



**IBM ILOG Diagram for .NET V2.0**

**Programming with IBM ILOG Diagram**

**for .NET Windows Forms and ASP.NET**

**Controls**

**June 2009**

© Copyright International Business Machines Corporation 1987, 2009.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.



## Contents

<b>Preface</b>	<b>Programming with IBM ILOG Diagram for .NET Windows Forms and ASP.NET</b>	
<b>Controls</b>	<b>7</b>	
<b>Creating a Basic Diagram Programmatically</b>		<b>11</b>
<b>Displaying Diagrams in a Windows Forms Application</b>		<b>15</b>
<b>Displaying Diagrams in a Diagram View</b>		<b>16</b>
<b>Controlling the Zoom Level in a Diagram View</b>		<b>17</b>
<b>Scrolling in a Diagram View</b>		<b>19</b>
<b>Showing the Whole View Content</b>		<b>20</b>
<b>Modifying the Appearance of a Diagram View</b>		<b>20</b>
<b>Displaying a Grid in a Diagram View</b>		<b>21</b>
<b>Displaying Rulers in a Diagram View</b>		<b>23</b>
<b>Using the Predefined Behavior of the Diagram View</b>		<b>25</b>
<b>Displaying Diagrams in an ASP.NET Application</b>		<b>27</b>
<b>Displaying a Diagram in a Basic Web Application</b>		<b>28</b>
<b>Displaying Diagrams in a Diagram View</b>		<b>28</b>
<b>Controlling the Zoom Level in a Diagram View</b>		<b>30</b>
<b>Showing the Whole View Content</b>		<b>32</b>
<b>Controlling the Image Generation</b>		<b>32</b>

Image Map Generation .....	32
Handling User Input Event in a Diagram View .....	35
<b>Displaying a Diagram in an AJAX Web Application .....</b>	<b>37</b>
Overview of the IBM ILOG Diagram for .NET AJAX Framework .....	38
Displaying a Diagram in an AJAX Diagram View .....	38
Tiling Mode .....	40
Using Predefined Interaction Tools in an AJAX Diagram View .....	41
Creating a New AJAX-enabled View Interactor .....	49
<b>Using Predefined Graphic Objects .....</b>	<b>59</b>
<b>Basic Shapes .....</b>	<b>60</b>
<b>Paths .....</b>	<b>64</b>
<b>Images .....</b>	<b>69</b>
<b>Text Objects .....</b>	<b>70</b>
<b>Link Objects .....</b>	<b>71</b>
Specifying the Connection Points .....	72
Specifying the Link Shape .....	73
Customizing the Arrows .....	76
Customizing the Link Appearance .....	77
Adding Text .....	78
Link Crossing .....	81
<b>Controls .....</b>	<b>82</b>
The Control Class .....	82
Basic Controls .....	84
Single Content Controls .....	84
Multiple Content Controls .....	84
<b>Panels .....</b>	<b>85</b>
The Panel class .....	85
Using Predefined Panels .....	89
<b>Subdiagrams .....</b>	<b>97</b>
<b>Graphic Symbols .....</b>	<b>100</b>
<b>Scale Objects .....</b>	<b>102</b>

The ScaleBase Class.....	102
Linear Scales.....	106
Circular Scales.....	106
<b>Gauges</b> .....	<b>107</b>
<b>User Symbols</b> .....	<b>110</b>
<b>Common Graphic Objects Rendering Features</b> .....	<b>113</b>
<b>Styling Graphic Objects Using Fill, Stroke and Filter Classes</b> .....	<b>119</b>
Filling and Stroking Graphic Objects.....	120
Editing Fill Objects Using the Fill Dialog Box.....	128
Applying a Filter to a Graphic Object.....	129
Editing Filter Effects Using the Filter Dialog Box.....	134
<b>Displaying Text in a Diagram</b> .....	<b>137</b>
<b>Understanding Graphic Object Visibility</b> .....	<b>145</b>
<b>Using Graphic Object Preferred Size</b> .....	<b>149</b>
<b>Understanding Graphic Object Events</b> .....	<b>153</b>
<b>Understanding Graphic Containers</b> .....	<b>157</b>
Introduction to Graphic Containers.....	157
Using the Predefined Graphic Containers.....	159
<b>Understanding Coordinate Systems</b> .....	<b>161</b>
Overview of Existing Coordinate Systems.....	161
Conversion Between Coordinate Systems.....	165
<b>Creating Diagrams with Nodes and Links</b> .....	<b>167</b>
Creating a Simple Diagram with Nodes and Links Programmatically.....	167
Introducing Link and Anchor Classes.....	171
Using Automatic Link Crossing Detection in a Graph.....	175
Creating a New Class of Anchor.....	178
<b>Handling Interactions in a Diagram View (WinForms)</b> .....	<b>181</b>
Understanding Events Dispatching in a Diagram View (WinForms).....	181

From the View to the Graphic Object . . . . .	182
Dispatching Events to Graphic Objects . . . . .	184
Stopping the Event Propagation . . . . .	188
Event Capture . . . . .	189
<b>Using Predefined Interaction Tools in a Diagram View . . . . .</b>	<b>192</b>
Setting an Interactor on a View . . . . .	193
Selection Interactor . . . . .	193
Zoom Interactor . . . . .	196
Rotate Interactor . . . . .	196
Pan Interactor. . . . .	197
Rectangular Shapes Creation Interactor . . . . .	197
Polypoints Shape Creation Interactor . . . . .	199
Link Creation Interactor . . . . .	201
Anchor Editing Interactor . . . . .	204
<b>Creating a New Interactor in a DiagramView (WinForms) . . . . .</b>	<b>204</b>
<b>Handling Selection in a Diagram . . . . .</b>	<b>217</b>
<b>Managing Selected Objects . . . . .</b>	<b>218</b>
<b>Listening to Selection Events. . . . .</b>	<b>219</b>
<b>Using Predefined Selection Graphic Objects . . . . .</b>	<b>225</b>
<b>Styling Selection Graphic Objects . . . . .</b>	<b>231</b>
<b>Creating Custom Selection Graphic Objects . . . . .</b>	<b>231</b>
<b>Building Diagrams and User Symbols Inside Visual Studio . . . . .</b>	<b>235</b>
<b>Creating Diagrams and User Symbols Using Visual Studio . . . . .</b>	<b>237</b>
<b>Adding Graphic Objects to Diagram or User Symbol. . . . .</b>	<b>241</b>
<b>Controlling the Zoom Level. . . . .</b>	<b>244</b>
<b>Selecting Graphic Objects . . . . .</b>	<b>244</b>
<b>Moving, Resizing, Rotating and Aligning Graphic Objects . . . . .</b>	<b>245</b>
<b>Changing Properties of Graphic Objects. . . . .</b>	<b>248</b>
<b>Setting Text to a Graphic Object . . . . .</b>	<b>252</b>
<b>Grouping Graphic Objects . . . . .</b>	<b>253</b>

Manipulating Panels and Other Containers.....	253
Controlling the Drawing Order of Graphic Objects .....	255
Inspecting the Structure of a Diagram.....	255
Showing and Hiding Objects .....	256
Creating Complex Path Objects .....	257
Cut, Copy and Paste .....	259
Graph, Link and Anchors .....	259
Graph Layout .....	259
Importing Vector Graphics in SVG or IVN Format .....	260
Printing a Diagram.....	260
Diagram Designer Commands .....	261
<b>XML Serialization .....</b>	<b>269</b>
Serializing and Deserializing a Diagram in XML .....	270
Understanding the XML Serialization Mechanism .....	271
Customizing the XML Serialization .....	277
<b>Animating Graphic Objects.....</b>	<b>293</b>
Animation Overview.....	294
Controlling Animation Execution .....	294
Animating a Property.....	295
Grouping Animations .....	297
Using Animation as a Timer .....	298
Animating a Graphic Object Along a Path.....	299
Animation Types .....	299
Creating a Custom Animation.....	300
<b>Printing a Diagram.....</b>	<b>303</b>
Setting up a Print Document.....	303
Using Predefined Printing Dialog Boxes.....	309
<b>Importing SVG Files.....</b>	<b>317</b>
<b>Improving the Design-Time Behavior of Your Graphic Object .....</b>	<b>325</b>

<b>Creating BPMN Diagrams</b> .....	<b>329</b>
<b>The BPMN Symbols</b> .....	<b>330</b>
<b>The BPMN Editor</b> .....	<b>334</b>
<b>Localizing an IBM ILOG Diagram for .NET Application</b> .....	<b>335</b>
<b>Creating a Localization Project</b> .....	<b>336</b>
<b>Translating the Resource Files</b> .....	<b>337</b>
<b>Creating the Satellite Assemblies</b> .....	<b>337</b>
<b>Index</b> .....	<b>1</b>



# ***Programming with IBM ILOG Diagram for .NET Windows Forms and ASP.NET Controls***

This section provides the essential programming information you need to build applications with IBM® ILOG® Diagram for .NET. You will find information about key programming concepts, as well as code samples and detailed explanations.

## **In This Section**

### *Creating a Basic Diagram Programmatically*

Explains how to create a basic diagram.

### *Displaying Diagrams in a Windows Forms Application*

Describes how to display diagrams in a Windows® Forms Application.

### *Displaying Diagrams in an ASP.NET Application*

Describes how to display diagrams in an ASP.NET Application.

### *Using Predefined Graphic Objects*

Presents the predefined graphic objects available in IBM ILOG Diagram for .NET.

### *Common Graphic Objects Rendering Features*

Introduces the common features that affect the rendering of the graphic object.

### *Styling Graphic Objects Using Fill, Stroke and Filter Classes*

Describes how to paint graphic objects, apply and edit filter effects.

### *Displaying Text in a Diagram*

Describes how to display basic or complex styled text inside a graphic object.

### *Understanding Graphic Object Visibility*

Explains how to change the visibility of a graphic object.

### *Using Graphic Object Preferred Size*

Explains how to use the preferred size of graphic objects.

### *Understanding Graphic Object Events*

Describes the events sent by the graphic objects.

### *Understanding Graphic Containers*

Introduces the Graphic Containers.

### *Understanding Coordinate Systems*

Describes the coordinate systems in IBM ILOG Diagram for .NET.

### *Creating Diagrams with Nodes and Links*

Explains how to create diagrams with nodes and links.

### *Handling Interactions in a Diagram View (WinForms)*

Explains how to create mouse and keyboard events in a graphic object and in a WinForms diagram view .

### *Handling Selection in a Diagram*

Explains how to handle selected objects in a diagram.

### *Building Diagrams and User Symbols Inside Visual Studio*

Introduces the Visual Studio .NET® Diagram Designer.

### *XML Serialization*

Describes the XML serialization framework available in IBM ILOG Diagram for .NET.

### *Animating Graphic Objects*

Explains how to use animation on graphic objects.

### *Printing a Diagram*

Introduces the API that allows you to print diagrams.

### *Importing SVG Files*

Explains how to import SVG documents in a diagram.

*Improving the Design-Time Behavior of Your Graphic Object*

Explains how to improve the design-time behavior of the graphic objects that you create.

*Creating BPMN Diagrams*

Describes how to create BPMN diagrams with IBM ILOG Diagram for .NET.

*Localizing an IBM ILOG Diagram for .NET Application*

Describes how to create a localized version of IBM ILOG Diagram for .NET.

*Using Graph Layout Algorithms*

Illustrates the Graph Layout functionality delivered with IBM ILOG Diagram for .NET.



## Creating a Basic Diagram Programmatically

In IBM® ILOG® Diagram for .NET you can create a diagram by assembling graphic objects. The graphic objects are subclasses of the `GraphicObject` class. The diagram itself is a graphic object called *container object* because it contains other graphic objects. Containers are subclasses of the abstract class `GraphicContainer`. The typical class used as the top container of your diagram is the `Group` class which is a container object that simply displays the graphic object it contains.

The following example shows how to create a diagram composed of three graphic objects: a rectangle (`Rect`), an ellipse (`Ellipse`) and a text object (`Text`). The three objects are added to a **Group** object.

```
Group CreateDiagram()
{
    // Creates the Group object that contains
    // the graphic objects that compose the diagram
    Group diagram = new Group();

    // Creates a rectangle object with text inside
    Rect rect = new Rect(0, 0, 100, 100);
    rect.Fill = new SolidFill(Color.Aquamarine);
    rect.Text = "A rectangle";
    rect.Radius = new Size2D(10, 10);

    // Creates an Ellipse object filled with a linear gradient
    Ellipse ellipse = new Ellipse(150, 0, 100, 100);
}
```

## Creating a Basic Diagram Programmatically

```
        ellipse.Stroke = new Stroke(Color.Blue, 3);
        ellipse.Fill = new LinearGradientFill(
            new Point2D(0, 0), new Point2D(0, 1),
            Color.Blue, Color.Yellow);

        // Creates a Text object
        Text text = new Text(new Point2D(125, 120), "My First Diagram");
        text.VerticalAlignment = VerticalTextAlignment.Bottom;
        text.HorizontalAlignment = HorizontalTextAlignment.Center;
        text.Font = new Font("Helvetica", 20);

        // Adds the objects in the diagram
        diagram.Objects.AddRange(
            new GraphicObject[] { rect, ellipse, text });

    return diagram;
}
Function CreateDiagram() As Group

    ' Creates the Group object that contains
    ' the graphic objects that compose the diagram

    Dim diagram As Group = New Group

    ' Creates a rectangle object with text inside

    Dim rect As Rect = New Rect(0, 0, 100, 100)
    rect.Fill = New SolidFill(Color.Aquamarine)
    rect.Text = "A rectangle"
    rect.Radius = New Size2D(10, 10)

    ' Creates an Ellipse object filled with a linear gradient

    Dim ellipse As Ellipse = New Ellipse(150, 0, 100, 100)
    ellipse.Stroke = New Stroke(Color.Blue, 3)
    ellipse.Fill = New LinearGradientFill(
        New Point2D(0, 0), New Point2D(0, 1), _
        Color.Blue, Color.Yellow)

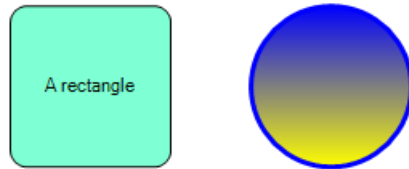
    ' Creates a Text object

    Dim text As Text = New Text(New Point2D(125, 120), "My First Diagram")
    text.VerticalAlignment = VerticalTextAlignment.Bottom
    text.HorizontalAlignment = HorizontalTextAlignment.Center
    text.Font = New Font("Helvetica", 20)

    ' Adds the objects in the diagram

    diagram.Objects.AddRange(New GraphicObject() {rect, ellipse, text})
    Return diagram
End Function
```

Here is the diagram you have just created:



## My First Diagram

The diagram you have created can now be used in a Windows® Forms application through the `DiagramView` class located in the `ILOG.Diagrammer.Windows.Forms` namespace. The **DiagramView** class is the Windows Forms control for displaying and interacting with a diagram in a Windows Forms application.

For more information, see *Displaying Diagrams in a Windows Forms Application*.

You can also display and interact with this diagram in a Web Form application using the `DiagramView` or `AjaxDiagramView`. These classes are Web controls that you can embed in an ASP.NET or ASP.NET Ajax application.

For more information, see *Displaying Diagrams in an ASP.NET Application*.





# *Displaying Diagrams in a Windows Forms Application*

IBM® ILOG® Diagram for .NET provides a Windows® Forms control to display diagrams into Windows applications: the `DiagramView` class.

The **DiagramView** control is responsible for drawing the view content and for dispatching the input events to the displayed graphic objects or to a view interactor.

In addition to that, the **DiagramView** class has a selection mechanism to manage and display selected objects.

## **In This Section**

### *Displaying Diagrams in a Diagram View*

Explains how to display diagrams in a diagram view.

### *Controlling the Zoom Level in a Diagram View*

Describes how to control the zoom level.

### *Scrolling in a Diagram View*

Describes how to use the scrolling feature.

### *Showing the Whole View Content*

Describes how to show the entire view content.

### *Modifying the Appearance of a Diagram View*

Describes how to modify the appearance of a diagram view.

### *Displaying a Grid in a Diagram View*

Describes how to display a grid in a diagram view.

### *Displaying Rulers in a Diagram View*

Describes how to display rulers in a diagram view.

### *Using the Predefined Behavior of the Diagram View*

Illustrates the predefined behaviors of the diagram view.

## **Related Sections**

### *Handling Interactions in a Diagram View (WinForms)*

Explains how to handle mouse and keyboard events in a graphic object and in a WinForms diagram view.

### *Handling Selection in a Diagram*

Explains how to handle selected objects in a diagram.

---

## **Displaying Diagrams in a Diagram View**

In order to display information, the **DiagramView** control must be connected to a graphic container, an instance of the **GraphicContainer** class. The graphic container associated with a **DiagramView** control can be set using the **Content** property of the **DiagramView** class.

The following example shows how to create a **DiagramView** displaying the content of a **Group** that contains an **Ellipse**:

```
public class TestForm : Form
{
    public TestForm() : base()
    {
        DiagramView view = new DiagramView();
        view.Dock = System.Windows.Forms.DockStyle.Fill;
        Group container = new Group();
        Ellipse ellipse = new Ellipse(0, 0, 100, 100);
        container.Objects.Add(ellipse);
        view.Content = container;
        Controls.Add(view);
        Size = new Size(200, 200);
    }
}
Public Class TestForm
    Inherits Form

    Public Sub New()
        MyBase.New
```

```

Dim view As DiagramView = New DiagramView
view.Dock = System.Windows.Forms.DockStyle.Fill
Dim container As Group = New Group
Dim ellipse As Ellipse = New Ellipse(0, 0, 100, 100)
container.Objects.Add(ellipse)
view.Content = container
Controls.Add(view)
Size = New Size(200, 200)
End Sub
End Class

```

When the control is connected to a graphic container, it listens to events coming from the container and is updated for each modification of the container. For example, when a new graphic object is added to or removed from the graphic container, the control is updated to display the new container content.

**Note:** The *DiagramView* only paints the content of the container it displays, not the container itself. For this reason, some properties set on the container will be ignored when the container is connected to a *DiagramView*. For example, the opacity, or the local transformation will be ignored.

---

## Controlling the Zoom Level in a Diagram View

The Transform property of the DiagramView is used to control the affine transformation applied to the view content. This transformation can be used to zoom in, zoom out, translate, or rotate the diagram displayed by the view.

The following example shows how to set a zoom factor of 2 on the view:

```

public class TestForm : Form
{
    public TestForm() : base()
    {
        DiagramView view = new DiagramView();
        view.Dock = System.Windows.Forms.DockStyle.Fill;
        Group container = new Group();
        Ellipse ellipse = new Ellipse(0, 0, 100, 100);
        container.Objects.Add(ellipse);
        view.Content = container;
        view.Transform = new Transform(2, 2, new Point2D(50, 50));
        Controls.Add(view);
        Size = new Size(200, 200);
    }
}
Public Class TestForm
    Inherits Form

    Public Sub New()

```

## Displaying Diagrams in a Windows Forms Application

```
MyBase.New
Dim view As DiagramView = New DiagramView
view.Dock = System.Windows.Forms.DockStyle.Fill
Dim container As Group = New Group
Dim ellipse As Ellipse = New Ellipse(0, 0, 100, 100)
container.Objects.Add(ellipse)
view.Content = container
view.Transform = new Transform(2, 2, new Point2D(50, 50))
Controls.Add(view)
Size = New Size(200, 200)
End Sub
End Class
```

Although the **Transform** property can be used to zoom in or zoom out the view, you may want to use the following methods of **DiagramView**, which are easier to use:

### Transformation Methods

Method	Description
Zoom	Zooms or un-zooms the view.
ZoomIn	Zooms in the view.
ZoomOut	Zooms out the view.
ShowAll	Zooms or un-zooms the view so that all the view content becomes visible.
ShowAllInRectangle	Zooms or un-zooms the view so that all the view content fits into the specified rectangle.
Translate	Translates the view.

All these methods result in a change of the **Transform** property. For various reasons, you may want to control the value of the **Transform** property, for example to limit the zoom factor. The **DiagramView** class has several properties that help you control the **Transform** property value. The following table lists the properties that can be used to control the **Transform** property:

Property	Description
MinimumZoom	The minimum value for the zoom.
MaximumZoom	The maximum value for the zoom.
KeepAspectRatio	Indicates if the view maintains the aspect ratio of its content.

Each time the **Transform** property changes on the view, the **TransformChanging** and **TransformChanged** events are sent. By means of these events, you can control and monitor the affine transformation applied to the view.

The following example shows how to listen to the **TransformChanging** event to set a veto when the transformation is not a scale:

```
DiagramView view = new DiagramView();
view.TransformChanging += new TransformChangeEventHandler(TransformChanging);
...
private void TransformChanging(object sender, TransformChangeEventArgs args)
{
    if (!args.Transform.IsScale)
        args.Cancel = true;
}
Dim view As New DiagramView
AddHandler view.TransformChanging, AddressOf Me.TransformChanging
...
Private Sub TransformChanging(ByVal sender As Object, ByVal args As
TransformChangeEventArgs)
    If Not args.Transform.IsScale Then
        args.Cancel = true
    End If
End Sub
```

---

## Scrolling in a Diagram View

When the **DiagramView** content is larger than the view, scroll bars can be displayed to enable the scrolling inside the view content. The default behavior shows the scroll bars only when they are needed, that is, when the bounds of the view content are larger than the view.

The following table lists the properties that can be used to control the scroll bars visibility and behavior:

Scroll Bars Properties

Property	Description
HScrollBar	Gets or sets the visibility of the horizontal scroll bar.
VScrollBar	Gets or sets the visibility of the vertical scroll bar.
InfiniteScroll	Indicates whether infinite scrolling is allowed using the scroll arrows.
InstantScroll	Indicates whether scrolling the view using the scroll bar is instantaneous.

ContentMargins	Gets or sets the margins used to enlarge the content bounds.
ContentBounds	Gets or sets the bounds in which scrolling will be possible.

---

### Changing the Scrolling Area

By default, the scrolling area is defined by the view content bounds. This allows you to scroll into the whole view content. You may want to use another area, for example in order to provide the scrolling in a portion of the view content bounds. To do this, use the **ContentBounds** property of the **DiagramView** class. You can also add margins to this area by setting the **ContentMargins** property. Margins added by setting this property are specified in pixels, whatever the zoom level is.

When the **InfiniteScroll** property is set to **true**, it is possible to scroll out of the area defined by the **ContentBounds** property by clicking the scroll arrows.

---

### Showing the Whole View Content

The **DiagramView** has a special mode that always shows the entire view content. This mode can be set by using the **AutoSizeContent** property. When this property is set to **true**, the scroll bars are no longer active and it is not possible to change the affine transformation of the view. In this mode, the view displays its content so that the area defined by the **ContentBounds** property is fully visible.

To make the **ContentBounds** area fit into the view, the **DiagramView** uses the **AutoSizeContentMode** property. When this property is set to **ResizeMode.Zoom**, the **DiagramView** zooms its content by adjusting its **Transform** property. When the property is set to **ResizeMode.Resize**, the **DiagramView** resizes its content so that it fits into the view.

---

### Modifying the Appearance of a Diagram View

The following tables show the appearance properties of the **DiagramView** control.

Appearance Properties

Property	Description
Antialiasing	Indicates whether the control uses anti-aliasing to display its content.
BackColor	The color used for the background of the control.

BackgroundImage	The image used for the background of the control.
BorderStyle	The border style of the control.

## Displaying a Grid in a Diagram View

The `DiagramView` class can display a grid below or above its graphic content. A grid is a subclass of the `Grid` class. Its purpose is to draw graphic decorations such as dots or lines at regular intervals to help positioning graphic objects. A **Grid** can be set by using the `Grid` property of the **DiagramView**.

The following example shows how to set a **Grid** that displays vertical and horizontal lines with a spacing of 100 pixels on a **DiagramView**:

```
DiagramView view = new DiagramView();
Grid g = new Grid();
g.HorizontalSpacing = 100f;
g.VerticalSpacing = 100f;
g.GridStyle = GridStyle.Lines;
g.HorizontalLineColor = Color.Blue;
g.VerticalLineColor = Color.Blue;
view.Grid = g;
Dim view as New DiagramView()
Dim g as New Grid()
g.HorizontalSpacing = 100f
g.VerticalSpacing = 100f
g.GridStyle = GridStyle.Lines
g.HorizontalLineColor = Color.Blue
g.VerticalLineColor = Color.Blue
view.Grid = g
```

### Setting the Grid Style

The style of the grid is controlled by the `GridStyle` property. This property can take the following values:

Grid Styles

Property	Description
<b>GridStyle.Dots</b>	Displays the grid with dots.
<b>GridStyle.Lines</b>	Displays the grid using lines.

**Note:** The *GridStyle* property can be set to a combination of the values listed above.

Depending on the grid style used, several properties can be used to customize the grid drawing. The following table shows a list of those properties:

### Grid Appearance Properties

Property	Description
DotColor	The color for the grid dots.
DotSize	The size for the grid dots.
HorizontalLineColor	The color for horizontal grid lines.
HorizontalLineStyle	The style for horizontal grid lines.
HorizontalLineWidth	The width for horizontal grid lines.
VerticalLineColor	The color for vertical grid lines.
VerticalLineStyle	The style for vertical grid lines.
VerticalLineWidth	The width for vertical grid lines.

The following example shows how to set a **Grid** that displays dots, vertical and horizontal lines with a spacing of 100 pixels on a **DiagramView**:

```
DiagramView view = new DiagramView();
Grid g = new Grid();
g.HorizontalSpacing = 100f;
g.VerticalSpacing = 100f;
g.GridStyle = GridStyle.Lines|GridStyle.Dots;
g.HorizontalLineColor = Color.Blue;
g.VerticalLineColor = Color.Blue;
g.DotColor = Color.Red;
g.DotSize = 3f;
view.Grid = g;
Dim view As DiagramView = New DiagramView
Dim g As Grid = New Grid
g.HorizontalSpacing = 100!
g.VerticalSpacing = 100!
g.GridStyle = (GridStyle.Lines Or GridStyle.Dots)
g.HorizontalLineColor = Color.Blue
g.VerticalLineColor = Color.Blue
g.DotColor = Color.Red
g.DotSize = 3!
view.Grid = g
```

---

### Setting the Grid Geometry

Whatever the grid style is, a grid displays decorations at regular intervals. To specify the grid intervals, use the `HorizontalSpacing` and `VerticalSpacing` properties. You can also specify the origin point of the grid by setting the `Origin` property.



The following example shows how to display dots each 10 pixels, with a grid origin point set to the point of coordinates (1, 1):

```
DiagramView view = new DiagramView();
Grid g = new Grid();
g.HorizontalSpacing = 10f;
g.VerticalSpacing = 10f;
g.GridStyle = GridStyle.Dots;
g.DotColor = Color.Red;
g.Origin = new Point2D(1, 1);
view.Grid = g;
Dim view as New DiagramView()
Dim g as New Grid()
g.HorizontalSpacing = 10f
g.VerticalSpacing = 10f
g.GridStyle = GridStyle.Dots
g.DotColor = Color.Red
g.Origin = New Point2D(1, 1)
view.Grid = g
```

---

### Using Grid to Snap Coordinates

When the Active property of the **Grid** is set to **true**, the grid is used by the **DiagramView** during interactions to snap points coordinate onto grid points. This feature is very useful to position and align graphic objects.

The method called to snap points is the Snap method. It is mainly called by the SnapToPoint method.

---

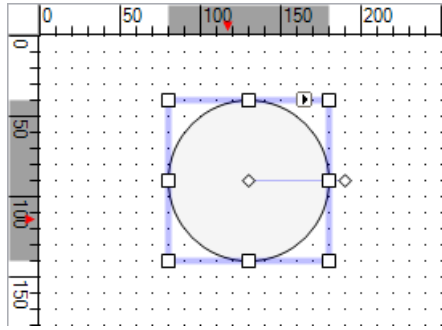
## Displaying Rulers in a Diagram View

A ruler is a scale that displays ticks and labels at regular intervals to help positioning graphic objects. It also shows the pointer location, as well as an area of interest. Here is an example of an horizontal ruler:



Horizontal and vertical rulers are respectively implemented by the DiagramHRuler and DiagramVRuler classes.

A DiagramView embeds both an horizontal and a vertical ruler. To show the rulers in the **DiagramView**, use the DiagramView.HRuler and DiagramView.VRuler properties. Note that you can connect other rulers to the same view by creating them and setting their DiagramRuler.DiagramView property. Here is a picture showing a view with its rulers displayed:



The following example shows how to set up rulers in a **DiagramView**:

```
DiagramView view = new DiagramView();
view.HRuler = RulerVisibility.Visible;
view.VRuler = RulerVisibility.Visible;
Dim view as New DiagramView()
view.HRuler = RulerVisibility.Visible
view.VRuler = RulerVisibility.Visible
```

## Styling the Rulers

When using the internal rulers of the **DiagramView**, use the following properties of **DiagramView** to style the rulers:

Property	Description
<b>HRuler</b>	Indicates whether the horizontal ruler is visible or not.
<b>VRuler</b>	Indicates whether the vertical ruler is visible or not.
<b>RulerBackColor</b>	Gets or set the background color of the rulers.
<b>RulerTextColor</b>	Gets or set the text color of the rulers.
<b>RulerTickColor</b>	Gets or set the ticks color of the rulers.
<b>RulerUnit</b>	Gets or set the unit of the rulers.
<b>RulerSelectionVisible</b>	Indicates whether the selection is displayed in the rulers.
<b>RulerSelectionColor</b>	Gets or set the color of the rulers selection area.
<b>RulerMarkerColor</b>	Gets or set the color of the rulers marker.

To style external rulers, see the **DiagramRuler** class in the reference manual.

---

## Using the Predefined Behavior of the Diagram View

A `DiagramView` has the following predefined behavior.

---

### Zooming In and Out

In the **DiagramView**, an application user can zoom in or out by using the mouse wheel while pressing the CTRL key. The zoom factor applied when the mouse wheel is used is defined by the `DefaultMouseWheelZoomFactor` property.

This feature can be disabled by setting the `WheelZoom` property of the **DiagramView** to `false`. In addition, minimum and maximum zoom values can be controlled by setting the `MinimumZoom` and `MaximumZoom` properties.

***Note:** This behavior is automatically disabled when the `AutoSizeContent` property is set to `true`.*

---

### Tooltips

Tooltips appear in the **DiagramView** when the mouse hovers a graphic object that has tooltip information defined on it. Set the `ShowObjectToolTips` property of the **DiagramView** to `false` to disable tooltips in the control.

***Note:** Some view interactors may disable tooltips.*



# *Displaying Diagrams in an ASP.NET Application*

IBM® ILOG® Diagram for .NET provides two WebForm controls to display diagrams into ASP.NET 2.0 applications: *DiagramView* and *AjaxDiagramView* classes.

The **DiagramView** Web control is a basic ASP.NET 2.0 control responsible for displaying diagrams in a basic Web application. It supports image map generation and limited interactions.

The **AjaxDiagramView** Web control is an advanced ASP.NET 2.0 control that extends the basic functionalities of the **DiagramView** class. It adds support for rich client-side user interaction and image tiling.

## **In This Section**

### *Displaying a Diagram in a Basic Web Application*

Explains the ready-to-use Web diagram view.

### *Displaying a Diagram in an AJAX Web Application*

Explains the ready-to-use AJAX Web diagram view.

## Displaying a Diagram in a Basic Web Application

The *DiagramView* Web control is a basic ASP.NET 2.0 control responsible for displaying diagrams in a basic Web application. It supports image map generation and limited interactions. In addition, the **DiagramView** class has a selection mechanism to manage and display selected objects.

### In This Section

#### *Displaying Diagrams in a Diagram View*

Explains how to display a diagram in a Web diagram view.

#### *Controlling the Zoom Level in a Diagram View*

Explains how to control the zooming factor in a Web diagram view.

#### *Showing the Whole View Content*

Explains how to display the whole diagram in a Web diagram view.

#### *Controlling the Image Generation*

Explains how to control the image generation.

#### *Image Map Generation*

Explains how to generate an image map in a Web diagram view.

#### *Handling User Input Event in a Diagram View*

Explains how to handle input events in a Web diagram view.

### Related Section

#### *Creating your First IBM ILOG Diagram for .NET AJAX Web Site*

Walks you through the process of creating your first IBM ILOG Diagram AJAX Web Site.

---

## Displaying Diagrams in a Diagram View

In order to display information, the **DiagramView** Web control must be connected to a graphic container, an instance of the *GraphicContainer* class. The graphic container associated with a **DiagramView** Web control can be set using the *Content* property of the **DiagramView** class.

The following example shows how to create a Page containing a **DiagramView** displaying the content of a Group that contains an Ellipse:

The `Default.aspx` file:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs"
Inherits="_Default" %>
```

## Displaying a Diagram in a Basic Web Application

```
<%@ Register Assembly="ILOG.Diagrammer.Web, Version=1.6.0.0, Culture=neutral,
PublicKeyToken=7906592bc7cc7340"
    Namespace="ILOG.Diagrammer.Web.UI" TagPrefix="cc1" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>My First Diagram</title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <cc1:DiagramView ID="DiagramView1" runat="server" BorderColor="Black"
BorderStyle="Solid"
                BorderWidth="1px" ContentSessionId="DiagramView1Content"
Height="200px" Width="300px" />
        </div>
    </form>
</body>
</html>
```

### The C# code-behind file:

```
using System;
using System.Web;
using System.Web.UI;
using ILOG.Diagrammer.Graphic;

public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (!IsPostBack) {
            Group container = new Group();
            Ellipse ellipse = new Ellipse(0, 0, 100, 100);
            container.Objects.Add(ellipse);
            DiagramView1.Content = container;
        }
    }
}
```

### The VB.NET code-behind file:

```
Imports System
Imports System.Web
Imports System.Web.UI
Imports ILOG.Diagrammer.Graphic

Public Class _Default
Inherits System.Web.UI.Page

    Protected Sub Page_Load(ByVal sender As Object, ByVal e As EventArgs)
        If Not IsPostBack Then
            Dim container As Group = New Group
            Dim ellipse As Ellipse = New Ellipse(0, 0, 100, 100)
            container.Objects.Add(ellipse)
            DiagramView1.Content = container
        End If
    End Sub
End Class
```

## Displaying Diagrams in an ASP.NET Application

```
End If
End Sub
End Class
```

While the **DiagramView** Web control is responsible for drawing the view content, it actually delegates the image generation to a specialized **HttpHandler** implementation which retrieves the graphic container from a cache. By default, the cache used is the Session, and the ID used to store the graphic container in the session must be set using the `ContentSessionId` property of the **DiagramView** class.

---

### Controlling the Zoom Level in a Diagram View

The `Transform` property of the **DiagramView** is used to control the affine transformation applied to the view content. This transformation can be used to zoom in, zoom out, translate, or rotate the diagram displayed by the view.

The following example shows how to set a zoom factor of 2 on the view:

The Default.aspx file:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs"
Inherits="_Default" %>

<%@ Register Assembly="ILOG.Diagrammer.Web, Version=1.6.0.0, Culture=neutral,
PublicKeyToken=7906592bc7cc7340"
Namespace="ILOG.Diagrammer.Web.UI" TagPrefix="cc1" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
<title>Untitled Page</title>
</head>
<body>
<form id="form1" runat="server">
<div>
<cc1:DiagramView ID="DiagramView1" runat="server" BorderColor="Black"
BorderStyle="Solid"
BorderWidth="1px" ContentSessionId="DiagramView1Content"
Height="200px" Width="300px" />
</div>
</form>
</body>
</html>
```

The C# code-behind file:

```
using System;
using System.Web;
using System.Web.UI;
using ILOG.Diagrammer.Graphic;
using ILOG.Diagrammer;
```



```
public partial class _Default : System.Web.UI.Page
{
    protected void Page_Init(object sender, EventArgs e) {
        DiagramView1.Transform = new Transform(2, 2, new Point2D(50, 50));
        if (!IsPostBack) {
            Group container = new Group();
            Ellipse ellipse = new Ellipse(0, 0, 100, 100);
            container.Objects.Add(ellipse);
            DiagramView1.Content = container;
        }
    }
}
```

The VB.NET code-behind file:

```
Imports System
Imports System.Web
Imports System.Web.UI
Imports ILOG.Diagrammer.Graphic
Imports ILOG.Diagrammer

Public Class _Default
    Inherits System.Web.UI.Page

    Protected Sub Page_Load(ByVal sender As Object, ByVal e As EventArgs)
        DiagramView1.Transform = new Transform(2, 2, new Point2D(50, 50))
        If Not IsPostBack Then
            Dim container As Group = New Group
            Dim ellipse As Ellipse = New Ellipse(0, 0, 100, 100)
            container.Objects.Add(ellipse)
            DiagramView1.Content = container
        End If
    End Sub
End Class
```

Although the **Transform** property can be used to zoom in or out the view, you may prefer to use the following methods of **DiagramView**, which are easier to use:

Transformation Methods:

Method	Description
Zoom	Zooms in or zooms out the view.
ZoomIn	Zooms in the view.
ZoomOut	Zooms out the view.
ShowAll	Zooms in or zooms out the view so that all the view content becomes visible.
ShowAllInRectangle	Zooms in or zooms out the view so that all the view content fits into the specified rectangle.
Translate	Translates the view.

All these methods results in a change of the **Transform** property. For various reasons you may want to control the value of the **Transform** property, for example if you want to limit the zoom factor. The **DiagramView** class has several properties that help you control the **Transform** property value. The following table lists the properties that can be used to control the **Transform** property:

Property	Description
MinimumZoom	The minimum value for the zoom.
MaximumZoom	The maximum value for the zoom.
KeepAspectRatio	Indicates whether the view maintains the aspect ratio of its content.

Each time the **Transform** property changes on the view, the **TransformChanging** and **TransformChanged** events are sent. Using those events, you can control and monitor the affine transformation applied to the view.

---

### Showing the Whole View Content

The **DiagramView** has a special mode that always shows the entire view content. This mode can be set by using the **AutoSizeContent** property. When this property is set to **true**, it is no longer possible to change the affine transformation of the view. In this mode, the view displays its content so that the area defined by the **ContentBounds** property is fully visible.

To make the **ContentBounds** area fit into the view, the **DiagramView** uses the **AutoSizeContentMode** property. When this property is set to **ResizeMode.Zoom**, the **DiagramView** zooms its content by adjusting its **Transform** property. When the property is set to **ResizeMode.Resize**, the **DiagramView** resizes its content so that it fits into the view.

---

### Controlling the Image Generation

The following table shows the image-related properties of the **DiagramView** Web control.

Property	Description
ImageFormat	The format of the generated image.

---

### Image Map Generation

The **DiagramView** Web control allows you to generate an image map to add a basic level of interactivity to a diagram image by means of the **RenderImageMap** property. When the property is set to **true**, the **DiagramView** Web control processes every objects in its graphic container to build an image map of the displayed hierarchy.

By default, an AREA tag is generated for every objects. The AREA Tooltip is initialized to the object tooltip and the SHAPE AREA attribute is initialized to the transformed bounding box of the object.

This default behavior can be customized thanks to the QueryMapArea event of the **DiagramView** class. This event is raised when a graphic object is processed to query the map area information in order to generate the corresponding AREA tag.

The QueryMapAreaEventArgs class provides data to the **QueryMapArea** event and enables you to provide the following information:

- ◆ The AREA tooltip, by means of the ToolTip property.
- ◆ The AREA href, by means of the HRef property.
- ◆ Some extra attributes, for example a "onmouseover" JavaScript event handler, by means of the ExtraAttributes property.
- ◆ The AREA geometry, by means of the Geometry property. By default, it is the transformed bounding box of the object in view coordinates.

You can prevent the generation of an AREA tag by means of the Cancel property of the **QueryMapAreaEventArgs** class. When this property is **true**, the current object is skipped and no area tag is generated for it.

***Note:** As mentioned above, the default behavior sets the tooltip attribute of the AREA tag to the object tooltip if the **ToolTip** property of the **QueryMapAreaEventArgs** is **null**. When the **ToolTip** property is set to **String.Empty**, the default behavior is disabled.*

The following example shows how to handle the **QueryMapArea** event. It displays a rotated rectangle that links to the IBM Web site and displays a tooltip. It also changes the text of a SPAN element when the mouse moves over the diamond thanks to the **ExtraAttributes** property of the **QueryMapAreaEventArgs**.

