GoInstruments for .NET

Instruments Library

for GoDiagram for .NET

User Guide

This guide provides information on using the classes provided in the **GoInstruments™ for Microsoft® .NET** library that is an add-on to **GoDiagram™ for Microsoft® .NET**.

January 2008

Northwoods Software Corporation
142 Main St.
Nashua, NH 03060

http://www.nwoods.com

Copyright © 1999-2008 Northwoods Software Corporation

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise without the prior written permission of the publisher.

Northwoods Software Corporation makes no representations that the use of its products in the manner described in this publication will not infringe on existing or future patent rights, nor do the descriptions contained in this publication imply the granting of licenses to make, use, or sell equipment or software in accordance with the description.

Possession, use, or copying of the software described in this publication is authorized only pursuant to a valid written license from Northwoods or an authorized sublicensor.

Neither Northwoods Software Corporation nor its employees are responsible for any errors that may appear in this publication. The information in this publication is subject to change without notice.

The following are trademarks of Northwoods Software Corporation: Northwoods Software, GoDiagram, GoLayout, GoInstruments, JGo, GO++, Sanscript, Flowgram, the Northwoods logo, and Fully Visual Programming.

All other trademarks and servicemarks are property of their respective holders.

PREFACE

Purpose of this guide:

This guide provides an overview of **GoInstruments for .NET**, a .NET class library containing classes used to implement dials, gauges or meters in a **GoDiagram for .NET** document. Versions of **GoInstruments** exist for Windows Forms and for Web Forms.

For more detailed information about the types, classes and interfaces, see the appropriate **GoDiagram** Class Reference Manual, a compiled HTML Help file describing **GoDiagram** and **GoLayout** as well as **GoInstruments**: **`GoWin.chm`** or **`GoWeb.chm`**.

Who should use this guide:

This guide is intended for application programmers using the **GoInstruments for .NET** library or the InstrumentDemo sample application to incorporate dials/gauges/meters or similar functionality in their **GoDiagram for .NET** application.

This manual assumes you are familiar with Microsoft .NET and **GoDiagram for .NET** programming concepts and terminology. If you are not, please refer to your Microsoft .NET and **GoDiagram for .NET** documentation or online help.

1. INTRODUCTION

The **GoInstruments for .NET** library is a set of classes built to display numeric values in a graphical manner as part of **GoDiagram for .NET** applications. The classes implement **GoObject**s that display a scale with regular markings and labels and they implement **GoObject**s that indicate a value on a scale.

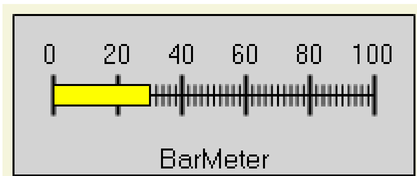
The objects can be customized and combined to present many different kinds of appearances. In the real world, depending on the industry, such things are called meters, dials, or gauges. Because there is so much customization that can be done, **GoInstruments** does not provide any prebuilt instruments. The InstrumentDemo sample application, however, does define many such objects for various purposes and with various appearances and behaviors.

GoInstruments is available for **GoDiagram Win** and **GoDiagram Web** and **GoDiagram Pocket**, but not for **GoDiagram Express**.

2. THE INSTRUMENTDEMO SAMPLE APPLICATION

Running the InstrumentDemo sample application is a good way to become familiar with the functionality built in the sample using the **GoInstruments** classes.

Most of the objects you see are instances of subclasses of the **Meter** class. Each **Meter** has a **Background** shape, a **Scale**, an **Indicator**, and a text **Label**.



The background shape defaults to a gray rectangle, but you can easily change the properties of the **GoRectangle**, or replace it with an instance of a different **GoObject** class.

The scale is a **GraduatedScale**. It has a line drawn from the scale's **Minimum** value to the **Maximum** value, along with major and minor tick marks that cross that path at regular value intervals, and with labels that display the value for the major tick marks. The **Meter** also has **Minimum** and **Maximum** properties; these are just the scale's properties exposed by the **Meter** for convenience.

The indicator is an **Indicator**. There are various kinds of **Indicator**s, but they all display a value on the scale by drawing something that intersects or ends at the scale at the point representing that value. The **Value** of the **Indicator** is also exposed as the **Value** of the **Meter**, for convenience in getting and setting the value.

