShowOutput(Symbol, Result1, ma1, MarketSeries, index);
        ShowOutput(symbol2, Result2, ma2, series2, index);
        ShowOutput(symbol3, Result3, ma3, series3, index);
    }
    
    private void ShowOutput(Symbol symbol, IndicatorDataSeries
    result,
        MovingAverage movingAverage, MarketSeries series, int index)
    {
        var index2 =
            series.OpenTime.GetIndexByTime(MarketSeries.OpenTime[index]);
        result[index] = movingAverage.Result[index2];

        string text = string.Format("{0} {1}\n", symbol.Code,
            Math.Round(result[index], symbol.Digits));
        ChartObjects.DrawText(symbol.Code, text, index,
            result[index],
            VerticalAlignment.Top, HorizontalAlignment.Right,
            Colors.Yellow);
    }