The Default.aspx file:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs"
Inherits="_Default" %>

<%@ Register Assembly="ILOG.Diagrammer.Web, Version=1.6.0.0, Culture=neutral,
PublicKeyToken=7906592bc7cc7340"
    Namespace="ILOG.Diagrammer.Web.UI" TagPrefix="cc1" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Untitled Page</title>
</head>
```

## Displaying Diagrams in an ASP.NET Application

```
<body>
  <form id="form1" runat="server">
    <div>
      <ccl:DiagramView ID="DiagramView1" runat="server"
RenderImageMap="true" BorderColor="Black" BorderStyle="Solid"
      BorderWidth="1px" ContentSessionId="DiagramView1Content"
Height="200px" Width="300px" OnQueryMapArea="DiagramView1_QueryMapArea" />
      <span id="message">Move the mouse pointer over the diamond.</span>
    </div>
  </form>
</body>
</html>
```

### The C# code-behind file:

```
using System;
using System.Web;
using System.Web.UI;
using ILOG.Diagrammer.Graphic;
using ILOG.Diagrammer;
using System.Drawing;

public partial class _Default : System.Web.UI.Page
{
    protected void Page_Init(object sender, EventArgs e) {
        if (!IsPostBack) {
            Group container = new Group();
            Rect rect = new Rect(100, 50, 100, 100);
            rect.Transform = new Transform(45, new Point2D(150, 100));
            rect.Stroke = new Stroke(Color.LightSteelBlue);
            rect.Fill = new SolidFill(Color.LightBlue);
            container.Objects.Add(rect);
            DiagramView1.Content = container;
        }
    }

    protected void DiagramView1_QueryMapArea(object sender,
        ILOG.Diagrammer.Web.UI.QueryMapAreaEventArgs e) {
        e.HRef = "http://www.ibm.com";
        e.ToolTip = "Click to visit IBM !";
        e.ExtraAttributes = "
onmouseover=\"document.getElementById('message').innerText='Go to IBM web
site';return true;\""+
        " onmouseout=\"document.getElementById('message').innerText="+
        "'Move the mouse pointer over the diamond.';return true;\"";
    }
}
```

### The VB.NET code-behind file:

```
Imports System
Imports System.Web
Imports System.Web.UI
Imports ILOG.Diagrammer.Graphic
Imports ILOG.Diagrammer
Imports System.Drawing

Public Class _Default
Inherits System.Web.UI.Page
```

```

Protected Sub Page_Init(ByVal sender As Object, ByVal e As EventArgs)
    If Not IsPostBack Then
        Dim container As Group = New Group
        Dim rect As Rect = New Rect(100, 50, 100, 100)
        rect.Transform = New Transform(45, New Point2D(150, 100))
        rect.Stroke = New Stroke(Color.LightSteelBlue)
        rect.Fill = New SolidFill(Color.LightBlue)
        container.Objects.Add(rect)
        DiagramView1.Content = container
    End If
End Sub

Protected Sub DiagramView1_QueryMapArea(ByVal sender As Object, _
                                         ByVal e As
ILOG.Diagrammer.Web.UI.QueryMapAreaEventArgs)
    e.HRef = "http://www.ibm.com"
    e.ToolTip = "Click to visit IBM !"
    e.ExtraAttributes = "
onmouseover=""document.getElementById('message').innerText='Go to IBM web
site';return true;""+ _
        " onmouseout=""document.getElementById('message').innerText="+_
        "'Move the mouse pointer over the diamond.';return true;""
End Sub
End Class

```

---

### Handling User Input Event in a Diagram View

The **DiagramView** Web control provides a simple way to handle user input events by means of the Click event of the **DiagramView** class. This event is raised when the user clicks on the view. You can then perform some post-processing in response to the click event.

The following example shows how to handle **Click** event to change the color of the objects targeted by the click. An image map is generated to provide tooltips.

The Default.aspx file:

```

<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs"
Inherits="_Default" %>

<%@ Register Assembly="ILOG.Diagrammer.Web, Version=1.6.0.0, Culture=neutral,
PublicKeyToken=7906592bc7cc7340"
    Namespace="ILOG.Diagrammer.Web.UI" TagPrefix="cc1" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <cc1:DiagramView ID="DiagramView1" runat="server"
RenderImageMap="true" BorderColor="Black" BorderStyle="Solid"
                BorderWidth="1px" ContentSessionId="DiagramView1Content"

```

## Displaying Diagrams in an ASP.NET Application

```
Height="200px" Width="300px" OnClick="DiagramView1_Click" />
    </div>
</form>
</body>
</html>
```

### The C# code-behind file:

```
using System;
using System.Web;
using System.Web.UI;
using ILOG.Diagrammer.Graphic;
using ILOG.Diagrammer;
using System.Drawing;

public partial class _Default : System.Web.UI.Page
{
    protected void Page_Init(object sender, EventArgs e) {
        if (!IsPostBack) {
            Group container = new Group();
            Rect rect = new Rect(100, 50, 100, 100);
            rect.ToolTip = "Click Me!";
            rect.Transform = new Transform(45, new Point2D(150, 100));
            rect.Stroke = new Stroke(Color.LightSteelBlue);
            rect.Fill = new SolidFill(Color.LightBlue);
            container.Objects.Add(rect);
            rect = new Rect(100, 50, 100, 100);
            rect.ToolTip = "Click Me!";
            rect.Stroke = new Stroke(Color.Salmon);
            rect.Fill = new SolidFill(Color.Coral);
            container.Objects.Add(rect);
            DiagramView1.Content = container;
        }
    }

    protected void DiagramView1_Click(object sender, ImageClickEventArgs e) {
        Shape hittest = DiagramView1.HitTestSelectable(new Point(e.X, e.Y)) as
        Shape;
        if (hittest != null) {
            Color c = ((SolidFill)hittest.Fill).Color;
            int r = (c.R + 20) % 255;
            int g = (c.G + 20) % 255;
            int b = (c.B + 20) % 255;
            Color newColor = Color.FromArgb(r, g, b);
            ((SolidFill)hittest.Fill).Color = newColor;
        }
    }
}
```

### The VB.NET code-behind file:

```
Imports System
Imports System.Web
Imports System.Web.UI
Imports ILOG.Diagrammer.Graphic
Imports ILOG.Diagrammer
Imports System.Drawing

Public Class _Default
```

```

Inherits System.Web.UI.Page

Protected Sub Page_Init(ByVal sender As Object, ByVal e As EventArgs)
    If Not IsPostBack Then
        Dim container As Group = New Group
        Dim rect As Rect = New Rect(100, 50, 100, 100)
        rect.ToolTip = "Click Me!"
        rect.Transform = New Transform(45, New Point2D(150, 100))
        rect.Stroke = New Stroke(Color.LightSteelBlue)
        rect.Fill = New SolidFill(Color.LightBlue)
        container.Objects.Add(rect)
        rect = New Rect(100, 50, 100, 100)
        rect.ToolTip = "Click Me!"
        rect.Stroke = New Stroke(Color.Salmon)
        rect.Fill = New SolidFill(Color.Coral)
        container.Objects.Add(rect)
        DiagramView1.Content = container
    End If
End Sub

Protected Sub DiagramView1_Click(ByVal sender As Object, ByVal e As
ImageClickEventArgs)
    Dim hittest As Shape = TryCast(DiagramView1.HitTestSelectable( _
        New Point(e.X, e.Y)), Shape)
    If Not (hittest Is Nothing) Then
        Dim c As Color = CType(hittest.Fill, SolidFill).Color
        Dim r As Integer = (c.R + 20) Mod 255
        Dim g As Integer = (c.G + 20) Mod 255
        Dim b As Integer = (c.B + 20) Mod 255
        Dim newColor As Color = Color.FromArgb(r, g, b)
        CType(hittest.Fill, SolidFill).Color = newColor
    End If
End Sub
End Class

```

---

## Displaying a Diagram in an AJAX Web Application

IBM® ILOG® Diagram for .NET comes with a set of classes that enables to build AJAX-enabled Web applications that provide rich client-side user interaction and a better user experience.

This framework is based on the ASP.NET AJAX extension. It requires by default the .NET Framework 3.5 SDK. In case you want to target a .NET Framework 2.0 web application and ASP.NET AJAX 1.0, IBM ILOG Diagram for .NET 2.0 provides the **ILOG.Diagrammer.Web.Ajax10.dll** assembly which has a dependency with ASP.NET AJAX 1.0. Note that in this case, the IBM ILOG Diagram for .NET 2.0 Project Templates cannot be used and the web controls must be added manually to the Visual Studio Toolbox.

The IBM ILOG Diagram for .NET AJAX components support the following browsers:

- ◆ Microsoft Internet Explorer 6
- ◆ Microsoft Internet Explorer 7

- ◆ Firefox 1.5
- ◆ Firefox 2.0
- ◆ Firefox 3.0

### In This Section

#### *Overview of the IBM ILOG Diagram for .NET AJAX Framework*

A brief introduction to the IBM ILOG Diagram for .NET AJAX.

#### *Displaying a Diagram in an AJAX Diagram View*

Explains how to show a diagram in an AJAX diagram view.

#### *Tiling Mode*

Explains how to using the tiling mode in an AJAX diagram view.

#### *Using Predefined Interaction Tools in an AJAX Diagram View*

Explains how to use the predefined interactors in an AJAX diagram view.

#### *Creating a New AJAX-enabled View Interactor*

Explains how to create a new AJAX-enabled view interactor.

---

## Overview of the IBM ILOG Diagram for .NET AJAX Framework

The IBM® ILOG® Diagram for .NET AJAX framework consists on a set of server-side ASP.NET components and JavaScript components based on the Microsoft ASP.NET 2.0 AJAX extension. This extension provides AJAX capabilities to ASP.NET 2.0 applications by means of server-side ASP.NET Web controls and JavaScript classes. It enables postback partial-refresh thanks to the **UpdatePanel** control, an asynchronous communication layer and provides an object oriented JavaScript library. For more information about the Microsoft ASP.NET 2.0 AJAX extension, see <http://ajax.asp.net>.

The IBM ILOG Diagram for .NET AJAX framework supports optimized refreshes when panning or zooming thanks to a tiled-based cache mechanism, rich user interaction by means of client-side interaction tools (editing capabilities like links or objects creation), client-side selection (with asynchronous contextual information).

---

## Displaying a Diagram in an AJAX Diagram View

Displaying a diagram in an AJAX-enabled ASP.NET application is the purpose of the AJAX diagram view component defined by the `AjaxDiagramView` class of the `ILOG.Diagrammer.Web.UI` namespace. This class inherits from **DiagramView** and as such inherits from all the features supported by it (see *Displaying Diagrams in a Diagram View* for a complete list of the features provided by the **DiagramView** class). It also implements the **IScriptControl** interface of the `System.Web.UI` namespace that allows you to add client capabilities to a Web server control by means of JavaScript components.



The following example shows how to create a Page containing an **AjaxDiagramView** Web control displaying the content of a Group that contains an Ellipse:

The Default.aspx file:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs"
Inherits="_Default" %>

<%@ Register Assembly="ILOG.Diagrammer.Web, Version=1.6.0.0, Culture=neutral,
PublicKeyToken=7906592bc7cc7340"
Namespace="ILOG.Diagrammer.Web.UI" TagPrefix="cc1" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
<title>Untitled Page</title>
</head>
<body>
<form id="form1" runat="server">
<asp:ScriptManager ID="ScriptManager1" runat="server" />
<div>
<asp:UpdatePanel ID="UpdatePanel1" runat="server">
<ContentTemplate>
<cc1:AjaxDiagramView ID="DiagramView1" runat="server" Width="300px"
Height="200px" ContentSessionId="DiagramView1Contents"/>
</ContentTemplate>
</asp:UpdatePanel>
</div>
</form>
</body>
</html>
```

The C# code-behind file:

```
using System;
using System.Data;
using System.Configuration;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
using ILOG.Diagrammer.Graphic;

public partial class _Default : System.Web.UI.Page
{
    protected void Page_Init(object sender, EventArgs e)
    {
        {
            if (!IsPostBack) {
                Group container = new Group();
                Ellipse ellipse = new Ellipse(0, 0, 100, 100);
                container.Objects.Add(ellipse);
                DiagramView1.Content = container;
            }
        }
    }
}
```

The VB.NET code-behind file:

```
Imports System
Imports System.Web
Imports System.Web.UI
Imports ILOG.Diagrammer.Graphic

Public Class _Default
Inherits System.Web.UI.Page

    Protected Sub Page_Init(ByVal sender As Object, ByVal e As EventArgs)
        If Not IsPostBack Then
            Dim container As Group = New Group
            Dim ellipse As Ellipse = New Ellipse(0, 0, 100, 100)
            container.Objects.Add(ellipse)
            DiagramView1.Content = container
        End If
    End Sub
End Class
```

### See Also

*Creating your First IBM ILOG Diagram for .NET AJAX Web Site*

---

### Tiling Mode

The default behavior of the `AjaxDiagramView` is to display its content in one image. While this behavior addresses most of the use cases, it may not be optimal in the case where the content is big and contains a kind of static layer in the background composed of objects that do not change (appearance or geometry) in the lifecycle of the application, and some objects that are displayed over it whose representation may change in the application lifecycle (a kind of dynamic layer). A typical example is the case of a map in the background with symbols whose appearance may change displayed over it.

The **AjaxDiagramView** Web control supports an advanced, tiled-based refresh mode that specifically addresses this use case by optimizing the display of the static content. In this mode, the **AjaxDiagramView** Web control handles two graphic container: one for the static layer (represented by the `StaticContent` property) and a second one for the dynamic layer (represented by the `Content` property).

This tiled-based refresh mode is activated by means of the `ImageTiling` property. When this property is **true**, the static content is considered as a grid where each cell (or tile) is drawn independently and asynchronously from the others. This grid structure allows a better user experience when panning or zooming the view giving a progressive load effect.

The tile size is by default 256 pixels in both directions. You can change the size by means of the `TileSize` property of the **AjaxDiagramView** class.

You can find a complete example that illustrates this feature in the `Samples\QuickStart\ImageTiling` directory.

---

## Using Predefined Interaction Tools in an AJAX Diagram View

This section explains how to use the predefined AJAX-enabled interaction tools provided in the IBM® ILOG® Diagram for .NET product.

The `AjaxDiagramView` Web control supports rich client-side user interactions by means of AJAX-enabled interaction tools. AJAX interaction tools are called view interactors.

### Setting an Interactor on an `AjaxDiagramView`

View interactors are provided as a set of classes and are defined by the `ViewInteractor` class of the `ILOG.Diagrammer.Web.UI` namespace. This is the base class for view interactor implementations.

The **`ViewInteractor`** class extends the ASP.NET 2.0 AJAX **`ExtenderControl`** class to add client behavior to an associated **`AjaxDiagramView`** Web control via the **`ILOG.Diagrammer.Web.UI.ViewBehavior`** JavaScript class. For more information on the **`ExtenderControl`** class, see <http://ajax.asp.net>.

In order to be used, an interactor must be set on a view. When an interactor is set on a view, all input events coming to the client-side representation of the **`AjaxDiagramView`** in the browser are forwarded to the JavaScript counterpart of the view interactor. Setting an interactor can be done either declaratively in the ASPX file, or programmatically in the code-behind file using the `Interactor` property of the **`AjaxDiagramView`** class.

The following example shows how to set a selection interactor on an AJAX diagram view Web control declaratively:

```
<body>
  <form id="form1" runat="server">
    <asp:ScriptManager ID="ScriptManager1" runat="server" />
    <div>
      <asp:UpdatePanel ID="UpdatePanel1" runat="server">
        <ContentTemplate>
          <cc1:AjaxDiagramView ID="DiagramView1" runat="server" Width="300px"
Height="200px" ContentSessionId="DiagramView1Contents"/>
          <cc1>SelectInteractor ID="SelectInteractor1" runat="server"
TargetControlID="DiagramView1" />
        </ContentTemplate>
      </asp:UpdatePanel>
    </div>
  </form>
</body>
```

The predefined AJAX-enabled view interactors available in the IBM ILOG Diagram for .NET library are:

- ◆ *Selection Interactor*
- ◆ *Zoom Interactor*
- ◆ *Pan Interactor* (support tiled-based cache mechanism)

- ◆ *Graphic Object Creation Interactor* (links and nodes)

**Note:** In the following sections, the interactor classes refer all to the `ILOG.Diagrammer.Web.UI` namespace.

### Selection Interactor

The selection interactor is defined by the `SelectInteractor` class and its JavaScript counterpart is the `ILOG.Diagrammer.Web.UI.SelectBehavior` JavaScript class. It allows you to select and move graphic objects displayed in a `DiagramView`.

#### *Selecting a Graphic Object*

There are several ways to select graphic objects depending on what you want to do:

- ◆ Single selection: to select only one graphic object, click the object with the left mouse button.
- ◆ Multiple selection on a per-object basis: to select several graphic objects one at a time, press the CTRL or Shift key and click with the left mouse button the graphic objects you want to select.

The multi selection capability is available provided it has been enabled on the view by means of the `AjaxDiagramView.MultipleSelection` property which is `true` by default.

- ◆ Area selection: to select all the graphic objects that intersect a given area, move the mouse pointer to an empty area of the view, press the left mouse button and drag the mouse to define the area while keeping the left mouse button pressed. When the selection area is as expected, release the mouse button. The selection interactor supports two kind of selection area, either via a rectangular area (the default mode) or via a freehand path, by means of the `SelectInteractor.AreaSelectionMode` property.
- ◆ Clear the selection: to deselect all the objects current selected, click an empty area of the view with the left mouse button.
- ◆ Deselect an object: to deselect one particular object from the current selection, press the CTRL modifier and click with the left mouse button the graphic object to deselect.

When a graphic object is selected, a semi-transparent filled rectangle is displayed on top of the selected object. The appearance of this rectangle can be customized by means of the `SelectionAnchorColor`, `SelectionColor`, `SelectionFillOn` and `SelectionStrokeWidth` properties of the `AjaxDiagramView` class.

#### *Moving a Graphic Object*

The selection interactor allows you to move one or several graphic objects in the view:

- ◆ To move one object: click the graphic object and drag the mouse to a new position.

- ◆ To move several objects: select the objects to move using one of the selection types described in *Select a Graphic Object*. Click the selected graphic objects and drag the mouse to a new position.

### **Resizing a Graphic Object**

The selection interactor allows you to resize a graphic object in the view. To resize a graphic object click the graphic object to select it, then drag a selection handle (each at one corner of the selection object) to adjust its size. The resizing capability is available provided it has been enabled on the view by means of the **AjaxDiagramView.ResizeSelection** property.

### **Changing the Connection Points of a Link**

The selection interactor allows you to change the connection points of a link. To change the selection points of a link click the link to select it, then drag a connection handle (one at each link extremity) over the destination node. When the mouse moves over a destination node, the available anchors are displayed. Move the dragged handle over the selected anchor and release the mouse button.

### **Deleting a Graphic Object**

The selection interactor allows you to delete one or several graphic objects in the view:

- ◆ To delete one object: click the graphic object and press the DELETE key.
- ◆ To delete several objects: select the objects to delete using one of the selection types described in *Select a Graphic Object* and press the DELETE key.

### **Providing Contextual Information to the Client**

The **SelectInteractor** class allows you to provide contextual information to the client asynchronously when a selection occurs. To implement such behavior, you have to implement the following instructions:

- ◆ Subscribe to the **QuerySelectionData** of the **AjaxDiagramView** class. This event is raised on the server to fetch information related to the current selection in order to post them back to the client. Data must be provided as a key-value pair where the key is the name of the property and the value is the value of the property. On the client, the dictionary is represented as a JavaScript object, each property in the dictionary being represented as a field of this object and accessible via the value of the property key.
- ◆ Define a JavaScript function in your ASPX that will receive the data of the **QuerySelectionData** event and set the **ClientSelectionCallback** property of the **AjaxDiagramView** control to the name of this function. This function will be invoked in response to a selection and the data is accessible from the **callbackArguments** field of the function parameter.

The following example shows how to handle the **QuerySelectionData** event to display the bounding box of the currently selected object.

The `Default.aspx` file:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs"
```

## Displaying Diagrams in an ASP.NET Application

```
Inherits="_Default" %>

<%@ Register Assembly="ILOG.Diagrammer.Web, Version=1.6.0.0, Culture=neutral,
PublicKeyToken=7906592bc7cc7340"
    Namespace="ILOG.Diagrammer.Web.UI" TagPrefix="cc1" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Untitled Page</title>
</head>
<body>
    <script type="text/javascript">
        function onSelection(args) {
            var msg = "";
            if (args.callbackArguments) {
                var bounds = args.callbackArguments.bounds;
                msg = "["+ bounds.x+", "+bounds.y+", "+bounds.w+", "+bounds.h+"]";
                msg = "Selected Object Bounds : " + msg;
            } else {
                msg = "No selection.";
            }
            document.getElementById("message").innerText = msg;
        }
    </script>
    <form id="form1" runat="server">
        <asp:ScriptManager ID="ScriptManager1" runat="server" />
        <div>
            <asp:UpdatePanel ID="UpdatePanel1" runat="server">
                <ContentTemplate>
                    <cc1:AjaxDiagramView ID="DiagramView1" runat="server"
ClientSelectionCallback="onSelection" Width="300px" Height="200px"
ContentSessionId="DiagramView1Contents"
OnQuerySelectionData="DiagramView1_QuerySelectionData"/>
                    <cc1>SelectInteractor ID="SelectInteractor1" runat="server"
TargetControlID="DiagramView1" />
                    <span id="message" />
                </ContentTemplate>
            </asp:UpdatePanel>
        </div>
    </form>
</body>
</html>
```

### The C# code-behind file:

```
using System;
using System.Web;
using System.Web.UI;
using ILOG.Diagrammer.Graphic;
using ILOG.Diagrammer;
using System.Drawing;

public partial class _Default : System.Web.UI.Page
{
    protected void Page_Init(object sender, EventArgs e)
    {
```

```

        if (!IsPostBack) {
            Group container = new Group();
            Ellipse ellipse = new Ellipse(0, 0, 100, 100);
            ellipse.Fill = new SolidFill(Color.Coral);
            container.Objects.Add(ellipse);
            Rect rect = new Rect(60, 80, 90, 50);
            rect.Fill = new SolidFill(Color.LawnGreen);
            container.Objects.Add(rect);
            DiagramView1.Content = container;
        }
    }
    protected void DiagramView1_QuerySelectionData(object sender,
        ILOG.Diagrammer.Web.UI.QuerySelectionDataEventArgs args) {
        GraphicObject obj = args.SelectedObject;
        args.Properties["bounds"] = obj.Bounds;
    }
}

```

The VB.NET code-behind file:

```

Imports System
Imports System.Web
Imports System.Web.UI
Imports ILOG.Diagrammer.Graphic
Imports ILOG.Diagrammer
Imports System.Drawing

Public Class _Default
    Inherits System.Web.UI.Page

    Protected Sub Page_Init(ByVal sender As Object, ByVal e As EventArgs)
        If Not IsPostBack Then
            Dim container As Group = New Group
            Dim ellipse As Ellipse = New Ellipse(0, 0, 100, 100)
            ellipse.Fill = New SolidFill(Color.Coral)
            container.Objects.Add(ellipse)
            Dim rect As Rect = New Rect(60, 80, 90, 50)
            rect.Fill = New SolidFill(Color.LawnGreen)
            container.Objects.Add(rect)
            DiagramView1.Content = container
        End If
    End Sub

    Protected Sub DiagramView1_QuerySelectionData(ByVal sender As Object, _
        ByVal args As ILOG.Diagrammer.Web.UI.QuerySelectionDataEventArgs)
        Dim obj As GraphicObject = args.SelectedObject
        args.Properties("bounds") = obj.Bounds
    End Sub
End Class

```

### Zoom Interactor

The zoom interactor is defined by the `ZoomInteractor` class and its JavaScript counterpart is the `ILOG.Diagrammer.Web.UI.ZoomBehavior` JavaScript class. It enables you to zoom in or zoom out a particular area of a view:

- ◆ **Zoom in:** to zoom in an area of a view, drag a rectangle corresponding to the area to zoom and release the mouse button to perform the zoom.

- ◆ **Zoom out:** to zoom out the view, press the Shift key and drag a rectangle corresponding to the area into which the current visible area should appear after the zooming-out operation.

It is possible to cancel the current operation pressing the ESC key while dragging the mouse to define the rectangular area to zoom.

### **Pan Interactor**

The interactor used to pan a view is defined by the `PanInteractor` class and its JavaScript counterpart is the **ILOG.Diagrammer.Web.UI.PanBehavior** JavaScript class.

To pan a view, press the left mouse button on the view and drag the mouse to display the new visible area.

It is possible to cancel the current operation by pressing the ESC key while dragging the mouse to define the new visible area.

### **Graphic Object Creation Interactor**

The interactor used to create graphic objects is defined by the `MakeObjectInteractor` class and its JavaScript counterpart is the **ILOG.Diagrammer.Web.UI.MakeObjectBehavior** JavaScript class.

This class allows you to create graphic objects by defining the object bounds on the client-side which are then post back to the server to effectively create the object.

The **MakeObjectInteractor** supports two different modes: the link mode and the graphic object mode via the `LinkMode` property. When this property is set to **true**, the interactor is configured to create links.

#### ***Creating links***

To create a link, proceed with the following steps:

- ◆ On the server, subscribe to the `MakeLink` event of the **MakeObjectInteractor**. This event is raised to query the creation of a new link via the `Link` property of the `MakeLinkEventArgs` event argument. Note that the link size and location are automatically initialized by the interactor and should not be done by the event handler.
- ◆ On the client, press the left mouse button on the starting point of the link (it can be an empty area or a graphic object) and drag the mouse to the ending point of the link (it can be an empty area or a graphic object) then release the mouse button.

By default, when a source or destination graphic object exists for the link, the interactor automatically connects the link. You can change this behavior by means of the `AutoConnect` property of the **MakeLinkEventArgs**. When this property is **false**, the link is not connected and the user is responsible for connecting it.

#### ***Creating objects***

To create graphic objects, proceed through the following steps:



- ◆ On the server, subscribe to the `MakeObject` event of the `MakeObjectInteractor` class. This event is raised to query the creation of a new object via the `GraphicObject` property of the `MakeObjectEventArgs` event argument. Note that the object size and location are automatically initialized by the interactor and should not be done by the event handler.
- ◆ On the client, define the bounds of the object to create pressing the left mouse button and drag the mouse to define a rectangle of the expected size then release the mouse button.

The following example shows how to handle the `MakeObject` and `MakeLink` events:

The `Default.aspx` file:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs"
Inherits="_Default" %>

<%@ Register Assembly="ILOG.Diagrammer.Web, Version=1.6.0.0, Culture=neutral,
PublicKeyToken=7906592bc7cc7340"
Namespace="ILOG.Diagrammer.Web.UI" TagPrefix="cc1" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
<title>Untitled Page</title>
</head>
<body>
<form id="form1" runat="server">
<asp:ScriptManager ID="ScriptManager1" runat="server" />
<div>
<asp:UpdatePanel ID="UpdatePanel1" runat="server">
<Triggers>
<asp:AsyncPostBackTrigger ControlID="MakeObjectInteractor1" />
</Triggers>
<ContentTemplate>
<cc1:AjaxDiagramView ID="DiagramView1" runat="server" Width="300px"
Height="200px" ContentSessionId="DiagramView1Contents" />
<cc1:MakeObjectInteractor ID="MakeObjectInteractor1" runat="server"
TargetControlID="DiagramView1" OnMakeLink="MakeObjectInteractor1_MakeLink"
OnMakeObject="MakeObjectInteractor1_MakeObject" />
<asp:CheckBox ID="CheckBox1" runat="server" AutoPostBack="True"
OnCheckedChanged="CheckBox1_CheckedChanged"
Text="Create Links" />
</ContentTemplate>
</asp:UpdatePanel>
</div>
</form>
</body>
</html>
```

The C# code-behind file:

```
using System;
using System.Web;
using System.Web.UI;
using ILOG.Diagrammer.Graphic;
using ILOG.Diagrammer;
using System.Drawing;
```

## Displaying Diagrams in an ASP.NET Application

```
public partial class _Default : System.Web.UI.Page
{
    protected void Page_Init(object sender, EventArgs e)
    {
        if (!IsPostBack) {
            Group container = new Group();
            Ellipse ellipse = new Ellipse(0, 0, 100, 100);
            ellipse.Fill = new SolidFill(Color.Coral);
            container.Objects.Add(ellipse);
            Rect rect = new Rect(60, 80, 90, 50);
            rect.Fill = new SolidFill(Color.LawnGreen);
            container.Objects.Add(rect);
            DiagramView1.Content = container;
        }
    }

    protected void MakeObjectInteractor1_MakeObject(object sender,
        ILOG.Diagrammer.Web.UI.MakeObjectEventArgs args) {
        Ellipse ellipse = new Ellipse(0, 0, 100, 100);
        ellipse.Fill = new SolidFill(Color.Coral);
        args.GraphicObject = ellipse;
    }

    protected void CheckBox1_CheckedChanged(object sender, EventArgs e) {
        MakeObjectInteractor1.LinkMode = CheckBox1.Checked;
    }

    protected void MakeObjectInteractor1_MakeLink(object sender,
        ILOG.Diagrammer.Web.UI.MakeLinkEventArgs args) {
        Link link = new Link();
        link.Stroke = new Stroke(Color.Black, 2f);
        args.Link = link;
    }
}
```

The VB.NET code-behind file:

```
Imports System
Imports System.Web
Imports System.Web.UI
Imports ILOG.Diagrammer.Graphic
Imports ILOG.Diagrammer
Imports System.Drawing

Public Class _Default
    Inherits System.Web.UI.Page

    Protected Sub Page_Init(ByVal sender As Object, ByVal e As EventArgs)
        If Not IsPostBack Then
            Dim container As Group = New Group
            Dim ellipse As Ellipse = New Ellipse(0, 0, 100, 100)
            ellipse.Fill = New SolidFill(Color.Coral)
            container.Objects.Add(ellipse)
            Dim rect As Rect = New Rect(60, 80, 90, 50)
            rect.Fill = New SolidFill(Color.LawnGreen)
            container.Objects.Add(rect)
            DiagramView1.Content = container
        End If
    End Sub
End Class
```

```

End Sub

Protected Sub MakeObjectInteractor1_MakeObject(ByVal sender As Object, _
    ByVal args As ILOG.Diagrammer.Web.UI.MakeObjectEventArgs)
    Dim ellipse As Ellipse = New Ellipse(0, 0, 100, 100)
    ellipse.Fill = New SolidFill(Color.Coral)
    args.GraphicObject = ellipse
End Sub

Protected Sub CheckBox1_CheckedChanged(ByVal sender As Object, _
    ByVal e As EventArgs)
    MakeObjectInteractor1.LinkMode = CheckBox1.Checked
End Sub

Protected Sub MakeObjectInteractor1_MakeLink(ByVal sender As Object, _
    ByVal args As ILOG.Diagrammer.Web.UI.MakeLinkEventArgs)
    Dim link As Link = New Link
    link.Stroke = New Stroke(Color.Black, 2F)
    args.Link = link
End Sub
End Class

```

---

### Creating a New AJAX-enabled View Interactor

The `AjaxDiagramView` Web control supports rich client-side user interactions by means of AJAX-enabled view interactors.

An interactor makes it possible to handle user's input in an **AjaxDiagramView**. When an interactor is set on a view, all input events received by the client-side view component (that is, all input events that occur in the browser on the view) are forwarded to the interactor which will then handle the events to accomplish its task.

This section explains how to create a new AJAX-enabled view interactor.

**Note:** *The following example extends and uses the ASP.NET 2.0 AJAX library and therefore assumes the knowledge of this technology is known. In particular, creating a new `ExtenderControl` is a prerequisite to understand this example.*

### Requirements

To write a new interactor, the first step is to define what the interactor should do when it is set on a view.

The example shows a basic interactor that displays contextual information in a span HTML element when the mouse moves over an object and changes its color when it is clicked. The information is fetched asynchronously.

The requirements to implement this interaction are:

- ◆ The information must be processed asynchronously.

- ◆ The new color must be processed synchronously since it requires an update of the image.
- ◆ The id of the span element must be specified on the server and known on the client.

### Creating the Interactor

To handle input events in a view, you have to create an interactor class. The base class for interactors is the `ViewInteractor` class which defines the basic functionalities and services to handle input events in an **AjaxDiagramView**.

In order to know the span element that displays the information, the interactor defines the **MsgElementID** property to store the element **Id**.

The following example shows the class definition as well as the **MsgElementID** property.

```
using System;
using System.Web;
using System.Web.UI;
using ILOG.Diagrammer.Web.UI;

public class MyInteractor : ViewInteractor
{
    private string _msgElementId;

    [IDReferenceProperty()]
    public string MsgElementID {
        get {
            return _msgElementId ?? string.Empty;
        }
        set {
            _msgElementId = value;
        }
    }
}

Imports System
Imports System.Web
Imports System.Web.UI
Imports ILOG.Diagrammer.Web.UI

Public Class MyInteractor
Inherits ViewInteractor
    Private _msgElementId As String

    <IDReferenceProperty()> _
    Public Property MsgElementID() As String
        Get
            If Not (_msgElementId Is Nothing) Then
                Return _msgElementId
            Else
                Return String.Empty
            End If
        End Get
        Set
            _msgElementId = value
        End Set
    End Property
End Class
```

## Inheriting the ViewInteractor Base Class

When subclassing the **ViewInteractor**, you must implement the following method:

- ◆ Create **CreateScriptBehaviorDescriptor** abstract method of the **ViewInteractor** class to return a **ScriptBehaviorDescriptor** instance that defines the instance of the client behavior type. In this example, the **ScriptBehaviorDescriptor** is initialized with the **MySample.MyInteractor** behavior type and the **ClientID** of the target control.

You may also have to override the following methods:

- ◆ **ConfigureScriptDescriptor** to configure the **ScriptBehaviorDescriptor** instance created in the method above to define the client behavior's properties. Client behavior's properties are declared to the script descriptor by means of the **AddProperty** method of the **ScriptBehaviorDescriptor** class. In this example, the **MsgElementID** property is added to the script descriptor to initialize the corresponding client behavior's property. Overriding this method is not required if you do not have any properties to add.
- ◆ **GetScriptReferences** to pass the location of the script library that defines the client behavior type, in this example `MyBehavior.js`, which is written later in this section.

```

        protected override ScriptBehaviorDescriptor
        CreateScriptBehaviorDescriptor(Control targetControl) {
            return new ScriptBehaviorDescriptor("MySample.MyBehavior",
            targetControl.ClientID);
        }

        protected override void ConfigureScriptDescriptor(ScriptBehaviorDescriptor
        descriptor) {
            base.ConfigureScriptDescriptor(descriptor);
            descriptor.AddProperty("messageElementId", MsgElementID);
        }

        protected override System.Collections.Generic.IEnumerable<ScriptReference>
        GetScriptReferences() {
            IEnumerable<ScriptReference> references = base.GetScriptReferences();
            List<ScriptReference> list = new List<ScriptReference>(references);
            list.Add(new ScriptReference(ResolveClientUrl("MyBehavior.js")));
            return list;
        }
    Protected Overloads Overrides Function CreateScriptBehaviorDescriptor(ByVal
    targetControl As Control) As ScriptBehaviorDescriptor
        Return New ScriptBehaviorDescriptor("MySample.MyBehavior",
        targetControl.ClientID)
    End Function

    Protected Overloads Overrides Sub ConfigureScriptDescriptor(ByVal descriptor As
    ScriptBehaviorDescriptor)
        MyBase.ConfigureScriptDescriptor(descriptor)
        descriptor.AddProperty("messageElementId", MsgElementID)
    End Sub

    Protected Overrides Function GetScriptReferences() As IEnumerable(Of
    ScriptReference)
        Dim references As IEnumerable(Of ScriptReference)
        references = MyBase.GetScriptReferences()
    
```

## Displaying Diagrams in an ASP.NET Application

```
Dim list As New List(Of ScriptReference) (references)
list.Add(New ScriptReference(ResolveClientUrl("FocusBehavior.js")))
Return list;
End Function
```

### Processing the Interaction on the Server

A view interactor can process the result of interactions in two ways depending on whether the client behavior sent an asynchronous callback or a synchronous postback.

In case of an asynchronous callback has been sent, the **ViewInteractor** class, which implements the **ICallbackEventHandler** interface, defines the **RaiseCallbackEvent** method. This method should be overridden by subclasses to implement the expected behavior.

In case of a synchronous postback has been sent, the **ViewInteractor** class, which implements the **IPostBackEventHandler** interface, defines the **RaisePostBackEvent** method. This method should be overridden by subclasses to implement the expected behavior.

In this example, the interactor overrides the **RaiseCallbackEventHandler** to get the contextual information of the clicked object and send it back to the server. Note that the data related to the client-side event is sent to the server as a JSON string and therefore requires to be deserialized first. Then a hit test is performed to get the clicked object. The result is then the object tooltip.

```
protected override void RaiseCallbackEvent(string eventArgument) {
    JavaScriptSerializer serializer = new JavaScriptSerializer();
    IDictionary param = serializer.DeserializeObject(eventArgument) as
IDictionary;
    if (param != null) {
        DiagramView view = View;
        int mouseX = (int)param["x"];
        int mouseY = (int)param["y"];
        GraphicObject obj = view.HitTestSelectable(new Point(mouseX, mouseY));
        if (obj != null) {
            _callbackResult = obj.ToolTip;
        }
    }
}
Protected Overloads Overrides Sub RaiseCallbackEvent(ByVal eventArgument As
String)
Dim serializer As JavaScriptSerializer = New JavaScriptSerializer
Dim param As IDictionary = CType(serializer.DeserializeObject(eventArgument),
IDictionary)
If Not (param Is Nothing) Then
    Dim view As DiagramView = View
    Dim mouseX As Integer = CType(param("x"), Integer)
    Dim mouseY As Integer = CType(param("y"), Integer)
    Dim obj As GraphicObject = view.HitTestSelectable(New Point(mouseX, mouseY))
    If Not (obj Is Nothing) Then
        _callbackResult = obj.ToolTip
    End If
End If
End Sub
```

It also overrides the **RaisePostBackEventHandler** method to change the **Fill** property of the clicked object. Similarly to the **RaiseCallbackEventHandler** method, the **RaisePostBackEventHandler** receives the client-side event data as a JSON string and therefore requires it is deserialized first. Then a hit test is performed to get the clicked object and its **Fill** property is changed.

```
protected override void RaisePostBackEvent(string eventArgument) {
    JavaScriptSerializer serializer = new JavaScriptSerializer();
    IDictionary param = serializer.DeserializeObject(eventArgument) as
    IDictionary;
    if (param != null) {
        DiagramView view = View;
        int mouseX = (int)param["x"];
        int mouseY = (int)param["y"];
        Shape obj = view.HitTestSelectable(new Point(mouseX, mouseY)) as Shape;
        if (obj != null) {
            SolidFill fill = (SolidFill)obj.Fill;
            Color c = fill.Color;
            Color newColor = Color.FromArgb((c.R + 25) % 255, (c.G + 25) % 255,
            (c.B + 25) % 255);
            fill.Color = newColor;
        }
    }
}
Protected Overloads Overrides Sub RaisePostBackEvent(ByVal eventArgument As
String)
    Dim serializer As JavaScriptSerializer = New JavaScriptSerializer
    Dim param As IDictionary = CType(serializer.DeserializeObject(eventArgument),
IDictionary)
    If Not (param Is Nothing) Then
        Dim view As DiagramView = View
        Dim mouseX As Integer = CType(param("x"), Integer)
        Dim mouseY As Integer = CType(param("y"), Integer)
        Dim obj As Shape = TryCast(view.HitTestSelectable(
            New Point(mouseX, mouseY)), Shape)
        If Not (obj Is Nothing) Then
            Dim fill As SolidFill = CType(obj.Fill, SolidFill)
            Dim c As Color = fill.Color
            Dim newColor As Color = Color.FromArgb((c.R + 25) Mod 255,
            (c.G + 25) Mod 255,
            (c.B + 25) Mod 255)
            fill.Color = newColor
        End If
    End If
End Sub
```

## Creating the Client-side Behavior

The client behavior of a view interactor is defined by the **ILOG.Diagrammer.Web.UI.ViewBehavior** JavaScript class. This class defines the basic functionalities required to handle input events in an **AjaxDiagramView** and to communicate with the server-side control.

The client behavior in this example therefore inherits from **ViewBehavior** and defines the **messageElementId** property that corresponds to the **MsgElementID** property of the **MyInteractor** class.

Note that since the interactor `GetScriptReferences` method specifies the `MyBehavior.js` file as the JavaScript file that contains the client behavior, all the JavaScript code below should be in a `MyBehavior.js` file.

The following example shows the client behavior class definition:

```
Type.registerNamespace('MySample');

MySample.MyBehavior = function(element) {
    MySample.MyBehavior.initializeBase(this, [element]);
    this._messageElementId = null;
    this._timer = null;
}

MySample.MyBehavior.prototype = {

    initialize : function() {
        MySample.MyBehavior.callBaseMethod(this, 'initialize');
    },

    dispose : function() {
        MySample.MyBehavior.callBaseMethod(this, 'dispose');
    },

    get_messageElementId : function() {
        return this._messageElementId;
    },

    set_messageElementId : function(value) {
        this._messageElementId = value;
    }
}

MySample.MyBehavior.registerClass('MySample.MyBehavior',
    ILOG.Diagrammer.Web.UI.ViewBehavior);
```

### Handling Input Events on the Client

Handling input events in an **AjaxDiagramView** is done by listening the events of the browser DOM. In this example, the interactor needs to listen to the `mousemove` and `mousedown` events and therefore defines the `onMouseDown` and `onMouseMove` event handlers and associates them with the corresponding events. The new implementation of the `initialize()` and `dispose()` methods are:

```
    initialize : function() {
        MySample.MyBehavior.callBaseMethod(this, 'initialize');
        $addHandlers(this.get_element(), {'mousedown': this.onMouseDown,
                                          'mousemove': this.onMouseMove},
this);
    },

    dispose : function() {
```



```

        $clearHandlers(this.get_element());
        MySample.MyBehavior.callBaseMethod(this, 'dispose');
    },

```

### Initiating a Postback From the Client Behavior

The **ViewBehavior** JavaScript class enables to initiate postback from an interactor by means of the `doPostBack()` method. This method initiates a postback passing optional parameters to the server view interactor control.

In this example, the interactor initiates a postback on the mousedown event to change the color of the clicked object. The following example shows the code of the `onMouseDown` method:

```

onMouseDown : function(e) {
    if (e.button == Sys.UI.MouseButton.leftButton) {
        var p = { 'x':e.offsetX, 'y':e.offsetY };
        this.doPostBack(p);
        e.preventDefault();
    }
},

```

### Initiating a Callback From the Client Behavior

In addition to the postback facility, the **ViewBehavior** class enables to initiate asynchronous callback from an interactor by means of the `callServerAsync()` method. This methods initiates an asynchronous communication between the client behavior and the server view interactor control. The response of the server control is handled in the `onCallbackReceived` method which must be overridden in the subclass to process the received information.

In this example, the interactor initiates an asynchronous callback when the mouse moves to get information on the possible hovered object. In order to reduce the number of requests, it uses a timer that is reset as soon as the mouse moves again. The following example shows the code of the `onMouseMove` and `onCallbackReceived` methods:

```

onMouseMove : function(e) {
    var p = { 'x':e.offsetX, 'y':e.offsetY };
    this.clearTimer();
    var inter = this;
    this._timer = setTimeout(function() {
        inter.callServerAsync(p);
        inter.clearTimer();
    }, 250);
    e.preventDefault();
},

onCallbackReceived : function(args) {
    var elt = document.getElementById(this.get_messageElementId());
    if (elt)
        elt.innerText = args;
},

```

## Displaying Diagrams in an ASP.NET Application

Here is the complete source code of the client behavior:

```
// JScript File
Type.registerNamespace('MySample');

MySample.MyBehavior = function(element) {
    MySample.MyBehavior.initializeBase(this, [element]);
    this._messageElementId = null;
    this._timer = null;
}

MySample.MyBehavior.prototype = {

    initialize : function() {
        MySample.MyBehavior.callBaseMethod(this, 'initialize');
        $addHandlers(this.get_element(), {'mousedown': this.onMouseDown,
                                          'mousemove' : this.onMouseMove},
this);
    },

    dispose : function() {
        $clearHandlers(this.get_element());
        MySample.MyBehavior.callBaseMethod(this, 'dispose');
    },

    get_messageElementId : function() {
        return this._messageElementId;
    },

    set_messageElementId : function(value) {
        this._messageElementId = value;
    },

    onMouseDown : function(e) {
        if (e.button == Sys.UI.MouseButton.leftButton) {
            var p = {'x':e.offsetX, 'y':e.offsetY };
            this.doPostBack(p);
            e.preventDefault();
        }
    },

    onMouseMove : function(e) {
        var p = {'x':e.offsetX, 'y':e.offsetY };
        this.clearTimer();
        var inter = this;
        this._timer = setTimeout(function() {
            inter.callServerAsync(p);
            inter.clearTimer();
        }, 250);
        e.preventDefault();
    },

    onCallbackReceived : function(args) {
        var elt = document.getElementById(this.get_messageElementId());
        if (elt)
            elt.innerText = args;
    },
};
```

```
clearTimer : function() {  
    if (this._timer) {  
        clearTimeout(this._timer);  
        this._timer = null;  
    }  
}  
}  
  
MySample.MyBehavior.registerClass('MySample.MyBehavior',  
ILOG.Diagrammer.Web.UI.ViewBehavior);
```



# *Using Predefined Graphic Objects*

This section presents the predefined graphic objects available in IBM® ILOG® Diagram for .NET. A complete set of graphic objects is provided, from basic shapes such as rectangles, ellipses to complex objects such as predefined gauges or charts. This section describes also the various graphic objects that are container objects (graphic objects containing other objects). By combining those predefined graphic objects you will be able to create new graphic objects that you can reuse through your application.

## **In This Section**

### *Basic Shapes*

Explains the ready-to-use graphic objects that represent basic shapes such as rectangle, ellipse, polyline, and so on.

### *Paths*

Introduces the Path object, a graphic object for displaying any type of shape.

### *Images*

Introduces the Image class.

### *Text Objects*

Introduces graphic objects that display text.

### *Link Objects*

Introduces the Link class, a graphic object for creating links between nodes.

## Using Predefined Graphic Objects

### *Controls*

Introduces the Control class.

### *Panels*

Introduces the Panel class and its subclasses.

### *Subdiagrams*

Introduces the SubDiagram class.

### *Graphic Symbols*

Introduces the GraphicSymbol class.

### *Scale Objects*

Introduces the CircularScale and LinearScale classes.

### *Gauges*

Introduces predefined graphic objects for dashboard displays.

### *User Symbols*

Introduces the UserSymbol class, a composite graphic object that can be created using Visual Studio .NET.

## **Related Sections**

### *Displaying Text in a Diagram*

Describes how to display basic or complex styled text inside a graphic object.

### *Building Diagrams and User Symbols Inside Visual Studio*

Introduces the Visual Studio .NET® Diagram Designer.

### *Creating a Simple Diagram with Nodes and Links Programmatically*

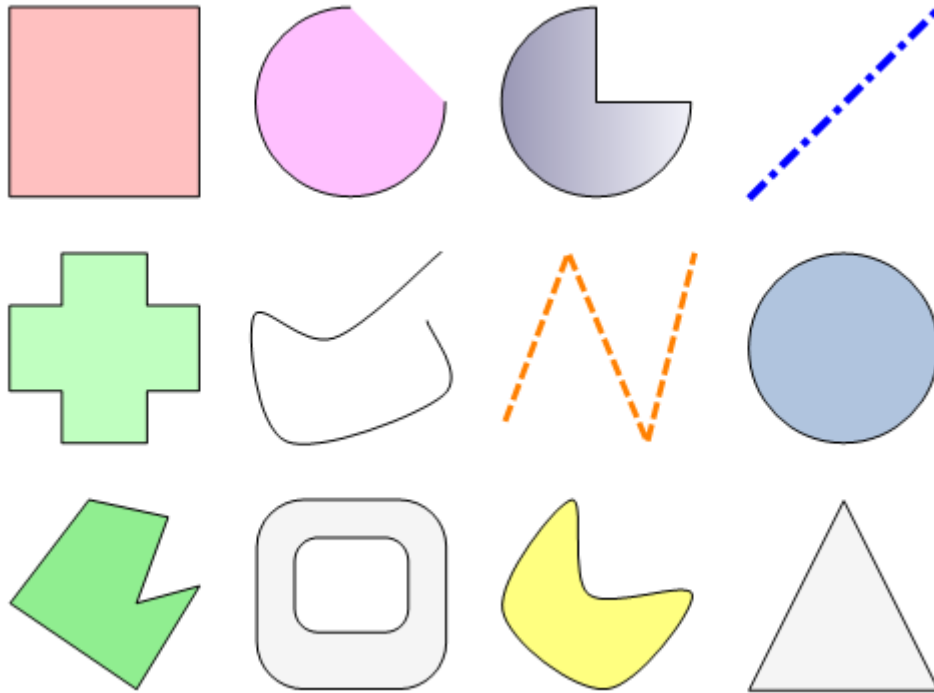
Explains the code needed to create a simple diagram.

---

## **Basic Shapes**

IBM® ILOG® Diagram for .NET provides a number of ready-to-use graphic objects that represent basic shapes. The following shapes are available: Circle, Ellipse, Rect, Arc, Pie, Line, Polyline, Polygon, BezierCurve, Curve, ClosedCurve, Basic2DShape and Path.

The following illustration shows some of the basic shapes:



All the basic shapes inherit from the **Shape** class and share the following common properties:

- ◆ **Fill**: describes how the interior of the shape is painted.
- ◆ **Stroke**: defines how the outline of the shape is painted.

The **Rect** class displays a rectangle. The following example creates a **Rect** object and shows how to specify the **Fill** and **Stroke** properties:

```
Rect rectangle = new Rect(0, 0, 100, 100);
rectangle.Radius = new Size2D(10, 10);
rectangle.Fill = new SolidFill(Color.LightBlue);
rectangle.Stroke = new Stroke(Color.Black, 2);
Dim rectangle As Rect = New Rect(0, 0, 100, 100)
rectangle.Radius = New Size2D(10, 10)
rectangle.Fill = New SolidFill(Color.LightBlue)
rectangle.Stroke = New Stroke(Color.Black, 2)
```

The following image shows the rendered rectangle:



To learn more about the **Fill** and **Stroke** properties of a basic shape, see *Filling and Stroking Graphic Objects*.

Like other graphic objects, the basic shapes share a number of properties that are common to all graphic objects, like displaying text inside the graphic object or specifying opacity for the object. To learn more about those properties see *Common Graphic Objects Rendering Features*.

---

### Basic Shapes With the Geometry Defined by a Rectangle

Some of the basic shapes such as the Rect, Ellipse, Pie, Arc and Basic2DShape objects have their geometry defined by a rectangle.

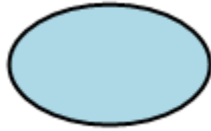
They all inherit from the same abstract base class, BoundedShape, and they share the Rectangle property that represents the rectangle that defines their bounds. For example, for an ellipse object, the **Rectangle** property defines the area where the ellipse is painted.