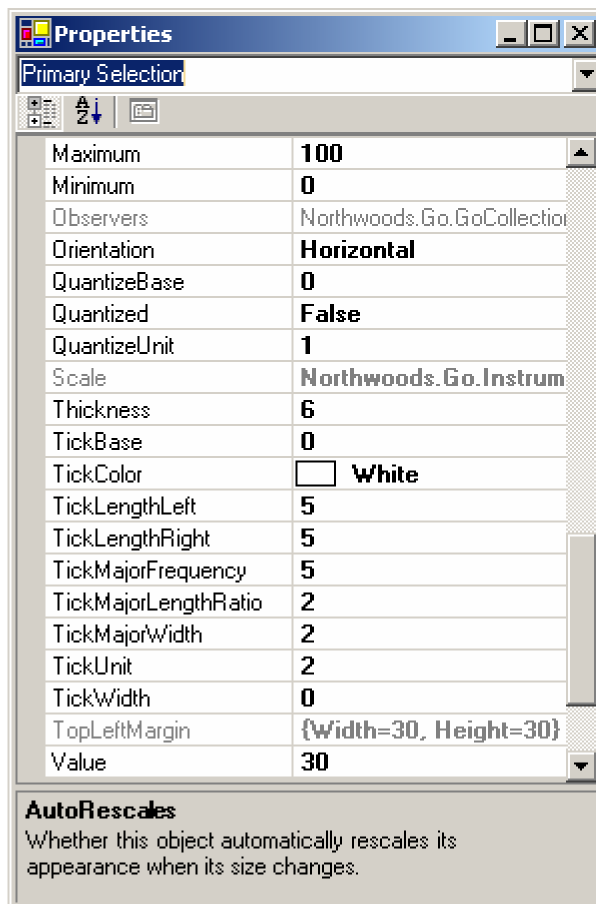
The label is often a caption, but can be used for other purposes such as displaying the current value in a textual format.

Click on an object to select it. If it is a meter, the current value will be displayed in the "Meter Value" text box. You can also modify the value by typing in this text box and then typing Enter or clicking elsewhere to move the focus out of that text box. Of course, modifying a meter value using the text box may be difficult to do when the value is continually changing while you type. You can click on the "Stop Animation" button to stop the timer that is modifying the meter values.

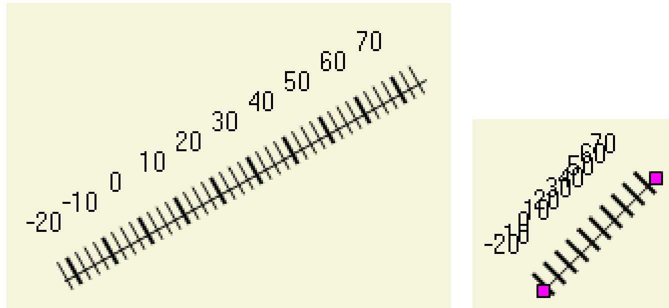
You can also modify the values of many meters interactively when the pointer is near the place where the indicator is indicating the current value. The cursor changes to a “Hand” cursor to inform the user that they can modify the value by direct manipulation. Note that doing such direct manipulation does not necessarily cause the meter to be selected in the view, so the “Meter Value” text box at the top of the application might not show the value of what you are changing.

You will notice that some of the meters are continually changing values. There are two timers that are running: one updates the clock and one updates some of the other meters. The timers are part of the example application—for efficiency, each meter is not responsible for creating its own **Timer**.

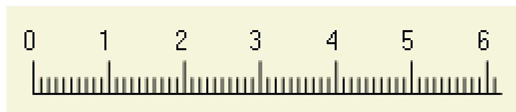
Please note that some features in the Windows Forms version of this sample application are not available in the Web Forms version. The use of **Timer**s to automatically update the instruments is one major difference. Although one can modify each web page to automatically update every few seconds, only when the web page displays clocks does it do so, and only after you click on the “Start AutoRefresh” button.



The InstrumentDemo application also makes all of the object properties visible by means of a **PropertyGrid** control. Click on the “Properties” button, or press the F4 key, to make the “Properties” window visible. These property values might not remain up-to-date as the application runs. However, you will be able to experiment interactively by changing many of the interesting property values, such as the ones involving the tick marks or tick labels on the scale.

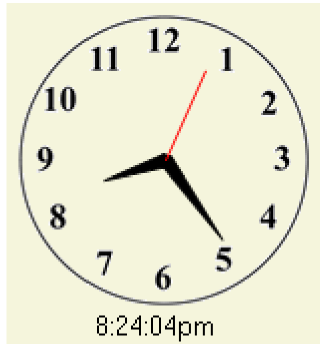


You can more directly manipulate scales by selecting and resizing the **GraduatedScaleLinear**, **GraduatedScaleElliptical**, and **Ruler** instances that are shown in the “Scales” tab. (You may need to scroll to the right to see them.) Note how scales automatically skip drawing the minor ticks when they become packed too close together.



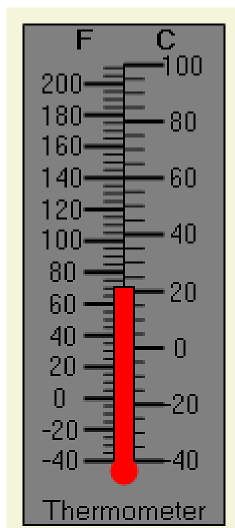
The ruler, which is implemented by a subclass of **GraduatedScaleLinear**, maintains a constant distance between the tick marks by continuously adjusting the **Maximum** value according to the **Length** of the scale. You can double-click on the ruler to toggle the ruler between metric and English units. Please note that the ruler can only be as accurate as the **Graphics** indication of how many pixels there are per inch. Your monitor or display may of course magnify the ruler quite arbitrarily.