The following example shows how to create an **Ellipse** object and how to specify the **Rectangle** property.

```
Ellipse ellipse = new Ellipse();
ellipse.Rectangle = new Rectangle2D(10, 10, 100, 60);
ellipse.Fill = new SolidFill(Color.LightBlue);
ellipse.Stroke = new Stroke(Color.Black, 2);
Dim ellipse As Ellipse = New Ellipse
ellipse.Rectangle = New Rectangle2D(10, 10, 100, 60)
ellipse.Fill = New SolidFill(Color.LightBlue)
ellipse.Stroke = New Stroke(Color.Black, 2)
```



The following image shows the rendered ellipse:




---

### Basic Shapes With the Geometry Defined by a Set of Points

Some of the basic shapes such as Polyline, Polygon, BezierCurve, Curve or ClosedCurve have their geometry defined by a set of points. They all inherit from the same abstract base class, PolyPoints, and share the following properties:

- ◆ **Points**: indicates the set of points that define the geometry of the shape.
- ◆ **IsClosed**: indicates if the shape is closed or not.
- ◆ **CanEditPoints**: indicates if the points can be modified by a mouse interaction.

The following example shows how to create a **Polyline** object and how to specify the **Points** property.

```
Polyline polyline = new Polyline();
polyline.Points.AddRange(new Point2D[] {
    new Point2D(0,0),
    new Point2D(100, 10),
    new Point2D(200,0),
    new Point2D(300,10)
});
polyline.Stroke = new Stroke(Color.Black, 2, DashStyle.Dash);
Dim polyline As Polyline = New Polyline
polyline.Points.AddRange(New Point2D() { _
    New Point2D(0, 0), _
    New Point2D(100, 10), _
    New Point2D(200, 0), _
    New Point2D(300, 10)})
polyline.Stroke = New Stroke(Color.Black, 2, DashStyle.Dash)
```

The following image shows the rendered polyline:



---

### The Path Object

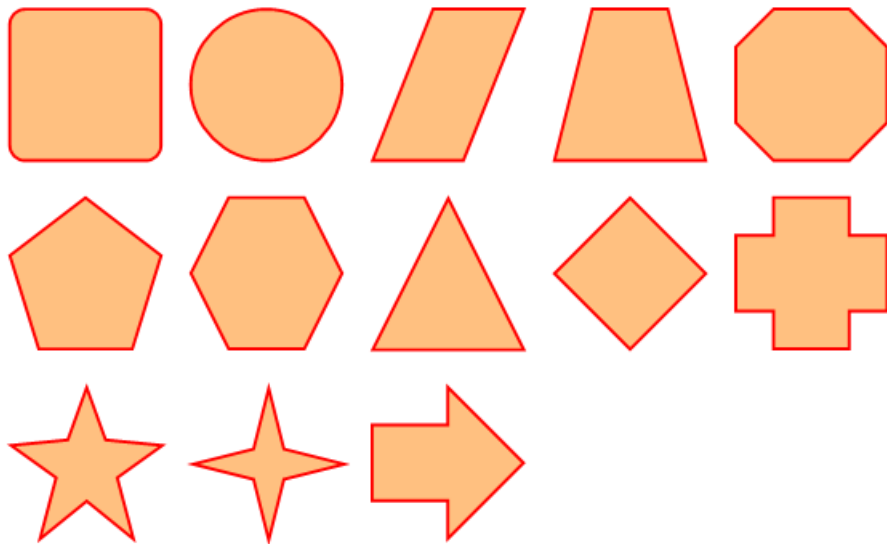
The Path object is a basic shape that enables you to draw complex curves and shapes. To learn more about the **Path** object see *The Path Object*.

---

### The Basic2DShape Object

The Basic2DShape object is a convenient graphic object that can display a predefined set of shapes. You specify the shape of the object through the ShapeType property. You can choose among different shapes: rectangle, ellipse, pentagon, arrow, star, and so on. The ControlValue property is a floating point value that allows you to modify the shape depending on the shape type. For example, for a cross it allows you to control the thickness of the cross.

The following illustration shows the possible shapes of the **Basic2DShape** class:



---

## Paths

The Path object is a graphic object that allows you to display any kind of shape.

The geometry of the **Path** object is defined by the PathData object. The **PathData** contains a collection of segments; each segment in this collection represents a segment of the path.

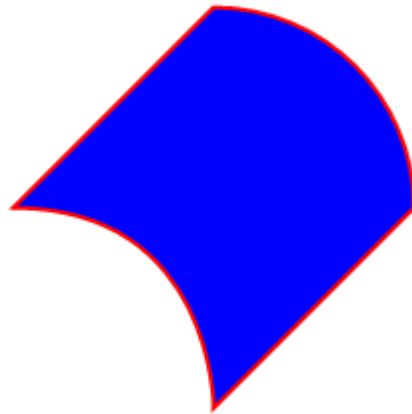
Segments can be lines, arcs, bezier or quadratic bezier segments. It is also possible to create compound paths (a path with multiple subpaths) to obtain effects such as "donut holes".

The following example shows how to create a **Path** object.

```
Path path = new Path();
path.Fill = new SolidFill(Color.Blue);
path.Stroke = new Stroke(Color.Red, 2);

path.Data.AddSegment(new StartSegment(110, 20));
path.Data.AddSegment(new ArcSegment(new Point2D(210,120),
    new Size2D(100,100), 90, false, true));
path.Data.AddSegment(new LineSegment(110, 220));
path.Data.AddSegment(new BezierSegment(110, 220, 110, 120, 10, 120));
path.Data.AddSegment(new CloseSegment());
Dim path As Path = New Path
path.Fill = New SolidFill(Color.Blue)
path.Stroke = New Stroke(Color.Red, 2)
path.Data.AddSegment(New StartSegment(110, 20))
path.Data.AddSegment(New ArcSegment(New Point2D(210, 120),
    New Size2D(100, 100), 90, False, True))
path.Data.AddSegment(New LineSegment(110, 220))
path.Data.AddSegment(New BezierSegment(110, 220, 110, 120, 10, 120))
path.Data.AddSegment(New CloseSegment)
```

The **Path** object you have just created should look like the following one.



## Using Predefined Graphic Objects

The segments that can be specified in the **PathData** objects are subclasses of the **PathSegment** class:

Class	Properties	Description
StartSegment	Point (Point2D)	Starts a new subpath at the given point coordinate specified by the <b>Point</b> property.
LineSegment	Point (Point2D)	Draws a line from the current point to the coordinate specified by the <b>Point</b> property.
CloseSegment		Closes the current subpath by drawing a straight line from the current point to the initial point of the current subpath.
BezierSegment	Point1(Point2D) Point2(Point2D) Point3(Point2D)	Draws a cubic Bézier curve from the current point to point <b>Point3</b> using <b>Point1</b> as the control point at the beginning of the curve and <b>Point2</b> as the control point at the end of the curve.
QuadraticBezierSegment	Point1(Point2D) Point2 (Point2D)	Draws a quadratic Bézier curve from the current point to point <b>Point2</b> using <b>Point1</b> as the control point.
ArcSegment	Point (Point2D), Size(Point2D), XRotation(double) LargeArc (bool) SweepFlag(bool)	Draws an elliptical arc from the current point to the point specified by the <b>Point</b> property. The size and orientation of the ellipse are defined by two radii ( <b>Size</b> property) and an <b>XRotation</b> , which indicates how the ellipse as a whole is rotated with respect to the current coordinate system.

The geometry of the **Path** object can also be specified through a formatted string that specifies the segments. The format of this string is the same format specified by the Scalable Vector Graphics (SVG) "path data" format. This format is a set of commands followed by parameters.

The commands are the following:

Command	Name	Parameters	Description
<b>M</b> (absolute) <b>m</b> (relative)	moveto	(x y)+	Starts a new subpath at the given (x,y) coordinate. <b>M</b> (uppercase) indicates that absolute coordinates will follow; <b>m</b> (lowercase) indicates that relative coordinates will follow. If a relative <b>moveto</b> ( <b>m</b> ) appears as the first element of the path, then it is treated as a pair of absolute coordinates. If a <b>moveto</b> is followed by multiple pairs of coordinates, the subsequent pairs are treated as implicit <b>lineto</b> commands.
<b>Z</b> or <b>z</b>	closepath	none	Closes the current subpath by drawing a straight line from the current point to the initial point of the current subpath.
<b>L</b> (absolute) <b>l</b> (relative)	lineto	(x y)+	Draws a line from the current point to the given (x,y) coordinate which becomes the new current point. <b>L</b> (uppercase) indicates that absolute coordinates will follow; <b>l</b> (lowercase) indicates that relative coordinates will follow. A number of coordinates pairs may be specified to draw a polyline. At the end of the command, the new current point is set to the final set of the provided coordinates.
<b>H</b> (absolute) <b>h</b> (relative)	horizontal lineto	x+	Draws a horizontal line from the current point (cpx, cpy) to (x, cpy). <b>H</b> (uppercase) indicates that absolute coordinates will follow; <b>h</b> (lowercase) indicates that relative coordinates will follow. Multiple x values can be provided (although usually this does not make sense). At the end of the command, the new current point becomes (x, cpy) for the final value of x.
<b>V</b> (absolute) <b>v</b> (relative)	vertical lineto	y+	Draws a vertical line from the current point (cpx, cpy) to (cpx, y). <b>V</b> (uppercase) indicates that absolute coordinates will follow; <b>v</b> (lowercase) indicates that relative coordinates will follow. Multiple y values can be provided (although usually this does not make sense). At the end of the command, the new current point becomes (cpx, y) for the final value of y.

Command	Name	Parameters	Description
<b>C</b> (absolute) <b>c</b> (relative)	curveto	(x1 y1 x2 y2 x y)+	Draws a cubic Bézier curve from the current point to (x,y) using (x1,y1) as the control point at the beginning of the curve and (x2,y2) as the control point at the end of the curve. <b>C</b> (uppercase) indicates that absolute coordinates will follow; <b>c</b> (lowercase) indicates that relative coordinates will follow. Multiple sets of coordinates may be specified to draw a polybézier. At the end of the command, the new current point becomes the final (x,y) coordinate pair used in the polybézier.
<b>S</b> (absolute) <b>s</b> (relative)	shorthand/ smooth curveto	(x2 y2 x y)+	Draws a cubic Bézier curve from the current point to (x,y). The first control point is assumed to be the reflection of the second control point on the previous command relative to the current point. (If there is no previous command or if the previous command was not a <b>C</b> , <b>c</b> , <b>S</b> or <b>s</b> , it is assumed that the first control point is coincident with the current point.) (x2,y2) is the second control point (that is, the control point at the end of the curve). <b>S</b> (uppercase) indicates that absolute coordinates will follow; <b>s</b> (lowercase) indicates that relative coordinates will follow. Multiple sets of coordinates may be specified to draw a polybézier. At the end of the command, the new current point becomes the final (x,y) coordinate pair used in the polybézier.
<b>Q</b> (absolute) <b>q</b> (relative)	quadraticBézier curvetoxxx	(x1 y1 x y)+	Draws a quadratic Bézier curve from the current point to (x,y) using (x1,y1) as the control point. <b>Q</b> (uppercase) indicates that absolute coordinates will follow; <b>q</b> (lowercase) indicates that relative coordinates will follow. Multiple sets of coordinates may be specified to draw a polybézier. At the end of the command, the new current point becomes the final (x,y) coordinate pair used in the polybézier.

Command	Name	Parameters	Description
<b>T</b> (absolute) <b>t</b> (relative)	Shorthand/ smooth quadratic Bézier curveto	(x y)+	Draws a quadratic Bézier curve from the current point to (x,y). The control point is assumed to be the reflection of the control point on the previous command relative to the current point. (If there is no previous command or if the previous command was not a <b>Q</b> , <b>q</b> , <b>T</b> or <b>t</b> , it is assumed that the control point is coincident with the current point.) <b>T</b> (uppercase) indicates that absolute coordinates will follow; <b>t</b> (lowercase) indicates that relative coordinates will follow. At the end of the command, the new current point becomes the final (x,y) coordinate pair used in the polybézier.
<b>A</b> (absolute) <b>a</b> (relative)	elliptical arc	(rx ry x-axis- rotation large-arc-flag sweep-flag x y)+	Draws an elliptical arc from the current point to (x, y). The size and orientation of the ellipse are defined by two radii (rx, ry) and an x-axis-rotation, which indicates how the ellipse as a whole is rotated with respect to the current coordinate system. The center (cx, cy) of the ellipse is calculated automatically to satisfy the constraints imposed by other parameters. <b>large-arc-flag</b> and <b>sweepflag</b> contribute to the automatic calculations and help determine how the arc is drawn.

The path we have created in the previous example can also be created through the following code:

```
Path path = new Path();
path.Fill = new SolidFill(Color.Blue);
path.Stroke = new Stroke(Color.Red, 2);
path.Data.SetGeometry("M110 20A100 100 90 0 1 210 120L110 220C110 220 110 120
10 120z");
Dim path As Path = New Path
path.Fill = New SolidFill(Color.Blue)
path.Stroke = New Stroke(Color.Red, 2)
path.Data.SetGeometry("M110 20A100 100 90 0 1 210 120L110 220C110 220 110 120
10 120z")
```

---

## Images

The Image object is a graphic object that can display various types of images. The object can display raster images such as GIF, JPEG, PNG, BMP and all the bitmap formats supported by the .NET Framework®.

It can also display vector graphics that can be specified by a graphic object or an SVG or IVN file.

## Using Predefined Graphic Objects

To display a raster image use the `SystemImage` property. This is the property that specified the .NET Framework® **Image** to display.

To display vector graphics use the `GraphicObject` property.

The content displayed by the **Image** object can also be specified by the `ImageUri` property that represents a URI to the raster or vector content.

The area where the image is displayed is specified by the `Rectangle` property. Within this rectangle the image may be scaled with various fitting options specified by the `KeepAspectRatio` and `Slice` properties.

The following example shows how to create an **Image** object:

```
Image img = new Image();
img.Rectangle = new Rectangle2D(0, 0, 100, 100);
img.KeepAspectRatio = AspectRatioAlignment.XMidYMid;
img.ImageUri = new Uri("http://www.ibm.com/logo.gif");
Dim img As Image = New Image
img.Rectangle = New Rectangle2D(0, 0, 100, 100)
img.KeepAspectRatio = AspectRatioAlignment.XMidYMid
img.ImageUri = New Uri("http://www.ibm.com/logo.gif")
```

---

## Text Objects

IBM® ILOG® Diagram for .NET allows you to display text in a diagram in different ways.

All graphic objects can display text by means of the `Text` property of the `GraphicObject` class. Through the `TextAppearance` property you can control the appearance of the text within the graphic object in terms of color, font and alignment.

To display text, you can also use the following classes:

- ◆ `Text`: to display a styled text aligned around an anchor point.
- ◆ `TextOnPath`: to display a text along a path.

The following illustration shows **Text** and **TextOnPath** objects.





## Diagrammer for .NET

For more information on displaying text in a diagram see *Displaying Text in a Diagram*.

---

## Link Objects

A **Link** is a graphic object used to draw a connection between two other graphic objects. Links are useful to build graphs like flow chart diagrams or business process diagrams.

The **Link** class is a subclass of PolyPoints. The link looks like a sequence of straight or curved segments between two objects.

### In This Section

#### *Specifying the Connection Points*

Explains how to connect a **Link**.

#### *Specifying the Link Shape*

Explains how to customize the **Link** shape.

### *Customizing the Arrows*

Explains how to customize the start and end arrows of a **Link**.

### *Customizing the Link Appearance*

Explains how to customize the **Link** appearance.

### *Adding Text*

Explains how to add text on a **Link**.

### *Link Crossing*

Explains how to enable link crossings.

## **Related Sections**

### *Creating a Simple Diagram with Nodes and Links Programmatically*

Explains the code needed to create a simple diagram.

### *Introducing Link and Anchor Classes*

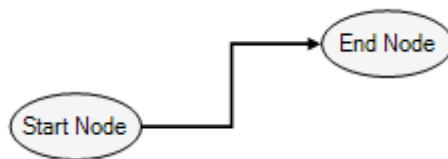
Explains the basic principles of the classes that are used to create graphs.

---

## **Specifying the Connection Points**

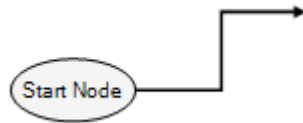
To connect a link between two graphic objects use the `StartAnchor` and `EndAnchor` properties. The values of these properties are `Anchor` objects contained in the collection stored in the `Anchors` property of the start or end object. When a link is connected to a graphic object, the start or end point is determined by the anchor. If the graphic object is moved or resized, the connection point will be recomputed automatically.

The following illustration shows an example of a link connected to graphic objects at both extremities.



A link can also be disconnected (that is, not connected to any graphic object) at one or both extremities. In that case, the start or end point is determined by the `StartPoint` or `EndPoint` property, respectively.

The following illustration shows an example of a link whose end is not connected.

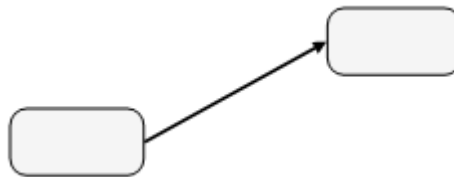


---

### Specifying the Link Shape

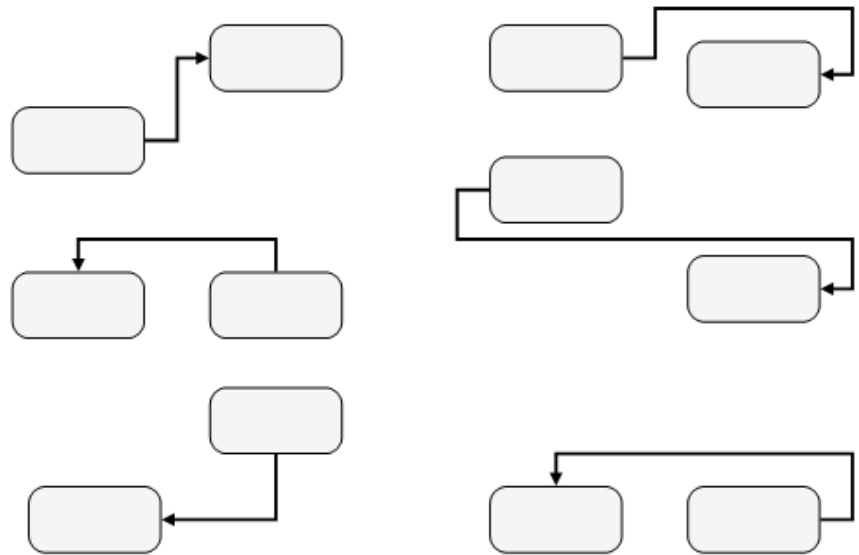
The **LinkShape** property controls the shape of the link. The type of this property is the `LinkShapeType` enumeration, and these are the possible values:

- ◆ **Straight:** the link is a straight line between the start point and the end point.

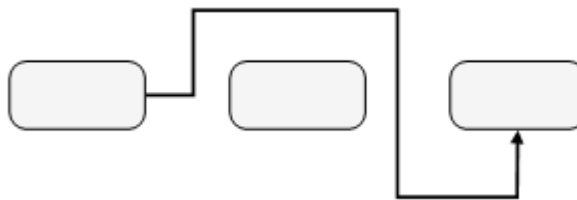


- ◆ **Orthogonal:** the link shape is computed automatically to a sequence of vertical or horizontal segments. There can be 2 to 5 segments, depending on the connection points and the bounds of the graphic objects to which the link is connected.

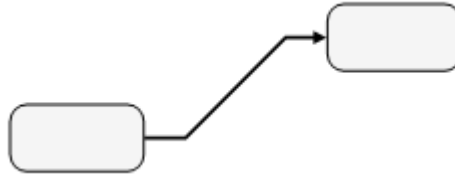
The following illustration shows the possible shapes of an orthogonal link.



*Note: When **LinkShape** is **Orthogonal** and if the **CanEditOrthogonalShape** property is **true**, it is possible to modify the link shape interactively, while keeping the link orthogonal, as illustrated in the following picture.*



- ◆ **Oblique:** the link shape is a sequence of one horizontal/vertical segment, one 45-degree segment, and another vertical/horizontal segment.



- ◆ **Free**: the link shape is determined by the Points property, as for a **PolyPoints** object.



The shape of the link can be further customized using the Radius, Curved and CurveTension properties.

When the **Radius** property is set to a positive value, the bends of the link are replaced by arcs of circle of the specified radius. The following illustration shows an example of an orthogonal link with a radius of 10.

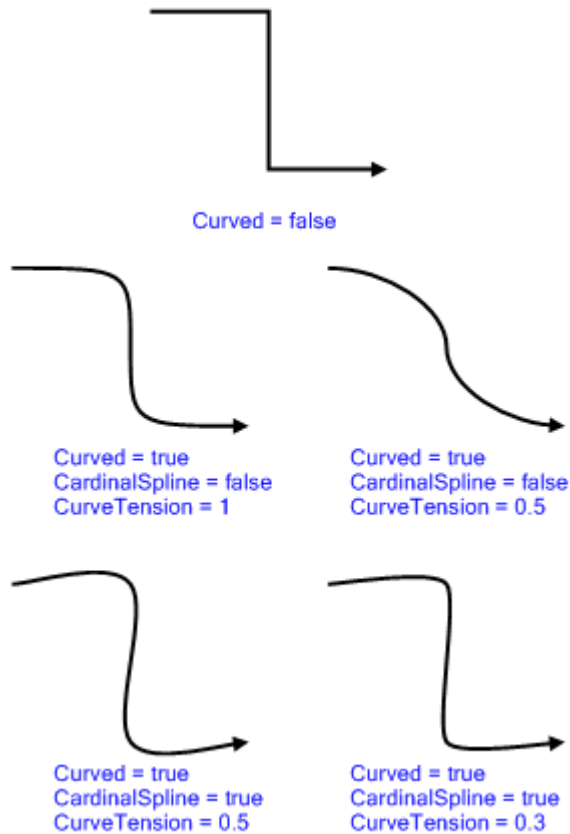


The **Curved**, CardinalSpline and **CurveTension** properties can be used to create curved links. If **Curved** is **true**, the segments of the link are replaced by a Bezier spline.

If **CardinalSpline** is **true**, the curve is a cardinal spline that goes through all the points of the link. If **CardinalSpline** is **false**, the curve is a Bezier spline that starts at the start point of the link, then goes through the middle points of the link segments, and finally ends at the end point of the link.

The **CurveTension** property can be used to adjust the shape of the curve by modifying the control points of the Bezier spline.

The following illustration shows examples of curved links.



---

## Customizing the Arrows

By default, a link has one arrow at its end.

You can customize the start or end arrows using the `StartArrow` and `EndArrow` properties. The values of these properties are instances of the `LinkArrow` abstract class, which has two predefined subclass, `CapArrow` and `ShapeArrow`.

### Cap Arrows

The **CapArrow** class is used to draw standard filled or open arrows. Since these arrows are drawn directly by the .NET Framework®, this is the most efficient choice when you do not need a special arrow shape.

In the following illustration the link has its **StartArrow** property set to a **CapArrow** object whose **Filled** property is **false**, and its **EndArrow** property set to a **CapArrow** object whose **Filled** property is **true**.

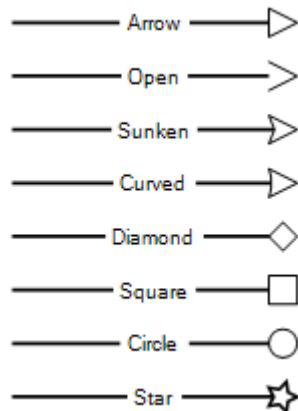


The size of the arrow can be changed using the **Size** property of the **CapArrow** object.

### Shape Arrows

The **ShapeArrow** class can be used when you need other arrow shapes than the standard filled and open arrows. The **ShapeArrow** class draws a **Shape** graphic object (or one of its subclasses: **Rect**, **Ellipse**, and so on) as the arrow of a link. In most cases, the shape of the arrow is specified through the **Shape** property of the **ShapeArrow** object.

The example below shows the predefined arrow shapes.



It is possible to define custom arrow shapes by setting the **CustomShape** property of the **ShapeArrow** to any **Shape** graphic object.

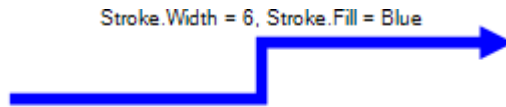
The size, color and stroke of the arrow can be changed using the **Size**, **Fill** and **Stroke** properties of the **ShapeArrow** object.

---

### Customizing the Link Appearance

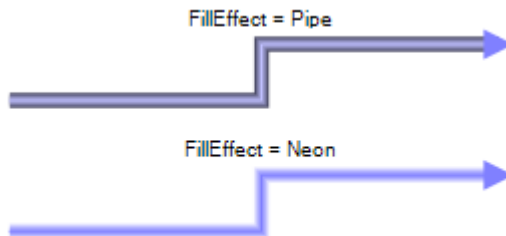
As the **Link** class is a subclass of **Shape**, the **Stroke** property can be used to customize the way the link is drawn. For example, you can use the **Fill** property of the stroke to change the color of the link and the **Width** property to modify the thickness of the link.

## Using Predefined Graphic Objects

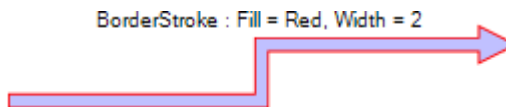


Additional customizations can be applied by using the **FillEffect**, **BorderStroke** and **BevelBorder** properties.

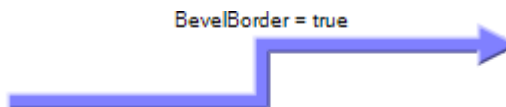
The **FillEffect** property defines a graphic effect. The value is a **LinkFillEffect** enumeration and the following illustration shows the possible values.



The **BorderStroke** property specifies a **Stroke** object that is used to draw a border around the link path.



The **BevelBorder** property specifies a 3D "bevel" border.



---

### Adding Text

Links can have a number of text items associated with them. The text items are defined by objects of type **LinkTextItem** and are added or removed using the collection contained in the **TextItems** property.



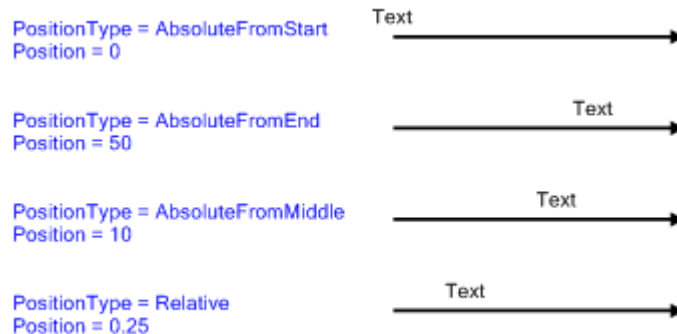
The text of each text item is set using the Text property of the **LinkTextItem** object.

**Note:** You can use the **Text** property of the link to set or get the text of the first text item.

### Text Position

The position of the text item along the path of the link is defined by the Position and PositionType properties.

- ◆ If **PositionType** is **Relative**, the **Position** property is interpreted as a relative position along the path. For example 0.5 means the middle of the link, 0.33 means a third of the link length, and so on.
- ◆ If **PositionType** is **AbsoluteFromStart**, **AbsoluteFromEnd** or **AbsoluteFromMiddle**, the **Position** property is interpreted as a fixed offset (in pixels) from the start, the end or the middle of the link.

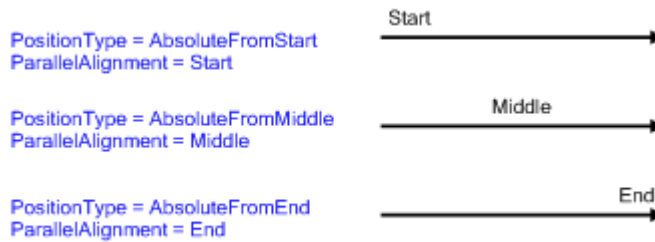


### Text Alignment

The ParallelAlignment property determines the alignment of the text, parallel to the link path, relative to the text position. The parallel alignment can have the following values:

- ◆ **Start:** the start of the text is aligned with the position.
- ◆ **Middle:** the middle of the text is aligned with the position.
- ◆ **End:** the end of the text is aligned with the position.

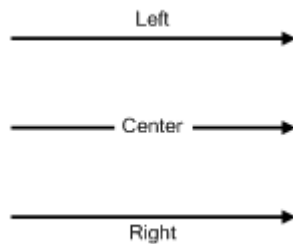
The example below shows the effects of the different values.



The `OrthogonalAlignment` property determines the alignment of the text orthogonally to the direction of the link path. The parallel alignment can have the following values:

- ◆ **Left:** the text is placed on the left of the link path, looking towards the end of the link.
- ◆ **Center:** the text is centered on the link path.
- ◆ **Right:** the text is placed on the right of the link path, looking towards the end of the link.

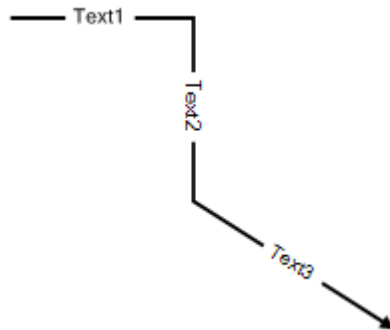
The example below shows the effects of the different values.



When the **OrthogonalAlignment** is **Left** or **Right**, the `Distance` property determines the distance between the link path and the text.

### Text Rotation

If the `AutoRotate` property of the **LinkTextItem** object is **true**, the text is rotated to follow the direction of the link segment, as in the following illustration.



It is also possible to rotate each text item to a fixed angle using the **Rotation** property of the **LinkTextItem** object. The value of this property is in degrees.

If **AutoRotate** is **true** and **Rotation** is non-0, the **Rotation** value is added to the automatic rotation angle.

### Text Appearance

To customize the appearance of the text, including font, color and rendering hints use the **TextAppearance** property of the **LinkTextItem**. For more information on the properties of the **TextAppearance** object see *Displaying Text in a Graphic Object*.

***Note:** The **Margins** property of the **TextAppearance** is taken into account when aligning the text item on the link. For example, if **OrthogonalAlignment** is **Left**, with a **Distance** of 0, and if **Margins** is 3 pixels in all directions, the text will be drawn 3 pixels away from the link.*

### Link/Text Intersections

The **CutAtText** property of the **Link** specifies whether the link path must be cut when it intersects a text item. By default, the property is **true**.

---

### Link Crossing

When links cross in a diagram, the crossings can be highlighted by drawing the link with a special shape at the crossing point. This shape can be customized using the **CrossingStyle** and **CrossingSize** properties of the **Link**. See *Using Automatic Link Crossing Detection in a Graph* for more information on how to enable and customize link crossing detection.

---

## Controls

Controls are containers whose geometry is defined by a rectangular area in which children will be placed. The base class for controls is the `Control` class. Controls add graphic decorations such as a border or a background and can implement specific behaviors.

### In This Section

#### *The Control Class*

Explains the basics of the `Control` class.

#### *Basic Controls*

Explains the basic controls.

#### *Single Content Controls*

Explains the basics of the `ContentControl` class.

#### *Multiple Content Controls*

Explains the basics of the `ObjectsControl` class.

---

## The Control Class

`Control` is the base class for controls in IBM® ILOG® Diagram for .NET. Controls are containers whose geometry is defined by a rectangular area in which children will be placed.

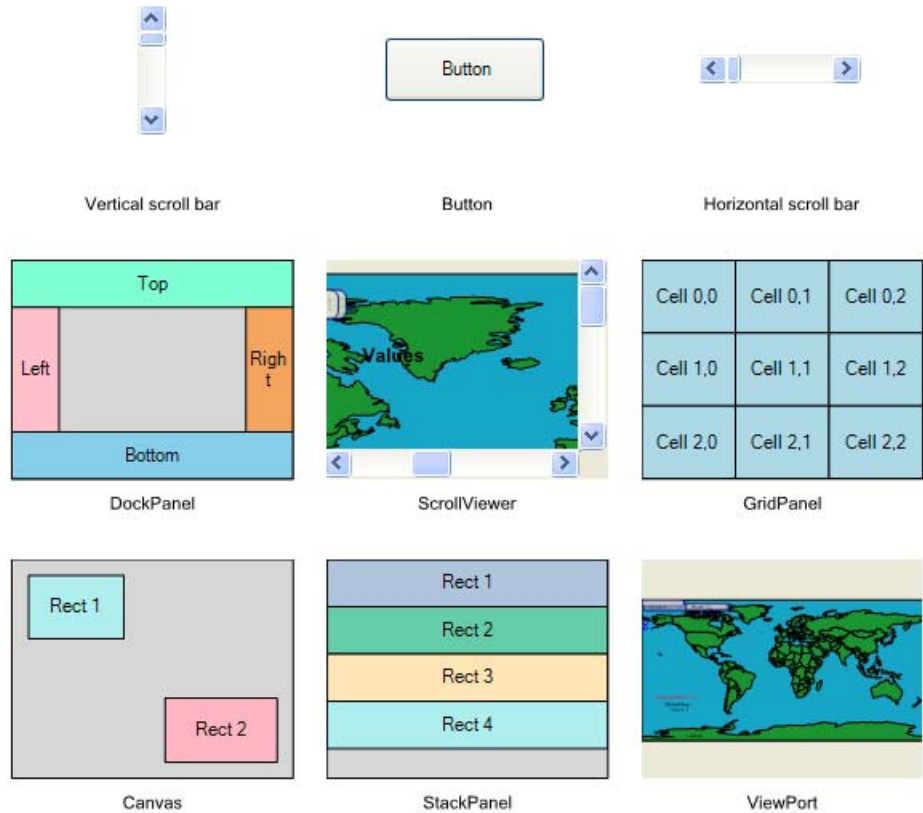
### Controls Overview

The **Control** class is an abstract class that has three different types of subclasses listed in the following table:

Super Class	Description	Examples
<code>Control</code>	Controls that do not have children.	<code>HScrollBar</code> , <code>VScrollBar</code>
<code>ContentControl</code>	Controls with a single child.	<code>Button</code> , <code>ScrollViewer</code> , <code>ViewPort</code>
<code>ObjectsControl</code>	Controls with several children.	<code>DockPanel</code> , <code>Canvas</code> , <code>GridPanel</code> , <code>StackPanel</code>

For details about the classes **DockPanel**, **Canvas**, **GridPanel** and **StackPanel**, see *Panels*.

The following illustration shows the available controls:



### Control Geometry

The geometry of the control is given by the `Rectangle` property. This geometry can be restricted by specifying a minimum and a maximum size using the `Control.MinimumSize` and `Control.MaximumSize` properties.

### Control Appearance

The **Control** class has the `Control.Background` property that can be used to change the control background.

The **Control** class has the `Control.Foreground` property that can be used to change the control foreground.

The `Control.VisualStyle` property defines whether the control should use visual styles to draw itself. By default, this value inherits from the control parent. Note that only controls supporting visual styles will be affected by changing this value. An example of control using visual styles is the **Button** class.

---

### Basic Controls

The only controls provided that do not have children are the `HScrollBar` and `VScrollBar` controls. The appearance, behavior, and API of these controls are the same that can be found on the Windows® Forms scroll bars.

The scroll bar values can be changed using the `ScrollBar.Minimum`, `ScrollBar.Maximum`, and `ScrollBar.Value` properties. The `ScrollBar.ValueChanged` event is raised when the scroll bar value changes.

---

### Single Content Controls

The base class for controls that contains a single child is the abstract `ContentControl` class. The `ContentControl.Content` property is used to get and set the control content, which can be any graphic object. The `ContentControl.ContentMargins` property can be used to change the margins around the control content.

Existing concrete subclasses of the **ContentControl** class are:

- ◆ `Button` : Displays a common button.
  - ◆ `ViewPort` : Displays its content or a portion of it with a specific transformation.
  - ◆ `ScrollViewer` : Allows you to scroll the control content.
- 

### Multiple Content Controls

The base class for controls that contains multiple children is the abstract `ObjectsControl` class. The `ObjectsControl.Objects` property is used to access the control children.

The only subclass of **ObjectsControl** is the `Panel` class, which is described in *Panels*.

---

## Panels

Panels are graphic objects that control the size and dimensions of their children. IBM® ILOG® Diagram for .NET provides a number of predefined panels as well as the ability to construct custom panels. Panels can be nested to build complex structures.

### In This Section

#### *The Panel class*

Explains the basics of the Panel class.

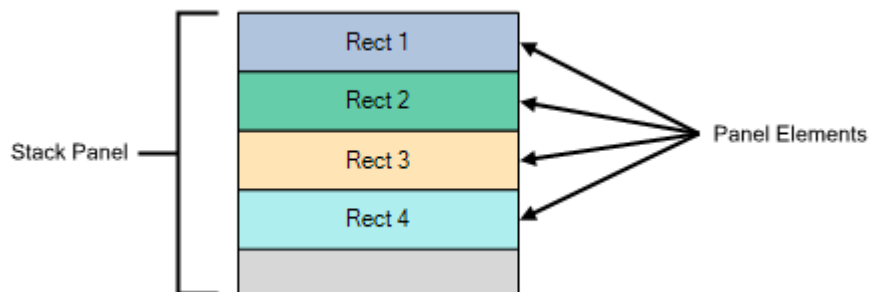
#### *Using Predefined Panels*

Describes the predefined panels.

---

### The Panel class

Panel is the base class for graphic objects that provide layout support in IBM® ILOG® Diagram for .NET. A panel is a rectangular container in which children are laid out. To layout children means to set their size and position inside the panel area. The following illustration shows a vertical StackPanel, a panel that stacks its children one on top of the other.



### Populating Panels

The **Panel** class has an **Objects** property that can be used to access the panel children. The following example shows how to create a **StackPanel** with several children:

```
StackPanel panel = new StackPanel();  
Rect rect1 = new Rect(0, 0, 100, 30);  
rect1.Text = "Rect 1";  
Rect rect2 = new Rect(0, 0, 100, 30);  
rect2.Text = "Rect 2";  
Rect rect1 = new Rect(0, 0, 100, 30);
```

## Using Predefined Graphic Objects

```
rect3.Text = "Rect 3";  
Rect rect1 = new Rect(0, 0, 100, 30);  
rect4.Text = "Rect 4";  
panel.Objects.Add(rect1);  
panel.Objects.Add(rect2);  
panel.Objects.Add(rect3);  
panel.Objects.Add(rect4);  
Dim panel As StackPanel = New StackPanel  
Dim rect1 As Rect = New Rect(0, 0, 100, 30)  
rect1.Text = "Rect 1"  
Dim rect2 As Rect = New Rect(0, 0, 100, 30)  
rect2.Text = "Rect 2"  
Dim rect1 As Rect = New Rect(0, 0, 100, 30)  
rect3.Text = "Rect 3"  
Dim rect1 As Rect = New Rect(0, 0, 100, 30)  
rect4.Text = "Rect 4"  
panel.Objects.Add(rect1)  
panel.Objects.Add(rect2)  
panel.Objects.Add(rect3)  
panel.Objects.Add(rect4)
```

### Defining Appearance Properties

The **Panel** class defines some common properties that can be used to customize its appearance.

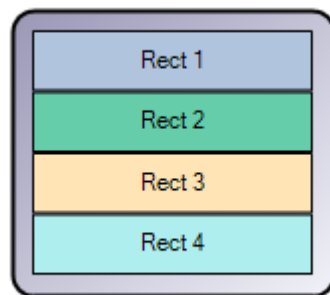
#### **Border**

The border of the panel is drawn as a rectangle around the layout area. The **Border** property allows you to specify a **Stroke** to draw the border of the panel. **Margins** can be added between the border and the layout area by specifying the **BorderMargins** property. If you want to change the border shape and draw a rounded rectangle, use the **CornerRadius** property.

#### **Background**

The **Background** property allows you to specify a **Fill** to draw the panel background.

The following illustration shows a vertical **StackPanel** with a rounded border and a gradient fill as background:





## Layout Properties

The **Panel** class also defines some properties that can be used to customize the layout. Those properties are used by most of the **Panel** subclasses.

### *Margins*

While defining the layout, it is possible to add margins around the children bounds. The **Panel** class defines the `ItemsMargins` property that can be used to set the same margins on each element of the panel. In addition, the `SetMargins` method can be used to set the margin of a specific element. In this case, both margins are added during the layout.

### *Alignments*

Each element of the panel has vertical and horizontal alignment. The horizontal alignment specifies how the element is resized in the horizontal direction during the layout; the vertical alignment specifies how the element is resized in the vertical direction.

You can get and set the horizontal and vertical alignments by using the following methods: `GetHorizontalAlignment`, `SetHorizontalAlignment`, `GetVerticalAlignment`, and `SetVerticalAlignment`.

The horizontal alignment is defined by the `HorizontalAlignment` enumeration. The following table lists the different values of this enumeration:

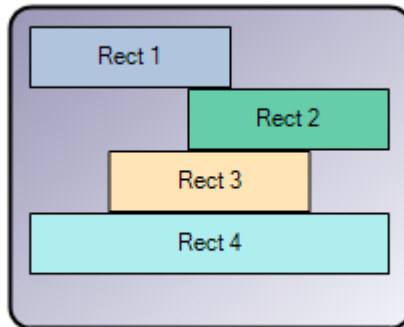
Value	Description
Left	The element is aligned to the left. The element size is defined by the element itself.
Right	The element is aligned to the right. The element size is defined by the element itself.
Center	The element is horizontally centered. The element size is defined by the element itself.
Stretch	The element is stretched in the horizontal direction to fit the size defined by the panel. The element size is defined by the panel. This is the default value.

The vertical alignment is defined by the `VerticalAlignment` enumeration. The following table lists the different values of this enumeration:

Value	Description
Top	The element is aligned to the top. The element size is defined by the element itself.
Bottom	The element is aligned to the bottom. The element size is defined by the element itself.

Value	Description
Center	The element is vertically centered. The element size is defined by the element itself.
Stretch	The element is stretched in the vertical direction to fit the size defined by the panel. The element size is defined by the panel. This is the default value.

Most of the Panel subclasses use either the horizontal alignment, or the vertical alignment, or both. For example, a vertical **StackPanel** uses only the horizontal alignment to layout its elements, whereas a horizontal **StackPanel** uses only the vertical alignment. The following illustration shows a vertical **StackPanel** where the first element has its horizontal alignment set to **HorizontalAlignment.Left**, the second element to **HorizontalAlignment.Right**, the third element to **HorizontalAlignment.Center**, and the last element to **HorizontalAlignment.Stretch**:



The following table lists the different types of panel and for each type it indicates the corresponding alignment:

Type o Panel	Use Horizontal Alignment	Use Vertical Alignment
Canvas	No	No
StackPanel	Yes, for vertical stack panels.	Yes, for horizontal stack panels.
GridPanel	Yes	Yes
DockPanel	Yes, for vertical elements.	Yes, for horizontal elements.

## Using Graphic Object Preferred Size

Depending on the type of panel you use, an element may have its size constrained or not by its panel. When element size is not constrained by the panel, it is defined by the element itself. In this case, if the `AutoSize` property of the element is set to **true**, the element can take its preferred size, given by the `GetPreferredSize` method. If the **AutoSize** property is set to **false**, the element keeps its original size.

The default behavior of the **GetPreferredSize** method is to return the size needed for the graphic object to fully display its `Text` property. Graphic objects that need a different size may change this behavior. For example, the preferred height of a vertical **StackPanel** is the sum of the heights of its elements.

For more details on the preferred sizes of the existing panels, see *Using Graphic Object Preferred Size*.

## Positioning Elements

The layout area of the panel is given by the `ClientRectangle` property. The left upper coordinate of this area is always the point (0, 0). To place its elements inside this area, the panel sets the bounds of each element by calling the `SetChildBounds` method.

## Optimizing Layout

The **Panel** automatically updates the layout when it appears to be invalid. The layout can be invalid for different reasons: the panel has been resized, or elements have been added to or removed from the panel.

For the seek of performance, it may be useful to temporarily disable the layout mechanism. Use the `SuspendLayout` method to suspend the layout and the `ResumeLayout` method to resume it.

---

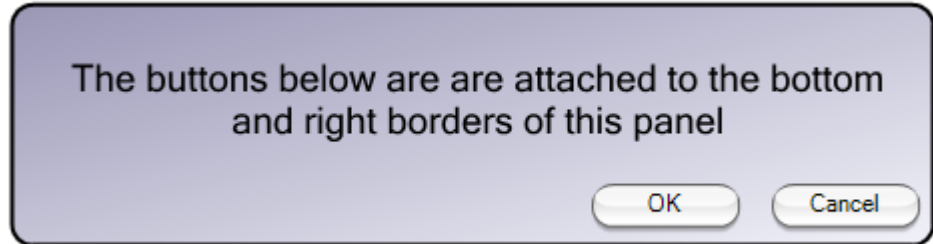
## Using Predefined Panels

IBM® ILOG® Diagram for .NET includes a comprehensive set of panel implementations that enable many complex layouts. These derived classes expose properties and methods that allow you to create the most standard user interface scenarios. If you cannot find a child arrangement behavior that meets your needs, you can create new layouts.

## The Canvas Class

The `Canvas` class enables absolute positioning of contents according to x and y coordinates. Elements are positioned relatively to the upper left corner of the layout area. Elements can be attached to a border of the panel using the `SetAnchor` method. When an element is attached to the **Canvas**, the distance between the element bounds and the attached border remains constant when the panel gets resized.

The following illustration shows a **Canvas** with two buttons.



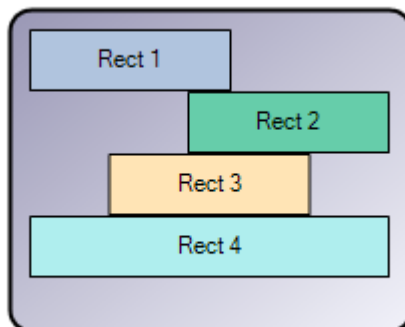
The following example shows how to create a **Canvas** with a **Rect** object attached to the right border of the **Canvas**:

```
Canvas panel = new Canvas();
panel.Rectangle = new Rectangle2D(0, 0, 500, 200);
Rect rect1 = new Rect(300, 100, 100, 30);
rect1.Text = "Rect 1";
panel.Objects.Add(rect1);
panel.SetAnchor(rect1, AnchorStyle.Right);
Dim panel As Canvas = New Canvas
panel.Rectangle = New Rectangle2D(0, 0, 500, 200)
Dim rect1 As Rect = New Rect(300, 100, 100, 30)
rect1.Text = "Rect 1"
panel.Objects.Add(rect1)
panel.SetAnchor(rect1, AnchorStyle.Right)
```

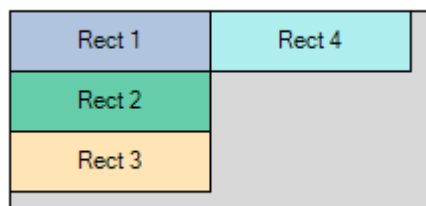
### The StackPanel Class

The **StackPanel** class stacks elements to an assigned direction. The **Orientation** property can be used to control content flow. The default value for this property is **Vertical**. When the **StackPanel** is vertical, it uses the horizontal alignment of its elements to layout them. When the **StackPanel** is horizontal, it uses the vertical alignment of its elements to layout them.

The following illustration shows a vertical **StackPanel** where the first element has its horizontal alignment set to **HorizontalAlignment.Left**, the second element to **HorizontalAlignment.Right**, the third element to **HorizontalAlignment.Center**, and the last element to **HorizontalAlignment.Stretch**:



When the flow of elements exceeds the panel size, the **StackPanel** can display its elements on several rows if the **FlowLayout** property has been set to **true**. The following illustration shows a vertical **StackPanel** with the **FlowLayout** set to **true**.



**Note:** When the **FlowLayout** property is set to **true**, the **VerticalAlignment.Stretch** and **HorizontalAlignment.Stretch** alignments can no longer be used.

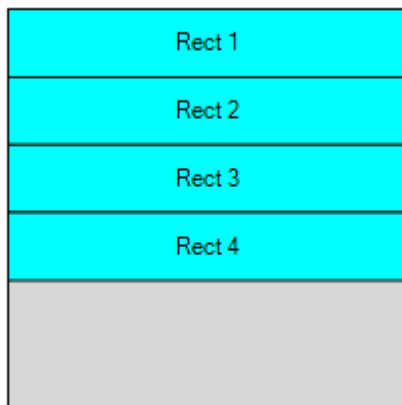
The following example shows how to create a **StackPanel** with four **Rect** objects stacked vertically.

```
StackPanel panel = new StackPanel();
panel.Rectangle = new Rectangle2D(0, 0, 200, 200);
Rect rect1 = new Rect();
rect1.Text = "Rect 1";
rect1.Fill = new SolidFill(Color.Aqua);
rect1.TextAppearance.Margins = new Margins(10);
rect1.AutoSize = true;
panel.Objects.Add(rect1);
```

## Using Predefined Graphic Objects

```
Rect rect2 = new Rect();
rect2.Text = "Rect 2";
rect2.Fill = new SolidFill(Color.Aqua);
rect2.TextAppearance.Margins = new Margins(10);
rect2.AutoSize = true;
panel.Objects.Add(rect2);
Rect rect3 = new Rect();
rect3.Text = "Rect 3";
rect3.Fill = new SolidFill(Color.Aqua);
rect3.TextAppearance.Margins = new Margins(10);
rect3.AutoSize = true;
panel.Objects.Add(rect3);
Rect rect4 = new Rect();
rect4.Text = "Rect 4";
rect4.Fill = new SolidFill(Color.Aqua);
rect4.TextAppearance.Margins = new Margins(10);
rect4.AutoSize = true;
panel.Objects.Add(rect4);
Dim panel As StackPanel = New StackPanel
panel.Rectangle = New Rectangle2D(0, 0, 200, 200)
Dim rect1 As Rect = New Rect
rect1.Text = "Rect 1"
rect1.Fill = New SolidFill(Color.Aqua)
rect1.TextAppearance.Margins = New Margins(10)
rect1.AutoSize = true
panel.Objects.Add(rect1)
Dim rect2 As Rect = New Rect
rect2.Text = "Rect 2"
rect2.Fill = New SolidFill(Color.Aqua)
rect2.TextAppearance.Margins = New Margins(10)
rect2.AutoSize = true
panel.Objects.Add(rect2)
Dim rect3 As Rect = New Rect
rect3.Text = "Rect 3"
rect3.Fill = New SolidFill(Color.Aqua)
rect3.TextAppearance.Margins = New Margins(10)
rect3.AutoSize = true
panel.Objects.Add(rect3)
Dim rect4 As Rect = New Rect
rect4.Text = "Rect 4"
rect4.Fill = New SolidFill(Color.Aqua)
rect4.TextAppearance.Margins = New Margins(10)
rect4.AutoSize = true
panel.Objects.Add(rect4)
```

The user interface you have just created should look like the following illustration:



### The GridPanel Class

A grid panel is composed of rows and columns that create a set of grid cells in which elements are positioned.

The rows and columns of the GridPanel object can be accessed through the Rows and Columns properties. Each row or column of the **GridPanel** is an instance of the GridElement class.

The **GridElement** class has a size and a policy that indicates how the grid element should manage its size. The sizing policy is defined by the UnitType property. The following table lists the possible values for this property:

Value	Description
<b>GridUnitType.Fixed</b>	The grid element has a fixed size.
<b>GridUnitType.AutoFixed</b>	The grid element has a fixed size which depends on the graphic objects it contains. The grid element size is dynamically adjusted when its contents changes.
<b>GridUnitType.Elastic</b>	The grid element has a size that depends on the remaining space in the <b>GridPanel</b> after measuring the fixed elements. The Size property represents a percentage of space allocated to this element. This is the default value.

The following example creates a **GridPanel** with three rows and three columns. The second row and column have their **UnitType** property set to **GridUnitType.Elastic**:

```
GridPanel panel = new GridPanel();
panel.Rows.Add(new GridElement(GridUnitType.Fixed, 50));
panel.Rows.Add(new GridElement(GridUnitType.Elastic, 100));
panel.Rows.Add(new GridElement(GridUnitType.Fixed, 50));
panel.Columns.Add(new GridElement(GridUnitType.Fixed, 50));
panel.Columns.Add(new GridElement(GridUnitType.Elastic, 100));
panel.Columns.Add(new GridElement(GridUnitType.Fixed, 50));
Dim panel As GridPanel = New GridPanel
panel.Rows.Add(New GridElement(GridUnitType.Fixed, 50))
panel.Rows.Add(New GridElement(GridUnitType.Elastic, 100))
panel.Rows.Add(New GridElement(GridUnitType.Fixed, 50))
panel.Columns.Add(New GridElement(GridUnitType.Fixed, 50))
panel.Columns.Add(New GridElement(GridUnitType.Elastic, 100))
panel.Columns.Add(New GridElement(GridUnitType.Fixed, 50))
```

To specify in which row or column an element should be placed, use the **SetRow** and **SetColumn** methods. Elements can span across multiple rows or columns using the **SetRowSpan** and **SetColumnSpan** methods.

### *Notes:*

1. *Several elements can share the same grid cell.*
2. *Elements whose row or column is negative can be positioned using absolute positioning.*

The **GridPanel** class uses both horizontal and vertical alignments of its elements to place them into grid cells.

The following example shows how to create a **GridPanel** with two rows and two columns in which **Rect** objects are placed:

```
GridPanel panel = new GridPanel();
panel.Rows.Add(new GridElement());
panel.Rows.Add(new GridElement());
panel.Columns.Add(new GridElement());
panel.Columns.Add(new GridElement());
Rect rect1 = new Rect();
rect1.Text = "Rect 1";
panel.SetObjectAt(rect1, 0, 0);
panel.Objects.Add(rect1);
Rect rect2 = new Rect();
rect2.Text = "Rect 2";
panel.SetObjectAt(rect2, 0, 1);
panel.Objects.Add(rect2);
Rect rect3 = new Rect();
rect3.Text = "Rect 3";
panel.SetObjectAt(rect3, 1, 0);
panel.Objects.Add(rect3);
Rect rect4 = new Rect();
rect4.Text = "Rect 4";
```



```

panel.SetObjectAt(rect4, 1, 1);
panel.Objects.Add(rect4);
Dim panel As GridPanel = New GridPanel
panel.Rows.Add(New GridElement)
panel.Rows.Add(New GridElement)
panel.Columns.Add(New GridElement)
panel.Columns.Add(New GridElement)
Dim rect1 As Rect = New Rect
rect1.Text = "Rect 1"
panel.SetObjectAt(rect1, 0, 0)
panel.Objects.Add(rect1)
Dim rect2 As Rect = New Rect
rect2.Text = "Rect 2"
panel.SetObjectAt(rect2, 0, 1)
panel.Objects.Add(rect2)
Dim rect3 As Rect = New Rect
rect3.Text = "Rect 3"
panel.SetObjectAt(rect3, 1, 0)
panel.Objects.Add(rect3)
Dim rect4 As Rect = New Rect
rect4.Text = "Rect 4"
panel.SetObjectAt(rect4, 1, 1)
panel.Objects.Add(rect4)

```

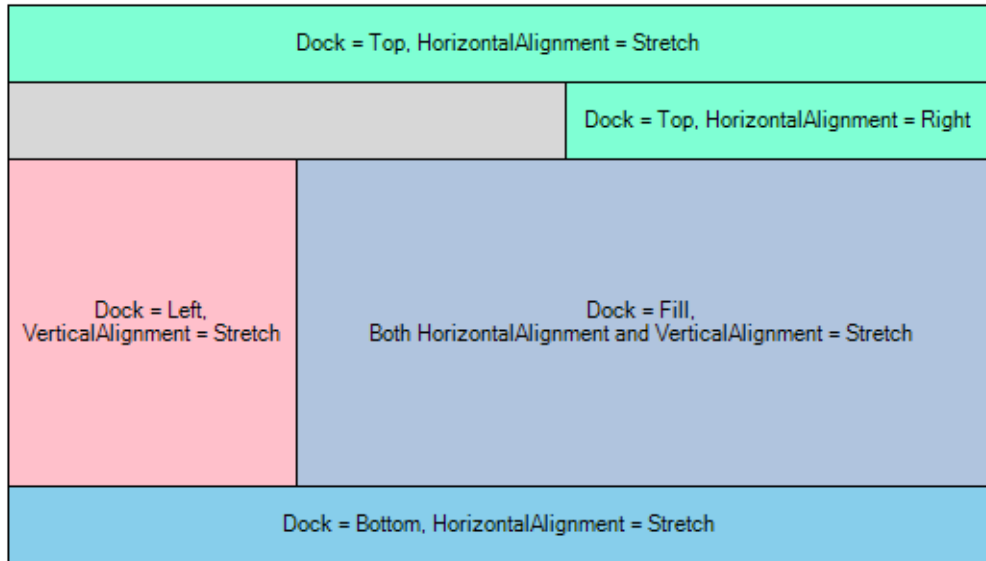
The application you have just created looks like the following illustration:



### The DockPanel Class

The **DockPanel** class positions elements along the borders of its layout area. Elements can be docked to the left, right, top or bottom of the panel, or they can fill the remaining area. During the layout, the **DockPanel** uses the horizontal alignment of its elements docked to the top or bottom, the vertical alignment of its elements docked to the left or right, and both the horizontal and vertical alignments of elements that fill the remaining area.

The following illustration shows a **DockPanel**.



To get or set the dock style of an element, use the `GetDock` and `SetDock` methods.

The following example shows how to create a **DockPanel** with two **Rect** objects: one docked to the top and one that fills the remaining area:

```
DockPanel panel = new DockPanel();
panel.Rectangle = new Rectangle2D(0, 0, 300, 300);
Rect rect1 = new Rect(0, 0, 100, 30);
rect1.Text = "Rect 1";
panel.Objects.Add(rect1);
panel.SetDock(rect1, ILOG.Diagrammer.Graphic.DockStyle.Top);
Rect rect2 = new Rect();
rect2.Text = "Rect 2";
panel.Objects.Add(rect2);
panel.SetDock(rect2, ILOG.Diagrammer.Graphic.DockStyle.Fill);
Dim panel As DockPanel = New DockPanel
panel.Rectangle = New Rectangle2D(0, 0, 300, 300)
Dim rect1 As Rect = New Rect(0, 0, 100, 30)
rect1.Text = "Rect 1"
panel.Objects.Add(rect1)
panel.SetDock(rect1, ILOG.Diagrammer.Graphic.DockStyle.Top)
Dim rect2 As Rect = New Rect
rect2.Text = "Rect 2"
panel.Objects.Add(rect2)
panel.SetDock(rect2, ILOG.Diagrammer.Graphic.DockStyle.Fill)
```

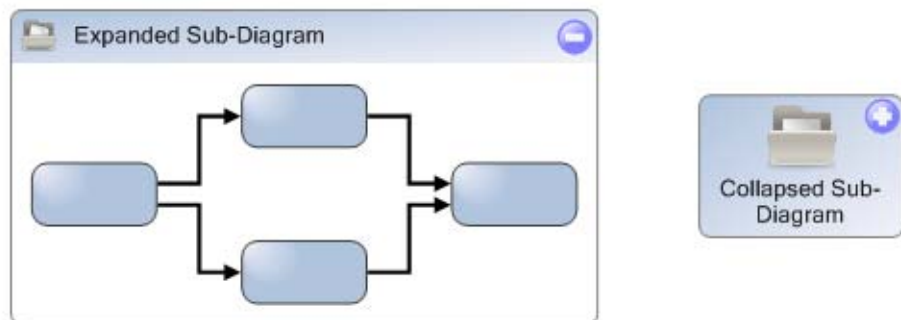
**Note:** Elements whose dock style is *DockStyle.None* can be positioned using absolute positioning.

## Subdiagrams

The `SubDiagram` class is used to create nested diagrams, that is, diagrams contained in other diagrams. You can also create nested diagrams using other graphic containers like the `Group` or the `Canvas` class, but the **SubDiagram** class provides the following additional capabilities:

- ◆ in-place expand and collapse
- ◆ scrolling
- ◆ customizable title bar
- ◆ customizable icons

The following illustration shows two **SubDiagram** objects. The first object is expanded and the second one is collapsed.



### Creating and Populating a SubDiagram

The `SubDiagram` class is a graphic container. The children of the container are defined by the `SubDiagramObjects` property.

The following example shows how to create a simple subdiagram containing two basic shapes:

```
SubDiagram subDiagram = new SubDiagram();
```

```
Rect rect1 = new Rect(10, 10, 80, 30);
subDiagram.SubDiagramObjects.Add(rect1);
Rect rect2 = new Rect(100, 100, 80, 30);
subDiagram.SubDiagramObjects.Add(rect2);
Dim subDiagram As SubDiagram = New SubDiagram
Dim rect1 As Rect = New Rect(10, 10, 80, 30)
subDiagram.SubDiagramObjects.Add(rect1)
Dim rect2 As Rect = New Rect(100, 100, 80, 30)
subDiagram.SubDiagramObjects.Add(rect2)
```

---

### Expanding and Collapsing a SubDiagram

A **SubDiagram** is initially expanded, which means that its children are visible. It can be collapsed by setting the `Expanded` property to `false`. Once the subdiagram is collapsed, its children are no longer visible and only a reduced rectangular shape is drawn.

---

### Customizing the Expanded Appearance

An expanded subdiagram consists of a title bar and a rectangular area where the children are displayed.

The title bar is initially located at the top of the subdiagram, although this position can be modified using the `TitlePosition` property. If the title bar is placed on the left or on the right, the title text is automatically rotated by 90 degrees.

The title bar contains three optional components: a text area, a button used to collapse the subdiagram, and an icon.

The title text can be changed using the `TitleText` property, and the appearance of the text can be customized through the `TitleTextAppearance`. (see *Displaying Text in a Diagram* for a description of the `TextAppearance` class).

The default collapse button displays a “minus” sign. You can customize the collapse button using the `CollapseButton` property.

The icon can be set using the `ExpandedIcon` property. By default, no icon is displayed. See *Subdiagram Icons and Buttons* for details on how to customize the button and the icon.

The background and the border of the title bar can be changed using the `TitleBackground` and `TitleBorder` properties.

The title bar can be completely hidden by setting the `TitleVisible` property to `false`.

The `ExpandedBackground` and `ExpandedBorder` properties define the overall background and border of the subdiagram, that is, the background visible beneath the children and the border that surrounds the children area and the title bar.

The `ExpandedRadius` property changes the corner radius of the expanded subdiagram border.

---

## Customizing the Collapsed Appearance

The collapsed subdiagram consists of a rectangle (usually small) which contains three optional components: a text, a button used to expand the subdiagram, and an icon.

The text can be changed using the `CollapsedText` property, and the appearance of the text can be customized through the `CollapsedTextAppearance`. (see *Displaying Text in a Diagram* for a description of the **TextAppearance** class).

The default expand button displays a “plus” sign. You can customize the expand button using the `ExpandButton` property.

The icon can be set using the `CollapsedIcon` property. By default, no icon is displayed. See *Subdiagram Icons and Buttons* for details on how to customize the button and the icon.

The background and the border of the collapsed subdiagram can be changed using the `CollapsedBackground` and `CollapsedBorder` properties

The `CollapsedRadius` property changes the corner radius of the collapsed subdiagram border.

---

## Subdiagram Icons and Buttons

The **CollapseButton**, **ExpandButton**, **ExpandedIcon** and **CollapsedIcon** properties accept values that are instances of the `SubDiagramIcon` class. This class describes the image, the size, the position and the visibility of a button or an icon displayed in a subdiagram.

The `Image` property specifies the image of the button or icon. The value of this property is a string that can be:

- ◆ A URI, for example “file://C:/work/images/button.png”, (the URI can point to a bitmap file or an SVG file);
- ◆ a string containing any IBM ILOG Diagram for .NET graphic object, serialized to XML (this is mainly used when you edit a **SubDiagram** object in Visual Studio or in the Diagram Editor).

The `Dock`, `HorizontalAlignment`, `VerticalAlignment` and `Margins` properties define the position of the button or icon. The meaning of these properties is the same as in the `DockPanel` class (see *The Panel class* and *The DockPanel Class*).

**Note:** For an expanded subdiagram, the position is defined relative to the title bar, not to the whole subdiagram object.

The `Size` property lets you resize the image of a button or icon. If the size is `0,0` (which is the default value), the image will have its default size.

You can hide a button or icon by setting its `Visible` property to `false`.

The `Color` property can be used to change the color of a button or icon, but note that this works only for vector images (that is, SVG files or IBM ILOG Diagram for .NET graphic objects), and that only the fill color of the first `Shape` child contained in the graphic object will be changed.

---

### Scrolling

The **SubDiagram** supports scrolling: if the bounds of the children are larger than the bounds of the subdiagram, a vertical and/or a horizontal scrollbar will be displayed to let you scroll the visible area of the subdiagram. The visibility of the scrollbars is controlled by the `HScrollBar` and `VScrollBar` properties. If needed, you can adjust the scrolling position using the `ContentOrigin` property.

---

### AutoBounds Mode

If the `AutoBounds` property is set to `true`, the subdiagram will automatically adjust its position and its size to fit the bounds of its children (taking into account the `ContentMargins` property). In that case, scrolling is disabled.

---

### Graph Layout

The **SubDiagram** container supports graph layout. If a graph layout algorithm is specified on an ancestor container (through the `GraphLayout` and/or `LinkLayout` properties), this graph layout can be applied recursively to the content of the subdiagram. A specific graph layout algorithm can be set on the subdiagram using its own **GraphLayout** and/or **LinkLayout** properties, in which case the local algorithm is applied to the subdiagram instead of the inherited layout.

When a graph layout algorithm is applied to a subdiagram, the children of the subdiagram are automatically translated so that the top-left corner of the children bounds corresponds to the top-left corner of the subdiagram. Moreover, if the `AutoSizeAfterGraphLayout` property is `true`, the size of the subdiagram is also adjusted to match the bounds of its children.

---

## Graphic Symbols

A simple way to create new graphical representation is to create a graphic symbol. A graphic symbol is a set of graphic objects that act as a single graphic object.

To create a graphic symbol, use an instance of the `GraphicSymbol` class and add other graphic objects to it through the `Objects` property. The **GraphicSymbol** class is very similar to the `Group` class that also allows you to group objects together. The **Group** class is mainly used to create logical group of graphic objects or to represent a complex diagram. The main difference between a **Group** and a **GraphicSymbol** is about selection. When you use the

selection tools, the graphic symbol can be selected only as a whole. It is not possible to select individual graphic objects that compose the graphic symbol. If you need to create a basic new graphic representation you can use the **GraphicSymbol** class; if you need to create a more complex graphic object that contains new properties and some specific logic, then you should create a user symbol.

For more information on the **UserSymbol** class see *User Symbols*.

The following example shows how to create a graphic symbol by assembling two basic shapes:

```
GraphicSymbol symbol = new GraphicSymbol();
Basic2DShape triangle = new Basic2DShape(15f, 0f, 80f, 70f);
triangle.ShapeType = Basic2DShapeType.Triangle;
triangle.ControlValue = 0.5f;
triangle.Fill = new SolidFill(Color.Blue);

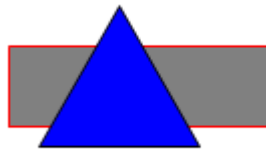
Rect rectangle = new Rect(0f, 20f, 130f, 40f);
rectangle.Fill = new SolidFill(Color.Gray);
rectangle.Stroke = new Stroke(Color.Red);

symbol.Objects.AddRange(new GraphicObject[] { rectangle, triangle });
Dim symbol As GraphicSymbol = New GraphicSymbol
Dim triangle As Basic2DShape = New Basic2DShape(15F, 0F, 80F, 70F)
triangle.ShapeType = Basic2DShapeType.Triangle
triangle.ControlValue = 0.5F
triangle.Fill = New SolidFill(Color.Blue)

Dim rectangle As Rect = New Rect(0F, 20F, 130F, 40F)
rectangle.Fill = New SolidFill(Color.Gray)
rectangle.Stroke = New Stroke(Color.Red)

symbol.Objects.AddRange(New GraphicObject() {rectangle, triangle})
```

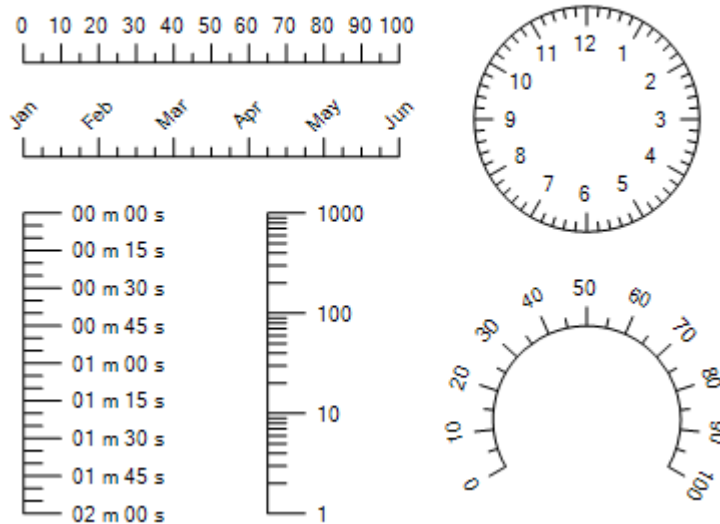
The graphic symbol you have just created should look like the following one.



---

## Scale Objects

A scale is a graphic object that draws a sequence of small lines (ticks) with labels along a base line. Scales are typically used to build symbols representing gauges and charts. The following illustration shows examples of scales.



There are two types of scales: linear scales represented by the class **LinearScale**, and circular scales represented by the class **CircularScale**. These two classes have a common base class, **ScaleBase**, which implements most of the common properties and behavior.

### In This Section

#### *The ScaleBase Class*

Describes the properties common to linear and circular scales.

#### *Linear Scales*

Describes the **LinearScale** class.

#### *Circular Scales*

Describes the **CircularScale** class.

---

### The ScaleBase Class

This section explains the common properties and behavior provided by the **ScaleBase** class.



A scale consists of:

- ◆ A base line that can be a straight segment or an elliptical arc.
- ◆ A sequence of small lines, orthogonal to the base line, called the ticks. There are two types of ticks: major ticks and minor ticks.
- ◆ A sequence of labels, drawn next to the major ticks.

### Scale Range and Tick Intervals

The range of the scale is defined by the `Minimum` and `Maximum` properties.

The number and the positions of the ticks are controlled by the `MajorTickInterval`, `MinorTickInterval`, `AutoMajorTicks` and `AutoMinorTicks` properties.

If `AutoMajorTicks` is **true**, the interval between major ticks is computed automatically according to the scale range. Otherwise, the interval between major ticks is specified by the `MajorTickInterval`.

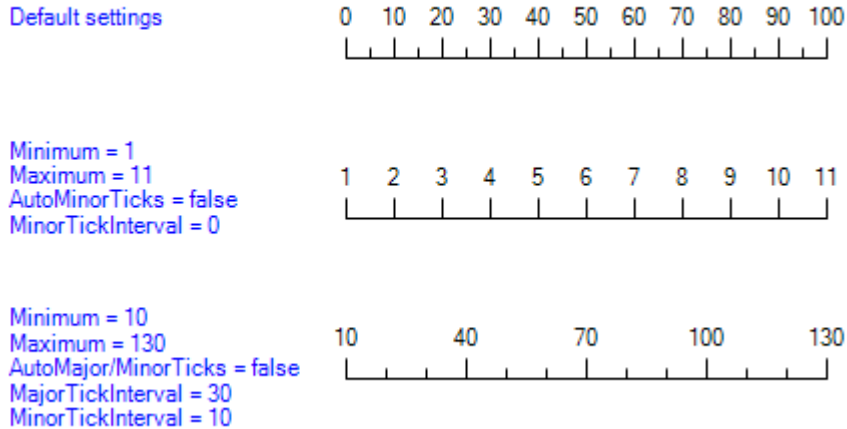
*Note: The interval is specified in the same value units as the **Minimum** and **Maximum** properties. For instance, if **Minimum** is 0 and **Maximum** is 1000, setting **MajorTickInterval** to 100 will create 10 major tick intervals (that is., 11 major ticks).*

The interval between minor ticks is controlled by the `AutoMinorTicks` and `MinorTickInterval` in the same way.

If `MajorTickInterval` is set to 0 and `AutoMajorTicks` is **false**, no ticks are displayed (either major or minor).

If `MinorTickInterval` is set to 0 and `AutoMinorTicks` is **false**, no minor ticks are displayed (but major ticks are displayed normally).

The following illustration shows some examples of range and tick interval settings:



### Tick Size

The `MajorTickSize` and `MinorTickSize` properties can be used to change the length of the tick segments. If **MajorTickSize/MinorTickSize** is set to 0, no major/minor ticks are displayed.

### Scale Labels

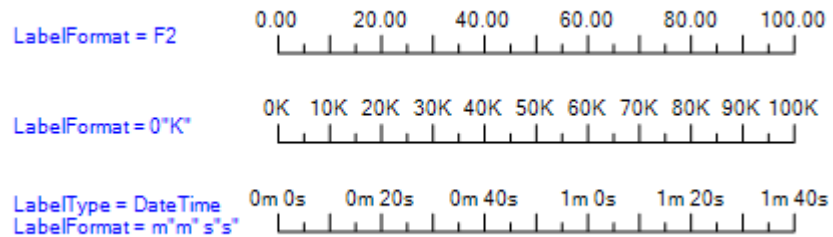
Labels are displayed near each major tick.

The `LabelType` property determines the text of the labels. This property is of type **ScaleLabelType** and these are the possible values:

- ◆ **Numeric**: The labels display the numeric values corresponding to each tick. The `LabelFormat` property can be used to format the values.
- ◆ **DateTime**: The value corresponding to each tick is considered as a number of seconds. It is converted to a `DateTime` object which is then formatted using the **LabelFormat** property.
- ◆ **Discrete**: The labels are specified explicitly using the `Labels` property. For example, if `Labels` contains the string "A", "B", "C", the first tick will have the label "A", the second tick will have the label "B" and the third tick will have the label "C". If the `Labels` property contains fewer elements than the number of ticks, the remaining ticks have no label. If the `Labels` property contains more elements than the number of ticks, the remaining elements are ignored.

The **LabelFormat** property can be used to customize the formatting of the labels (when **LabelType** is Numeric or DateTime). The possible format strings are defined by the .NET Framework® composite formatting feature. The format can be a simple format string (for example, "G", "F2", "000", "mm:ss"), or a complete format item (for example "{0,10:F2}") (if the format is a simple string, it is automatically surrounded by "{0:" and "}" to form a complete format item).

The following illustration shows some typical examples of label formats:



See the .NET Framework® documentation for details on Composite Formatting.

The font, color and alignment of the labels are controlled by the **LabelAppearance** property. This property is of type **TextAppearance**. For more details on the **TextAppearance** object, see *Displaying Text in a Diagram*.

The **LabelDistance** property controls the distance between the end of the major tick lines and the label.

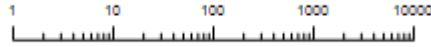
Labels can be rotated using the **AutoRotateLabels** and **LabelRotation** properties. If **AutoRotateLabels** is **true**, the labels are rotated automatically to be orthogonal to the tick lines. **LabelRotation** can be used to specify a fixed angle (in degrees) by which the labels must be rotated. Both properties can be specified at the same time, in which case the **LabelRotation** angle is added to the automatic angle.

Finally, the **SkipOverlappingLabels** property specifies that the scale can skip (that is, not draw) some labels to make sure that they do not overlap. This property is **true** by default, it can be set to **false** to make sure that all labels are drawn.

## Logarithmic Scales

Logarithmic scales are scales where each major tick interval represents a value interval that is **n** times the previous major interval. To create a logarithmic scale, set its **LogBase** property to the desired **n** value. The most commonly used log base is 10, but any positive number can be used.

The following illustration shows an example of a logarithmic scale where  $\text{LogBase} = 10$ :



When **LogBase** is non-0, the **Minimum** and **Maximum** values of the scale are automatically adjusted to the closest power or the log base, and the tick intervals are computed automatically, so the **AutoMajorTicks**, **AutoMinorTicks**, **MajorTickInterval** and **MinorTickInterval** properties are ignored.

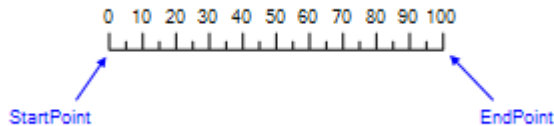
---

### Linear Scales

This section describes the properties specific to the **LinearScale** class.

#### Linear Scale Geometry

The base line of a linear scale is a straight line segment, defined by its **StartPoint** and **EndPoint** properties. Linear scales can take any direction: horizontal, vertical or oblique. The minimum value corresponds to the start point, and the maximum value corresponds to the end point.



#### Tick Direction

The direction of the ticks of a linear scale is controlled by the **TickDirection** property, and the possible values are:

- ◆ **Left:** The ticks are on the left of the base line, when looking towards the end point. For example, if the scale is horizontal, with the end point on the right of the start point, then the ticks are above the scale.
- ◆ **Right:** The ticks are on the right of the base line, when looking towards the end point. For example, if the scale is horizontal, with the end point on the right of the start point, then the ticks are below the scale.

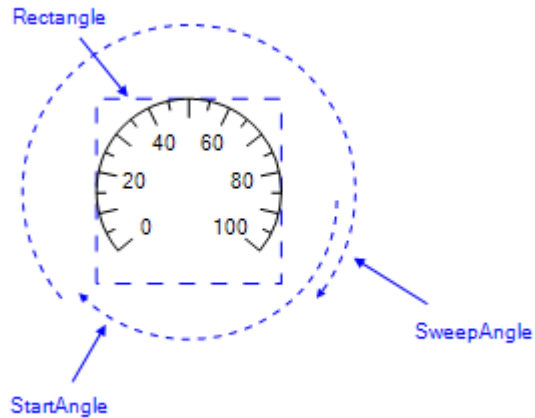
---

### Circular Scales

This section describes the properties specific to the **CircularScale** class.

### Circular Scale Geometry

The base line of a circular scale is an elliptical arc, defined by its `Rectangle`, `StartAngle` and `SweepAngle` properties.



### Tick Direction

The direction of the ticks of a circular scale is controlled by the `TickDirection` property, and the possible values are:

- ◆ Outside: The ticks are outside the elliptical arc.
- ◆ Inside: The ticks are inside the elliptical arc.

---

## Gauges

IBM® ILOG® Diagram for .NET offers predefined graphic objects that can be used for dashboard applications. These graphic objects are grouped in the `ILOG.Diagrammer.Gauges` namespace. This namespace contains several types of circular and linear gauges, various buttons, check boxes and knobs, several clocks with various styles. It also contains a basic chart object for displaying bar chart, pie or donut chart.

The following illustration shows some of the circular gauges and knobs.

## Using Predefined Graphic Objects



BlackGauge



PlasticGauge



DoubleGauge



PlasticGauge



VistaGauge



VistaKnob



TwistRoundGauge



TwistGauge

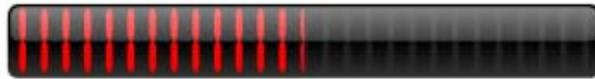


TwistKnob

The following illustration shows some linear gauges and other useful objects:



BlackLinearGauge



BlackProgressBar



MacGauge



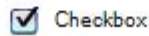
MacLinearGauge



OsXButton



BlackRadioButton



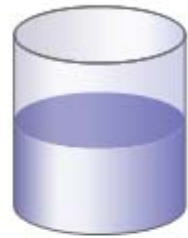
OsXCheckBox



Thermometer



TwistLinearGauge



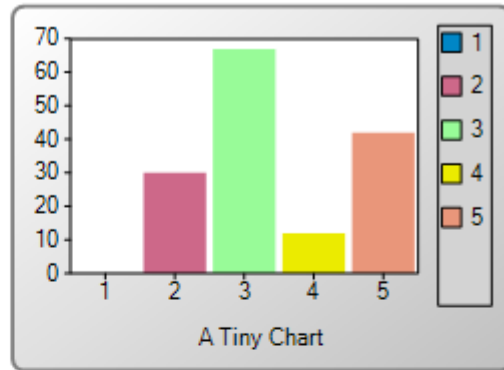
CylinderGauge

## Using Predefined Graphic Objects

The following illustration shows some clock and chart objects:



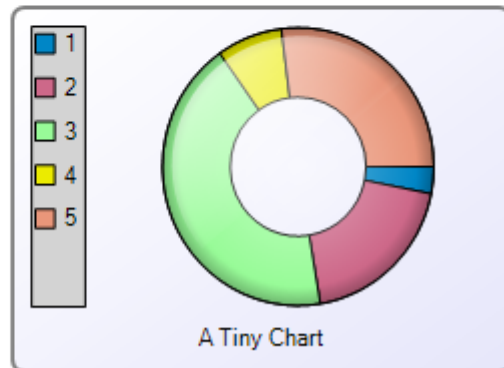
WorldClock



TinyChart



TwistClock



TinyChart

---

## User Symbols

IBM® ILOG® Diagram for .NET allows you to easily create new graphic objects by combining existing graphic object. Although you can create a new graphic object by code by subclassing the GraphicObject class, the recommended way for creating a new graphic object is to create a "user symbol". A "user symbol" is a subclass of the UserSymbol class. You do not directly create a **UserSymbol** by code, instead you can use the Diagram Designer of IBM ILOG Diagram for .NET that allows you to create your user symbol by assembling predefined graphic objects (or other user symbols that you have previously



created ) using a point-and-click editor. The Diagram Designer of IBM ILOG Diagram for .NET is a fully functional graphical editor integrated inside Visual Studio that will automatically generate the code corresponding to the user symbol that you assemble in the editor, thus you do not need to code the appearance of your symbol. Since the Diagram Designer is incorporated inside Visual Studio, you will be able to code and debug the logic of your symbol using your favorite .NET language and the IBM ILOG Diagram for .NET API.

The tutorial *Creating your First IBM ILOG Diagram for .NET Windows Forms Application* will step you through the creation of your first user symbol. You should also refer to *Building Diagrams and User Symbols Inside Visual Studio* to learn more about the Diagram Designer.



# Common Graphic Objects Rendering Features

The graphic objects of IBM® ILOG® Diagram for .NET are subclasses of the `GraphicObject` class and share a number of common features that affect the rendering of the graphic object. These features are introduced in this section.

---

## Visibility

Any graphic object can be made invisible through the `Visibility` property. The visibility of a graphic object can have three values:

- ◆ **Visible:** the object is visible. This is the default value.
- ◆ **Hidden:** the object is not visible but the area where the graphic object is displayed is still taken into account by the parent container to compute its size.
- ◆ **Collapsed:** the object is not visible but not taken into account by its parent.

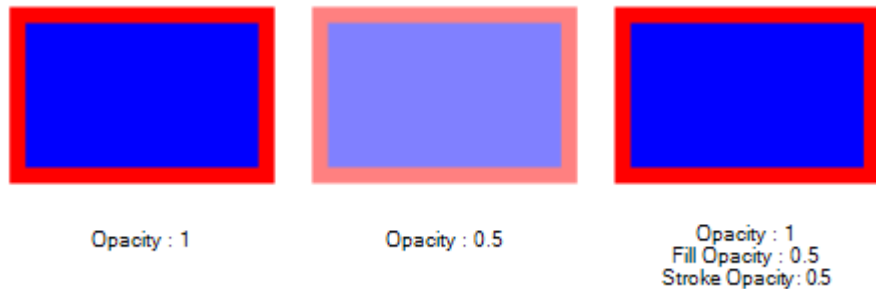
For more information on visibility options see *Understanding Graphic Object Visibility*.

## Opacity

Any graphic object can be made partially transparent through the **Opacity** property. When the value is 0, the object is completely transparent and thus not visible at all. When the value is 1, the object is completely opaque.

**Note:** In order to display a semi-transparent object, in most cases the graphic object is rendered in a temporary bitmap before rendered on the screen. Displaying a semi-transparent graphic object is time and memory consuming.

An alternative to the graphic object opacity is to use the **Opacity** property of the Fill object used to paint the graphic object, but the result may not be the same. The following illustration shows the same rectangle with various opacity values:



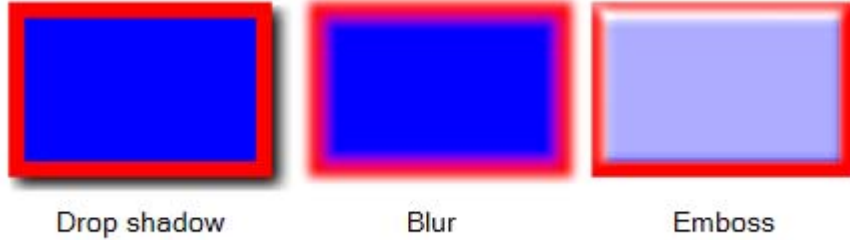
For more information on how to paint a graphic object see *Filling and Stroking Graphic Objects*.

---

## Image Filter

IBM ILOG Diagram for .NET allows you to change the appearance of a graphic object by specifying image filter effects on a graphic object.

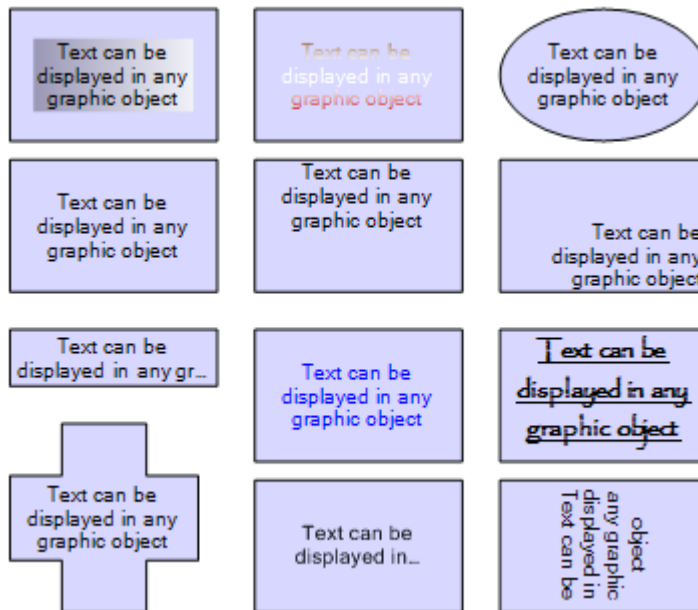
A Filter object can be specified on each graphic object through the Filter property of the GraphicObject class. A filter consists of a sequence of filter operations on an image, such as blur or lighting effect, and produces an image that can be used as the input for the next operation. The following illustration shows the same graphic object with three different image filters.



For more information see *Applying a Filter to a Graphic Object*.

## Text and Text Appearance

You can display text in any graphic object using the Text property of the GraphicObject class. The TextAppearance property allows you to control the layout and rendering options of the text inside the graphic object. The following image displays various graphic objects with text.



For more information on how to display text in a diagram see *Displaying Text in a Diagram*.

---

### Transform

Each graphic object can be rotated, zoomed, translated or skewed by specifying an affine transformation through the Transform property.

Such a coordinate transformation can be represented by a 3-row by 3-column matrix with an implicit last row of [ 0 0 1 ].

This matrix transforms source coordinates (x,y) into destination coordinates (x',y') by considering them as a column vector and multiplying the coordinate vector by the matrix according to the following process:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} m11 & m12 & dx \\ m21 & m22 & dy \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} m11*x + m12*y + dx \\ m21*x + m22*y + dy \\ 1 \end{bmatrix}$$

The following example creates a rectangle rotated by 45 degrees.

```
Rect CreateRotatedRectangle()  
{  
    Rect rectangle = new Rect(0, 0, 100, 100);  
    rectangle.Fill = new SolidFill(Color.Gray);  
    Transform t = Transform.Identity;  
    t = t.RotateAt(45, new Point2D(50,50));  
    rectangle.Transform = t;  
    return rectangle;  
}  
Function CreateRotatedRectangle() As Rect  
    Dim rectangle As Rect = New Rect(0, 0, 100, 100)  
    rectangle.Fill = New SolidFill(Color.Gray)  
    Dim t As Transform = Transform.Identity  
    t = t.RotateAt(45, New Point2D(50, 50))  
    rectangle.Transform = t  
    Return rectangle  
End Function
```

For more information on coordinate systems and transformations see *Understanding Coordinate Systems*.

---

### Clipping

Any graphic object can be clipped by any shape. A clipping path on a graphic object restricts the region to which the graphic object can be painted. All the drawings that are outside the region bounded by the currently active clipping path are not drawn.

You can specify a clipping path by creating a ClipPath object and set it through the Clip property of the GraphicObject class.

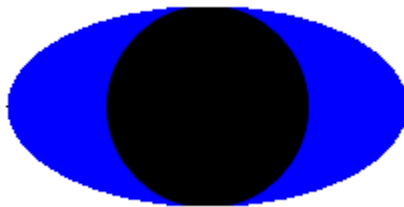
The following example creates a group containing a rectangle and a circle. The group itself is clipped with a clipping path that is an ellipse.

```
Group group = new Group();
Rect rect = new Rect(0,0, 200,200);
rect.Fill = new SolidFill(Color.Blue);
Circle circle = new Circle(100,100, 50);
circle.Fill = new SolidFill(Color.Black);
group.Objects.Add(rect);
group.Objects.Add(circle);

// clip with an ellipse
ClipPath clip = new ClipPath();
clip.Path.AddEllipse(new Rectangle2D(0, 50, 200, 100));

group.Clip = clip;
Dim group As Group = New Group
Dim rect As Rect = New Rect(0, 0, 200, 200)
rect.Fill = New SolidFill(Color.Blue)
Dim circle As Circle = New Circle(100, 100, 50)
circle.Fill = New SolidFill(Color.Black)
group.Objects.Add(rect)
group.Objects.Add(circle)
' clip with an ellipse
Dim clip As ClipPath = New ClipPath
clip.Path.AddEllipse(New Rectangle2D(0, 50, 200, 100))
group.Clip = clip
```

The group you have just created should produce the following result:







# ***Styling Graphic Objects Using Fill, Stroke and Filter Classes***

IBM® ILOG® Diagram for .NET library provides the Fill and Stroke classes that allow you to paint most of the graphic objects. By means of the Filter class it is also possible to apply filter effects on those objects.

## **In This Section**

### *Filling and Stroking Graphic Objects*

Describes how to paint the interior and the outline of graphic objects.

### *Editing Fill Objects Using the Fill Dialog Box*

Describes how to edit a Fill object using the predefined dialog box.

### *Applying a Filter to a Graphic Object*

Describes how to apply filter effects to graphic objects.

### *Editing Filter Effects Using the Filter Dialog Box*

Describes how edit the filter effects of a graphic object using the predefined dialog box.

## Filling and Stroking Graphic Objects

All basic shapes of IBM® ILOG® Diagram for .NET such as circles, ellipses, rectangles, polygons and more generally most of the graphic objects of the library can be filled and stroked. Filling an object means painting the interior of its shape while stroking an object means painting the outline of the shape.

To paint a graphic object use the **Fill** class and all its subclasses.

These classes allow you to paint the interior or the outline of a shape with a simple color, a texture defined by a raster image, various kinds of gradients or a predefined set of hatches.

To paint the outline of a shape use the **Stroke** class.

The **Stroke** class allows you to paint the outline of a shape with a line or a thick line with dashes. The **Stroke** class itself contains a **Fill** object to specify how the outline is painted.

---

### Painting the Interior of a Shape

You can paint a graphic object by means of the **Fill** class and all its subclasses. These classes allow you to paint the interior or the outline of a shape with a simple color, a texture defined by a raster image, different types of gradients or a predefined set of hatches.

For example, a **Fill** object is used to paint the interior of an ellipse or the foreground of a text.

### Painting with a Solid Color

A **SolidFill** object is used to paint an area with a solid **Color**. The **SolidFill** object can be created by specifying a **Color** object or the red, green and blue components.