Also on the “Scales” tab is an example of a timeline. What you see is actually a TimelineGroup, consisting of two Timeline scales. The example code shows how you can easily add more Timelines to this group for greater detail, with intervals of 6 hours and of 1 hour.

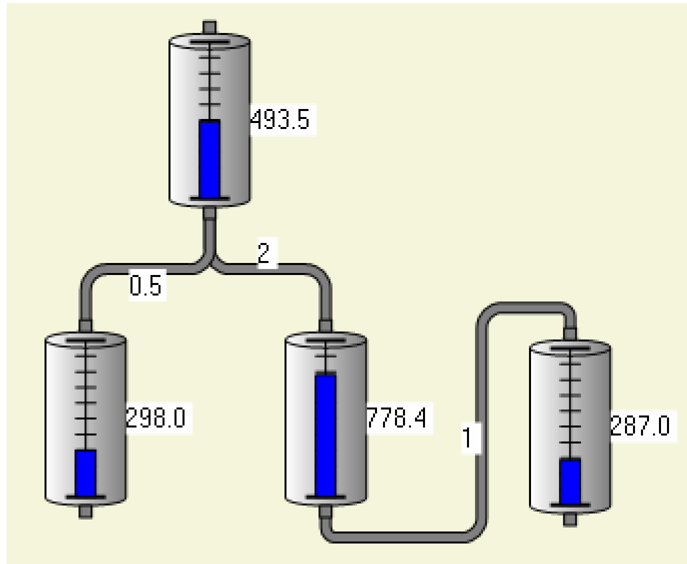


The clock at the top left corner of the "Clocks" tab normally displays the current time. However, you can speed up the clock, stop it, or reset the clock to the current time by using the buttons at the top of the Windows Forms application.

The **Clock** example class is what implements this functionality. This is a subclass of the **Meter** class where the **Background** object is actually a GIF image stored as a resource. As you expand the clock by resizing you will notice that the image will get fuzzy due to the stretching of the image. Resizing the other meters, however, leaves them all crisp looking because they are do not use **Image**s.



The thermometer at the left side of the "Complex Meters 2" tab is actually a meter that has an additional scale in it. The regular scale displays Celsius; the extra scale displays Fahrenheit. The scales have exactly the same **StartPoint** and **EndPoint**s, but the Celsius scale only displays ticks on the right side and the Fahrenheit scale only displays ticks on the left side.



A simple manufacturing facility consisting of tanks and pipes is shown in the "Factory" tab. Fluid flows out of tanks through the pipes that come out the bottom; fluid flows in through pipes coming in the top. The capacity of the tanks is 1000 units; the current volume is shown both with a bar indicator as well as textually. Furthermore you can change a tank value either by dragging the bar indicator or by editing the text value in-place. A text label on each pipe shows the maximum flow through the pipe; this too the user can edit in-place.

3. THE NORTHWOODS.GO.INSTRUMENTS LIBRARY

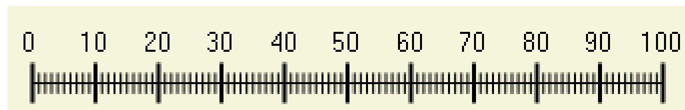
The **GoInstruments** library principally consists of the basic **GraduatedScale** and **Indicator** classes and the **Meter** class that is a group—a collection of objects including a scale and an indicator.

In your application you will normally define and instantiate subclasses of **Meter**. Your **Meter** subclass is where you will create and customize a **GraduatedScale**, typically an instance of **GraduatedScaleLinear** or **GraduatedScaleElliptical**, and an **Indicator**, typically an instance of **IndicatorNeedle**, **IndicatorBar**, **IndicatorBarElliptical**, **IndicatorSlider**, **IndicatorSliderElliptical**, or **IndicatorKnob**. Your **Meter** subclass is also responsible for making sure the sizes and relative positions of the parts of the group are the way you would like them to be.

GraduatedScale

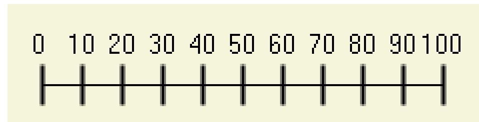
A graduated scale displays a range of values along a line. The **IGraduatedScale** interface defines these values to be double-precision floating-point numbers. There are methods for getting a **PointF** position in document coordinates for a value, and for getting a value for a given point. (Remember that document coordinates are single-precision floating-point numbers.) The extreme values are specified by the **Minimum** and **Maximum** properties.

The standard implementation of **IGraduatedScale** is the abstract class **GraduatedScale**. This class inherits from the **GoShape** class. It draws a line along a path with small crossings (“ticks”) marking intermediate value points. One end of the path represents the **Minimum** value and the other end represents the **Maximum** value.



There are a number of properties that govern the appearance of tick marks. Tick marks can be either major or minor; major tick marks are meant to mark more significant values at regular intervals. Major tick marks are typically bigger, in width or in length, and can be labeled with the value that they represent.

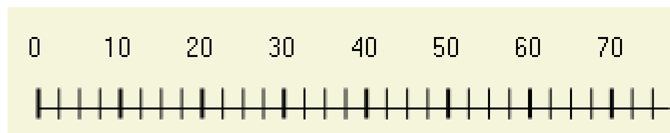
When the main path is too short to hold so many tick marks, only the major tick marks are drawn:



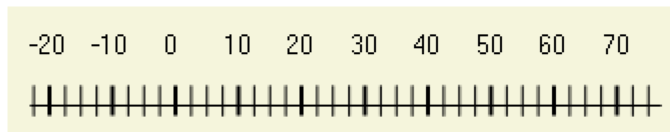
The values and frequency of tick marks are governed by two properties: **TickBase** and **TickUnit**. The **TickBase** property is normally the first desired major tick mark, and that is often the same as the **Minimum** property. Additional tick marks are positioned at each point along the path of the scale that represents a multiple of the **TickUnit** beyond the **TickBase**.

Tick Mark Values

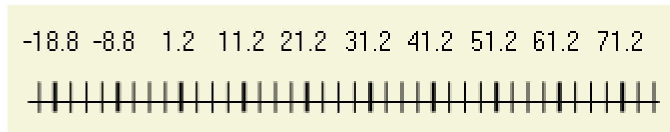
Tick marks become major marks every **TickMajorFrequency** marks. Thus a **Minimum** of 0, a **Maximum** of 77, a **TickBase** of 0, a **TickUnit** of 2.5, and a **TickMajorFrequency** of 4 results in a scale that might appear as:



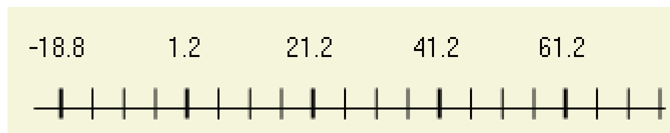
Changing the **Minimum** to –23 results in:



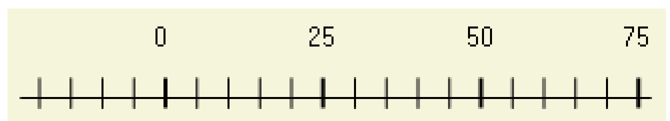
Changing the **TickBase** to 1.2 results in:



Changing the **TickUnit** to 5 results in:



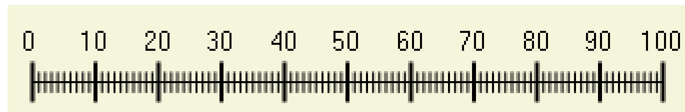
Changing the **TickBase** back to 0 and the **TickMajorFrequency** to 5 results in:



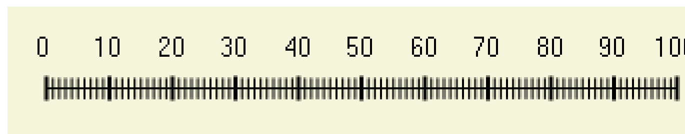
Tick Mark Appearance

The appearance of the tick marks is controlled by a number of different properties: **TickColor**, **TickWidth**, **TickLengthLeft**, **TickLengthRight**, **TickMajorWidth**, and **TickMajorLengthRatio**.

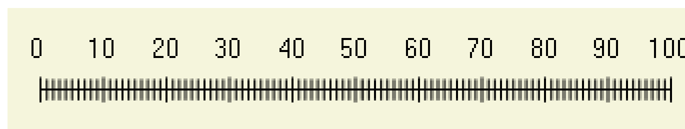
The default values for these tick properties results in a scale that might appear as:



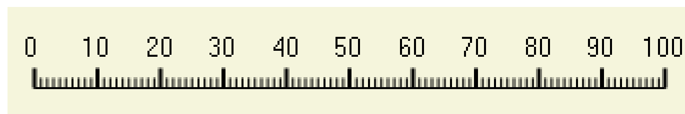
By default major tick marks are twice as long as minor ones. By changing **TickMajorLengthRatio** to 1.2, for example, you get:



Notice that the major tick marks are wider than the minor ones. Changing **TickMajorWidth** to 1 results in:



As the scale's path is drawn from the **Minimum** value to the **Maximum** value, there are two sides: left and right. You can control the length of each tick mark to either side. For example, changing the default scale's **TickLengthRight** property to 0 results in:



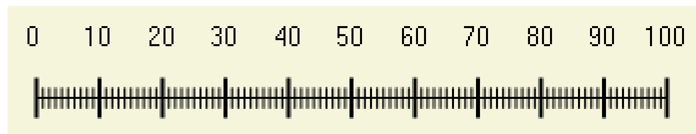
Major Tick Mark Labels

When the **TickLabels** property is true (as it is by default), the value at each major tick mark is drawn as well. The scale draws each label by using an instance of **GoText** that is held as the value of the **LabelTemplate** property. The scale calls the **GetLabelString** method to compute the text to be displayed and it calls the **GetLabelCenter** method to compute the middle position for the label. The **PaintLabel** method then sets the **LabelTemplate**'s **GoText.Text** property to the label text string, sets the **GoObject.Center** position of the text object, and then paints the text object.