The following example shows how to paint an **Ellipse** object with a **SolidFill**.

```
Ellipse ellipse = new Ellipse(0, 0, 100, 100);
SolidFill myFill = new SolidFill(Color.Blue);
ellipse.Fill = myFill;
Dim ellipse As Ellipse = New Ellipse(0, 0, 100, 100)
Dim myFill As SolidFill = New SolidFill(Color.Blue)
ellipse.Fill = myFill
```

This is how the **Ellipse** object looks like.



### Painting with a Linear Gradient

A `LinearGradientFill` allows you to paint an area with a gradient allong a line.

The `P1` and `P2` properties define the start and end point of the line. The two points define the axis of the gradient and allow you to create a gradient with any type of angle.

By default, **P1** and **P2** are specified in the coordinate space of the object, this means that a point of (0, 0) corresponds to the top-left corner of the object bounding area while a value of (1, 1) corresponds to the bottom-right corner.

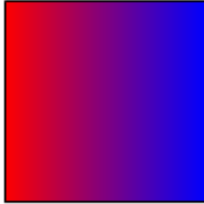
The linear gradient can have two or more colors. The start and end colors of the gradient are defined by the `StartColor` and `EndColor` property. Additional colors between **P1** and **P2** can be specified by adding a `GradientStop` object to the gradient through the `GradientStops` property. A **GradientStop** holds the color and the location of this color between **P1** and **P2**.

The following example shows how to create a horizontal linear gradient from red to blue and how to use it to fill a `Rect` object.

```
Rect rectangle = new Rect(0, 0, 100, 100);
LinearGradientFill myFill = new LinearGradientFill();
myFill.P1 = new Point2D(0, 0);
myFill.P2 = new Point2D(1, 0);
myFill.StartColor = Color.Red;
myFill.EndColor = Color.Blue;
rectangle.Fill = myFill;
Dim rectangle As Rect = New Rect(0, 0, 100, 100)
Dim myFill As LinearGradientFill = New LinearGradientFill
myFill.P1 = New Point2D(0, 0)
myFill.P2 = New Point2D(1, 0)
myFill.StartColor = Color.Red
myFill.EndColor = Color.Blue
rectangle.Fill = myFill
```

## Styling Graphic Objects Using Fill, Stroke and Filter Classes

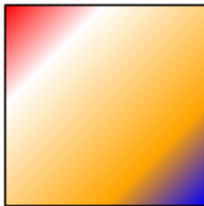
This code will produce the following result:



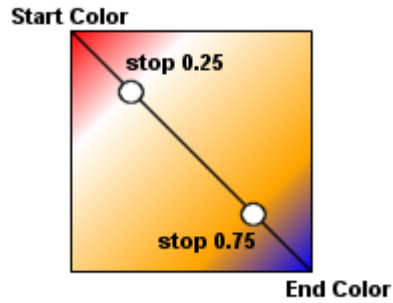
This following example shows how to create a linear gradient with additional gradient colors. The **P1** and **P2** properties are set so that the gradient line starts at the top-left corner and ends at the bottom-right corner:

```
Rect rectangle = new Rect(0, 0, 100, 100);
LinearGradientFill myFill = new LinearGradientFill();
myFill.P1 = new Point2D(0, 0);
myFill.P2 = new Point2D(1, 1);
myFill.StartColor = Color.Red;
myFill.EndColor = Color.Blue;
myFill.GradientStops.AddRange(
    new GradientStop[] {
        new GradientStop(Color.White, 0.25f),
        new GradientStop(Color.Orange, 0.75f) });
rectangle.Fill = myFill;
Dim rectangle As Rect = New Rect(0, 0, 100, 100)
Dim myFill As LinearGradientFill = New LinearGradientFill
myFill.P1 = New Point2D(0, 0)
myFill.P2 = New Point2D(1, 1)
myFill.StartColor = Color.Red
myFill.EndColor = Color.Blue
myFill.GradientStops.AddRange( _
    New GradientStop() { _
        New GradientStop(Color.White, 0.25F), _
        New GradientStop(Color.Orange, 0.75F)})
rectangle.Fill = myFill
```

This code will produce the following result:

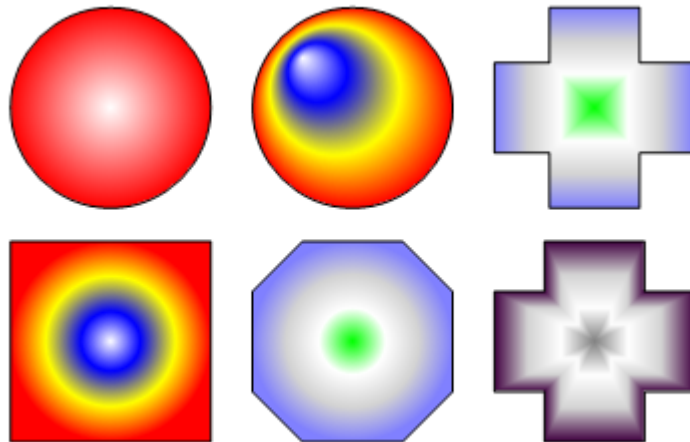


The following illustration shows the axis and the gradient stops in the previous gradient.



### Painting with a Path Gradient

A `PathGradientFill` allows you to paint an area with a gradient from the outline of a path to the center of the path. The following illustration shows various path gradients.



**PathGradientFill** can have various shapes defined by the `GradientShape` property, such as ellipse, circle or rectangle. When the **PathGradientFill** is used to fill a graphic object, the shape can be the one of the graphic object itself.

The start and end colors of the gradient are defined by the `StartColor` and `EndColor` properties. The **StartColor** will be used for the outline of the shape and the **EndColor** for the center of the shape. By means of the `GradientStops` property, you can add additional colors between the outline and the center by adding `GradientStop` object to the gradient. A **GradientStop** holds the color and the location of this color between the outline and the center.

It is possible to change the location of the center of the gradient through the `CenterPoint` property.

### Painting with a Texture

A `TextureFill` allows you to paint an area with an image. You specify the image through the `Image` property. **TextureFill** contains a predefined set of images for texture representing various things such as wood, granite, marble and so on. The image fills the area to paint according to the `Bounds` property. This property defines the location and size of the tiles to be painted. The tiling theoretically extends a series of such rectangles to infinity in X and Y (positive and negative), with rectangles starting at  $(x + m * \text{width}, y + n * \text{height})$  for each possible integer value for `m` and `n`. When the value of `CoordinateSpace` property is **ObjectBoundingBox** (which is the default value), the value of the `Bounds` property must be interpreted in the coordinate system of the graphic object being painted. In this coordinate system (0, 0) is mapped to the top-left corner of the graphic object and (1, 1) to the bottom-right corner of the graphic object. When the `Bounds` property is (0, 0, 1, 1) the image is stretched to fit the graphic object bounding box.

The following example fills a `Rect` object with a **TextureFill**:

```
Rect rectangle = new Rect(0, 0, 100, 100);
TextureFill myFill = new TextureFill();
myFill.Bounds = new Rectangle2D(0, 0, 1, 1);
myFill.Image = TextureFill.Wood1;

rectangle.Fill = myFill;
return rectangle;
Dim rectangle As Rect = New Rect(0, 0, 100, 100)
Dim myFill As TextureFill = New TextureFill
myFill.Bounds = New Rectangle2D(0, 0, 1, 1)
myFill.Image = TextureFill.Wood1
rectangle.Fill = myFill
Return rectangle
```

This code will produce the following result:



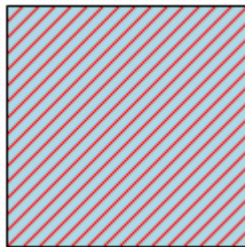
### Painting with Hatches

A HatchFill allows you to paint an area with a predefined set of hatches. The hatch style is specified by the HatchStyle property. ForeColor represents the color of the hatch and BackColor the color of the background.

The following example shows how to create a rectangle filled with a **HatchFill**:

```
Rect rectangle = new Rect(0, 0, 100, 100);
HatchFill myFill = new HatchFill();
myFill.BackColor = Color.LightBlue;
myFill.ForeColor = Color.Red;
myFill.HatchStyle = HatchStyle.BackwardDiagonal;
rectangle.Fill = myFill;
return rectangle;
Dim rectangle As Rect = New Rect(0, 0, 100, 100)
Dim myFill As HatchFill = New HatchFill
myFill.BackColor = Color.LightBlue
myFill.ForeColor = Color.Red
myFill.HatchStyle = HatchStyle.BackwardDiagonal
rectangle.Fill = myFill
Return rectangle
```

This code will produce the following result:



### Common Fill Features

#### *Fill Opacity*

Each Fill subclass allows you to specify opacity through the Opacity property. The value of the opacity can be from 0 to 1. Value 0 means that the fill is completely transparent, value 1 means that the fill is completely opaque. If a color is specified in the **Fill** object as an alpha value, then the opacity of the color is multiplied with the opacity of the fill.

**Note:** *The opacity of the fill is different from the opacity of a graphic object (**Opacity** property of **GraphicObject**). It is more efficient to use the opacity of the fill than the opacity of the graphic object.*

#### *Immutable Fill*

By default, a **Fill** object is mutable. When you modify the properties of the **Fill** object, for example the colors of a gradient, every graphic object that uses this **Fill** object is notified and repainted. Any **Fill** object can be made immutable by calling the Freeze method. If a **Fill** object is not mutable, an exception will be thrown if you try to change a property. The advantage of having a **Fill** object non mutable is that graphic objects using this fill no longer need to listen to changes of the fill. This implies better performances and less memory consumption.

### Painting the Outline of a Shape

The Stroke object contains all the information necessary to paint the outline of a shape. In a **Stroke** object you find:

- ◆ A **Fill** object that represents the color, gradient or texture used to paint the outline of the shape specified by the Fill property.
- ◆ The thickness of the outline specified by the Width property.
- ◆ The ability to draw dashed lines though the DashStyle, DashOffset, DashPattern and DashCap properties.

The following example shows how to create a stroked rectangle with dashed lines:

```
Rect CreateRectangleWithStroke()  
{  
  
    Rect rect = new Rect(320F, 150F, 200F, 130F);  
    rect.Fill = new SolidFill(Color.WhiteSmoke);  
  
    // Create the Stroke object  
  
    Stroke stroke = new Stroke();  
    stroke.DashCap = DashCap.Triangle;  
    stroke.DashStyle = DashStyle.Dash;  
    stroke.Fill = new SolidFill(Color.LightBlue);  
    stroke.Width = 6F;  
}
```



```

        // Sets the stroke in the rect
        rect.Stroke = stroke;

        return rect;
    }
Function CreateRectangleWithStroke() As Rect

    Dim rect As Rect = New Rect(320F, 150F, 200F, 130F)
    rect.Fill = New SolidFill(Color.WhiteSmoke)

    ' create the Stroke object

    Dim stroke As Stroke = New Stroke
    stroke.DashCap = DashCap.Triangle
    stroke.DashStyle = DashStyle.Dash
    stroke.Fill = New SolidFill(Color.LightBlue)
    stroke.Width = 6F

    ' Sets the stroke in the rect
    rect.Stroke = stroke

    Return rect
End Function

```

The code produces the following result:



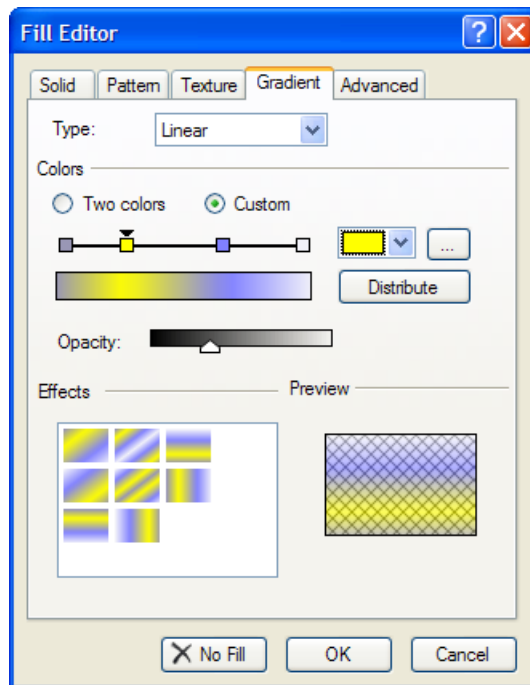
### ***Immutable Stroke***

By default, a **Stroke** object is mutable. When you modify the properties of the **Stroke**, for example the line width, every graphic object that uses this **Stroke** is notified and repainted. Any **Stroke** object can be made immutable by calling the Freeze method. If a **Stroke** object is not mutable, an exception will be thrown if you try to change a property. The advantage of having a **Stroke** object non mutable is that graphic objects using this stroke no longer need to listen to changes of the stroke. This implies better performances and less memory consumption.

## Editing Fill Objects Using the Fill Dialog Box

In IBM® ILOG® Diagram for .NET the interior of a shape is painted using a **Fill** object. Subclasses of the **Fill** class are the **SolidFill**, to paint with a solid color, **LinearGradientFill** and **PathGradientFill** to paint with a gradient, **TextureFill** to paint with a texture and **HatchFill** to paint hatches.

IBM ILOG Diagram for .NET provides a predefined dialog box that allows you to edit a **Fill** object and choose between the various possibilities. This dialog box is defined by the class **FillDialog** located in the **ILOG.Diagrammer.Windows.Forms** namespace. The following illustration shows the dialog box editing a linear gradient:



The dialog box proposes several tabs to select the type of fill. Optionally, an Advanced tab allows you to modify all the properties of the **Fill** object through a property sheet.

Here is a method that edits a fill through the dialog box:

```
Fill EditFill(Fill fill)
{
    // Create the dialog
    FillDialog dialog = new FillDialog();

    // Specify the fill to edit
```

```

    if (fill != null)
        dialog.Fill = fill;

    // Show the dialog
    if (dialog.ShowDialog() == System.Windows.Forms.DialogResult.OK)
    {
        // Returns the resulting fill
        return dialog.Fill;
    }
    // In other cases returns the old fill
    return fill;
}
Function EditFill(ByVal fill As Fill) As Fill
    ' Create the dialog
    Dim dialog As FillDialog = New FillDialog

    ' Specify the fill to edit
    If Not (fill Is Nothing) Then
        dialog.Fill = fill
    End If

    ' Show the dialog
    If dialog.ShowDialog = System.Windows.Forms.DialogResult.OK Then
        ' returns the resulting fill
        Return dialog.Fill
    End If
    ' In other case return the old fill
    Return fill
End Function

```

---

## Applying a Filter to a Graphic Object

For each graphic object you can apply filter effects that consist of a series of graphic operations that are applied to the original drawing of the graphic object and produce a modified graphical result. When a filter effect is specified on a graphic object, the result of the filter effect is rendered on the target device instead of the original rendering of the graphic object.

The following illustration shows the process:



A filter effect on a graphic object is specified using the `Filter` class. You can set a filter effect through the `Filter` property of the `GraphicObject` class.

The filter effects that can be applied are inspired from those defined in the Scalable Vector Graphics (SVG) specification of the W3C.

A **Filter** object consists of a series of filter operations. Each operation performs a single fundamental graphical operation such as a blur or a lighting effect and produces a resulting image that can be used as the input for the next operation. The class `FilterEffect` is the abstract base class for these operations.

Source graphic and the source alpha are two important notions.

The source graphic represents the initial drawing of the graphic object to which the filter is applied.

The source alpha represents only the alpha channel of the source graphics.

The various filter primitive operations are defined by the following subclasses of the **FilterEffect** class:

- ◆ **FeBlend**: This filter composites two images together using commonly used imaging software blending modes.
- ◆ **FeColorMatrix**: Applies a matrix transformation to the RGBA colors and alpha values of every pixel of the input image.
- ◆ **FeComponentTransfer**: Applies a component-wise remapping of every pixel of the input image and can be used for operations like brightness adjustment, contrast adjustment, color balance or threshold.
- ◆ **FeComposite**: This filter performs the combination of the two input images pixel-wise in image space using one of the Porter-Duff composition operations: `over`, `in`, `atop`, `out`, `xor`. Additionally, a component-wise arithmetic operation (with the result clamped between [0..1]) can be applied.
- ◆ **FeConvolveMatrix**: applies a matrix convolution filter effect. A convolution combines pixels in the input image with neighboring pixels to produce a resulting image.
- ◆ **FeDiffuseLighting**: This filter primitive lights an image using the alpha channel as a bump map. The resulting image is an RGBA opaque image based on the light color with  $\alpha = 1.0$  everywhere. The lighting calculation follows the standard diffuse component of the Phong lighting model.
- ◆ **FeDisplacementMap**: This filter primitive uses the pixels values from the second image to displace the image from the first input image.
- ◆ **FeFlood**: This filter primitive creates a rectangle filled with a specified color and opacity.
- ◆ **FeGaussianBlur**: This filter primitive performs a Gaussian blur on the input image.

- ◆ **FeImage**: This filter primitive refers to another graphic object or raster image, which is rendered into an RGBA raster and becomes the result of the filter primitive.
- ◆ **FeMerge**: This filter primitive merges input image layers on top of each other using the over operator.
- ◆ **FeMorphology**: This filter primitive performs fattening or thinning of artwork. It is particularly useful for fattening or thinning an alpha channel.
- ◆ **FeOffset**: This filter primitive offsets the input image relative to its current position in the image space by the specified vector.
- ◆ **FeSpecularLighting**: This filter primitive lights a source graphic using the alpha channel as a bump map. The resulting image is an RGBA image based on the light color. The lighting calculation follows the standard specular component of the Phong lighting model.
- ◆ **FeTile**: This filter primitive fills a target rectangle with a repeated, tiled pattern of an input image.
- ◆ **FeTurbulence**: This filter primitive creates an image using the Perlin turbulence function. It enables the synthesis of artificial textures like clouds or marble. For a detailed description of the Perlin turbulence function, see *Texturing and Modeling*, Ebert et al, AP Professional, 1994.

The **Filter** and **FilterEffect** classes are located in the `ILOG.Diagrammer` namespace, subclasses of the **FilterEffect** class are in the `ILOG.Diagrammer.Filters`.

---

### An Example of Filter Effect

The following example shows how to create an embossing and a drop shadow effect.

```
// Create the filter
Filter filter = new Filter();
filter.FilterRegion = new Rectangle2D(0f, 0f, 1.2f, 1.2f);

// Gaussian blur on source alpha
FeGaussianBlur blur = new FeGaussianBlur(4f, 4f);
blur.In = new FeSourceAlpha();

// Offset the blur by 4,4
FeOffset offsetBlur = new FeOffset(4f, 4f);
offsetBlur.In = blur;

// Lighting on blur
FeSpecularLighting lighting
    = new FeSpecularLighting(5f, 0.9f, 20f, new DistantLight(225, 30,
    Color.White));

// Limit lighting to non zero source alpha
FeComposite feComposite1 = new FeComposite(CompositionOperator.In);
feComposite1.Inputs.Add(lighting);
feComposite1.Inputs.Add(new FeSourceAlpha());
```

## Styling Graphic Objects Using Fill, Stroke and Filter Classes

```
// Compose lighting and source graphic
FeComposite feComposite2 = new FeComposite(CompositionOperator.Arithmetic);
feComposite2.K2 = 1F;
feComposite2.K3 = 1F;
feComposite2.Inputs.Add(new FeSourceGraphic());
feComposite2.Inputs.Add(feComposite1);

// Merge the result and the shadow
FeMerge feMerge = new FeMerge();
feMerge.Inputs.Add(offsetBlur);
feMerge.Inputs.Add(feComposite2);

// Add effects in the filter
filter.Effects.Add(blur);
filter.Effects.Add(offsetBlur);
filter.Effects.Add(lightning);
filter.Effects.Add(feComposite1);
filter.Effects.Add(feComposite2);
filter.Effects.Add(feMerge);
' Create the filter
Dim filter As Filter = New Filter
filter.FilterRegion = New Rectangle2D(0F, 0F, 1.2F, 1.2F)

' Gaussian blur on source alpha
Dim blur As FeGaussianBlur = New FeGaussianBlur(4F, 4F)
blur.In = New FeSourceAlpha

' Offset the blur by 4,4
Dim offsetBlur As FeOffset = New FeOffset(4F, 4F)
offsetBlur.In = blur

' Lighting on blur
Dim lighting As FeSpecularLighting = _
    New FeSpecularLighting(5F, 0.9F, 20F, New DistantLight(225, 30,
Color.White))

' Limit lighting to non zero source alpha
Dim feComposite1 As FeComposite = New FeComposite(CompositionOperator.In)
feComposite1.Inputs.Add(lighting)
feComposite1.Inputs.Add(New FeSourceAlpha())

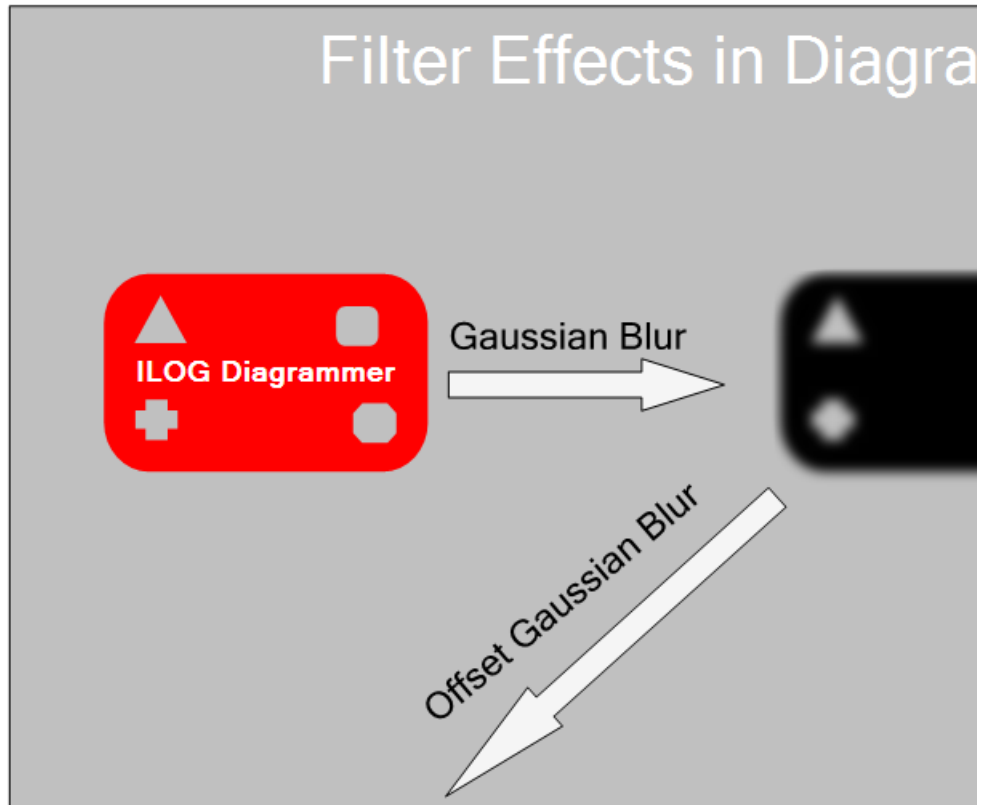
' Compose lighting and source graphic
Dim feComposite2 As FeComposite = New
FeComposite(CompositionOperator.Arithmetic)
feComposite2.K2 = 1F
feComposite2.K3 = 1F
feComposite2.Inputs.Add(New FeSourceGraphic())
feComposite2.Inputs.Add(feComposite1)

' Merge the result and the shadow
Dim feMerge As FeMerge = New FeMerge
feMerge.Inputs.Add(offsetBlur)
feMerge.Inputs.Add(feComposite2)

' Add effects in the filter
filter.Effects.Add(blur)
filter.Effects.Add(offsetBlur)
```

```
filter.Effects.Add(lightning)
filter.Effects.Add(feComposite1)
filter.Effects.Add(feComposite2)
filter.Effects.Add(feMerge)
```

The following illustration shows how this filter is applied to a graphic object:



Computing a filter effect on a graphic object can be time consuming. The time to compute the filter depends on the number and type of filter effects primitives. It also depends on the size of the graphic object on the output device and the resolution used for intermediate images.

One easy way to accelerate the time needed to compute a filter effect is to specify a maximum resolution for the filter. This can be done through the Resolution property of the Filter class. This property specifies the maximum size in pixels of intermediate images used to compute the filter effect results. The default value is set to -1,-1 which means that the filter will use the most appropriate size to get the best rendering result, of course a small size

for intermediate images will result in the graphic object being pixelized on screen when the object is zoomed above this rendering size.

**Notes:**

1. A filter object can be shared by several graphic objects.
2. IBM ILOG Diagram for .NET contains a predefined dialog box for editing the filter effects of a graphic object. This dialog box is described in *Editing Filter Effects Using the Filter Dialog Box*.

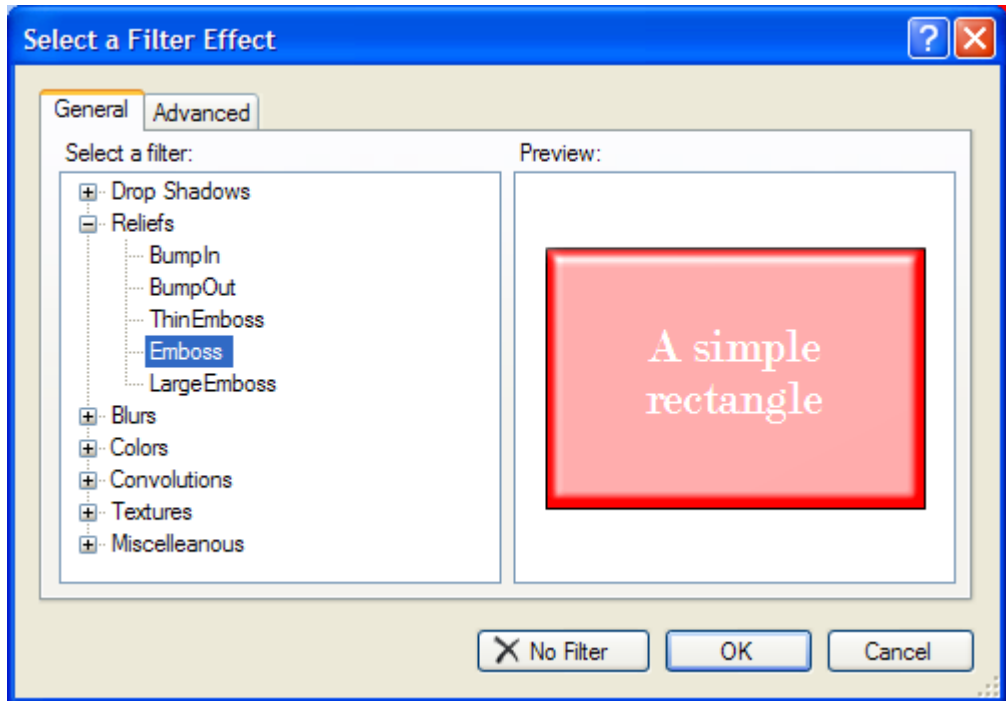
---

### Editing Filter Effects Using the Filter Dialog Box

Each graphic object of IBM® ILOG® Diagram for .NET can be associated with filter effects defined by the Filter class. For more details on applying filter effects on a graphic object, see *Applying a Filter to a Graphic Object*.

IBM ILOG Diagram for .NET provides a predefined dialog box that proposes a set of predefined filter effects and allows you to edit a **Filter** object. This dialog box is defined by the class FilterDialog located in the ILOG.Diagrammer.Windows.Forms namespace. The following illustration shows the dialog box that previews an emboss filter on a rectangle object:





The General tab displays a tree of predefined filters. The Advanced tab allows you to modify all the properties of the **Filter** object through a property sheet.

Here is a method that edits a filter through the dialog box:

```

Filter EditFilter(Filter filter, GraphicObject preview)
{
    // Create the dialog
    FilterDialog dialog = new FilterDialog();

    // Specify the filter to edit
    dialog.Filter = filter;

    // Specify the graphic object used to preview the result
    dialog.PreviewObject = preview;

    // Show the dialog
    if (dialog.ShowDialog() == DialogResult.OK)
    {
        // returns the resulting filter
        return dialog.Filter;
    }
    // in other cases returns the old filter
    return filter;
}
Function EditFilter(ByVal filter As Filter, ByVal preview As GraphicObject) As
Filter

```

## Styling Graphic Objects Using Fill, Stroke and Filter Classes

```
' Create the dialog
Dim dialog As FilterDialog = New FilterDialog

' Specify the filter to edit

dialog.Filter = filter

' Specify the graphic object used to preview the result
dialog.PreviewObject = preview

' Show the dialog
If dialog.ShowDialog = DialogResult.OK Then

    ' returns the resulting filter
    Return dialog.Filter

End If

' in other cases returns the old filter
Return filter
End Function
```

## *Displaying Text in a Diagram*

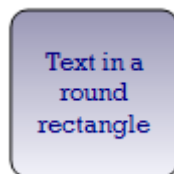
IBM® ILOG® Diagram for .NET offers various ways to display textual information in a diagram. You may display simple multiline text inside a graphic object or create more complex styled text using dedicated graphic objects such as the Text or TextOnPath objects.

---

### Displaying Text in a Graphic Object

A graphic object can display textual information specified by the Text property of the GraphicObject class. The text that you specify through the **Text** property is fitted in the bounding rectangle of the graphic object. Many options allow specifying the alignment of the text within this rectangle, the margin, the colors as well as various ways to wrap the text in the rectangle.

The following illustration shows a text wrapped within a rectangular object.

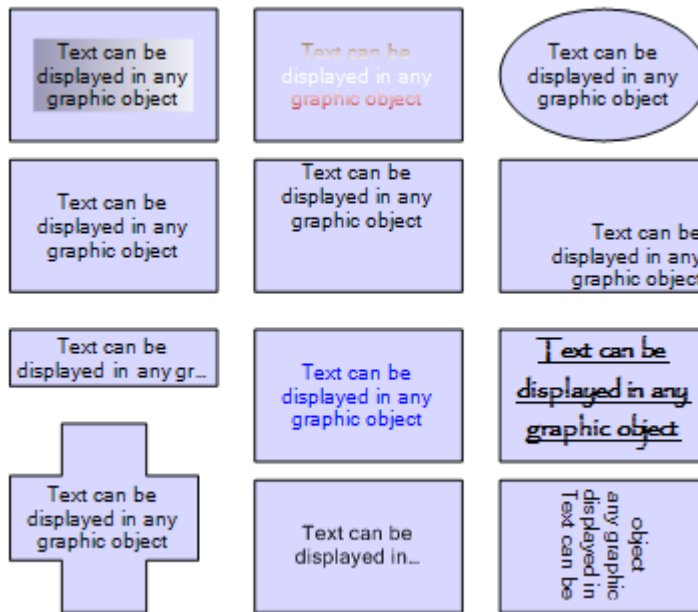


## Displaying Text in a Diagram

Once the text is specified through the **Text** property, the various options that allow controlling the appearance of the text is defined by the `TextAppearance` class.

**TextAppearance** regroups the properties to specify the style of the text such as the `Font`, the `Background` and `Foreground` properties as well as properties to specify the way the text is laid out inside bounding rectangle of the graphic object such as the `Margins`, the `HorizontalAlignment`, the `VerticalAlignment` or the `Trimming` properties.

The following shows some of the **TextAppearance** options:



Every **GraphicObject** holds a **TextAppearance** instance in its **TextAppearance** property. Like other styling objects such as the `Fill` or `Stroke` objects, the same **TextAppearance** instance can be used by several objects allowing sharing the same text style in several graphic objects.

The following code creates an `Ellipse` object displaying some text inside the ellipse.

```
public Ellipse CreateEllipseWithText ()
{
    Ellipse ellipse = new Ellipse(520F, 160F, 140F, 120F);
    ellipse.Fill = new SolidFill(Color.WhiteSmoke);
    ellipse.Text = "Welcome to IBM ILOG Diagram for .NET";

    TextAppearance appearance = new TextAppearance();
    appearance.Font = new System.Drawing.Font("Arial", 12F);
    appearance.Foreground = new SolidFill(Color.Blue);
    appearance.Margins = new ILOG.Diagrammer.Margins(5f);
}
```

```

        appearance.VerticalAlignment = VerticalTextAlignment.Center;
        appearance.HorizontalAlignment = HorizontalTextAlignment.Center;

        ellipse.TextAppearance = appearance;
        return ellipse;
    }
    Public Function CreateEllipseWithText() As Ellipse

        Dim ellipse As Ellipse = New Ellipse(520F, 160F, 140F, 120F)
        ellipse.Fill = New SolidFill(Color.WhiteSmoke)
        ellipse.Text = "Welcome to IBM ILOG Diagram for .NET"

        Dim appearance As TextAppearance = New TextAppearance
        appearance.Font = New System.Drawing.Font("Arial", 12F)
        appearance.Foreground = New SolidFill(Color.Blue)
        appearance.Margins = New ILOG.Diagrammer.Margins(5F)
        appearance.VerticalAlignment = VerticalTextAlignment.Center
        appearance.HorizontalAlignment = HorizontalTextAlignment.Center

        ellipse.TextAppearance = appearance
        Return ellipse
    End Function

```

The ellipse you have just created will look like the following:



**Note:** Some graphic objects such as the `Rect` object that simply displays a rectangle have an `AutoSize` property. When a **`Rect`** object has its **`AutoSize`** property set to **`true`**, then the object automatically computes its size depending on the text that it displays. This feature can be particularly useful when creating a custom graphic object.

The example below shows how to use a `StackPanel` that layouts vertically several rectangles containing text to create a kind of table of text.

```
private static StackPanel CreateStackPanelWithRectangles()
{
    StackPanel stackPanel = new StackPanel();
    stackPanel.Rectangle = new Rectangle2D(0, 0, 100, 100);
    stackPanel.AutoSize = true;
    stackPanel.Border = new Stroke(Color.Black);
    stackPanel.CornerRadius = new CornerRadius(6F);

    Rect header = new Rect();
    header.Text = "Header";
    header.AutoSize = true;
    header.Fill = new LinearGradientFill(new Point2D(0, 0),
                                        new Point2D(0, 1),
                                        Color.Silver, Color.White);

    Rect rect1 = new Rect();
    rect1.Text = "Text 1";
    rect1.Fill = null;
    rect1.AutoSize = true;

    Rect rect2 = new Rect();
    rect2.Fill = null;
    rect2.AutoSize = true;
    rect2.Text = "Text 2";

    Rect rect3 = new Rect();
    rect3.Fill = null;
    rect3.AutoSize = true;
    rect3.Text = "Text 3";
    stackPanel.Objects.AddRange(new GraphicObject[] { header, rect1, rect2,
rect3 });

    return stackPanel;
}
Private Shared Function CreateStackPanelWithRectangles() As StackPanel

    Dim stackPanel As StackPanel = New StackPanel
    stackPanel.Rectangle = New Rectangle2D(0, 0, 100, 100)
    stackPanel.AutoSize = True
    stackPanel.Border = New Stroke(Color.Black)
    stackPanel.CornerRadius = New CornerRadius(6F)

    Dim header As Rect = New Rect
```

```

header.Text = "Header"
header.AutoSize = True
header.Fill = New LinearGradientFill(New Point2D(0, 0), New Point2D(0, 1), _
    Color.Silver, Color.White)

Dim rect1 As Rect = New Rect
rect1.Text = "Text 1"
rect1.Fill = Nothing
rect1.AutoSize = True

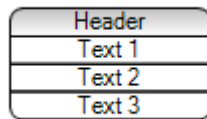
Dim rect2 As Rect = New Rect
rect2.Fill = Nothing
rect2.AutoSize = True
rect2.Text = "Text 2"

Dim rect3 As Rect = New Rect
rect3.Fill = Nothing
rect3.AutoSize = True
rect3.Text = "Text 3"

stackPanel.Objects.AddRange(New GraphicObject() {header, rect1, rect2,
rect3})
Return stackPanel
End Function

```

The stack panel you have just created should look like the following one:



For more information on how to layout objects with Panels read the section *Panels*.

---

## Displaying Text using Text and TextOnPath Objects

You can also display textual information in a diagram using dedicated objects like the `Text` and `TextOnPath`.

The `Text` class allows you to display text at a specified location. The location of the text depends on the `TextLocation` property that specifies the anchor point of the text and the `VerticalAlignment` and `HorizontalAlignment` properties that specify how the text is aligned with respect to this anchor point.

In addition to text drawn along a straight line, IBM ILOG Diagram for .NET also includes to specify text drawn along a path. To specify a block of text to be drawn along a path you use the `TextOnPath` object and specify the path through its `Path` property.

The following illustration shows a **Text** and a **TextOnPath** object.



Both **Text** and **TextOnPath** objects can be filled and stroked. You stroke the outline of each letter using the `Stroke` property and fill the content using the `Fill` property.

The following example creates the **Text** and **TextOnPath** objects.

```
GraphicObject[] CreateTextAndTextOnPath()
{
    TextOnPath textOnPath = new TextOnPath();
    textOnPath.Text = "IBM ILOG Diagram for .NET";
    textOnPath.Font = new Font("Garamond", 35F, FontStyle.Bold);

    LinearGradientFill gradient
        = new LinearGradientFill(new Point2D(0, 0),
                                new Point2D(0, 1),
                                Color.Yellow,
                                Color.Red);

    textOnPath.Fill = gradient;
    textOnPath.Stroke = new Stroke(Color.Olive);

    textOnPath.Path.SetGeometry(
        "M100 200 C 100 100 250 100 250 200 S400 300 400 200");

    Text text = new Text();
    text.Text = "IBM ILOG Diagram for .NET";
    text.TextLocation = new Point2D(70f, 330f);
    text.Fill = gradient;
    text.Stroke = textOnPath.Stroke;
    text.Font = textOnPath.Font;
    text.HorizontalAlignment = HorizontalTextAlignment.Left;
    text.VerticalAlignment = VerticalTextAlignment.Center;

    return new GraphicObject[] { text, textOnPath };
}
```



```

}
Function CreateTextAndTextOnPath() As GraphicObject()
    Dim textOnPath As TextOnPath = New TextOnPath
    textOnPath.Text = "IBM ILOG Diagram for .NET"
    textOnPath.Font = New Font("Garamond", 35.0F, FontStyle.Bold)

    Dim gradient As LinearGradientFill = _
        New LinearGradientFill(New Point2D(0, 0), _
            New Point2D(0, 1), _
            Color.Yellow, _
            Color.Red)

    textOnPath.Fill = gradient
    textOnPath.Stroke = New Stroke(Color.Olive)
    textOnPath.Path.SetGeometry(_
        "M100 200 C 100 100 250 100 250 200 S400 300 400 200")
    Dim text As Text = New Text

    text.Text = "IBM ILOG Diagram for .NET"
    text.TextLocation = New Point2D(70.0F, 330.0F)
    text.Fill = gradient
    text.Stroke = textOnPath.Stroke
    text.Font = textOnPath.Font
    text.HorizontalAlignment = HorizontalTextAlignment.Left
    text.VerticalAlignment = VerticalTextAlignment.Center

    Return New GraphicObject() {text, textOnPath}
End Function

```

Textual information can also be displayed in the links of a graph by the Link object. Please refer to the section *Link Objects*.

## Displaying Text in a Diagram

## Understanding Graphic Object Visibility

The visibility of a graphic object can be changed by using the `Visibility` property of the `GraphicObject` class. The type of this property is given by the `Visibility` enumeration. The values for this enumeration are defined in the following table:

Value	Description
Visible	The graphic object is visible.
Hidden	The graphic object is not visible.
Collapsed	The graphic object is not visible and its bounds are no longer used

The difference between **Hidden** and **Collapsed** is that when an object visibility is set to **Collapsed**, its geometry is no longer used during bounding boxes computations.

The following example illustrates this difference:

```
Group container = new Group();
Ellipse ellipse = new Ellipse(0, 0, 100, 100);
Rect rectangle = new Rectangle(200, 200, 100, 100);
container.Objects.Add(ellipse);
container.Objects.Add(rect);

Console.WriteLine(container.Bounds);
// Output : 0, 0, 300, 300
```

## Understanding Graphic Object Visibility

```
rectangle.Visibility = Visibility.Hidden;
Console.WriteLine(container.Bounds);
// Output : 0, 0, 300, 300

rectangle.Visibility = Visibility.Collapsed;
Console.WriteLine(container.Bounds);
// Output : 0, 0, 100, 100
Dim container As Group = New Group
Dim ellipse As Ellipse = New Ellipse(0, 0, 100, 100)
Dim rectangle As Rect = New Rect(200, 200, 100, 100)
container.Objects.Add(ellipse)
container.Objects.Add(rect)

Console.WriteLine(container.Bounds)
' Output : 0, 0, 300, 300

rectangle.Visibility = Visibility.Hidden
Console.WriteLine(container.Bounds)
' Output : 0, 0, 300, 300

rectangle.Visibility = Visibility.Collapsed
Console.WriteLine(container.Bounds)
' Output : 0, 0, 100, 100
```

The **Visibility** property is just a hint to specify the visibility of a graphic object. The real visibility is given by the **IsVisible** and **IsCollapsed** properties, which can be overridden. As a consequence, an object is said to be visible only when its **IsVisible** property returns **true**. Similarly, an object is said to be collapsed only when its **IsCollapsed** property returns **true**.

An object can be collapsed because its geometry cannot be computed. For example, a Polyline with zero points is collapsed, even though its **Visibility** property is set to **Visible**.

When an object visibility is changed, the associated graphic event must contain the **VisibilityMask** value. See *Raising Graphic Object Events* for details.





## *Using Graphic Object Preferred Size*

A graphic object that has content to display may have a preferred size, that is, a size at which its content is rendered at best. For example, a rectangular graphic object displaying text has a preferred size that enables the text to be displayed entirely.

By using the preferred size of graphic objects you are able to build complex hierarchies of graphic objects using panels or other types of containers.

The `AutoSize` property is used to specify whether a graphic object should use its preferred size. The following code shows how to create a `Rect` object displaying a text inside. The specific size of the **Rect** object is not given. Instead, the size is automatically computed by setting the **AutoSize** property:

```
Rect r = new Rect();
r.Text = "This is an auto sized Rect";
t.TextAppearance.Margins = new Margins(10);
r.AutoSize = true;
Dim r as Rect = new Rect()
r.Text = "This is an auto sized Rect"
t.TextAppearance.Margins = new Margins(10)
r.AutoSize = True
```

The **AutoSize** property is just a hint that allows you to specify if an object should take its preferred size, the real properties are the `HasPreferredWidth` and `HasPreferredHeight` properties. These properties can be overridden in subclasses in order to implement specific behaviors.

**Note:** Even though the **AutoSize** property has been set, the container of the graphic object has the responsibility of resizing it. As a consequence, setting the **AutoSize** property may not be enough to have the graphic object resized to its preferred size, as it depends on its parent behavior. In particular, when graphic objects are displayed into panels, the panel always uses its settings first to determine its children size. For details on panels, see *Panels*.

To get the preferred size of a graphic object, use the `GetPreferredSize` method. This method is called internally by the existing containers. You may want to override it, if you need to change the preferred size of an object.

The default behavior for the `GraphicObject` class is to compute a size that allows you to display the `Text` property entirely, taking into account the `TextAppearance` property of the graphic object. Other graphic objects, like panels, have implemented different behaviors based on their content: their preferred size is computed so that their children are displayed using their preferred size, if any. For details on panels preferred sizes, see *Panels*.

Because it is powerful, the layout system is also complex, and finding the right setting to get the right layout may not be obvious. The following rules may help you understand which settings to use to get the desired layout:

- ◆ A graphic object using its preferred size has its size constrained by its content.
- ◆ A graphic object located in a panel may have its size constrained by the panel, depending on the panel settings. This may conflict with the previous rule. In this case, the panel settings always win.
- ◆ A graphic object may have its width constrained by its contents, and its height constrained by its parent, or vice-versa.

The following example shows how to create a `StackPanel` with three **Rect** objects inside. The **StackPanel** has its **AutoSize** property set to **true**, which means that it should take its preferred size. In the case of a vertical stack panel, the preferred size is as follows:

Width = Maximum width of children whose horizontal alignment is not **HorizontalAlignment.Stretch**.

Height = Sum of the children heights.

Each **Rect** has also its **AutoSize** property set to **true**, to make it take its preferred size. In the case of a basic graphic object, the preferred size is determined by the text displayed by the object.

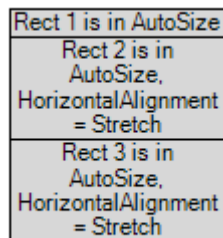
Only the first **Rect** object has a horizontal alignment different from **HorizontalAlignment.Stretch**. This means that the width of the stack panel has the same width as this object. The other two **Rect** objects are using the default horizontal alignment,



that is, **HorizontalAlignment.Stretch**. In this case, their width size will be constrained by the panel width.

```
StackPanel panel = new StackPanel();
panel.AutoSize = true;
Rect rect1 = new Rect();
rect1.Text = "Rect 1 is in AutoSize";
rect1.AutoSize = true;
panel.Objects.Add(rect1);
panel.SetHorizontalAlignment(rect1, HorizontalAlignment.Center);
Rect rect2 = new Rect();
rect2.Text = "Rect 2 is in AutoSize, HorizontalAlignment = Stretch";
rect2.AutoSize = true;
panel.Objects.Add(rect2);
Rect rect3 = new Rect();
rect3.Text = "Rect 3 is in AutoSize, HorizontalAlignment = Stretch";
rect3.AutoSize = true;
panel.Objects.Add(rect3);
Dim panel As StackPanel = New StackPanel
panel.AutoSize = true
Dim rect1 As Rect = New Rect
rect1.Text = "Rect 1 is in AutoSize"
rect1.AutoSize = true
panel.Objects.Add(rect1)
panel.SetHorizontalAlignment(rect1, HorizontalAlignment.Center)
Dim rect2 As Rect = New Rect
rect2.Text = "Rect 2 is in AutoSize, HorizontalAlignment = Stretch"
rect2.AutoSize = true
panel.Objects.Add(rect2)
Dim rect3 As Rect = New Rect
rect3.Text = "Rect 3 is in AutoSize, HorizontalAlignment = Stretch"
rect3.AutoSize = true
panel.Objects.Add(rect3)
```

The following illustration shows the compiled application:



Using Graphic Object Preferred Size

## *Understanding Graphic Object Events*

Among the various events sent by graphic objects, we can distinguish between:

- ◆ The events raised after a property change.
- ◆ The events raised to notify the framework of a graphic change.