Thus all of the **GoText** properties can be set to affect the appearance of each label. For example, you may wish to set the **LabelTemplate**'s **GoText.Bold**

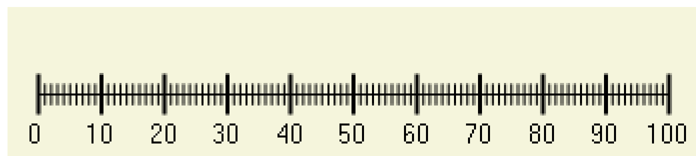
property to true and **GoText.FontSize** to 12. Remember that the appearance can be affected for the one shared **GoText** object, but not any editing behavior, because the user may not modify any label.

The **LabelDistance** property controls the distance from the center of the label. The default value is 10. Changing this property to 20 results in:

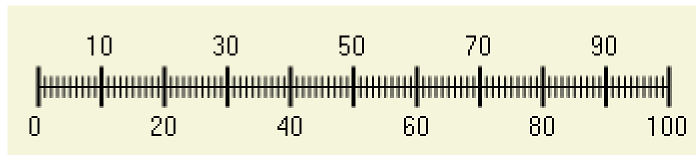


You can see that the labels are now much further from the tick marks. If you have a vertical scale and large values (i.e. many digits of precision) you will need to increase the **LabelDistance** to keep the numbers from overlapping with the tick marks.

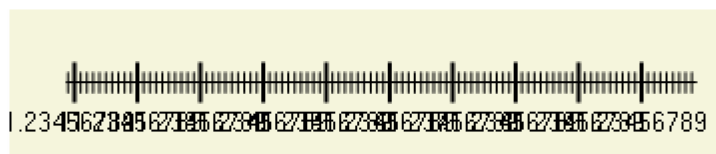
Labels can also be positioned on the left or on the right of the scale. A value for the **LabelStyle** property of **GraduatedScaleLabelStyle.Right** results in:



A value of **GraduatedScaleLabelStyle.AlternateStartRight** results in:



You can easily change the formatting of the major tick label values by setting the **LabelFormat** property. This is a .NET formatting string; the default is "G". When the values are very long, such as when the **TickBase** is set to 1.23456789, you will often get overlapping labels:



But by setting **LabelFormat** to "G3", specifying a precision of 3 digits, the result appears as:

Another way of reducing the chance of label overlap, in addition to using alternating **LabelStyle** and a truncating **LabelFormat**, is to reduce the frequency at which major ticks have labels. The value of **LabelFrequency** is normally 1, meaning every major tick has a label. But set it to 2 and you can set the **LabelFormat** to display more precision, such as "G7", and still not have overlap:

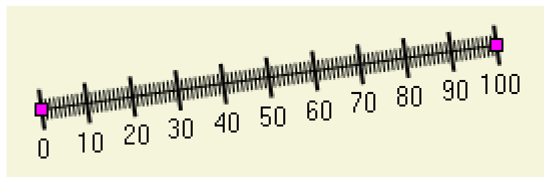
Besides formatting the value for each major tick mark according to the **LabelFormat** directive, you can also set **LabelChoices** to an **ArrayList** of objects that are converted to strings for the integral values starting at zero. When **LabelChoices** is non-null, **GetLabelString** rounds the value to an integer and uses that integer as an index into the **LabelChoices** list. Values less than zero use the first item (index zero); values beyond the end of the list use the last item. An example of this is given later, when talking about **KnobMeter**.

Finally, one can override the **GetLabelString** method to produce whatever you like. An example of this is in the **Timeline** class in the InstrumentDemo sample.

GraduatedScaleLinear

The above examples were all screenshots of a **GraduatedScaleLinear**, a scale that has a path that is a straight line.

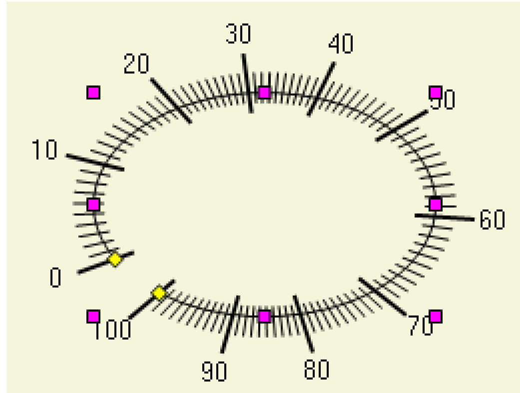
The distinguishing properties of a **GraduatedScaleLinear** are the **StartPoint** and **EndPoint** properties. These coincide with the resize handles:



GraduatedScaleElliptical

The other kind of scale is **GraduatedScaleElliptical**, a scale that has a path along an ellipse.