The events raised after a property change are typical .NET events, whereas the events raised to notify the framework of a graphic change are specific to IBM® ILOG® Diagram for .NET. This section focuses on the second type of events.

---

### **Raising Graphic Object Events**

Graphic object events are useful for the framework to know which portion of the screen should be redrawn after a change in the objects hierarchy. When an object gets modified, it should raise two events: one before the change and one after the change. To raise those events, a graphic object uses the `OnGraphicChanging` and `OnGraphicChanged` methods.

The following code example shows the `Radius` property of the `Circle` class to illustrate this:

```
public class Circle : Shape
{
    ...
    public float Radius
    {
        get { return _radius; }
    }
}
```

## Understanding Graphic Object Events

```
        set
        {
            OnGraphicChanging(GraphicChange.GeometryBounds);
            _radius = value;
            OnGraphicChanged(GraphicChange.GeometryBounds);
        }
    }
    ...
}
Public Class Circle
    Inherits Shape
    ...
    Public Property Radius As Single
        Get
            Return _radius
        End Get
        Set
            OnGraphicChanging(GraphicChange.GeometryBounds)
            _radius = value
            OnGraphicChanged(GraphicChange.GeometryBounds)
        End Set
    End Property
    ...
End Class
```

The parameter for the **OnGraphicChanging** and **OnGraphicChanged** methods is one of the **GraphicChange** values. This enumeration defines the possible changes for a graphic object. In the example above, changing the radius of the circle changes its geometry bounds, thus the **GraphicChange.GeometryBounds** value is used.

The following table lists the main values that can be used:

Enumeration Value	Description
<b>Appearance</b>	The graphic object appearance has changed.
<b>StyledBounds</b>	The graphic object styled bounds have changed.
<b>Bounds</b>	The graphic object bounds have changed.
<b>GeometryBounds</b>	The graphic object geometry bounds have changed.
<b>Transform</b>	The graphic object transformer has changed.

**Note:** Raising graphic object events is an advanced feature that is needed only if you have to create a new graphic object by code. If you create your new graphic object by composing existing graphic objects, the events notification will be handled by the framework.

---

## Listening to Graphic Object Events

Listening to graphic object events is useful to track changes in a graphic objects hierarchy. For example, you may want to track the location of an object, its size, or whatever. To listen to graphic events sent by an object, use the Changing and Changed events.

The following code example shows how to listen to geometry bounds events, that is, events that notify a change in the object geometry bounds:

```
Rect r = new Rect(0, 0, 100, 100);
r.Changed += new GraphicChangeEventHandler(ObjectChanged);

private void ObjectChanged(object sender, GraphicObjectEventArgs args)
{
    if (args.IsBoundsChangeEvent)
        Console.WriteLine("New bounds : " + args.Bounds);
}
Dim r As Rect = New Rect(0, 0, 100, 100)
AddHandler r.Changed, AddressOf Me.ObjectChanged

Private Sub ObjectChanged(ByVal sender As Object, _
                          ByVal args As GraphicObjectEventArgs)
    If args.IsBoundsChangeEvent Then
        Console.WriteLine("New bounds", args.Bounds)
    End If
End Sub
```

Note that the BoundsChanging and BoundsChanged events are also raised when the bounds of graphic objects are changed.

The ChildChanging and ChildChanged events are raised by any container whose children are being changed. Listening to those events allows you to track changes in a entire graphic objects hierarchy, as shown in the following code example:

```
Group g = new Group();
g.ChildChanged += new GraphicChangeEventHandler(ChildChanged);

private void ChildChanged(object sender, GraphicObjectEventArgs args)
{
    Console.WriteLine("Name of the object : " + args.Source.Name);
    if (args.IsBoundsChangeEvent)
        Console.WriteLine("New bounds : " + args.Bounds);
}
Dim g As Group = New Group
AddHandler g.ChildChanged, AddressOf Me.ChildChanged

Private Sub ChildChanged (ByVal sender As Object, _
                          ByVal args As GraphicObjectEventArgs)
    Console.WriteLine("Name of the object : " + args.Source.Name)
    If args.IsBoundsChangeEvent Then
        Console.WriteLine("New bounds", args.Bounds)
    End If
End Sub
```

## Understanding Graphic Object Events

In the code example above, the Source property is used to know which graphic object is the event source.

# *Understanding Graphic Containers*

Graphic objects are by nature hierarchical: a drawing can be decomposed in several subdrawings. A graphic container is a graphic object that contains other graphic objects. The base class for graphic containers is the abstract class `GraphicContainer`. IBM® ILOG® Diagram for .NET provides different concrete subclasses of the **GraphicContainer** class in order to achieve the most complex scenarios.

## **In This Section**

### *Introduction to Graphic Containers*

Explains the basics of the **GraphicContainer** class.

### *Using the Predefined Graphic Containers*

Explains how to choose between the various containers available in IBM ILOG Diagram for .NET.

---

## **Introduction to Graphic Containers**

The `GraphicContainer` class is the base class for graphic objects that contain children. A container is responsible for the drawing and the hit testing of its children.

The **GraphicContainer** class does not define any API to add or remove children, which is done in subclasses. Instead, it provides a set of helpful methods that should be called by concrete graphic container subclasses.

---

### Graphic Container Events

The **GraphicContainer** class defines several events that can be used to track the container changes.

The **ChildrenHierarchyChange** event is raised when a change has occurred in the container children hierarchy. The change may have occurred on the direct children of the container, or on one of their descendants. The event argument sent with the event gives accurate information about the change that occurred. See the **ChildrenHierarchyChangeEventArgs** class for details. The following code shows how to monitor changes in the children hierarchy of a **Group**:

```
public Group CreateGroup()
{
    Group group = new Group();
    group.ChildrenHierarchyChanged +=
        new ChildrenHierarchyChangeEventHandler(HierarchyChanged);
    Rect rect1 = new Rect(0, 0, 100, 100);
    rect1.Text = "Rect 1";
    group.Objects.Add(rect1);
    Rect rect2 = new Rect(200, 200, 100, 100);
    rect2.Text = "Rect 2";
    group.Objects.Add(rect2);
    return group;
}

private void HierarchyChanged(object sender, ChildrenHierarchyChangeEventArgs
e)
{
    if (e.Action == ChildrenHierarchyChangeAction.Add)
        System.Console.WriteLine("Adding children");
    else
        System.Console.WriteLine("Removing children");
}

Public Function CreateGroup() As Group
    Dim group As Group = New Group
    AddHandler group.ChildrenHierarchyChanged, AddressOf Me.HierarchyChanged
    Dim rect1 As Rect = New Rect(0, 0, 100, 100)
    rect1.Text = "Rect 1"
    group.Objects.Add(rect1)
    Dim rect2 As Rect = New Rect(200, 200, 100, 100)
    rect2.Text = "Rect 2"
    group.Objects.Add(rect2)
    Return group
End Function

Private Sub HierarchyChanged(ByVal sender As Object, _
    ByVal e As ChildrenHierarchyChangeEventArgs)
    If (e.Action = ChildrenHierarchyChangeAction.Add) Then
        System.Console.WriteLine("Adding children")
    End If
End Sub
```



```
Else
    System.Console.WriteLine("Removing children")
End If
End Sub
```

The `ChildChanging` and `ChildChanged` events are raised respectively before and after a descendant of the graphic container changed. These events can be used to monitor graphic changes on the container descendants.

The `ChildInvalidated` event is raised when one of the container descendants needs to be repainted. This event is used by the `DiagramView` to invalidate its display.

---

### Graphic Container Coordinate System

The **GraphicContainer** class uses an internal affine transformation to draw its children. This transformation can be accessed using the `ChildTransform` property. For details on coordinate systems, see XREF Understanding Coordinate Systems.

---

### Logical Children versus Children

The **GraphicContainer** class introduces the notion of logical children. Logical children can be added to or removed from their container, whereas children cannot. For example, the `ScrollViewer` class uses scroll bars to scroll its content. These scroll bars cannot be removed from the **ScrollViewer**, whereas the **ScrollViewer** content can be changed. The **ScrollViewer** content is a logical child of the **ScrollViewer**, whereas the **ScrollViewer** scroll bars are not. Both children and logical children react to events.

Logical children can be accessed through the `LogicalChildren` property, and children can be accessed through the `Children` property.

---

## Using the Predefined Graphic Containers

IBM® ILOG® Diagram for .NET provides a large set of predefined graphic containers that can be mixed to achieve the most complex scenarios. Understanding the main features of each graphic container is the key to choose which container suits best for a particular use.

---

### Controls

Controls are containers whose geometry is defined by a rectangular area in which children will be placed. The base class for controls is the `Control` class. Controls add graphic decorations such as a border or a background and can implement specific behaviors.

The **Control** class is an abstract class that has three different types of subclasses listed in the following table:

Super Class	Description	Examples
<b>Control</b>	Controls that do not have children.	HScrollBar, VScrollBar
ContentControl	Controls with a single child.	Button, ScrollViewer, ViewPort
ObjectsControl	Controls with several children.	DockPanel, Canvas, GridPanel, StackPanel

For details about controls, see *Controls*.

For details about the classes **DockPanel**, **Canvas**, **GridPanel** and **StackPanel**, see *Panels*.

### Composite Objects

Composite objects are containers whose geometry is defined by their children. The base class for composite objects is the Composite object class. The children of the composite objects can be accessed through the Objects property. Composite objects are containers that simply group their children without adding any graphic decoration or behavior.

The **Composite** class is an abstract class that has several subclasses listed in the following table:

Class	Description
Group	A simple group of graphic objects that can be used to group objects sharing the same z-order.
GraphicSymbol	Allows you to create simple symbols. See <b>GraphicSymbols</b> for details.
UserSymbol	Allows you to create complex symbols in Visual Studio.NET®. See <b>UserSymbol</b> for details.

# Understanding Coordinate Systems

Understanding coordinate systems in IBM® ILOG® Diagram for .NET is useful to convert coordinates expressed in a coordinate system into another coordinate system.

## In This Section

### *Overview of Existing Coordinate Systems*

Describes the coordinate systems defined by IBM ILOG Diagram for .NET.

### *Conversion Between Coordinate Systems*

Explains how to convert coordinates.

---

## Overview of Existing Coordinate Systems

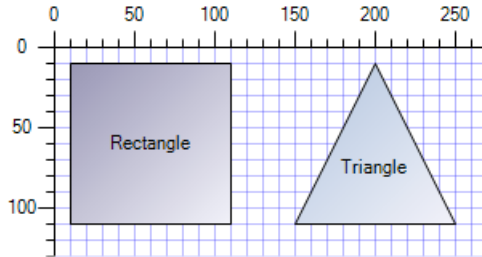
IBM® ILOG® Diagram for .NET defines three coordinate systems:

- ◆ *The Geometry Coordinate System*
- ◆ *The Container Coordinate System*
- ◆ *The View Coordinate System*

---

### The Geometry Coordinate System

The geometry of an object is defined by  $(x, y)$  points expressed in the geometry coordinate system. The following illustration shows a rectangle where the top-left point is  $(10, 10)$  and the size is  $(100, 100)$ , and a triangle defined by the points  $(200, 10)$ ,  $(250, 110)$ , and  $(150, 110)$ :



Each graphic object has its own way of defining its geometry. A **Rect** object has the **Rect.Rectangle** property, a **Polygon** has the **PolyPoints.Points** property, and so on.

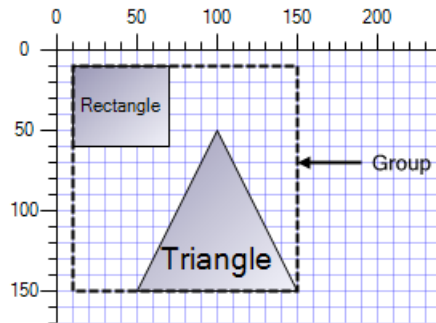
For details on how to specify the geometry of a particular object, see the reference manual of this object.

The bounds of a graphic object expressed in the geometry coordinate system are called geometry bounds. To get or set the geometry bounds of a graphic object, use **GeometryBounds** property. Set this property to modify the graphic object geometry so that it fits in the new geometry bounds. For example, setting the geometry bounds of a **Polygon** will move its points, while setting the geometry bounds of a **Rect** object will just set the **Rect.Rectangle** property to the new bounds.

The geometry coordinate system does not use the local object transformation and thus setting the **Transform** property has no impact on the geometry coordinate system.

## The Container Coordinate System

The container coordinate system is the coordinate system in which a container displays its children. This coordinate system takes into account the children local transformations. The following illustration shows a group composed of a rectangle and a triangle:

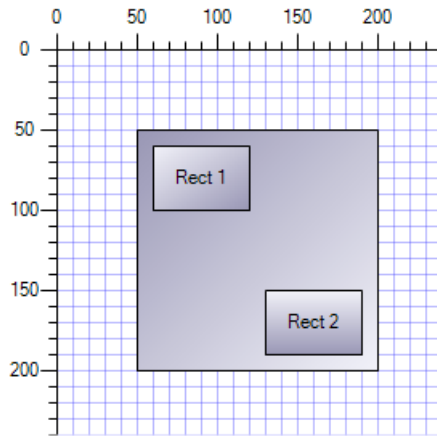


The rectangle geometry bounds are  $(x = 10, y = 10, \text{width} = 60, \text{height} = 60)$ , there is no local transformation. Thus, its bounds expressed in the container coordinate space are the same.

The triangle geometry bounds are  $(50, 50, 50, 50)$ . The local transformation applied to the triangle is the composition of a translation of  $(-50, -50)$  and a scale transformation of factor 2. The resulting bounds expressed in the container coordinate space are  $(50, 50, 100, 100)$ .

In addition to the local graphic object transformation that can be set using the **Transform** property, graphic containers apply another transformation to their children. This transformation is given by the **ChildTransform** property. The property is used internally by some containers to transform the coordinate system in which their children are drawn. The following illustration shows a Canvas that contains two rectangles. The Canvas uses its

**ChildTransform** property to have its upper left corner defined as the point (0, 0) in its children coordinate system:



The Canvas bounds are (50, 50, 150, 150). In the container coordinate system, the point (50, 50) is transformed to the point (0, 0). As a result, the bounds of the rectangles named "Rect 1" and "Rect 2" are respectively (10, 10, 60, 40) and (80, 100, 60, 40).

The bounds of a graphic object expressed in the container coordinate system are called bounds. To get or sets the bounds of a graphic object, use the Bounds property. Set this property to modify the graphic object so that it fits in the new bounds. The way the graphic object is modified to fit in the new bounds depends on the following parameters:

- ◆ graphic object type,
- ◆ graphic object local transformation,
- ◆ ResizeMode property.

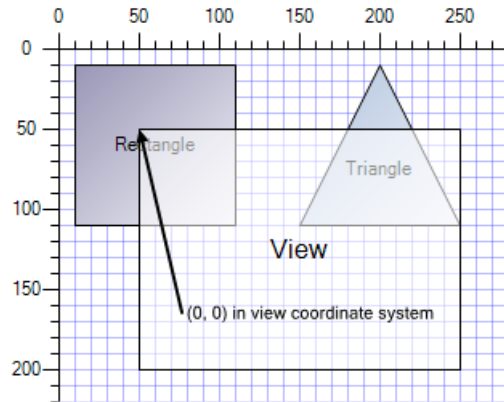
Depending on those parameters, the graphic object will either modify its geometry bounds, or its local transformation to fit in the new bounds.

---

### The View Coordinate System

When graphic objects are displayed using diagram views, an additional transformation is used to enable the scrolling and the zooming into the view. The following illustration shows a Winforms **DiagramView** that contains a rectangle and a triangle whose bounds are respectively (10, 10, 100, 100) and (150, 10, 100, 100). The diagram view has been

translated by the vector (-50, -50) so that the point (0, 0) in the view coordinate system corresponds to the point (50, 50) in the diagram coordinate system:



The diagram view transformation can be set by using the Transform property for the Winforms version, and the Transform property for the ASP.NET version.

---

## Conversion Between Coordinate Systems

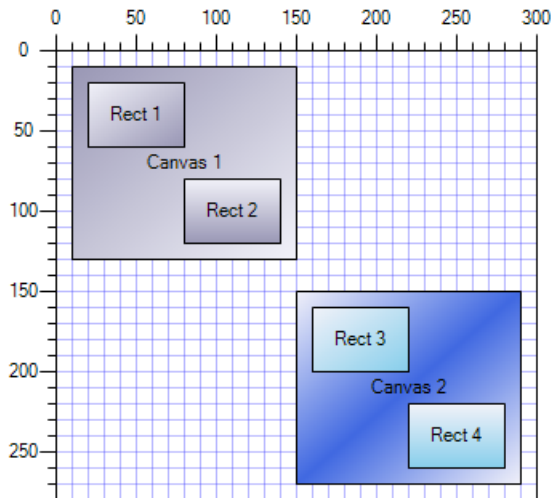
To convert coordinates expressed in a coordinate system to another coordinate system, the following methods can be used:

Class	Method	Description
GraphicObject	GetGeometryToContainerTransform	Gets the transformation to convert geometry coordinates into the specified container coordinate system.
<b>GraphicObject</b>	GetGeometryToViewTransform	Gets the transformation to convert geometry coordinates into the specified view coordinate system.
GraphicContainer	GetTransformToContainer	Gets the transformation to convert children coordinates into the specified container coordinate system.

Class	Method	Description
<b>GraphicContainer</b>	GetTransformToView	Gets the transformation to convert children coordinates into the specified view coordinate system.
<b>GraphicObject</b>	GetRelativeBounds	Gets the bounds of this object expressed in the specified container coordinate system.

Note that conversions must be made between graphic objects that have a common ancestor; otherwise an exception is thrown.

The following illustration shows two Canvas, each containing two Rect objects. Assume that the two Canvas share a common ancestor, so that the coordinate conversion between them is possible.



If you want to express the bounds of "Rect 3" in the container coordinate system of "Canvas 1", you can code:

```
Rectangle2D bounds = rect3.GetRelativeBounds(canvas1);
Dim bounds As Rectangle2D = rect3.GetRelativeBounds(canvas1)
```

If you want to compute the transformation that converts from the container coordinate system of "Canvas 1" to the container coordinate system of "Canvas 2", you can code:

```
Transform t = canvas1.GetTransformToContainer(canvas2);
Dim t As Transform = canvas1.GetTransformToContainer(canvas2)
```



# ***Creating Diagrams with Nodes and Links***

In this section you are going to see how to create diagrams with nodes and links.

## **In This Section**

### *Creating a Simple Diagram with Nodes and Links Programmatically*

Explains the code needed to create a simple diagram.

### *Introducing Link and Anchor Classes*

Explains the basic principles of the classes that are used to create graphs.

### *Using Automatic Link Crossing Detection in a Graph*

Explains how to automatically detect and display link crossing in a graph.

### *Creating a New Class of Anchor*

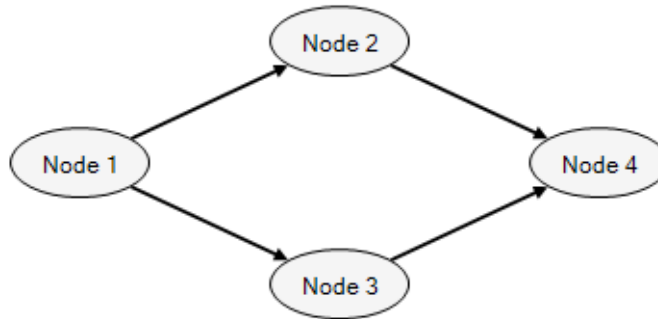
Explains how to write a new Anchor subclass.

---

## **Creating a Simple Diagram with Nodes and Links Programmatically**

This section explains the code needed to create a simple diagram that depicts a graph of objects.

This is the graph that you are going to create:



A graph is a set of objects (called nodes) connected together through links. In IBM® ILOG® Diagram for .NET, creating a graph involves the following classes:

- ◆ The nodes of the graph are instances of any subclass of `GraphicObject`.
- ◆ The links of the graph are instances of the class `Link`.
- ◆ The connections between nodes and links are done through the `Anchor` class.

---

### Creating Nodes

The graphic objects representing the nodes of a graph are created in the usual way. This example uses `Ellipse` objects.

The following example shows how to create a node.

```
// The group argument is the container in which the graph is created.
//
private GraphicObject CreateNode(Group group, string text,
                                float x, float y)
{
    // Create the ellipse object:
    //
    Ellipse ellipse = new Ellipse(new Rectangle2D(x, y, 80, 40));
    ellipse.Text = text;
    ellipse.Fill = new SolidFill(Color.WhiteSmoke);

    // Add a shape anchor to the ellipse:
    //
    ShapeAnchor anchor = new ShapeAnchor();
    ellipse.Anchors.Add(anchor);

    // Add the ellipse to the group:
    //
    group.Objects.Add(ellipse);

    return ellipse;
}
```

## Creating a Simple Diagram with Nodes and Links Programmatically

```
}
'' The group argument is the container in which the graph is created.
''
Private Function CreateNode(ByVal group As Group, ByVal text As String,
    ByVal x As Single, ByVal y As Single) As GraphicObject

    ' Create the ellipse object:
    ,
    Dim ellipse As Ellipse = New Ellipse(New Rectangle2D(x, y, 80, 40))
    ellipse.Text = text
    ellipse.Fill = New SolidFill(Color.WhiteSmoke)

    ' Add a shape anchor to the ellipse:
    ,
    Dim anchor As ShapeAnchor = New ShapeAnchor
    ellipse.Anchors.Add(anchor)

    ' Add the ellipse to the group:
    ,
    group.Objects.Add(ellipse)
    Return ellipse
End Function
```

**Note:** A *ShapeAnchor* has been added to the node. This anchor will be used to connect the links.

---

### Creating Links

The following example shows how to create a link between two nodes.

```
private Link CreateLink(Group group, GraphicObject startNode, GraphicObject
endNode)
{
    // Create the link:
    //
    Link link = new Link();
    link.ShapeType = LinkShapeType.Straight;

    // Set the start and end anchors:
    //
    link.StartAnchor = startNode.Anchors[0];
    link.EndAnchor = endNode.Anchors[0];

    group.Objects.Add(link);

    return link;
}
Private Function CreateLink(ByVal group As Group, ByVal startNode As
GraphicObject,
    ByVal endNode As GraphicObject) As Link

    ' Create the link:
    ,
    Dim link As Link = New Link
    link.ShapeType = LinkShapeType.Straight
```

## Creating Diagrams with Nodes and Links

```
' Set the start and end anchors:
,
link.StartAnchor = startNode.Anchors(0)
link.EndAnchor = endNode.Anchors(0)
group.Objects.Add(link)
Return link
End Function
```

In this example, every node has only one anchor, and all the links that start or end on the same node are connected to the same anchor. It is also possible to add several anchors to a node and to connect each link to a different anchor.

---

### Creating the Graph

Finally, the following example shows how to create the whole graph.

```
private Group CreateGraph()
{
    // Create the container of the graph:
    //
    Group group = new Group();

    // Create the four nodes:
    //
    GraphicObject node1 = CreateNode(group, "Node 1", 30, 90);
    GraphicObject node2 = CreateNode(group, "Node 2", 180, 20);
    GraphicObject node3 = CreateNode(group, "Node 3", 180, 160);
    GraphicObject node4 = CreateNode(group, "Node 4", 330, 90);

    // Create the four links:
    //
    CreateLink(group, node1, node2);
    CreateLink(group, node1, node3);
    CreateLink(group, node2, node4);
    CreateLink(group, node3, node4);

    return group;
}
Private Function CreateGraph() As Group

    ' Create the container of the graph:
    ,
    Dim group As Group = New Group

    ' Create the four nodes:
    ,
    Dim node1 As GraphicObject = CreateNode(group, "Node 1", 30, 90)
    Dim node2 As GraphicObject = CreateNode(group, "Node 2", 180, 20)
    Dim node3 As GraphicObject = CreateNode(group, "Node 3", 180, 160)
    Dim node4 As GraphicObject = CreateNode(group, "Node 4", 330, 90)

    ' Create the four links:
    ,
```

```

    CreateLink(group, node1, node2)
    CreateLink(group, node1, node3)
    CreateLink(group, node2, node4)
    CreateLink(group, node3, node4)
    Return group
End Function

```

---

## Introducing Link and Anchor Classes

This section explains the basic principles of the classes that are used to create graphs.

In this section, we will focus on the **Link** and **Anchor** classes.

The relationships between nodes, anchors and links are the following:

- ◆ Each graphic object owns a collection of anchors in its **Anchors** property.
- ◆ A link can be attached to two anchors through its **StartAnchor** and **EndAnchor** properties.
- ◆ The links that are attached to an anchor can be retrieved through the **Links** property of the anchor.

---

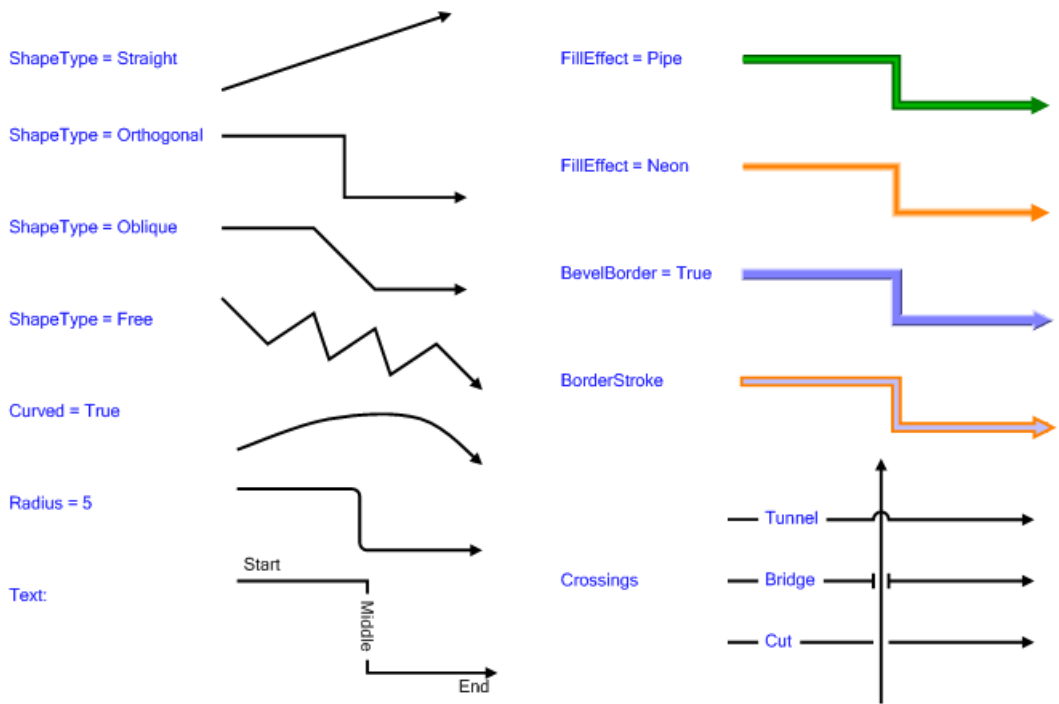
### Links

Links are graphic objects that implement the **ILink** interface. This interface defines the following properties:

- ◆ **StartAnchor** defines the **Anchor** to which the start of the link is attached. This property can be null.
- ◆ **EndAnchor** defines the **Anchor** to which the end of the link is attached. This property can be null.
- ◆ **StartPoint** defines the position of the start point of the link when it is not attached to an anchor, that is, when **StartAnchor** is null.
- ◆ **EndPoint** defines the position of the end point of the link when it is not attached to an anchor, that is, when **EndAnchor** is null.

The **Link** class is the only class in the IBM® ILOG® Diagram for .NET library that implements the **ILink** interface. The **Link** class has many options to control the appearance of the link and the text items attached to it.

The following illustration shows some samples of **Link** objects.



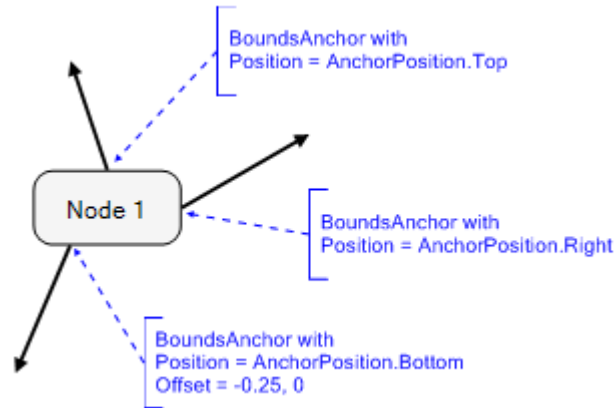
For more details on how to use and configure **Link** objects see *Link Objects*.

---

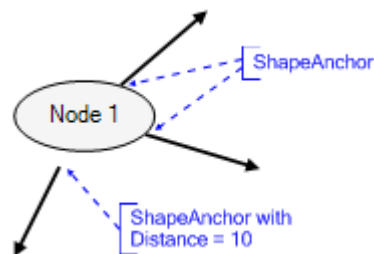
### Anchors

An anchor determines the exact position of the start and end point of a link. The base class **Anchor** is an abstract class. This section describes the predefined subclasses of **Anchor**.

- ◆ The **BoundsAnchor** connects a link to a fixed position on the bounding rectangle of a graphic object. The position of the connection point is determined by the **Position** property of the anchor, which is an enumeration of type **AnchorPosition** whose possible values are **Top**, **Bottom**, **Left**, **Right**, **Center**, **TopLeft**, and so on. The connection point can be further modified using the **Offset** property.

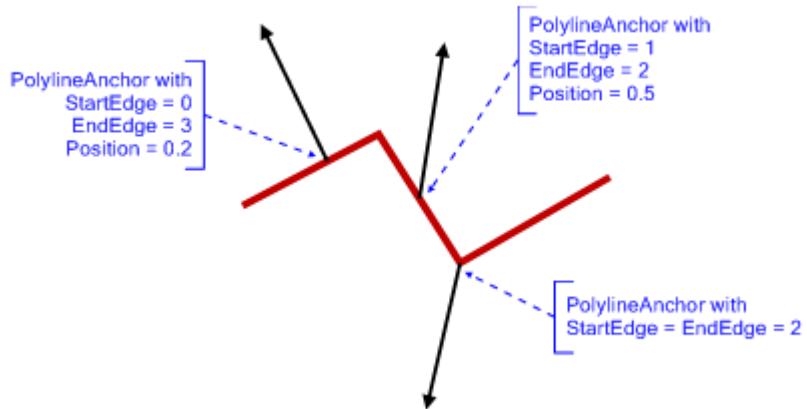


- ◆ The ShapeAnchor class connects a link to the outline of the shape of a graphic object. The connection point depends on the geometry of the last segment of the link: the last segment is considered to point to the center of the node's shape, and the connection point is the intersection between this virtual segment and the outline of the shape. The distance between the connection point and the shape can be modified using the Distance property.



**Note:** The same *ShapeAnchor* can provide different connection points for different links, as in the example above.

- ◆ The `PolylineAnchor` class connects to a point along a linear graphic object like a `Polyline`. The connection point is a point of the polyline that lies between two edges of the polyline defined by the `StartEdge` and `EndEdge` properties. The `Position` property specifies the linear position along the path of the polyline.



The **PolylineAnchor** can be used with any graphic object that implements the `IPolyPoints` interface.

**Note:** Since the *Link* class implements this interface, it is possible to connect a link to another link

You can create your own subclass of **Anchor** if the predefined anchor classes do not fit your needs. To do this see *Creating a New Class of Anchor*.

---

### Specifying Anchors on a Graphic Object

To add an anchor on a graphic object, use the **Add** method of the collection contained in the **Anchors** property of the graphic object.

Each class of graphic object has a set of default anchors, returned by the `GetDefaultAnchors` method of the graphic object. These default anchors are not contained in the **Anchors** collection of the graphic object: they are only used by the `CreateLinkInteractor` and `EditAnchorsInteractor` objects to propose the possible anchors (in addition to any anchors that have been added to the **Anchors** collection as explained above). Once the user has selected one of the default anchors, this anchor will actually be added to the graphic object.

The default anchors mechanism can be disabled by setting the `UseDefaultAnchors` property of the graphic object to **false**. The interactors will not propose any more the default anchors



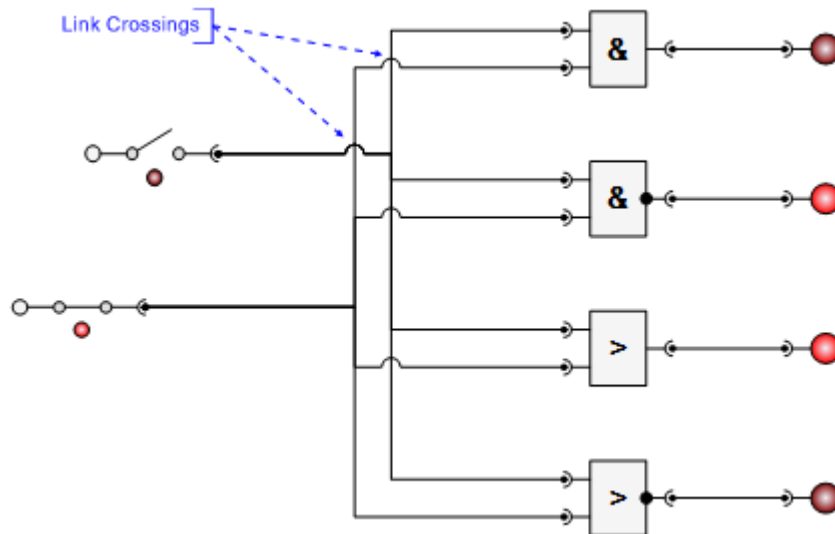
of the graphic object; they will only propose the anchors that have been added to the **anchors** collection.

---

## Using Automatic Link Crossing Detection in a Graph

IBM® ILOG® Diagram for .NET provides options to automatically detect and display link crossing in a graph. Link crossing detection is used when you need to graphically show the points where two different links cross. This feature is often used, for instance, in electrical schematics diagrams.

The following illustration shows a graph where link crossing detection is enabled.




---

### Enabling Link Crossing Detection

Link crossing detection is controlled by the `LinkCrossings` property of the `GraphicContainer` class. The value of this property is an instance of the class `LinkCrossings`.

To enable link crossing detection in a graphic container, set the `Enabled` property of the `LinkCrossings` object to `LinkCrossingsEnableMode.Enabled`.

Link crossings will then be detected and displayed on all the links contained in the graphic container. Link crossings concern all the objects that implement the `ICrossable` interface. In the IBM ILOG Diagram for .NET library, the only class that implements this interface is the `Link` class, so only crossings between links are detected.

---

### Enabling or Disabling Link Crossing Detection on SubContainers

You will typically enable link crossing detection on the top-level graphic container that contains your diagram. This will also enable link crossing detection on all subcontainers.

You can also choose to enable or disable link crossing detection on specific subcontainers by setting the **Enabled** property of the **LinkCrossings** object to one of the values of the **LinkCrossingsEnableMode** enumeration. The possible values are:

- ◆ **LikeParent:** This value means that link crossing detection is inherited from the parent container, that is, it is enabled if the **LinkCrossings** object of one of the ancestors of the container has its **Enabled** property set to **Enabled**.
- ◆ **Enabled:** This value means that link crossing detection is enabled for this container, regardless of the settings on other containers.
- ◆ **Disabled:** This value means that link crossing detection is disabled for this container, regardless of the settings on other containers.

The default value for all graphic containers is **LikeParent**, so if you enable link crossing detection for the top-level container, it is also enabled automatically for all subcontainers.

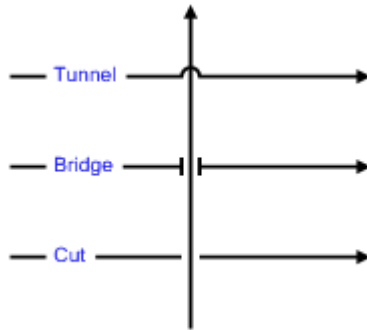
---

### Changing the Appearance of Link Crossings

You can choose the way link crossings are displayed in a graphic container by setting the **Style** property of the **LinkCrossings** object of the container. The value of this property is an enumeration of type **LinkCrossingsStyle**, and the possible values are:

- ◆ **Tunnel:** This value means that crossings are drawn as small arcs of circle (see the picture below).
- ◆ **Bridge:** This value means that crossings are drawn as two short segments orthogonal to the link's path on each side of the crossing (see the picture below).
- ◆ **Cut:** This value means that the path of the link is cut at the crossing (see the picture below).
- ◆ **None:** This value means that no special crossings are drawn, that is, the links cross normally.
- ◆ **Default:** This value means that the crossing style is inherited from the parent container, if any. If there is no parent container, the default crossing style is **Tunnel**.

The following illustration shows the possible crossing types.



You can also change the size of the link crossings by setting the `Size` property of the `LinkCrossings` object. The default size is 10.

Instead of changing the appearance of crossings for all the links in a container, the appearance can be changed for an individual **Link** using the `CrossingStyle` and `CrossingSize` properties of the link. This will override the values of the **Style** and **Size** properties of the `LinkCrossings` object of the graphic container. However, this is not generally recommended because diagrams look nicer if all link crossings have the same appearance and size.

---

### Changing the Orientation of Link Crossings

When there are many link crossings in a diagram, the display will generally look nicer if all the crossings have the same orientation. For example, if all the links are made of horizontal or vertical segments, placing all the crossings on horizontal segments is preferable to having some crossings horizontal and some vertical.

The orientation of link crossings is controlled by the `Orientation` property of the `LinkCrossings` object of the graphic container. The value of this property is an enumeration of type `LinkCrossingsOrientation`, and the possible values are:

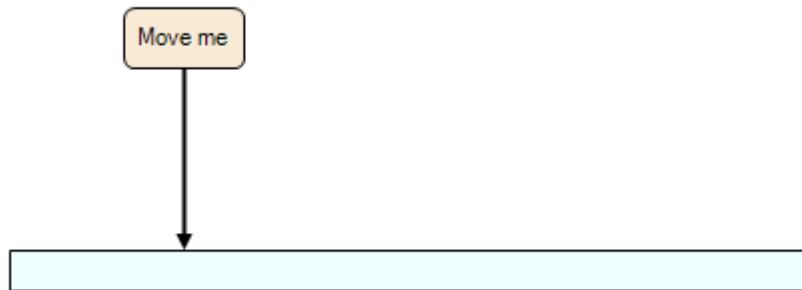
- ◆ **Horizontal:** This value means that all crossings will be placed on horizontal segments (if the segments are not strictly horizontal or vertical, the closest orientation is used).
- ◆ **Vertical:** This value means that all crossings will be placed on vertical segments (if the segments are not strictly horizontal or vertical, the closest orientation is used).
- ◆ **Any:** This value means that crossings are placed on the first segment on which they are detected, regardless of its orientation.

## Creating a New Class of Anchor

In the section *Introducing Link and Anchor Classes* you have been introduced to the basic principles of the **Anchor** class and to its predefined subclasses.

In some cases, the predefined anchors might not fulfill your needs and you may have to write a new **Anchor** subclass. This section explains how to do this.

You are going to create a class called **OrthogonalAnchor**. This custom anchor class can be used to connect a link to a graphic object such that the link stays vertical regardless of the position of the other end of the link, as shown in the following illustration.



The first step is to create the subclass of **Anchor**.

```
public class OrthogonalAnchor : Anchor
{
    Public Class OrthogonalAnchor
        Inherits Anchor
}
```

In your custom **Anchor** class, you will override two methods: **NeedsReferencePoint** and **GetPoint**.

The **NeedsReferencePoint** method simply returns **true**.

```
public override bool NeedsReferencePoint ()
{
    return true;
}
Public Overrides Function NeedsReferencePoint () As Boolean
    Return True
End Function
```

This means that your custom anchor needs a reference point to compute the connection point of a link. The reference point is the "other end" of the link (in the example above, it will be the center of the "Move me" rectangle). The reference point is used to determine the X position of the connection point.

The **GetPoint** method is the place where the anchor computes the connection point.

```
public override void GetPoint(Point2D referencePoint, out Point2D
connectionPoint,
out Point2D originPoint)
{
    // Get the bounds of the owner object.
    //
    Rectangle2D bounds =
IQueryOutlinePath(Owner).GetOutlinePath(Transform.Identity).GetBounds();

    connectionPoint = new Point2D();
    originPoint = new Point2D();

    connectionPoint.X = referencePoint.X;
    connectionPoint.Y = bounds.Y;
    originPoint.X = connectionPoint.X;
    originPoint.Y = bounds.Y + bounds.Height / 2;
}
Public Overrides Sub GetPoint(ByVal referencePoint
As Point2D, _
                                ByRef connectionPoint As Point2D, _
                                ByRef originPoint As Point2D)
    ' Get the bounds of the owner object.
    '
    Dim bounds As Rectangle2D _
        = CType(Owner,
IQueryOutlinePath).GetOutlinePath(Transform.Identity).GetBounds()
    connectionPoint = New Point2D
    originPoint = New Point2D

    connectionPoint.X = referencePoint.X
    connectionPoint.Y = bounds.Y
    originPoint.X = connectionPoint.X    originPoint.Y = (bounds.Y +
(bounds.Height / 2))
End Sub
```

By using this method, proceed as follows:

1. Retrieve the bounds of the graphic object that owns the anchor (this is the wide rectangle in the example). The owner object is contained in the Owner property.
2. Compute the connection point: its X coordinate is the same of the X coordinate of the reference point, and its Y coordinate is the top of the bounding rectangle of the owner object.
3. Compute the origin point: it has the same X coordinate as the connection point, but its Y coordinate is the vertical center of the owner object. Note that the origin point is important because it determines the side on which the link is connected (this is used, in

## Creating Diagrams with Nodes and Links

particular, to compute the shape of orthogonal links). You can think of the origin point as "where the link points to". In this example, the link has to be vertical and must point somewhere below the connection point.

This example can be found in the Samples/QuickStart/CustomAnchor directory. (The code of the sample is slightly more complex because it deals with the cases where the reference point is not directly above the graphic object.)

# ***Handling Interactions in a Diagram View (WinForms)***

In this section you are going to see how to handle interactions in a diagram view (WinForms).

## **In This Section**

*Understanding Events Dispatching in a Diagram View (WinForms)*

Shows how to handle input events.

*Using Predefined Interaction Tools in a Diagram View*

Explains how to use the predefined interaction tools.

*Creating a New Interactor in a DiagramView (WinForms)*

Shows how to create a subclass of ViewInteractor through an example.

---

## **Understanding Events Dispatching in a Diagram View (WinForms)**

This section explains how input events are handled by a diagram view, how they are dispatched to a diagram and how to handle events in a graphic object. When an input event is received by a diagram view, the event is dispatched to the view interactor or to the graphic objects according to different routing strategies. When an event is routed to graphic objects

through the graphic object hierarchy, the handlers that have been set on the hierarchy elements are notified to perform some actions in response to this event.

### In This Section

#### *From the View to the Graphic Object*

Explains how events are handled by the diagram view.

#### *Dispatching Events to Graphic Objects*

Explains how events are dispatched to graphic objects.

#### *Stopping the Event Propagation*

Explains how to stop event propagation.

#### *Event Capture*

Explains how a graphic object can capture events.

---

## From the View to the Graphic Object

When an input event is received by a diagram view, the event is dispatched to the graphic objects contained in the view graphic container. The graphic object that receives the event may respond to this event either directly in its implementation or via the event handlers set on the graphic object.

A view dispatches input events according to the following rules:

- ◆ If a view interactor is set on the view, the event is dispatched to the interactor and the event dispatching phase ends.
- ◆ The event is sent to the graphic object targeted by the input event, that is, the graphic object that responded to the hit testing. Then, the graphic object notifies its event handlers.

Listening to an event is performed by attaching an event handler to the corresponding `GraphicObject` event and by implementing the expected behavior in the event handler.