The ellipse is determined by the **Bounds** of the scale and the **StartAngle** and **SweepAngle** properties. By default there are the usual eight resize handles, plus two additional ones for controlling the **StartAngle** and the **SweepAngle** if the scale is **Reshapable**.



In the above example, the **StartAngle** is 160 and the **SweepAngle** is 340. The **SweepAngle** may be negative, to have the path go in the counter-clockwise direction.

Indicator

An indicator displays a particular value on a graduated scale. The abstract **Indicator** class has a **Value** property and a **Scale** property.

But there are many different ways for an indicator to do its job. The primary ways provided by **GoInstruments** are: needle, bar, slider, and knob.

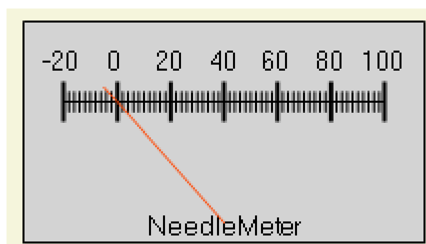
Since the **Indicator** class inherits from **GoShape**, you can customize the appearance by setting the **Pen** and/or **Brush** properties.

Needle

A needle indicator, **IndicatorNeedle**, is basically a simple shape that is drawn from the **PivotPoint** to the point on the **Scale** representing the indicator's **Value**.

The **IndicatorNeedleStyle** enumeration lists the basic needle shapes.

The following example **Meter** shows an **IndicatorNeedle** of style **IndicatorNeedleStyle.Line**, with a **Value** of zero, on a **GraduatedScaleLinear** scale. The indicator's **Pen** is **Pens.OrangeRed**.



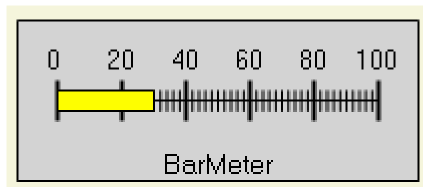
This next example **Meter** shows an **IndicatorNeedle** of style **IndicatorNeedleStyle.Kite**, with a **Value** of about 66, on a **GraduatedScaleElliptical** scale. The indicator's **Thickness** is 12 and its **Brush**

is **Brushes.Orange**. The scale's **TickColor** and **LabelTemplate.TextColor** are orange too.

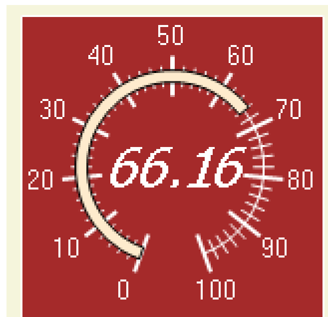
Bar

A bar indicator, **IndicatorBar**, displays as a rectangular or annular bar drawn from the scale's **Minimum** value to the indicator's **Value**. Its width (across its path) is specified by the **Thickness** property. It can also be shifted in position, relative to the scale's minimum value point, by setting the **StartOffset** property.

The following **Meter** screenshot shows an **IndicatorBar** with a **Brushes.Yellow** brush and a **Value** of 30.



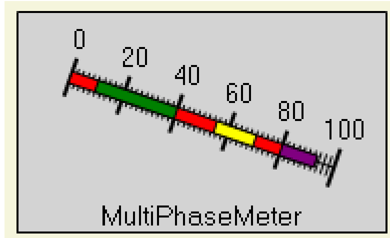
For elliptical scales, you need to use the **IndicatorBarElliptical** class. The following example shows such a bar with a **Thickness** of 6. The **Bounds** of the indicator have been inflated by half the **Thickness** to cause the bar to be centered on the elliptical path of the scale.



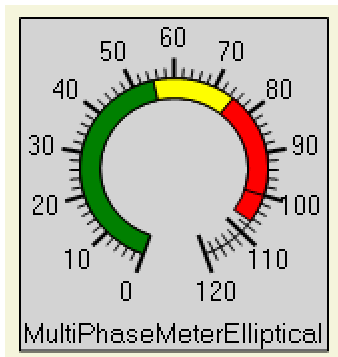
The **IndicatorBar** class also supports painting parts of the bar in different colors. A **Phase** is a structure that holds a **Color** and starting and ending values, **Min** and **Max**. You can add several **Phase**s to an **IndicatorBar**. When the

indicator's **Value** is less than a **Phase**'s **Min** value, that phase is not drawn**.** When the **Value** is between the **Min** and the **Max**, the part of the phase from the minimum up to the value is drawn. When the **Value** is greater than the **Max**, the whole phase is drawn.

The **IndicatorBar** keeps **Phase**s in an ordered list, so that if there is any overlap, later phases will be drawn on top of earlier ones. Any gaps between the phases will be filled in by the standard bar, which is actually drawn first.

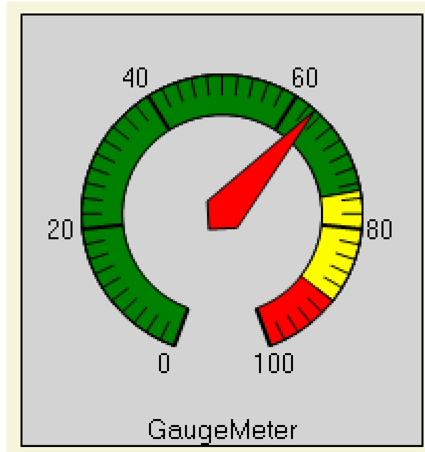


In the **MultiPhaseMeter** example, three **Phase**s have been added to a regular **IndicatorBar**: green from 10 to 40, yellow from 55 to 70, and purple from 80 to 95. The indicator's **Brush** is the default: **Brushes.Red**.



The **MultiPhaseMeterElliptical** example also has three **Phase**s: green from 0 to 55, yellow from 55 to 75, and red from 75 to 100. Note the line at 100, which is caused by the indicator's **Pen** (the default black pen) to demarcate the end of the third phase.

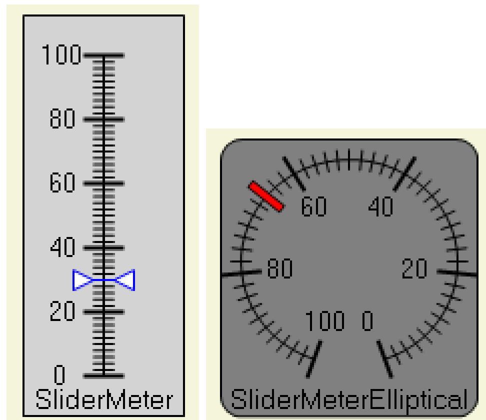
Indicators need not have changing values, so you can use **IndicatorBar**s in a static manner to mark regions of a scale. The **GaugeMeter** example creates an additional **IndicatorBar**, with two additional phases (green and yellow), that has a **Value** equal to the **Maximum** of the **Scale**.



Slider

A slider indicator, **IndicatorSlider**, is basically like a fancy tick mark. As a regular **GoShape**, you can set its **Pen** and **Brush**. You can control its size by setting the **Dimensions** property. The dimension's "width" controls how far off the path of the scale the slider should extend; the "height" controls how long it is along the path.

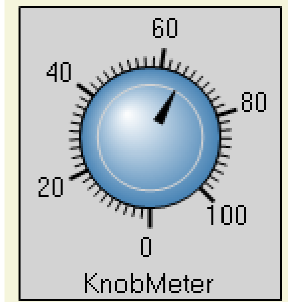
The **IndicatorSliderStyle** enumeration lists the different pre-defined shapes that the slider can take. The following examples use **IndicatorSliderStyle.Triangles** and **IndicatorSliderStyle.Bar**. As with **IndicatorBar**, there is a separate **IndicatorSliderElliptical** class to work with elliptical scales.



Knob

A knob, **IndicatorKnob**, is always elliptical, and should normally be circular. The value is shown by a thin triangle, whose color is determined by the **MarkerColor** property.

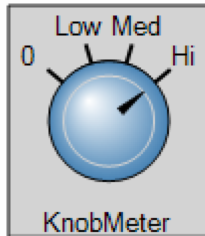
The **KnobMeter** example class uses an **IndicatorKnob**, but customizes its **Brush** to be a **PathGradientBrush** to give a lighting/reflection effect.



You can also use the **GraduatedScale.LabelChoices** property to provide non-numerically oriented labels. For example, if you initialize the **KnobMeter**'s **Scale** as follows:

```
scale.StartAngle = 220;
scale.SweepAngle = 100;
scale.TickMajorFrequency = 1;
scale.LabelChoices =
    new ArrayList(new String[]{"0", "Low", "Med", "Hi"});
scale.Maximum = scale.LabelChoices.Count-1;
```

If you also set the **KnobMeter**'s **Indicator.Quantized** property to true and **QuantizeUnit** to 1, you will get:



The user will only be able to choose among the four choices, resulting in a **Value** from zero to three. Explanation about quantization is in the next section.

Quantization

When the **Indicator.Value** is set, the value is first passed to the **ValidValue** method to ensure its validity. By default it will make sure the value is between the scale's **Minimum** and **Maximum**.

In addition, **ValidValue** will call the **QuantizeValue** method to allow the value to be forced to take discrete values. Although you can override **QuantizeValue** to get any behavior you like, three properties cover the most common cases:

Quantized, **QuantizeBase** and **QuantizeUnit**. The latter two properties behave in a manner similar to **GraduatedScale**'s **TickBase** and **TickUnit** properties. The standard implementation of **QuantizeValue** will make sure the value is **QuantizeBase** plus a multiple of **QuantizeUnit**, if **Quantized** is true. **Quantized** is false by default.