The following example shows how to listen to `MouseEnter` and `MouseLeave` events to implement a rollover effect on a graphic object:

```
public partial class Form1 : Form
{
    private static Stroke DefaultStroke = new Stroke(Color.Black, 3);
    private static Stroke HighlightStroke =
        new Stroke(Color.LightSteelBlue,
            3, System.Drawing.Drawing2D.DashStyle.Dash);
    public Form1()
    {
        InitializeComponent();

        DiagramView diagramView1 = new DiagramView();
        diagramView1.Dock = System.Windows.Forms.DockStyle.Fill;
    }
}
```



## Understanding Events Dispatching in a Diagram View (WinForms)

```
this.Controls.Add(diagramView1);
// Create the graphic object hierarchy
Group content = new Group();
Rect shape = new Rect(new Rectangle2D(30, 60, 110, 60));
shape.Stroke = DefaultStroke;
shape.Name = "shape";
shape.Fill = new SolidFill(Color.Coral);
content.Objects.Add(shape);
diagramView1.Content = content;
shape.MouseEnter += new ObjectEventHandler(OnShapeMouseEnter);
shape.MouseLeave += new ObjectEventHandler(OnShapeMouseLeave);
}

void OnShapeMouseLeave(object sender, EventArgs e)
{
    Shape shape = e.Target as Shape;
    if (shape != null)
    {
        shape.Stroke = DefaultStroke;
    }
}

void OnShapeMouseEnter(object sender, EventArgs e)
{
    Shape shape = e.Target as Shape;
    if (shape != null)
    {
        shape.Stroke = HighlightStroke;
    }
}
}

Public Class Form1
Inherits Form
Private Shared DefaultStroke As Stroke = New Stroke(Color.Black, 3)
Private Shared HighlightStroke As Stroke = _
    New Stroke(Color.LightSteelBlue, _
        3, System.Drawing.Drawing2D.DashStyle.Dash)

Public Sub New()
    InitializeComponent
    Dim diagramView1 As DiagramView = New DiagramView
    diagramView1.Dock = System.Windows.Forms.DockStyle.Fill
    Me.Controls.Add(diagramView1)
    Dim content As Group = New Group
    Dim shape As Rect = New Rect(New Rectangle2D(30, 60, 110, 60))
    shape.Stroke = DefaultStroke
    shape.Name = "shape"
    shape.Fill = New SolidFill(Color.Coral)
    content.Objects.Add(shape)
    diagramView1.Content = content
    AddHandler shape.MouseEnter, AddressOf OnShapeMouseEnter
    AddHandler shape.MouseLeave, AddressOf OnShapeMouseLeave
End Sub

Sub OnShapeMouseLeave(ByVal sender As Object, ByVal e As EventArgs)
    Dim shape As Shape = TryCast(e.Target, Shape)
    If Not (shape Is Nothing) Then
        shape.Stroke = DefaultStroke
    End If
End Sub
```

## Handling Interactions in a Diagram View (WinForms)

```
        shape.Stroke = DefaultStroke
    End If
End Sub

Sub OnShapeMouseEnter(ByVal sender As Object, ByVal e As EventArgs)
    Dim shape As Shape = TryCast(e.Target, Shape)
    If Not (shape Is Nothing) Then
        shape.Stroke = HighlightStroke
    End If
End Sub
End Class
```

---

### Dispatching Events to Graphic Objects

In IBM® ILOG® Diagram for .NET, a diagram is a hierarchical structure similar to a tree. The top-level container is a root node which contains child nodes that might contain themselves other children. This tree is called graphic object hierarchy. When an event is dispatched to the graphic object, it goes throughout the hierarchy to reach the target graphic object.

The event dispatching is divided in two different phases:

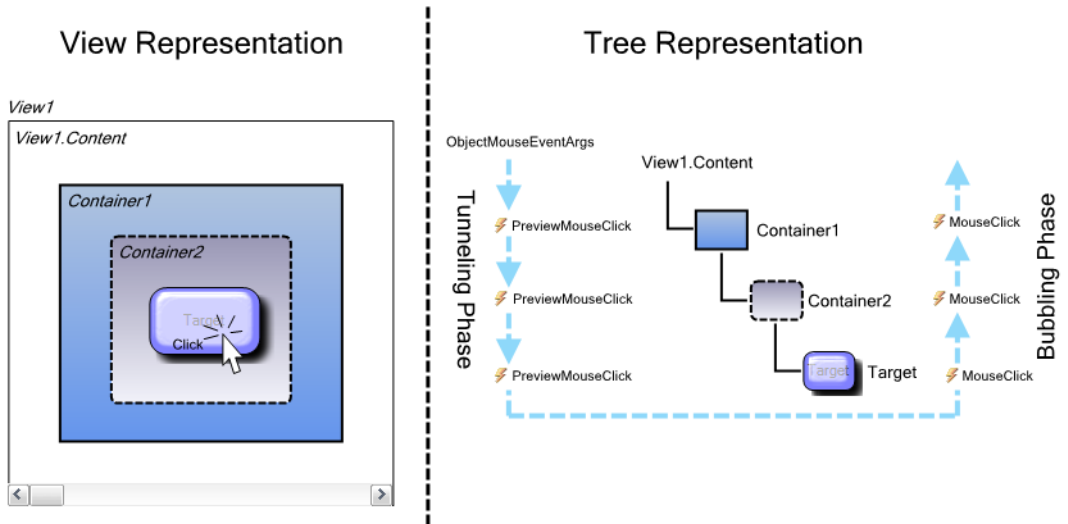
◆ tunneling phase

From the top-level container to the target graphic object (that is, from top to bottom).

◆ bubbling phase

From the target graphic object to the top-level container (that is, from bottom to top).

The following picture shows the route followed by an event during the event dispatching phase when the user clicks a graphic object contained in a container, itself being a child of a top-level container:



### Tunneling Phase

The tunneling phase corresponds to the top-to-bottom routing part. In this phase, the input event goes down the hierarchy from the top-level container to the target graphic object, notifying the corresponding event handlers at each level of the hierarchy.

To distinguish between an event in the tunneling phase and in the bubbling phase, the name of the event sent during the tunneling phase is prefixed by Preview.

### Bubbling Phase

The bubbling phase corresponds to the bottom-to-top routing part. In this phase, the input event goes up the hierarchy from the target graphic object to the top-level container, notifying the corresponding event handlers at each level of the hierarchy.

Because an event is routed throughout the hierarchy, an event handler must be able to know the target of the event (for example, the graphic object on which the user clicks). For this purpose, the event data define the Target property which returns the **GraphicObject** that received the event.

Similarly, an event handler must be able to know the current **GraphicObject** that is notified when the event travels throughout the hierarchy (for example, an intermediate container in the hierarchy that contains the event target). For this purpose, the event data define the CurrentTarget property which returns the current **GraphicObject** of the hierarchy that is being notified of the event.

### **Notes:**

1. *There are two exceptions to the event dispatching mechanism described above: the `MouseEnter` and `MouseLeave` events. These events are not dispatched by the standard route: there is no tunneling nor bubbling phases. Instead, they are dispatched directly to the graphic object.*
2. *The following example illustrates the tunneling and bubbling phases, tracing the `MouseClicked` event dispatching throughout a hierarchy of graphic objects.*

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();

        DiagramView diagramView1 = new DiagramView();
        diagramView1.Dock = System.Windows.Forms.DockStyle.Fill;
        this.Controls.Add(diagramView1);
        // Create the graphic object hierarchy
        Group content = new Group();
        Canvas container1 = new Canvas();
        container1.Name = "container1";
        container1.Size = new Size2D(300, 250);
        container1.Location = new Point2D(5, 5);
        container1.Background = new SolidFill(Color.LightSteelBlue);
        Canvas container2 = new Canvas();
        container2.Name = "container2";
        container2.Size = new Size2D(260, 180);
        container2.Location = new Point2D(10, 10);
        container2.Background = new SolidFill(Color.Coral);
        Rect shape = new Rect(new Rectangle2D(30, 60, 110, 60));
        shape.Name = "shape";
        shape.Fill = new SolidFill(Color.CadetBlue);
        container2.Objects.Add(shape);
        container1.Objects.Add(container2);
        content.Objects.Add(container1);
        diagramView1.Content = content;
        // Wire PreviewMouseClicked event handlers
        container1.PreviewMouseClicked +=
            new ObjectMouseEventHandler(OnObjectPreviewMouseClicked);
        container2.PreviewMouseClicked +=
            new ObjectMouseEventHandler(OnObjectPreviewMouseClicked);
        shape.PreviewMouseClicked +=
            new ObjectMouseEventHandler(OnObjectPreviewMouseClicked);
        // Wire MouseClick event handlers
        container1.MouseClick +=
            new ObjectMouseEventHandler(OnObjectMouseClick);
        container2.MouseClick +=
            new ObjectMouseEventHandler(OnObjectMouseClick);
        shape.MouseClick +=
            new ObjectMouseEventHandler(OnObjectMouseClick);
    }

    void OnObjectMouseClick(object sender, ObjectMouseEventArgs e)
```

## Understanding Events Dispatching in a Diagram View (WinForms)

```
{
    MessageBox.Show(" [Event target: " +
        e.Target.Name + "] [CurrentTarget:" +
        e.CurrentTarget.Name + "]", "Bubbling phase");
}

void OnObjectPreviewMouseClicked(object sender, ObjectMouseEventArgs e)
{
    MessageBox.Show(" [Event target: " +
        e.Target.Name + "] [CurrentTarget:" +
        e.CurrentTarget.Name + "]", "Tunneling phase");
}
}
Public Class Form1
Inherits Form

Public Sub New()
    InitializeComponent
    Dim diagramView1 As DiagramView = New DiagramView
    diagramView1.Dock = System.Windows.Forms.DockStyle.Fill
    Me.Controls.Add(diagramView1)
    Dim content As Group = New Group
    Dim container1 As Canvas = New Canvas
    container1.Name = "container1"
    container1.Size = New Size2D(300, 250)
    container1.Location = New Point2D(5, 5)
    container1.Background = New SolidFill(Color.LightSteelBlue)
    Dim container2 As Canvas = New Canvas
    container2.Name = "container2"
    container2.Size = New Size2D(260, 180)
    container2.Location = New Point2D(10, 10)
    container2.Background = New SolidFill(Color.Coral)
    Dim shape As Rect = New Rect(New Rectangle2D(30, 60, 110, 60))
    shape.Name = "shape"
    shape.Fill = New SolidFill(Color.CadetBlue)
    container2.Objects.Add(shape)
    container1.Objects.Add(container2)
    content.Objects.Add(container1)
    diagramView1.Content = content
    AddHandler container1.PreviewMouseClicked, AddressOf OnObjectPreviewMouseClicked
    AddHandler container2.PreviewMouseClicked, AddressOf OnObjectPreviewMouseClicked
    AddHandler shape.PreviewMouseClicked, AddressOf OnObjectPreviewMouseClicked
    AddHandler container1.MouseClick, AddressOf OnObjectMouseClicked
    AddHandler container2.MouseClick, AddressOf OnObjectMouseClicked
    AddHandler shape.MouseClick, AddressOf OnObjectMouseClicked
End Sub

Sub OnObjectMouseClicked(ByVal sender As Object, _
    ByVal e As ObjectMouseEventArgs)
    MessageBox.Show(" [Event target: " + _
        e.Target.Name + "] [CurrentTarget:" + _
        e.CurrentTarget.Name + "]", "Bubbling phase")
End Sub

Sub OnObjectPreviewMouseClicked(ByVal sender As Object, _
    ByVal e As ObjectMouseEventArgs)
    MessageBox.Show(" [Event target: " + _
        e.Target.Name + "] [CurrentTarget:" + _
```

```
                e.CurrentTarget.Name + "]", "Tunneling phase")
    End Sub
End Class
```

---

### Stopping the Event Propagation

Event handlers are able to stop the current event dispatching. For this purpose, the event data sent to the event handlers define the `Consumed` property. When this property is set to **true**, the event is marked as handled and the event propagation stops immediately. If an event is marked as **Consumed** during the bubbling phase, the bubbling phase stops and the event dispatching ends. If the event is marked as **Consumed** during the tunneling phase, the tunneling phase stops and the event dispatching ends with no bubbling phase. Typically, input events should be marked as **Consumed** as soon as an event handler has handled the event to implement the expected behavior. Then, the dispatching is no longer needed.

The following example illustrates how a container consumed the **MouseClicked** event in the tunneling phase to change the stroke of the clicked graphic object. The event is marked as **Consumed** in the tunneling phase so that the event propagation is reduced to its minimum: only the top-level container is notified and the bubbling phase is not executed. It is a typical scenario when you care for performances.

```
public partial class Form1 : Form
{
    private static Stroke DefaultStroke = new Stroke(Color.Black, 3);
    private static Stroke HighlightStroke = new Stroke(Color.Green, 3,
                                                       System.Drawing.Drawing2D.DashStyle.Dash);

    public Form1()
    {
        InitializeComponent();

        DiagramView diagramView1 = new DiagramView();
        diagramView1.Dock = System.Windows.Forms.DockStyle.Fill;
        this.Controls.Add(diagramView1);
        // Create the graphic object hierarchy
        Group content = new Group();
        Canvas container1 = new Canvas();
        container1.Size = new Size2D(300, 250);
        container1.Location = new Point2D(5, 5);
        container1.Background = new SolidFill(Color.LightSteelBlue);
        Rect shape = new Rect(new Rectangle2D(30, 60, 110, 60));
        shape.Stroke = DefaultStroke;
        shape.Fill = new SolidFill(Color.CadetBlue);
        container1.Objects.Add(shape);
        Ellipse ellipse = new Ellipse(new Rectangle2D(100, 130, 90, 70));
        ellipse.Fill = new SolidFill(Color.Coral);
        ellipse.Stroke = DefaultStroke;
        container1.Objects.Add(ellipse);
        content.Objects.Add(container1);
        diagramView1.Content = content;
        container1.PreviewMouseClicked += new
ObjectMouseEventHandler(OnPreviewMouseClicked);
    }

    void OnPreviewMouseClicked(object sender, ObjectMouseEventArgs e)
```

## Understanding Events Dispatching in a Diagram View (WinForms)

```
    {
        if (e.Target != e.CurrentTarget)
        {
            Shape s = (Shape)e.Target;
            s.Stroke = s.Stroke == DefaultStroke ?
                HighlightStroke : DefaultStroke;
            e.Consumed = true;
        }
    }
}

Public Class Form1
Inherits Form
    Private Shared DefaultStroke As Stroke = New Stroke(Color.Black, 3)
    Private Shared HighlightStroke As Stroke = _
        New Stroke(Color.Green, 3, System.Drawing.Drawing2D.DashStyle.Dash)

    Public Sub New()
        InitializeComponent
        Dim diagramView1 As DiagramView = New DiagramView
        diagramView1.Dock = System.Windows.Forms.DockStyle.Fill
        Me.Controls.Add(diagramView1)
        Dim content As Group = New Group
        Dim container1 As Canvas = New Canvas
        container1.Size = New Size2D(300, 250)
        container1.Location = New Point2D(5, 5)
        container1.Background = New SolidFill(Color.LightSteelBlue)
        Dim shape As Rect = New Rect(New Rectangle2D(30, 60, 110, 60))
        shape.Stroke = DefaultStroke
        shape.Fill = New SolidFill(Color.CadetBlue)
        container1.Objects.Add(shape)
        Dim ellipse As Ellipse = New Ellipse(New Rectangle2D(100, 130, 90, 70))
        ellipse.Fill = New SolidFill(Color.Coral)
        ellipse.Stroke = DefaultStroke
        container1.Objects.Add(ellipse)
        content.Objects.Add(container1)
        diagramView1.Content = content
        AddHandler container1.PreviewMouseClicked, AddressOf OnPreviewMouseClicked
    End Sub

    Sub OnPreviewMouseClicked(ByVal sender As Object, _
        ByVal e As MouseEventArgs)
        If Not (e.Target = e.CurrentTarget) Then
            Dim s As Shape = CType(e.Target, Shape)
            If s.Stroke = DefaultStroke Then
                s.Stroke = HighlightStroke
            Else
                s.Stroke = DefaultStroke
            End If
            e.Consumed = True
        End If
    End Sub
End Class
```

---

### Event Capture

In some cases, after a specific event has been received, a graphic object may require to receive all the input events that are coming next, whatever the event target is. This is called

## Handling Interactions in a Diagram View (WinForms)

*event capture*. It means that a graphic object that gets the event capture will receive all the input events received from the view until it releases the capture. When the graphic object that got the capture has completed its events handling it must release the mouse capture.

A diagram view distinguishes two types of capture: the mouse capture to handle all mouse events, and the keyboard capture to handle all keyboard events.

To get the mouse capture, a graphic object must call the `CaptureMouse(GraphicObject)`, while to release the mouse capture it must call the `ReleaseMouseCapture` method. These methods are in the `DiagramView` class.

To get the keyboard capture, a graphic object must call the `CaptureKey(GraphicObject)` method, while to release the keyboard capture it must call the `ReleaseKeyCapture` method. These methods are in the **DiagramView** class.

The following example illustrates how to set the key capture. A `Rect` instance is initialized with the key capture so that every character entered using the keyboard appears in the panel. The capture ends when the `ESC` key is pressed.

```
public partial class Form1 : Form
{
    Rect buffer;
    DiagramView diagramView1;
    public Form1()
    {
        InitializeComponent();
        diagramView1 = new DiagramView();
        diagramView1.AutoSizeContent = true;
        diagramView1.AutoSizeContentMode = ResizeMode.Resize;
        diagramView1.Dock = System.Windows.Forms.DockStyle.Fill;
        diagramView1.KeepAspectRatio = false;
        this.Controls.Add(diagramView1);
        // Create the graphic object hierarchy
        Group content = new Group();
        ScrollViewer scviewer = new ScrollViewer();
        SolidFill whiteFill = new SolidFill(Color.White);
        whiteFill.Freeze();
        scviewer.Background = whiteFill;
        buffer = new Rect();
        buffer.Location = new Point2D(0, 0);
        buffer.Stroke = null;
        buffer.Fill = whiteFill;
        buffer.AutoSize = true;
        buffer.TextAppearance.VerticalAlignment = VerticalTextAlignment.Top;
        buffer.TextAppearance.HorizontalAlignment =
            HorizontalTextAlignment.Left;
        scviewer.Content = buffer;
        scviewer.VerticalScrollBar.MouseUp +=
            new ObjectMouseEventHandler(OnScrollBarMouseUp);
        scviewer.HorizontalScrollBar.MouseUp +=
            new ObjectMouseEventHandler(OnScrollBarMouseUp);
        content.Objects.Add(scviewer);
        diagramView1.Content = content;
        diagramView1.CaptureKey(buffer);
        buffer.KeyDown += new ObjectKeyEventHandler(buffer_KeyDown);
    }
}
```



## Understanding Events Dispatching in a Diagram View (WinForms)

```
        buffer.KeyPress += new ObjectKeyPressEventHandler(buffer_KeyPress);
    }

    void OnScrollBarMouseUp(object sender, ObjectMouseEventArgs e)
    {
        diagramView1.CaptureKey(buffer);
    }

    void buffer_KeyDown(object sender, ObjectKeyEventArgs e)
    {
        if (e.KeyCode == Keys.Escape)
        {
            e.ReleaseKeyCapture();
        }
    }

    void buffer_KeyPress(object sender, ObjectKeyPressEventArgs e)
    {
        char c = e.KeyChar;
        Rect r = (Rect)sender;
        StringBuilder buffer = new StringBuilder(r.Text);
        if (c == 13)
            buffer.AppendLine("");
        else
            buffer.Append(c);
        r.Text = buffer.ToString();
    }
}

Public Class Form1
Inherits Form
Private buffer As Rect
Private diagramView1 As DiagramView

Public Sub New()
    InitializeComponent
    diagramView1 = New DiagramView
    diagramView1.AutoSizeContent = True
    diagramView1.AutoSizeContentMode = ResizeMode.Resize
    diagramView1.Dock = System.Windows.Forms.DockStyle.Fill
    diagramView1.KeepAspectRatio = False
    Me.Controls.Add(diagramView1)
    Dim content As Group = New Group
    Dim scviewer As ScrollViewer = New ScrollViewer
    Dim whiteFill As SolidFill = New SolidFill(Color.White)
    whiteFill.Freeze
    scviewer.Background = whiteFill
    buffer = New Rect
    buffer.Location = New Point2D(0, 0)
    buffer.Stroke = Nothing
    buffer.Fill = whiteFill
    buffer.AutoSize = True
    buffer.TextAppearance.VerticalAlignment = VerticalTextAlignment.Top
    buffer.TextAppearance.HorizontalAlignment = HorizontalTextAlignment.Left
    scviewer.Content = buffer
    AddHandler scviewer.VerticalScrollBar.MouseUp, AddressOf OnScrollBarMouseUp
    AddHandler scviewer.HorizontalScrollBar.MouseUp, AddressOf
OnScrollBarMouseUp
    content.Objects.Add(scviewer)
```

## Handling Interactions in a Diagram View (WinForms)

```
        diagramView1.Content = content
        diagramView1.CaptureKey(buffer)
        AddHandler buffer.KeyDown, AddressOf buffer_KeyDown
        AddHandler buffer.KeyPress, AddressOf buffer_KeyPress
    End Sub

    Sub OnScrollBarMouseUp(ByVal sender As Object, ByVal e As
ObjectMouseEventArgs)
        diagramView1.CaptureKey(buffer)
    End Sub

    Sub buffer_KeyDown(ByVal sender As Object, ByVal e As ObjectKeyEventArgs)
        If e.KeyCode = Keys.Escape Then
            e.ReleaseKeyCapture
        End If
    End Sub

    Sub buffer_KeyPress(ByVal sender As Object, ByVal e As
ObjectKeyPressEventArgs)
        Dim c As Char = e.KeyChar
        Dim r As Rect = CType(sender, Rect)
        Dim buffer As StringBuilder = New StringBuilder(r.Text)
        If c = 13 Then
            buffer.AppendLine("")
        Else
            buffer.Append(c)
        End If
        r.Text = buffer.ToString
    End Sub
End Class
```

---

## Using Predefined Interaction Tools in a Diagram View

Interaction tools make it possible to handle user's input in a diagram view. These interaction tools are called interactors.

### In This Section

#### *Setting an Interactor on a View*

Explains the set a view interactor on a diagram view.

#### *Selection Interactor*

Describes the SelectInteractor class.

#### *Zoom Interactor*

Describes the ZoomInteractor class.

#### *Rotate Interactor*

Describes the RotateInteractor class.

#### *Pan Interactor*

Describes the `PanInteractor` class.

### *Rectangular Shapes Creation Interactor*

Describes the `CreateGraphicObjectInteractor` class.

### *Polypoints Shape Creation Interactor*

Describes the `CreatePolyPointsInteractor` class.

### *Link Creation Interactor*

Describes the `CreateLinkInteractor` class.

### *Anchor Editing Interactor*

Describes the `EditAnchorsInteractor` class.

---

## Setting an Interactor on a View

Interactors are provided as a set of classes. The base class of all interactors is the `ViewInteractor` class which defines the basic facility to handle events in a view.

In order to be used, an interactor must be set on a view. When an interactor is set on a view, all input events coming to the view are forwarded to the interactor. To do so, you use the `Interactor` property of the `DiagramView` class.

The following example shows how to set a selection interactor on a view:

```
DiagramView view = new DiagramView();
ViewInteractor interactor = new SelectInteractor();
view.Interactor = interactor;
Dim view As DiagramView = New DiagramView
Dim interactor As ViewInteractor = New SelectInteractor
view.Interactor = interactor
```

---

## Selection Interactor

The selection interactor is defined by the `SelectInteractor` class. It allows you to select, move, resize, rotate, copy, or reparent graphic objects displayed in a `DiagramView`. It also allows you to connect existing links to nodes, as well as to edit the text displayed by a graphic object.

For details about selecting objects in a diagram view, see *Handling Selection in a Diagram*.

## Select a Graphic Object

There are several ways to select graphic objects depending on what you want to do:

- ◆ **Single selection:** to select only one graphic object, click the object with the left mouse button.

- ◆ **Multiple selection on a per-object basis:** to select several graphic objects one at a time, press the CTRL or Shift key and click with the left mouse button the graphic objects you want to select.
- ◆ **Area selection:** to select all the graphic objects that intersect a given rectangular area, move the mouse pointer to an empty area of the view, press the left mouse button and drag the mouse to define the area while keeping the left mouse button pressed. When the selection area is as expected, release the mouse button.
- ◆ **Clear the selection:** to deselect all the objects current selected, click an empty area of the view with the left mouse button.
- ◆ **Deselect an object:** to deselect one particular object from the current selection, press the CTRL or Shift modifier and click with the left mouse button the graphic object to deselect.

When a graphic object is selected, a selection graphic object is displayed on top of the selected object. The selection graphic object enables the user to perform operations specific to the selected graphic object, like resize and rotate operations for objects that support it.

See *Handling Selection in a Diagram* for more information.

### **Move, Copy, or Reparent a Graphic Object**

The selection interactor allows you to move one or several graphic objects in the view:

- ◆ **To move one object:** click the graphic object and drag the mouse to a new position.
- ◆ **To move several objects:** select the objects to move using one of the selection types described in *Select a Graphic Object*. Click the selected graphic objects and drag the mouse to a new position.

Several key modifiers can be used to control the move operation. The table below lists those modifiers:

Key modifier	Description
ALT	By default, the selection interactor snaps the mouse pointer to the view grid if such a grid is active. If you press the ALT key while performing the operation you disable the snap-to-grid behavior and this allows you to freely move or resize the selection.
CTRL	To copy the selected objects instead of simply moving them, press the CTRL key during the interaction.

Key modifier	Description
SHIFT	When moving a graphic object over a panel, the panel is highlighted to show that dropping the graphic object at this location will place it into the panel. To disable this behavior, press the SHIFT key during the interaction.
ESC	When moving graphic objects, pressing the ESC key will cancel the interaction.

### Connect a Link to a Node

The **SelectInteractor** allows you to connect or reconnect existing links to nodes. To do so, you must first select the link you want to connect to display the selection handles of the link. Then, simply drag one of these link extremities over a node to display the anchors to which the link can be connected. For details about connecting links or creating links, see *Link Creation Interactor*.

### Edit the Text of a Graphic Object

By double-clicking a graphic object, the **SelectInteractor** displays an editing box in which you can type a new text. To validate the editing press the ENTER key. To cancel the editing, press the ESCAPE key.

### Configure the SelectInteractor

The **SelectInteractor** behavior can be customized by using the **SelectionStyle** class. An instance of this class can be retrieved on the view on which the interactor has been set by using the **DiagramView.SelectionStyle** property for the Windows® Forms version, and the **DiagramView.SelectionStyle** property for the ASP.NET version.

The following table lists the properties of the SelectionStyle class that can be used to configure the **SelectInteractor** behavior:

Property	Description
CanMove	Indicates whether graphic objects can be moved using the interactor. The default value is <b>true</b> .
CanReparent	Indicates whether graphic objects can be reparented using the interactor. The default value is <b>true</b> .
CanCopy	Indicates whether graphic objects can be copied using the interactor. The default value is <b>true</b> .
CanEditText	Indicates whether graphic objects text can be edited using the interactor. The default value is <b>true</b> .

Property	Description
CanConnect	Indicates whether link can be connected or reconnected using the interactor. The default value is <b>true</b> .
CanRotate	Indicates whether graphic objects can be rotated using the interactor. The default value is <b>true</b> .
CanResize	Indicates whether graphic objects can be resized using the interactor. The default value is <b>true</b> .
MultipleSelection	Indicates whether multiple selection is allowed using the interactor. The default value is <b>true</b> .
RectangleSelection	Indicates whether multiple selection is allowed using the interactor by dragging a rectangle around objects. The default value is <b>true</b> .
InstantEditing	Indicates whether interactions are instantaneous. The default value is <b>true</b> .
SnapToGrid	Indicates whether the interactions should take into account the grid defined on the view to snap graphic objects onto the grid points. The default value is <b>true</b> .

### See Also

*Handling Selection in a Diagram*

---

### Zoom Interactor

The interactor used to zoom a view is defined by the `ZoomInteractor` class. It enables you to zoom in or zoom out a particular area of a view:

- ◆ **Zoom in:** to zoom in an area of a view, drag a rectangle corresponding to the area to zoom and release the mouse button to perform the zoom.
- ◆ **Zoom out:** to zoom out the view, press the Shift key and drag a rectangle corresponding to the area into which the current visible area should appear after the zooming-out operation.

It is possible to cancel the current operation pressing the ESC key while dragging the mouse to define the rectangular area to zoom.

### See Also

*Displaying Diagrams in a Windows Forms Application*

---

### Rotate Interactor

The interactor used to rotate a view is defined by the `RotateInteractor` class.

To rotate the view, press the left mouse button and set the angle of the rotation moving the mouse. The current angle is shown on the reticule that is drawn.

It is possible to cancel the current operation by pressing the ESC key while dragging the mouse to define the rotation angle.

### See Also

*Displaying Diagrams in a Windows Forms Application*

---

### Pan Interactor

The interactor used to pan a view is defined by the `PanInteractor` class.

To pan a view, press the left mouse button on the view and drag the mouse to display the new visible area.

It is possible to cancel the current operation by pressing the ESC key while dragging the mouse to define the new visible area.

### See Also

*Displaying Diagrams in a Windows Forms Application*

---

### Rectangular Shapes Creation Interactor

The interactor used to create rectangular shapes is defined by the `CreateGraphicObjectInteractor` class.

To create a shape, press the left mouse button and drag the mouse to define the rectangle corresponding to the size of the graphic object you want to create.

### Creating Shapes

To create a shape, the **`CreateGraphicObjectInteractor`** class delegates the shape creation to a shape factory. This factory determines the type of shape to create and is specified by means of the `Factory` property.

The following predefined factories are provided with the **`CreateGraphicObjectInteractor`**:

- ◆ `Rect`
- ◆ `Ellipse`
- ◆ `Arc`
- ◆ `Pie`

The following code shows how to configure the interactor to create ellipses:

```
DiagramView view = new DiagramView();
CreateGraphicObjectInteractor inter = new CreateGraphicObjectInteractor();
inter.Factory = CreateGraphicObjectInteractor.EllipseFactory;
view.Interactor = inter;
Dim view As DiagramView = New DiagramView
Dim inter As CreateGraphicObjectInteractor = New CreateGraphicObjectInteractor
```

## Handling Interactions in a Diagram View (WinForms)

```
inter.Factory = CreateGraphicObjectInteractor.EllipseFactory
view.Interactor = inter
```

### Creating Custom Shapes

Custom shape types are supported by implementing the `IGraphicObjectFactory` interface so that the `CreateInstance` method returns an instance of the custom type.

The following code shows the implementation of the default arc factory:

```
class ArcShapeFactory : IGraphicObjectFactory
{
    public GraphicObject CreateInstance(IServiceProvider serviceProvider)
    {
        Arc arc = new Arc();
        arc.StartAngle = 90f;
        arc.SweepAngle = 270f;
        return arc;
    }
}
Class ArcShapeFactory
Implements IGraphicObjectFactory

Public Function CreateInstance(ByVal serviceProvider _
                               As IServiceProvider) As GraphicObject
    Dim arc As Arc = New Arc
    arc.StartAngle = 90F
    arc.SweepAngle = 270F
    Return arc
End Function
End Class
```

### Customizing the Created Shape

The `CreateGraphicObjectInteractor` class defines the `ObjectCreated` event that enables event handlers to customize the created shape.

The following code shows how an event handler sets the `Text` property of every new shapes:

```
public Form1() {
    InitializeComponent();
    CreateGraphicObjectInteractor inter =
        new CreateGraphicObjectInteractor ();
    inter.ObjectCreated +=
        new GraphicObjectCreatedEventHandler(OnObjectCreated);
    diagramView1.Interactor = inter;
}

void OnObjectCreated(object sender, GraphicObjectCreatedEventArgs e) {
    e.GraphicObject.Text = "Hello World !";
}
Public Sub New()
    InitializeComponent
    Dim inter As CreateGraphicObjectInteractor = New CreateGraphicObjectInteractor
    AddHandler inter.ObjectCreated, AddressOf OnObjectCreated
    diagramView1.Interactor = inter
End Sub
```



```
Sub OnObjectCreated(ByVal sender As Object, ByVal e As
GraphicObjectCreatedEventArgs)
    e.GraphicObject.Text = "Hello World !"
End Sub
```

**See Also***Using Predefined Graphic Objects***Polypoints Shape Creation Interactor**

The interactor used to create polypoints shapes is defined by the `CreatePolyPointsInteractor` class.

Depending on the type of shape, there are several ways to create a polypoints shape:

- ◆ Freehand shape: press the left mouse button and drag the mouse to define the outline of the shape. Release the mouse button to end the interaction and create the shape.
- ◆ Point-by-point creation mode: click the left mouse button for each point of the shape. Double-click to end the interaction and create the shape.

**Creating a Polypoints Shape**

To create a polypoints shape, the `CreatePolyPointsInteractor` class delegates the shape creation to a shape factory. This factory determines the type of shape to create and is specified by means of the `Factory` property.

The following predefined factories are provided with the `CreatePolyPointsInteractor`:

- ◆ Curve
- ◆ Closed Curve
- ◆ Polygon
- ◆ Polyline

The following code shows how to configure the interactor to create closed curves:

```
DiagramView view = new DiagramView();
CreatePolyPointsInteractor inter = new CreatePolyPointsInteractor();
inter.Factory = CreatePolyPointsInteractor.ClosedCurveFactory;
view.Interactor = inter;
Dim view As DiagramView = New DiagramView
Dim inter As CreatePolyPointsInteractor = New CreatePolyPointsInteractor
inter.Factory = CreatePolyPointsInteractor.ClosedCurveFactory
view.Interactor = inter
```

**Creating a Custom Polypoints Shape**

Custom polypoints shape types are supported by implementing the `IPolyPointsShapeFactory` interface so that the `CreateInstance` method returns an instance of the custom type.

The following code shows a basic implementation of a factory that creates polygon:

```
class PolygonFactory : IPolyPointsShapeFactory
```

## Handling Interactions in a Diagram View (WinForms)

```
{
    public IPolyPointsShape CreateInstance(IServiceProvider serviceProvider)
    {
        IPolyPointsShape polygon = new Polygon();
        return polygon;
    }

    public int MinimumNumberOfPoints
    {
        get { return 2; }
    }

    public int MaximumNumberOfPoints
    {
        get { return int.MaxValue; }
    }

    GraphicObject IGraphicObjectFactory.CreateInstance(IServiceProvider
serviceProvider)
    {
        return CreateInstance(serviceProvider) as GraphicObject;
    }
}
Class PolygonFactory
Implements IPolyPointsShapeFactory

    Public Function CreateInstance(ByVal serviceProvider As IServiceProvider) As
IPolyPointsShape
        Dim polygon As IPolyPointsShape = New Polygon
        Return polygon
    End Function

    Public ReadOnly Property MinimumNumberOfPoints() As Integer
        Get
            Return 2
        End Get
    End Property

    Public ReadOnly Property MaximumNumberOfPoints() As Integer
        Get
            Return Integer.MaxValue
        End Get
    End Property

    Function IGraphicObjectFactory.CreateInstance(ByVal serviceProvider As
IServiceProvider) As GraphicObject
        Return TryCast(CreateInstance(serviceProvider), GraphicObject)
    End Function
End Class
```

### Customizing the Created Shape

The **CreatePolyPointsInteractor** class defines the `ObjectCreated` event that enables event handlers to customize the created shape.

The following code shows how an event handler sets the **Text** property of every new polypoints shapes:

```
public Form1() {
    InitializeComponent();
    CreatePolyPointsInteractor inter = new CreatePolyPointsInteractor();
    inter.ObjectCreated +=
        new GraphicObjectCreatedEventHandler(OnObjectCreated);
    diagramView1.Interactor = inter;
}

void OnObjectCreated(object sender, GraphicObjectCreatedEventArgs e) {
    e.GraphicObject.Text = "Hello World !";
}

Public Sub New()
    InitializeComponent
    Dim inter As CreatePolyPointsInteractor = _
        New CreatePolyPointsInteractor
    AddHandler inter.ObjectCreated, AddressOf OnObjectCreated
    diagramView1.Interactor = inter
End Sub

Sub OnObjectCreated(ByVal sender As Object,
    ByVal e As GraphicObjectCreatedEventArgs)
    e.GraphicObject.Text = "Hello World !"
End Sub
```

## See Also

*Using Predefined Graphic Objects*

---

### Link Creation Interactor

The interactor used to create links is defined by the `CreateLinkInteractor` class.

To create a new link, the user first clicks the start point of the link, then the end point. Alternatively, the user can press the left mouse button on the start point, drag the mouse, then release the mouse button on the end point.

New links can be created in an empty area of the diagram view, or they can be connected to existing graphic objects. To create a connected link, move the mouse cursor over the start object. The connection points (called anchors) attached to the object are drawn as small circles. Then, move the mouse cursor over the anchor to which the link must be connected and press the left mouse button. The same process is repeated for the end object.

### Moving or Duplicating Anchors

While connecting a link to a graphic object, it is possible to move an anchor if it is not at the desired position. To do this, proceed as follows:

1. press the left mouse button over an anchor to connect the link to its start or end anchor,
2. press the Shift key and drag the anchor to its new position,
3. release the Shift key.

You can complete the operation by connecting the link normally and releasing the left mouse button.

It is also possible to duplicate the anchor. To do this, follow the same steps required for moving an anchor and press the CTRL button instead of the Shift button.

***Note:** Not all types of anchors can be moved: anchors of type `BoundsAnchor` can be moved, while anchors of type `ShapeAnchor` cannot. The interactor displays a tooltip to indicate if it is possible to move and/or duplicate an anchor.*

### Creating Custom Link Type

To create a link object, the **CreateLinkInteractor** class delegates the shape creation to a link factory. This factory determines the type of link to create and is specified by means of the `LinkFactory` property. If no factory has been set, which is the case by default, the **CreateLinkInteractor** class creates new instances of the `Link` class.

The link factory is an implementation of the `IGraphicObjectFactory` interface so that the `CreateInstance` method returns an instance of the custom type.

The following code shows a basic implementation of a factory that creates an instance of a custom link class called **MyLink**:

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        CreateLinkInteractor inter = new CreateLinkInteractor();
        inter.LinkFactory = new MyLinkFactory();
        diagramView1.Interactor = inter;
    }
}

public class MyLinkFactory : IGraphicObjectFactory
{
    public GraphicObject CreateInstance(IServiceProvider serviceProvider)
    {
        return new MyLink();
    }
}

Public Class Form1
Inherits Form

    Public Sub New()
        InitializeComponent
        Dim inter As CreateLinkInteractor = New CreateLinkInteractor
        inter.LinkFactory = New MyLinkFactory
        diagramView1.Interactor = inter
    End Sub
End Class
```

```

Public Class MyLinkFactory
Implements ILinkFactory

    Public Function CreateInstance( _
        ByVal serviceProvider As IServiceProvider) As ILOG.Diagrammer.ILink
        Return New MyLink
    End Function

    Function IGraphicObjectFactory.CreateInstance( _
        ByVal serviceProvider As IServiceProvider) As
ILOG.Diagrammer.GraphicObject
        Return TryCast(CreateInstance(serviceProvider), GraphicObject)
    End Function
End Class

```

### Customizing the Created Shape

The **CreateLinkInteractor** class defines the **ObjectCreated** event that enables event handlers to customize the created shape.

The following code shows how an event handler sets the **ShapeType** and **CrossingStyle** properties of every new link:

```

public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        diagramView1.Content = new Group();
        diagramView1.Content.LinkCrossings.Enabled =
            LinkCrossingsEnableMode.Enabled;
        CreateLinkInteractor inter = new CreateLinkInteractor();
        diagramView1.Interactor = inter;
        inter.ObjectCreated +=
            new GraphicObjectCreatedEventHandler(OnObjectCreated);
    }

    void OnObjectCreated(object sender, GraphicObjectCreatedEventArgs e)
    {
        Link link = e.GraphicObject as Link;
        link.ShapeType = LinkShapeType.Orthogonal;
        link.CrossingStyle = LinkCrossingsStyle.Bridge;
    }
}
Public Class Form1
Inherits Form

Public Sub New()
    InitializeComponent
    diagramView1.Content = New Group
    diagramView1.Content.LinkCrossings.Enabled = LinkCrossingsEnableMode.Enabled
    Dim inter As CreateLinkInteractor = New CreateLinkInteractor
    diagramView1.Interactor = inter
    AddHandler inter.ObjectCreated, AddressOf OnObjectCreated
End Sub

Sub OnObjectCreated(ByVal sender As Object, ByVal e As
GraphicObjectCreatedEventArgs)

```

## Handling Interactions in a Diagram View (WinForms)

```
Dim link As Link = TryCast(e.GraphicObject, Link)
link.ShapeType = LinkShapeType.Orthogonal
link.CrossingStyle = LinkCrossingsStyle.Bridge
End Sub
End Class
```

### See Also

*Link Objects*

*Introducing Link and Anchor Classes*

*Creating a Simple Diagram with Nodes and Links Programmatically*

---

### Anchor Editing Interactor

The interactor used to modify the anchors of graphic objects is defined by the `EditAnchorsInteractor` class. The anchor editing interactor allows you to add, remove, move and duplicate anchors on a graphic object.

To modify the anchors of a graphic object, move the mouse cursor over the object. The existing anchors of the object are drawn as small red circles, as in the link creation interactor.

To add a new anchor, move the mouse cursor over an area where no anchor exists, press the CTRL key and press the left mouse button while holding down the CTRL key. This will add a new `BoundsAnchor` at the desired location.

To remove an anchor, move the mouse cursor over that anchor, press the CTRL key, and press the left mouse button while holding down the CTRL key.

To move an anchor, move the mouse cursor over that anchor, press the left mouse button and drag the anchor to its new position.

To duplicate an anchor, move the mouse cursor over that anchor, press the CTRL key, press the left mouse button and drag the duplicated anchor to its new position while holding down the CTRL key.

The interactor displays a tooltip that explains the possible actions at any given time.

---

## Creating a New Interactor in a DiagramView (WinForms)

IBM® ILOG® Diagram for .NET enables you to define interaction tools on a diagram view to provide a rich user experience by means of interactors.

An interactor makes it possible to handle user's input in a `DiagramView`. When an interactor is set on a view, all input events received by the view are forwarded to the interactor which will then handle the events to accomplish its task.

This section explains how to subclass the `ViewInteractor` class to create a new interactor through a step-by-step example.

---

## Requirements

When you write a new interactor, the first step is to define what the interactor should do when it is set on a view.

The example shows a basic interactor that creates circle objects when the user presses the left mouse button and drags the mouse.

The requirements to implement this interaction are:

- ◆ Circle objects are instances of the `Ellipse` class.
- ◆ To define the circle position and size, the user presses the left mouse button on the view to define the circle center and drags the mouse to define the circle radius.
- ◆ The user must be able to cancel the operation by pressing the ESC key.
- ◆ There must be a graphical feedback of the current operation.
- ◆ The feedback rendering must be customizable (like the line width or the line color).

---

## Creating the Interactor

To handle input events in a view, you have to create an interactor class. The base class for interactors is the **ViewInteractor** class which defines the basic functionalities and services to handle input events in a **DiagramView**.

In order to customize the rendering of the interaction feedback, the interactor defines the **GhostStroke** property as an instance of the `Stroke` class. The **Stroke** class encapsulates all the graphical attributes that define the ghost appearance.

The following example shows the class definition as well as the **GhostStroke** property.

```
public class CircleInteractor : ViewInteractor
{
    // the interaction starting point
    Point2D _startPt = Point2D.Empty;
    // the circle radius
    float _radius;
    // the ghost stroke
    private Stroke _stroke = new Stroke(Color.Black, 1, DashStyle.Dot);

    public CircleInteractor() : this(MouseButtons.Left)
    {
    }

    public CircleInteractor(MouseButtons button) : base(button)
    {
    }

    public Stroke GhostStroke
    {
        get {
            return _stroke;
        }
    }
}
```

## Handling Interactions in a Diagram View (WinForms)

```
    }
    set {
        _stroke = value;
    }
}
...
}
Public Class CircleInteractor
    Inherits ViewInteractor
    ' the interaction starting point
    Private _startPt As Point2D = Point2D.Empty
    ' the circle radius
    Private _radius As Single
    ' the ghost stroke
    Private _stroke As Stroke = New Stroke(Color.Black, 1, DashStyle.Dot)

    Public Sub New()
        MyClass.New(MouseButtons.Left)
    End Sub

    Public Sub New(ByVal button As MouseButtons)
        MyBase.New(button)
    End Sub

    Public Property GhostStroke() As Stroke
        Get
            Return _stroke
        End Get
        Set(ByVal value As Stroke)
            _stroke = value
        End Set
    End Property
...
End Class
```

---

### Handling Input Events in a ViewInteractor Subclass

A view interactor consists of implementing a behavior in response to input events that occur in a view. When a user input event occurs in the view, the current view interactor is notified of the event so that it performs the appropriate action.

It is possible to handle input events in a **ViewInteractor** subclass by overriding one of the following methods:

- ◆ OnMouseDown(MouseEventArgs)
- ◆ OnMouseMove(MouseEventArgs)
- ◆ OnMouseUp(MouseEventArgs)
- ◆ OnMouseClicked(MouseEventArgs)
- ◆ OnMouseDoubleClick(MouseEventArgs)
- ◆ OnMouseEnter(EventArgs)



- ◆ OnMouseLeave(EventArgs)
- ◆ OnMouseWheel(MouseEventArgs)
- ◆ OnKeyDown(KeyEventArgs)
- ◆ OnKeyUp(KeyEventArgs)
- ◆ OnKeyPress(KeyPressEventArgs)

In this example, to implement the behavior of the interactor as defined by the requirements, the following actions need to be performed:

- ◆ Memorize the location of the pointer when the mouse button is pressed.
- ◆ Compute the circle radius when the mouse is moved while the mouse button is still pressed.
- ◆ Create the circle and add it to the view content when the mouse button is released.
- ◆ Cancel the current interaction when the ESC key is pressed.

Therefore, the interactor must handle the MouseDown, MouseMove, MouseUp and KeyDown events.

The following example shows the implementation of the input events handlers :

```
protected override void OnMouseDown(MouseEventArgs e)
{
    if (e.Button == Buttons)
    {
        _startPt = new Point2D(e.X, e.Y);
        StartInteraction();
    }
    base.OnMouseDown(e);
}

protected override void OnMouseMove(MouseEventArgs e)
{
    if (InOperation)
    {
        Rectangle bounds = GetGhostBounds();
        Point2D currentPt = new Point2D(e.X, e.Y);
        _radius = GetDistance(_startPt, currentPt);
    }
    base.OnMouseMove(e);
}

protected override void OnMouseUp(MouseEventArgs e)
{
    if (InOperation)
    {
        if (_radius > 3)
            DoIt();
        StopInteraction(true);
    }
    base.OnMouseUp(e);
}
```

## Handling Interactions in a Diagram View (WinForms)

```
}

protected override void OnKeyDown(KeyEventArgs e)
{
    if (e.KeyCode == Keys.Escape)
    {
        StopInteraction(false);
    }
    base.OnKeyDown(e);
}

Protected Overloads Overrides Sub OnMouseDown(ByVal e As MouseEventArgs)
    If e.Button = Buttons Then
        _startPt = New Point2D(e.X, e.Y)
        StartInteraction
    End If
    MyBase.OnMouseDown(e)
End Sub

Protected Overloads Overrides Sub OnMouseMove(ByVal e As MouseEventArgs)
    If InOperation Then
        Dim bounds As Rectangle = GetGhostBounds
        Dim currentPt As Point2D = New Point2D(e.X, e.Y)
        _radius = GetDistance(_startPt, currentPt)
    End If
    MyBase.OnMouseMove(e)
End Sub

Protected Overloads Overrides Sub OnMouseUp(ByVal e As MouseEventArgs)
    If InOperation Then
        If _radius > 3 Then
            DoIt
        End If
        StopInteraction(True)
    End If
    MyBase.OnMouseUp(e)
End Sub

Protected Overloads Overrides Sub OnKeyDown(ByVal e As KeyEventArgs)
    If e.KeyCode = Keys.Escape Then
        StopInteraction(False)
    End If
    MyBase.OnKeyDown(e)
End Sub
```

---

### Visual Feedback During the Interaction

An additional requirement is the visual feedback of the interaction while the user drags the mouse to define the circle radius. Such feedback is called *ghost drawing* in the IBM ILOG Diagram for .NET framework and is natively supported by the **ViewInteractor** API by means of the `DrawGhost(DrawingContext)` method.