In the above example of a **KnobMeter** using a **GraduatedScale** with a list of **LabelChoices**, it is commonplace to make sure the **Indicator** is **Quantized**. However, if **Indicator.Quantized** is false, the user would be able to choose fractional values, such as 1.5, in between the "Low" and "Med" major tick marks. This might be desired when the values are not an enumerated set of fixed choices but a continuous range of values.

Meter

Meter is the base class for most meters, that is, groups consisting of four child objects: a **Background**, a **GraduatedScale**, an **Indicator**, and a **Label**.

The default implementations of **CreateScale** and **CreateIndicator** do nothing but return null, so these two methods are normally overridden to create and initialize some kind of **GraduatedScale** and some kind of **Indicator**. You can create instances of **Meter** and do the creation and assignment of the scale and the indicator explicitly, but it is more common to define a class to inherit from **Meter** so that you can keep together the code for creating and for laying out the meter's child objects.

Meter.LayoutChildren is implemented to set the **Bounds** of the **Scale** and of the **Indicator** to be a rectangle that fits inside the **Background**'s **Bounds** leaving room for the **TopLeftMargin** and the **BottomRightMargin**. These margins default to a size of 10x10, but you may want to increase them depending on where the scale labels are and how large they are.

LayoutChildren also positions the **Label**'s **LabelRelativeSpot** to be at the **LabelSpot** compared to the **Background**. The default values for those spots are **GoObject.MiddleBottom**, causing the **Label** to be placed inside the **Background** centered along the bottom edge. Specify a **LabelRelativeSpot** of **GoObject.MiddleTop** to position the **Label** outside of the **Background**, underneath it.

Typical subclasses of **Meter** will override **LayoutChildren** to do nothing if **Initializing** is false, call the base method, and then do any additional adjustments of the indicator and/or scale based on the their new sizes and positions.

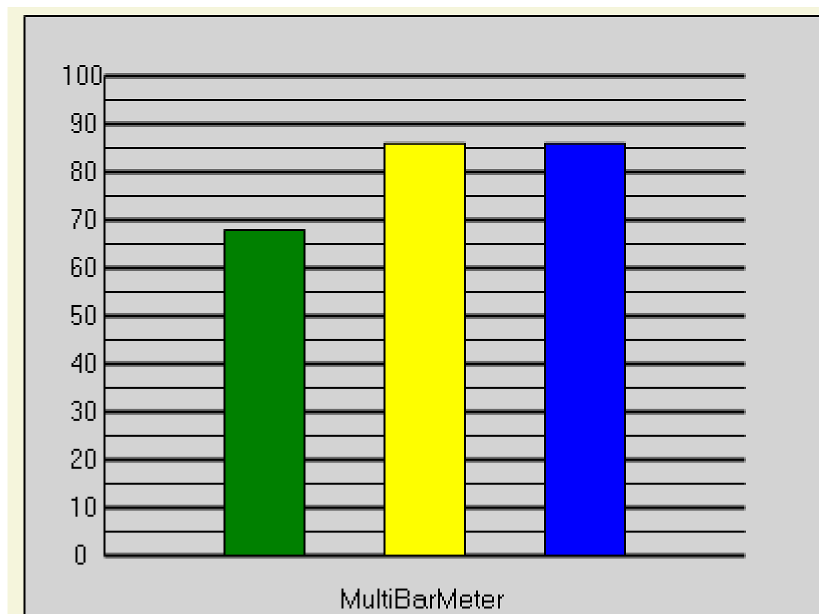
For the convenience of classes that inherit from **Meter**, there is an **Orientation** property that can be used by those kinds of meters that want to be vertical or horizontal. The standard implementation of **Meter**'s methods does not use the **Orientation** property at all.

For your convenience in referring to the various properties of the **Indicator** and the **Scale**, **Meter** exposes many of their properties as its own.

MultipleIndicatorMeter

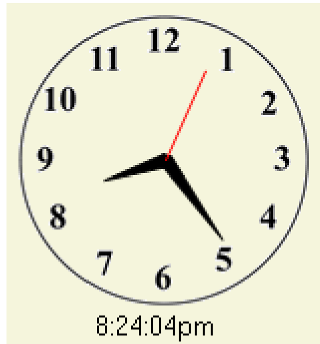
It is moderately common to have a **Meter** with more than one **Indicator**. The **MultipleIndicatorMeter** class is designed to keep track of a list of **Indicators**. The standard **Meter.Indicator** property is overridden to refer to the first **Indicator** in that list.

An example of this is the **MultiBarMeter**. The example has three **IndicatorBar**s, of different colors, each controllable by the user.



Another example is the **Clock**. A **Clock** has three indicators, the different "hands".

Each hand has different sizes and appearances, of course. But additionally they are defined as classes inheriting from **IndicatorNeedle** so that they can override the **ValidValue** method. This is what implements the automatic behavior of "wrapping" around when reaching 60 (seconds or minutes).



Furthermore the hour hand can use the same 0-60 scale even though it wraps around twice to show 1-12 hours. This is accomplished internally by multiplying the hour value by 5 to get the actual indicator value.