The **DrawGhost(DrawingContext)** method is called when the view is redrawn so that the interactor has a visual feedback on the view. The default implementation does nothing and the subclasses are free to implement it.

In this example, the interactor draws a circle to illustrate the current circle geometry defined by the current values of the center and radius.

The following example shows the implementation of the **DrawGhost(DrawingContext)** method:

```
protected override void DrawGhost(DrawingContext context)
{
    if (InOperation)
    {
        Graphics g = context.Graphics;
        Rectangle bounds = GetGhostGeometry();
        using (Pen pen = _stroke.GetPen(
            new Rectangle2D(bounds.X, bounds.Y,
                bounds.Width, bounds.Height),
            View.Transform))
        {
            g.DrawEllipse(pen, bounds);
        }
    }
    else
    {
        base.DrawGhost(context);
    }
}

Protected Overloads Overrides Sub DrawGhost(ByVal context As DrawingContext)
If InOperation Then
    Dim g As Graphics = context.Graphics
    Dim bounds As Rectangle = GetGhostGeometry
    ' Using
    Dim pen As Pen = _stroke.GetPen( _
        New Rectangle2D(bounds.X, bounds.Y, _
            bounds.Width, bounds.Height), _
        View.Transform)
    Try
        g.DrawEllipse(pen, bounds)
    Finally
        CType(pen, IDisposable).Dispose()
    End Try
Else
    MyBase.DrawGhost(context)
End If
End Sub
```

As mentioned above, the **DrawGhost** method is called when the view is invalidated. However, it is the responsibility of the interactor to invalidate the view area corresponding to the ghost bounds.

Therefore, the code of the input event handlers should be modified so that the view is invalidated when the mouse is pressed or dragged to draw the ghost, and when the mouse is released to erase the ghost.

The following example shows the complete code of the interactor:

```
using System;
```

## Handling Interactions in a Diagram View (WinForms)

```
using System.Windows.Forms;
using System.Drawing;
using System.Drawing.Drawing2D;
using ILOG.Diagrammer;
using ILOG.Diagrammer.Graphic;
using ILOG.Diagrammer.Windows.Forms;

namespace CustomInteractor
{
    public class CircleInteractor : ViewInteractor
    {
        Point2D _startPt = Point2D.Empty;
        float _radius;
        Stroke _stroke = new Stroke(Color.Black, 1, DashStyle.Dot);

        public CircleInteractor()
            : this(MouseButtons.Left)
        {
        }

        public CircleInteractor(MouseButtons button)
            : base(button)
        {
        }

        public Stroke GhostStroke
        {
            get
            {
                return _stroke;
            }
            set
            {
                _stroke = value;
            }
        }

        protected override void OnMouseDown(MouseEventArgs e)
        {
            if (e.Button == Buttons)
            {
                _startPt = new Point2D(e.X, e.Y);
                StartInteraction();
            }
            base.OnMouseDown(e);
        }

        protected override void OnMouseMove(MouseEventArgs e)
        {
            if (InOperation)
            {
                Rectangle bounds = GetGhostBounds();
                Point2D currentPt = new Point2D(e.X, e.Y);
                _radius = GetDistance(_startPt, currentPt);
                if (bounds.IsEmpty)
                    bounds = GetGhostBounds();
            }
        }
    }
}
```

```

        else
            bounds = Rectangle.Union(bounds, GetGhostBounds());
            InvalidateView(bounds);
        }
        base.OnMouseMove(e);
    }

protected override void OnMouseUp(MouseEventArgs e)
{
    if (InOperation)
    {
        if (_radius > 3)
            DoIt();
        Rectangle bounds = GetGhostBounds();
        StopInteraction(true);
        InvalidateView(bounds);
    }
    base.OnMouseUp(e);
}

protected override void OnKeyDown(KeyEventArgs e)
{
    if (e.KeyCode == Keys.Escape)
    {
        Rectangle bounds = GetGhostBounds();
        View.Invalidate(bounds);
        StopInteraction(false);
    }
    base.OnKeyDown(e);
}

protected Rectangle GetGhostGeometry()
{
    Rectangle bounds =
        new Rectangle((int)_startPt.X - (int)Math.Floor(_radius),
            (int)_startPt.Y - (int)Math.Floor(_radius),
            (int)(2 * _radius + 1),
            (int)(2 * _radius + 1));
    return bounds;
}

protected Rectangle GetGhostBounds()
{
    Rectangle bounds = GetGhostGeometry();
    using (Pen pen = _stroke.GetPenForBounds(View.Transform))
    {
        float width = pen.Width;
        bounds.Inflate((int)width, (int)width);
        bounds.Offset((int)(-width / 2), (int)(-width / 2));
    }
    return bounds;
}

protected override void DrawGhost(DrawingContext context)
{
    if (InOperation)
    {
        Graphics g = context.Graphics;

```

## Handling Interactions in a Diagram View (WinForms)

```
        Rectangle bounds = GetGhostGeometry();
        using (Pen pen = _stroke.GetPen(
            new Rectangle2D(bounds.X, bounds.Y,
                bounds.Width, bounds.Height),
            View.Transform))
        {
            g.DrawEllipse(pen, bounds);
        }
    }
    else
    {
        base.DrawGhost(context);
    }
}

protected override void StopInteraction(bool validate)
{
    _startPt = Point2D.Empty;
    _radius = 0;
    base.StopInteraction(validate);
}

protected virtual void DoIt()
{
    Rectangle rect = GetGhostGeometry();
    Ellipse ellipse = new Ellipse();
    Transform t = View.Content.Transform;
    t = t * View.Transform;
    t = t.Inverse();

    // Computes the points of the rect in the container coordinates
    Point2D[] points = GetPointsFromRectangle(rect);
    t.TransformPoints(points);
    // Sets the object's size
    if (rect.Width != 0f || rect.Height != 0f)
    {
        float w = GetDistance(points[0], points[1]);
        float h = GetDistance(points[1], points[2]);
        ellipse.Size = new Size2D(w, h);
    }
    // Sets the object's location
    Point2D center = t.TransformPoint(_startPt);
    ellipse.Move(center, ContentAlignment.MiddleCenter);
    GraphicObject[] obj = new GraphicObject[] { ellipse };
    IManageChildren container = View.Content as IManageChildren;
    if (container != null)
        container.AddChildren(obj);
    if (SelectOnCreate) {
        View.Selection.SetSelectedObjects(obj);
    }
}

static private float GetDistance(Point2D p1, Point2D p2)
{
    double a = (p2.X - p1.X);
    double b = (p2.Y - p1.Y);
    return (float)Math.Sqrt(a * a + b * b);
}
```

## Creating a New Interactor in a DiagramView (WinForms)

```
static public Point2D[] GetPointsFromRectangle(Rectangle2D rect)
{
    return new Point2D[] {
        new Point2D(rect.Left, rect.Top),
        new Point2D(rect.Right, rect.Top),
        new Point2D(rect.Right, rect.Bottom),
        new Point2D(rect.Left, rect.Bottom) };
}

Imports System
Imports System.Windows.Forms
Imports System.Drawing
Imports System.Drawing.Drawing2D
Imports ILOG.Diagrammer
Imports ILOG.Diagrammer.Graphic
Imports ILOG.Diagrammer.Windows.Forms

Public Class CircleInteractor
    Inherits ViewInteractor
    Private _startPt As Point2D = Point2D.Empty
    Private _radius As Single
    Private _stroke As Stroke = New Stroke(Color.Black, 1, DashStyle.Dot)
    Public Sub New()
        MyClass.New(MouseButtons.Left)
    End Sub

    Public Sub New(ByVal button As MouseButtons)
        MyBase.New(button)
    End Sub

    Public Property GhostStroke() As Stroke
        Get
            Return _stroke
        End Get
        Set(ByVal value As Stroke)
            _stroke = value
        End Set
    End Property

    Protected Overloads Overrides Sub OnMouseDown(ByVal e As MouseEventArgs)
        If e.Button = Buttons Then
            _startPt = New Point2D(e.X, e.Y)
            StartInteraction()
        End If
        MyBase.OnMouseDown(e)
    End Sub

    Protected Overloads Overrides Sub OnMouseMove(ByVal e As MouseEventArgs)
        If InOperation Then
            ' get the old bounds. Needed to compute the view area to
invalidate.
            Dim bounds As Rectangle = GetGhostBounds()
            Dim currentPt As Point2D = New Point2D(e.X, e.Y)
            _radius = GetDistance(_startPt, currentPt)
            ' compute the area to invalidate.

```

## Handling Interactions in a Diagram View (WinForms)

```
        If bounds.IsEmpty Then
            bounds = GetGhostBounds()
        Else
            bounds = Rectangle.Union(bounds, GetGhostBounds)
        End If
        ' Invalidate the view.
        InvalidateView(bounds)
    End If
    MyBase.OnMouseMove(e)
End Sub

Protected Overloads Overrides Sub OnMouseUp(ByVal e As MouseEventArgs)
    If InOperation Then
        If _radius > 3 Then
            DoIt()
        End If
        Dim bounds As Rectangle = GetGhostBounds()
        StopInteraction(True)
        InvalidateView(bounds)
    End If
    MyBase.OnMouseUp(e)
End Sub

Protected Overloads Overrides Sub OnKeyDown(ByVal e As KeyEventArgs)
    ' if the user press the Escape key, cancel the interaction.
    If e.KeyCode = Keys.Escape Then
        Dim bounds As Rectangle = GetGhostBounds()
        View.Invalidate(bounds)
        StopInteraction(False)
    End If
    MyBase.OnKeyDown(e)
End Sub

Protected Function GetGhostGeometry() As Rectangle
    Dim bounds As Rectangle = _
        New Rectangle(CType(_startPt.X, Integer) - _
            CType(Math.Floor(_radius), Integer), _
            CType(_startPt.Y, Integer) - _
            CType(Math.Floor(_radius), Integer), _
            CType((2 * _radius + 1), Integer), _
            CType((2 * _radius + 1), Integer))

    Return bounds
End Function

Protected Function GetGhostBounds() As Rectangle
    Dim bounds As Rectangle = GetGhostGeometry()
    ' expand the geometry bounds by the stroke width.
    Dim pen As Pen = _stroke.GetPenForBounds(View.Transform)
    Try
        Dim width As Single = pen.Width
        bounds.Inflate(CType(width, Integer), CType(width, Integer))
        bounds.Offset(CType((-width / 2), Integer), CType((-width / 2),
Integer))
    Finally
        CType(pen, IDisposable).Dispose()
    End Try
    Return bounds
End Function
```



## Creating a New Interactor in a DiagramView (WinForms)

```

Protected Overloads Overrides Sub DrawGhost(ByVal context As
DrawingContext)
    If InOperation Then
        Dim g As Graphics = context.Graphics
        Dim bounds As Rectangle = GetGhostGeometry()
        ' Using
        Dim pen As Pen = _stroke.GetPen(New Rectangle2D(bounds.X, bounds.Y,
-
                                                bounds.Width, _
                                                bounds.Height), _
                                                View.Transform)

        Try
            g.DrawEllipse(pen, bounds)
        Finally
            CType(pen, IDisposable).Dispose()
        End Try
    Else
        MyBase.DrawGhost(context)
    End If
End Sub

C
Protected Overloads Overrides Sub StopInteraction(ByVal validate As
Boolean)
    ' reset fields to default values
    _startPt = Point2D.Empty
    _radius = 0
    MyBase.StopInteraction(validate)
End Sub

Protected Overridable Sub DoIt()
    Dim rect As Rectangle = GetGhostGeometry()
    Dim ellipse As Ellipse = New Ellipse
    Dim t As Transform = View.Content.Transform
    t = t * View.Transform
    t = t.Inverse

    ' Computes the points of the rect in the container coordinates
    Dim points As Point2D() = GetPointsFromRectangle(rect)
    t.TransformPoints(points)
    ' Sets the object's size
    If Not (rect.Width = 0.0F) OrElse Not (rect.Height = 0.0F) Then
        Dim w As Single = GetDistance(points(0), points(1))
        Dim h As Single = GetDistance(points(1), points(2))
        ellipse.Size = New Size2D(w, h)
    End If
    ' Sets the object's location
    Dim center As Point2D = t.TransformPoint(_startPt)
    ellipse.Move(center, ContentAlignment.MiddleCenter)

    Dim obj As GraphicObject() = New GraphicObject() {ellipse}
    Dim container As IManageChildren = CType(View.Content, IManageChildren)
    If Not (container Is Nothing) Then
        container.AddChildren(obj)
    End If
    If SelectOnCreate Then
        View.Selection.SetSelectedObjects(obj)
    End If

```

## Handling Interactions in a Diagram View (WinForms)

```
End Sub

Private Shared Function GetDistance(ByVal p1 As Point2D, ByVal p2 As
Point2D) As Single
    Dim a As Double = (p2.X - p1.X)
    Dim b As Double = (p2.Y - p1.Y)
    Return CType(Math.Sqrt(a * a + b * b), Single)
End Function

Public Shared Function GetPointsFromRectangle(ByVal rect As Rectangle2D) As
Point2D()
    Return New Point2D() {New Point2D(rect.Left, rect.Top), _
        New Point2D(rect.Right, rect.Top), _
        New Point2D(rect.Right, rect.Bottom), _
        New Point2D(rect.Left, rect.Bottom)}
End Function

End Class
```

# *Handling Selection in a Diagram*

Selecting a graphic object in a diagram can be useful to:

- ◆ differentiate this object from non-selected objects by changing its appearance,
- ◆ interact with this object by adding interactive decorations.

IBM® ILOG® Diagram for .NET provides several classes that help you manage the selection in a diagram view. Most of these classes are shared by the WinForms and ASP.NET version of the diagram view.

## **In This Section**

### *Managing Selected Objects*

Explains how to add or remove objects to or from the selection using the `SelectionService` class.

### *Listening to Selection Events*

Explains how to listen to selection events.

### *Using Predefined Selection Graphic Objects*

Lists the predefined selection graphic objects.

### *Styling Selection Graphic Objects*

Explains how to change the rendering of selection graphic objects.

### *Creating Custom Selection Graphic Objects*

Explains how to create custom selection graphic objects.

---

## Managing Selected Objects

IBM® ILOG® Diagram for .NET provides an API to manage the selection of graphic objects in a diagram view. This API is mainly composed by the SelectionService class, which contains methods to access the selected objects as well as methods to add or remove objects to or from the selection.

In addition to the API, built-in interactors are provided to manage the selection by interacting directly in the DiagramView rather than using the API. For more information on how to use this interactor, see *Using Predefined Interaction Tools in a Diagram View*.

The objects that are selected in a diagram view can be accessed through the Selection property for the WinForms version and the Selection property for the ASP.NET® version. The type of this property is **SelectionService**.

The following example shows how to create a WinForms **DiagramView** with a few objects and how to select them:

```
public class MyForm : Form
{
    public MyForm()
    {
        // Creates a DiagramView and dock it in the form
        DiagramView view = new DiagramView();
        view.Dock = System.Windows.Forms.DockStyle.Fill;
        Controls.Add(view);

        // Fills the view content with a Rect and an Ellipse
        Group g = new Group();
        Rect r = new Rect(10, 10, 100, 100);
        Ellipse e = new Ellipse(150, 50, 100, 100);
        g.Objects.Add(r);
        g.Objects.Add(e);
        view.Content = g;

        // Selects the Rect object
        view.Selection.AddSelectedObject(r);
    }
}
Public Class MyForm
    Inherits Form

    Public Sub New()
        MyBase.New

        ' Creates a DiagramView and dock it in the form
        Dim view As DiagramView = New DiagramView
        view.Dock = System.Windows.Forms.DockStyle.Fill
        Controls.Add(view)
```

```

    ' Fills the view content with a Rect and an Ellipse
    Dim g As Group = New Group
    Dim r As Rect = New Rect(10, 10, 100, 100)
    Dim e As Ellipse = New Ellipse(150, 50, 100, 100)
    g.Objects.Add(r)
    g.Objects.Add(e)
    view.Content = g

    ' Selects the Rect object
    view.Selection.AddSelectedObject(r)
End Sub
End Class

```

**Notes:**

1. *If you change the whole selection, it is better to use the `SetSelectedObjects` method for performance reasons.*
2. *You can set the `Selection` property to share the selection between different diagram views.*

The **SelectionService** has the notion of primary selection. When the selection contains many graphic objects, the primary selection can be used as the reference for commands that work on the selected objects and that need a reference object. For example, when aligning several objects, the objects are aligned with the primary selection. The primary selection can be get and set using the `PrimarySelection` property. The primary selection is automatically managed by the selection service, which means that there is always a primary selection when the selection is not empty.

---

## Listening to Selection Events

Each time the selection changes in a diagram view, an event is raised so that listeners can be notified of the change.

`SelectionChanged` is the event raised for the WinForms diagram view, `SelectionChanged` is the event raised for the ASP.NET® diagram view. The event argument carried with the event contains information about the nature of the change that occurred in the selection.

The following example shows how to listen to selection events in a WinForms `DiagramView`:

```

public class MyForm : Form
{
    public MyForm()
    {
        // Creates a DiagramView and dock it in the form
        DiagramView view = new DiagramView();
        view.Dock = System.Windows.Forms.DockStyle.Fill;
        Controls.Add(view);
    }
}

```

## Handling Selection in a Diagram

```
// Fills the view content with a Rect and an Ellipse
Group g = new Group();
Rect r = new Rect(10, 10, 100, 100);
Ellipse e = new Ellipse(150, 50, 100, 100);
g.Objects.Add(r);
g.Objects.Add(e);
view.Content = g;

// Listens to selection changed events
view.SelectionChanged +=
    new SelectionChangedEventHandler(SelectionChanged);
// Selects the Rect object
view.Selection.AddSelectedObject(r);
}

void SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    switch (e.Action)
    {
        case ILOG.Diagrammer.SelectionAction.Add:
            Console.WriteLine("Objects added to selection !");
            break;
        case ILOG.Diagrammer.SelectionAction.Remove:
            Console.WriteLine("Objects removed from selection !");
            break;
        case ILOG.Diagrammer.SelectionAction.Primary:
            Console.WriteLine("Primary selection changed !");
            break;
        case ILOG.Diagrammer.SelectionAction.Reset:
            Console.WriteLine("Selection Reset !");
            break;
    }
}
}

Public Class MyForm
    Inherits Form

    Public Sub New()
        MyBase.New

        ' Creates a DiagramView and dock it in the form
        Dim view As DiagramView = New DiagramView
        view.Dock = System.Windows.Forms.DockStyle.Fill
        Controls.Add(view)

        ' Fills the view content with a Rect and an Ellipse
        Dim g As Group = New Group
        Dim r As Rect = New Rect(10, 10, 100, 100)
        Dim e As Ellipse = New Ellipse(150, 50, 100, 100)
        g.Objects.Add(r)
        g.Objects.Add(e)
        view.Content = g

        ' Listens to selection changed events
        AddHandler view.SelectionChanged, AddressOf Me.SelectionChanged

        ' Selects the Rect object
```

```

        view.Selection.AddSelectedObject(r)
    End Sub

    Private Sub SelectionChanged(ByVal sender As Object, _
                                ByVal e As SelectionChangedEventArgs)
        Select Case (e.Action)
            Case ILOG.Diagrammer.SelectionAction.Add
                Console.WriteLine("Objects added to selection !")
            Case ILOG.Diagrammer.SelectionAction.Remove
                Console.WriteLine("Objects removed from selection !")
            Case ILOG.Diagrammer.SelectionAction.Primary
                Console.WriteLine("Primary selection changed !")
            Case ILOG.Diagrammer.SelectionAction.Reset
                Console.WriteLine("Selection Reset !")
        End Select
    End Sub
End Class

```

---

## The SelectionGraphic Class

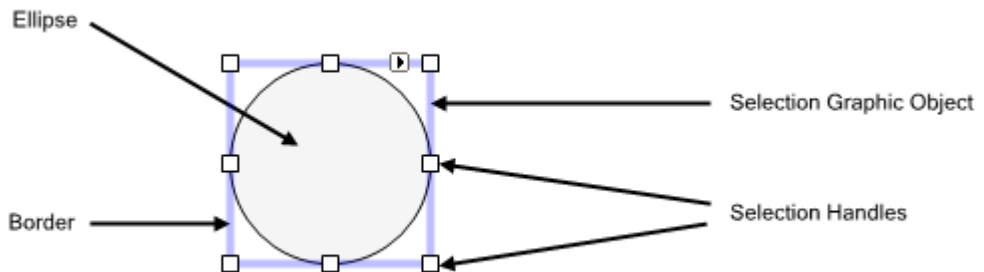
A selection graphic object is a graphic object that is drawn on top of the selected object it refers to. The base class for selection graphic object is the SelectionGraphic class.

Selection graphic objects are managed by the DiagramView: They are created, displayed, and disposed by the **DiagramView**.

---

## Example of Selection Graphic Object

The following illustration shows a selected Ellipse with its associated selection graphic object:



The selection graphic displays a border that can be used to move the selected object and the selection handles that allow you to resize the selected object.

The following example shows how to create and select an **Ellipse** in a **DiagramView**:

## Handling Selection in a Diagram

```
using System.Windows.Forms;
using ILOG.Diagrammer.Windows.Forms;
using ILOG.Diagrammer.Graphic;

public class SelectedEllipse : Form
{
    public SelectedEllipse()
    {
        // Creates the DiagramView
        DiagramView view = new DiagramView();
        view.Dock = System.Windows.Forms.DockStyle.Fill;
        Controls.Add(view);

        // Creates the group displayed by the view
        Group group = new Group();
        Ellipse e = new Ellipse(100, 100, 100, 100);
        group.Objects.Add(e);

        // Displays the group in the view
        view.Contents = group;

        // Selects the ellipse
        view.Selection.AddSelectedObject(e);
    }
}

Imports System.Windows.Forms
Imports ILOG.Diagrammer.Windows.Forms
Imports ILOG.Diagrammer.Graphic

Public Class SelectedEllipse
    Inherits Form

    Public Sub New()
        MyBase.New

        ' Creates the DiagramView
        Dim view As DiagramView = New DiagramView
        view.Dock = System.Windows.Forms.DockStyle.Fill
        Controls.Add(view)

        ' Creates the group displayed by the view
        Dim group As Group = New Group
        Dim e As Ellipse = New Ellipse(100, 100, 100, 100)
        group.Objects.Add(e)

        ' Displays the group in the view
        view.Contents = group

        ' Selects the ellipse
        view.Selection.AddSelectedObject(e)
    End Sub
End Class
```



## Life Cycle of Selection Graphic Objects

For each selected object in a **DiagramView**, a selection graphic object is created and displayed. Similarly, when an object is deselected, its associated selected graphic object is automatically disposed.

To create the selection graphic object associated with a graphic object, the **DiagramView** uses the following mechanism:

- ◆ The `CreateSelectionGraphic` event is raised to enable the creation of the selection graphic object by the event handlers.
- ◆ If no selection graphic object has been created in the previous step, the **SelectionGraphicAttribute** attribute is searched for on the graphic object type that is currently selected. This attribute contains the type of the selection graphic object that should be used for that particular type. If such an attribute is present on a graphic object type, the **DiagramView** creates a selection graphic object using the type specified in the attribute. This only applies when you create new graphic objects.

**Note:** The `GraphicObject` class has the **SelectionGraphicAttribute** attribute set with the `ReshapeSelectionGraphic` class as selection graphic object type. This means that the creation of a selection graphic object will always succeed. If you want to prevent an object from being selected by setting a **null** selection graphic object in the `CreateSelectionGraphic` event handler, do not forget to set the **CreateSelectionGraphicEventArgs.Cancel** to **true**.

The following example shows how to specify a selection graphic object by using the **SelectionGraphicAttribute** attribute:

```
[SelectionGraphic(typeof(MyGraphicObjectSelectionGraphic))]
public class MyGraphicObject : GraphicObject
{
    ...
}

public class MyGraphicObjectSelectionGraphic : SelectionGraphic
{
    ...
}
<SelectionGraphic(GetType(MyGraphicObjectSelectionGraphic))> _
Public Class MyGraphicObject
    Inherits GraphicObject
    ...
End Class

Public Class MyGraphicObjectSelectionGraphic
    Inherits SelectionGraphic
    ...
End Class
```

## Handling Selection in a Diagram

The following example shows how to use the **CreateSelectionGraphic** to create a specific selection graphic object:

```
using System.Windows.Forms;
using ILOG.Diagrammer;
using ILOG.Diagrammer.Windows.Forms;
using ILOG.Diagrammer.Graphic;

public class SelectedEllipse : Form
{
    public SelectedEllipse()
    {
        // Creates the DiagramView
        DiagramView view = new DiagramView();
        view.Dock = System.Windows.Forms.DockStyle.Fill;
        Controls.Add(view);

        // Adds the event handler for the selection
        // graphic object creation
        view.CreateSelectionGraphic +=
            new CreateSelectionGraphicEventHandler(CreateSelectionGraphic);

        // Creates the group displayed by the view
        Group group = new Group();
        Ellipse e = new Ellipse(100, 100, 100, 100);
        group.Objects.Add(e);

        // Displays the group in the view
        view.Contents = group;

        // Select the ellipse
        view.Selection.AddSelectedObject(e);
    }

    void CreateSelectionGraphic(object sender, CreateSelectionGraphicEventArgs
e)
    {
        // Creates a new selection graphic object
        // and sets it to the e.SelectionGraphic property
        // or set the e.Cancel property to true to deny selection
    }
}

Imports System.Windows.Forms
Imports ILOG.Diagrammer
Imports ILOG.Diagrammer.Windows.Forms
Imports ILOG.Diagrammer.Graphic
Public Class SelectedEllipse
    Inherits Form

    Public Sub New()
        MyBase.New

        ' Creates the DiagramView
        Dim view As DiagramView = New DiagramView
        view.Dock = System.Windows.Forms.DockStyle.Fill
        Controls.Add(view)
```

```

' Add the event handler for the selection
' graphic object creation
AddHandler view.CreateSelectionGraphic, _
    AddressOf Me.CreateSelectionGraphic

' Creates the group displayed by the view
Dim group As Group = New Group
Dim e As Ellipse = New Ellipse(100, 100, 100, 100)
group.Objects.Add(e)

' Displays the group in the view
view.Contents = group

' Select the ellipse
view.Selection.AddSelectedObject(e)
End Sub

Private Sub CreateSelectionGraphic(ByVal sender As Object, _
    ByVal e As
CreateSelectionGraphicEventArgs)
' Creates a new selection graphic object
' and set it to the e.SelectionGraphic property
' or set the e.Cancel property to true to deny selection
End Sub
End Class

```

When selected objects are deselected, the associated selection graphic objects are disposed by the **DiagramView**.

---

## Using Predefined Selection Graphic Objects

The following table lists the selection graphic objects available in IBM® ILOG® Diagram for .NET:

Graphic Object Class	Selection Graphic Class	Description
All	ReshapeSelectionGraphic	Provides handles around the object bounding box to enable the resize of the object.
Arc, Pie	ArcSelectionGraphic	Extends the <b>ReshapeSelectionGraphic</b> by adding handles to manipulate the arc angles.
Basic2DShape	Basic2DShapeSelectionGraphic	Extends the <b>ReshapeSelectionGraphic</b> by adding handles to manipulate the control value.
BezierCurve	BezierSelectionGraphic	Provides handles to manipulate the points of the Bezier curve.

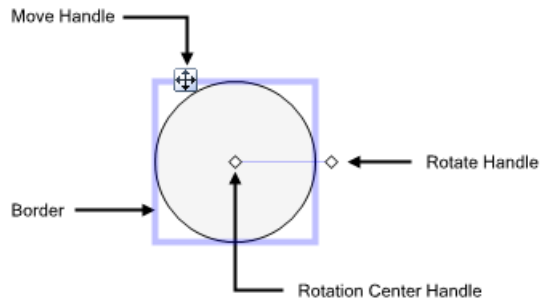
Graphic Object Class	Selection Graphic Class	Description
Circle	CircleSelectionGraphic	Provides handles to change the circle radius.
Control	ControlSelectionGraphic	Graphic selection object dedicated to controls.
Group	GroupSelectionGraphic	Provides a handle to move the group.
Line, LinearScale	LineSelectionGraphic	Provides two handles to manipulate the start and end points of the line.
Link	LinkSelectionGraphic	Provides handles to manipulate the points that define the link.
Path	PathSelectionGraphic	Provides handles to manipulate the points that define the path.
Polyline, Polygon, Curve, ClosedCurve	PolyPointsSelectionGraphic	Provides handles to manipulate the points of the selected object.
Panel	PanelSelectionGraphic	Graphic selection object dedicated to panels.
Text	TextSelectionGraphic	Graphic selection object dedicated to text objects.
TextOnPath	TextOnPathSelectionGraphic	Graphic selection object dedicated to <b>TextOnPath</b> objects.
UserSymbol	UserSymbolSelectionGraphic	Graphic selection object dedicated to user symbols.

---

### Common Behaviors

Most of the selection graphic objects inherit from the DefaultSelectionGraphic class.

The following illustration shows a **DefaultSelectionGraphic** object:



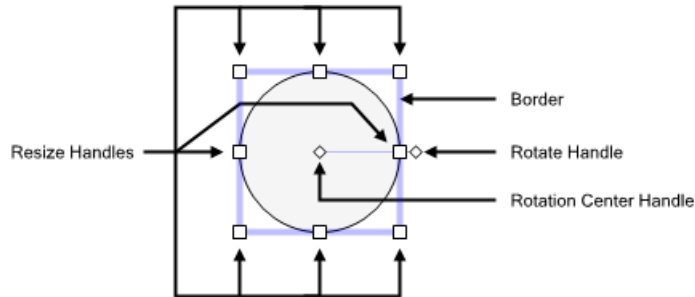
The predefined behaviors of the **DefaultSelectionGraphic** class are described in the following table:

Command	Interaction
Rotate the selected object.	Click and drag the rotate selection handle.
Move the rotation center.	Click and drag the rotation center selection handle.
Copy the selected object.	Click and drag the selection graphic border or the move selection handle with the CTRL key pressed.
Reparent the selected object.	Click and drag the selection graphic border or the move selection handle with the Shift key pressed.
Copy and reparent the selected object.	Click and drag the selection graphic border or the move selection handle with the CTRL and Shift key pressed.
Disable Grid Snapping during current interaction.	Press the ALT key during the interaction.
Cancel the current interaction.	Press the ESC key during the interaction.

### Resizing Behaviors

The **ReshapeSelectionGraphic** enhances the **DefaultSelectionGraphic** class by adding eight new handles around the selected object bounding box to allow you to resize it. Most of the predefined selection graphic objects inherit from the **ReshapeSelectionGraphic**.

The following illustration shows a **ReshapeSelectionGraphic**:



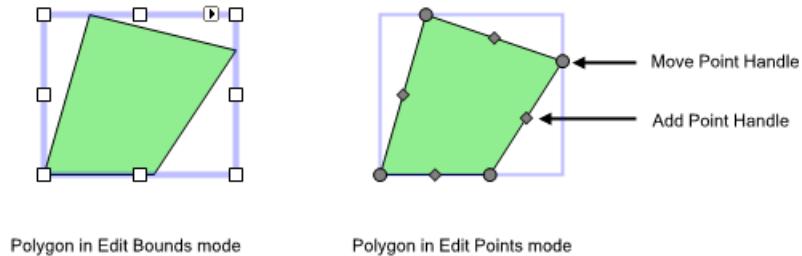
The predefined behaviors of the **ReshapeSelectionGraphic** class are described in the following table:

Command	Interaction
Change the width.	Click and drag the middle-left or middle-right selection handle.
Change the height.	Click and drag the top-center or bottom-center selection handle.
Change both the width and the height.	Click and drag the top-left, top-right, bottom-left or bottom-right selection handle.
Keep the aspect ratio during resizing.	Click and drag one the resize selection handles with the Shift key pressed.
Keep the center fixed during resizing.	Click and drag one the resize selection handles with the CTRL key pressed.

### PolyPoints Editing Behaviors

The **PolyPointsSelectionGraphic** class is the selection graphic class responsible for editing of **PolyPoints** subclasses such as **Polyline**, **Polygon**, **Curve**, and so on. As a subclass of the **ReshapeSelectionGraphic** class, the **PolyPointsSelectionGraphic** class provides handles to resize the selected object. The **PolyPointsSelectionGraphic** also provides a mode in which each point of the selected object has its own selection handle. In this mode, additional handles are also provided to create new points in the middle of each vertex. To switch from one mode to another, simply click the selected object.

The following illustration shows two selected polygons:



Polygon in Edit Bounds mode

Polygon in Edit Points mode

The predefined behaviors of the **PolyPointsSelectionGraphic** class are described in the following table:

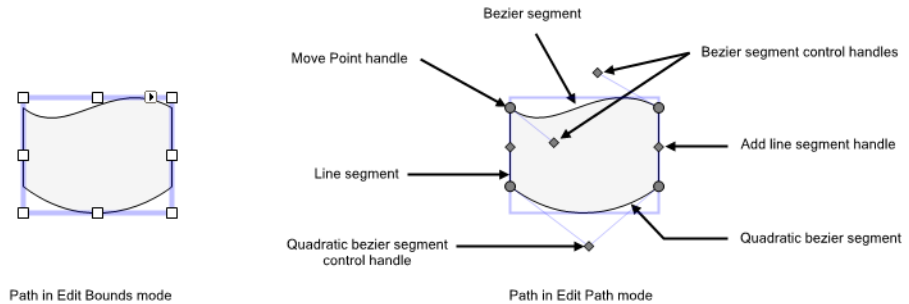
Command	Interaction
Move a point.	Click and drag the selection handle of the point to move it.
Create a new point between two existing points.	Click and drag the diamond shaped selection handle located in the middle of the two points.
Duplicate an existing point.	Click and drag the selection handle of the point to duplicate with the CTRL key pressed.
Remove an existing point.	Click the selection handle of the point to remove with the CTRL key pressed.
Switch to Edit Bounds mode.	Click the selected object while in Edit Points mode.
Switch to Edit Points mode.	Click the selected object while in Edit Bounds mode.

### Path Editing Behaviors

The **PathSelectionGraphic** class is the selection graphic class responsible for editing of graphic objects whose geometry is defined by a **PathData** such as **Path** and **TextOnPath** classes. As a subclass of the **ReshapeSelectionGraphic** class, the **PathSelectionGraphic** class provides handles to resize the selected object. The **PathSelectionGraphic** also provides a mode in which each segment of the selected object path data displays its own selection handles. In this mode, additional handles are also provided to create new segments in the middle of each segment. To switch from one mode to another, simply click the selected object.

The following illustration shows a selected **Path** object and a selected **TextOnPath** object:

## Handling Selection in a Diagram



The predefined behaviors of the **PathSelectionGraphic** class are described in the following table:

Command	Interaction
Move a point.	Click and drag the selection handle of the point to move it. If the Shift key is pressed, smooth curves will not be preserved.
Split a line segment into two line segments.	Click and drag the diamond shaped selection handle located in the middle of the line segment.
Duplicate an existing point of the selected object to create a new line segment.	Click and drag the selection handle of the point to duplicate with the CTRL key pressed.
Convert a line segment into a quadratic Bezier segment.	Click and drag the diamond shaped selection handle located in the middle of the line segment with the CTRL key pressed.
Convert a quadratic Bezier segment into a Bezier segment.	Click and drag the quadratic Bezier segment control selection handle with the CTRL key pressed.
Convert a quadratic Bezier segment into a line segment.	Click the quadratic Bezier segment control selection handle with the CTRL key pressed.
Convert a Bezier segment into a quadratic Bezier segment.	Click one of the Bezier segment control selection handles with the CTRL key pressed.
Remove an existing point of the selected object.	Click the selection handle of the point to remove with the CTRL key pressed.
Switch to Edit Bounds mode.	Click the selected object while in Edit Points mode.
Switch to Edit Points mode.	Click the selected object while in Edit Bounds mode.



---

## Styling Selection Graphic Objects

The rendering of selection graphic objects can be changed at the view level. The common graphic properties have been gathered in the `SelectionMode` class to enable the selection style to be changed in a global way. The following table lists the properties of the `SelectionMode` class that can be used to change the rendering of selection graphic objects:

Property	Description
<code>DashStyle</code>	Gets or sets the dash style used for selection graphic border.
<code>ForeColor</code>	Gets or sets the color used for selection graphic border.
<code>HandleForeColor</code>	Gets or sets the color used for selection graphic handles.
<code>HandleBackColor</code>	Gets or sets the background color used for selection graphic handles.
<code>PrimarySelectionHandleForeColor</code>	Gets or sets the color used for primary selection graphic handles.
<code>PrimarySelectionHandleBackColor</code>	Gets or sets the background color used for primary selection graphic handles.
<code>HandleSize</code>	Gets or sets the size for selection graphic handles.
<code>HighlightColor</code>	Gets or sets the color used to highlight a selection graphic object.
<code>HighlightSize</code>	Gets or sets the size of the border drawn when the selection graphic is highlighted.

In a WinForms diagram view, use the `SelectionMode` property to access the selection style. In an ASP.NET diagram view, use the `SelectionMode` property.

If you need to control the selection graphic rendering more accurately, see *Creating Custom Selection Graphic Objects*.

---

## Creating Custom Selection Graphic Objects

You need to create a custom selection graphic object when:

- ◆ you have created a new graphic object and want to add specific editing capabilities,

## Handling Selection in a Diagram

- ◆ you want to change the selection graphic object of an existing class in order to modify its behavior or appearance.

In both cases, you need to create a subclass of the `SelectionGraphic` class, the base class for selection graphic objects. Depending on the purpose of your selection graphic object, you can decide to inherit directly from the **SelectionGraphic** class or from an existing subclass.

If you want to change the appearance of an object that you have selected you need to modify its properties and subclass the **SelectionGraphic** class.

The following example shows how to change the color of an `Ellipse` when it is selected:

```
public class ShapeSelectionGraphic : SelectionGraphic
{
    private Fill _oldFill;

    public ShapeSelectionGraphic(GraphicObject obj) : base(obj)
    {
        Shape s = obj as Shape;
        // Store current fill
        _oldFill = s.Fill;
        // Change the fill
        s.Fill = new SolidFill(Color.Blue);
    }

    protected override void Dispose(bool disposing)
    {
        // Restore fill before selection
        if (disposing)
            ((Shape)SelectedObject).Fill = _oldFill;
        base.Dispose(disposing);
    }
}

Public Class ShapeSelectionGraphic
    Inherits SelectionGraphic

    Private _oldFill As Fill

    Public Sub New(ByVal obj As GraphicObject)
        MyBase.New(obj)
        Dim s As Shape = CType(obj,Shape)
        ' Store current fill
        _oldFill = s.Fill
        ' Change the fill
        s.Fill = New SolidFill(Color.Blue)
    End Sub

    Protected Overrides Sub Dispose(ByVal disposing As Boolean)
        ' Restore fill before selection
        If disposing Then
            CType(SelectedObject,Shape).Fill = _oldFill
        End If
        MyBase.Dispose(disposing)
    End Sub
End Class
```

If you want to add decorations or handles to manipulate the selected object, you may subclass the `DefaultSelectionGraphic` class or one of its subclasses. The **`DefaultSelectionGraphic`** class defines an API to manage selection handles. Each handle has a location and a specific rendering and behavior associated. The **`DefaultSelectionGraphic`** class defines three handles:

- ◆ to move the selected object
- ◆ to rotate the selected object
- ◆ to change the rotation center

Each subclass of the **`DefaultSelectionGraphic`** class adds new handles to implement specific behaviors. For example, the `ReshapeSelectionGraphic` class adds eight handles to resize the selected object.

The following table lists the methods that you may have to override when subclassing the **`DefaultSelectionGraphic`** class:

Method	Description
<code>GetHandlePoints</code>	Returns a list containing the location of the selection handles.
<code>IsHandleVisible</code>	Indicates whether the specified handle is visible or not.
<code>GetHandleCursor</code>	Gets the cursor associated with the specified handle.
<code>GetHandleType</code>	Returns the type of the selection handle.
<code>GetHandleDescription</code>	Gives textual information about the specified handle.
<code>MoveHandle</code>	Is called when the specified handle is moved.
<code>OnInteractionStarted</code>	Is called when an interaction is started.
<code>OnInteractionValidated</code>	Is called when an interaction is validated.
<code>OnInteractionCanceled</code>	Is called when an interaction is cancelled.

To see the complete implementation of a subclass of the **`DefaultSelectionGraphic`** class, see the sample located in `QuickStarts\CustomSelectionGraphic`.



## ***Building Diagrams and User Symbols Inside Visual Studio***

IBM® ILOG® Diagram for .NET offers a powerful foundation for quickly creating advanced graphical user interfaces. In addition to a fully programmable Software Development Kit (SDK), IBM ILOG Diagram for .NET has a deep integration in Visual Studio to automate the production of applications without coding. This integration allows you to create new graphical symbols and diagrams directly inside Visual Studio.

Several Visual Studio templates are available to help you get started with a new Visual Studio project or incorporate diagrams in your existing project.

Several samples are provided with the product to show you how to create user symbols and diagrams inside Visual Studio.

- ◆ Process Control, located in Samples/Applications/ProcessControl.
- ◆ Tunnel Monitoring, located in Samples/Applications/TunnelMonitoring.
- ◆ Analog Clock, located in Samples/QuickStart/AnalogClock.
- ◆ User Symbols Library, located in Samples/QuickStart/UserSymbolLibrary.

You may also read the following tutorial that guides you through the creation of a user symbol:

*Creating an IBM ILOG Diagram for .NET Windows Forms Application and User Symbol*

### **In This Section**

#### *Creating Diagrams and User Symbols Using Visual Studio*

Introduces the *Diagram Designer*.

#### *Adding Graphic Objects to Diagram or User Symbol*

Describes how to add graphic objects to a diagram or a user symbol

#### *Controlling the Zoom Level*

Describes how to control the zoom level.

#### *Selecting Graphic Objects*

Describes how to select graphic objects.

#### *Moving, Resizing, Rotating and Aligning Graphic Objects*

Explains how to move, resize, rotate and align a graphic object.

#### *Changing Properties of Graphic Objects*

Explains how to use the Properties Window.

#### *Setting Text to a Graphic Object*

Describes how to set text to a graphic object.

#### *Grouping Graphic Objects*

Explain how to group objects.

#### *Manipulating Panels and Other Containers*

Explains how to manipulate panels and other ontainers.

#### *Controlling the Drawing Order of Graphic Objects*

Explains how to control the drawing order of graphic objects.

#### *Inspecting the Structure of a Diagram*

Introduces the Document Outline view.

#### *Showing and Hiding Objects*

Describes how to show and hide objects.

#### *Creating Complex Path Objects*

Explains how to create a Path object.

#### *Cut, Copy and Paste*

Briefly refers to the cut, copy and paste functionalities.

#### *Graph, Link and Anchors*

Describes how to handle Link and Anchor objects.

*Graph Layout*

Introduces to the graph layouts.

*Importing Vector Graphics in SVG or IVN Format*

Explains how to import files in the SVG or IVN format.

*Printing a Diagram*

Shows how to print a diagram.

*Diagram Designer Commands*

Lists the commands available with the Diagram Designer.

**Related Sections**

*Creating your First IBM ILOG Diagram for .NET AJAX Web Site*

Walks you through the process of creating your first IBM ILOG Diagram for .NET AJAX Web Site.

*Creating an IBM ILOG Diagram for .NET Windows Forms Application and User Symbol*

Walks you through the process of creating your first IBM ILOG Diagram for .NET Windows® Forms application.

---

## Creating Diagrams and User Symbols Using Visual Studio

Integrated within Visual Studio, the Diagram Designer is a point-and-click editor that allows you to design diagrams and new graphic objects (user symbols) by dragging graphic objects to the Design view.

You can use the Diagram Designer to:

- ◆ create diagrams by assembling graphic objects,
- ◆ create new graphic objects: the user symbols.

In both cases, the .NET code corresponding to your diagrams or your symbols will be generated. If you create a new diagram, the related code corresponds to a subclass of the Group object. If you create a new graphic object, the related code is represented by subclass of the UserSymbol class.

The Diagram Designer is composed of the following elements:

- ◆ Design View

Is the view where you design your diagram or user symbol. In the Design view you drag the graphic objects that compose the diagram or the symbol being designed.

- ◆ **Toolbox**

Presents a set of predefined graphic objects that you can drag to the Design view.
- ◆ **Properties Window**

Displays the properties of the graphic objects that you have selected in the Design view. From the Properties Window you can modify all the properties of the graphic objects and access to the events fired by the graphic objects.
- ◆ **Diagram Toolbar**

Enables operations like controlling the zoom of the Design view, creating some graphic objects, grouping objects, changing the text properties of the selected graphic objects and more. To open the Diagram Toolbar select View>Toolbars>IBM ILOG Diagram from the menu bar.
- ◆ **Diagram Menu**

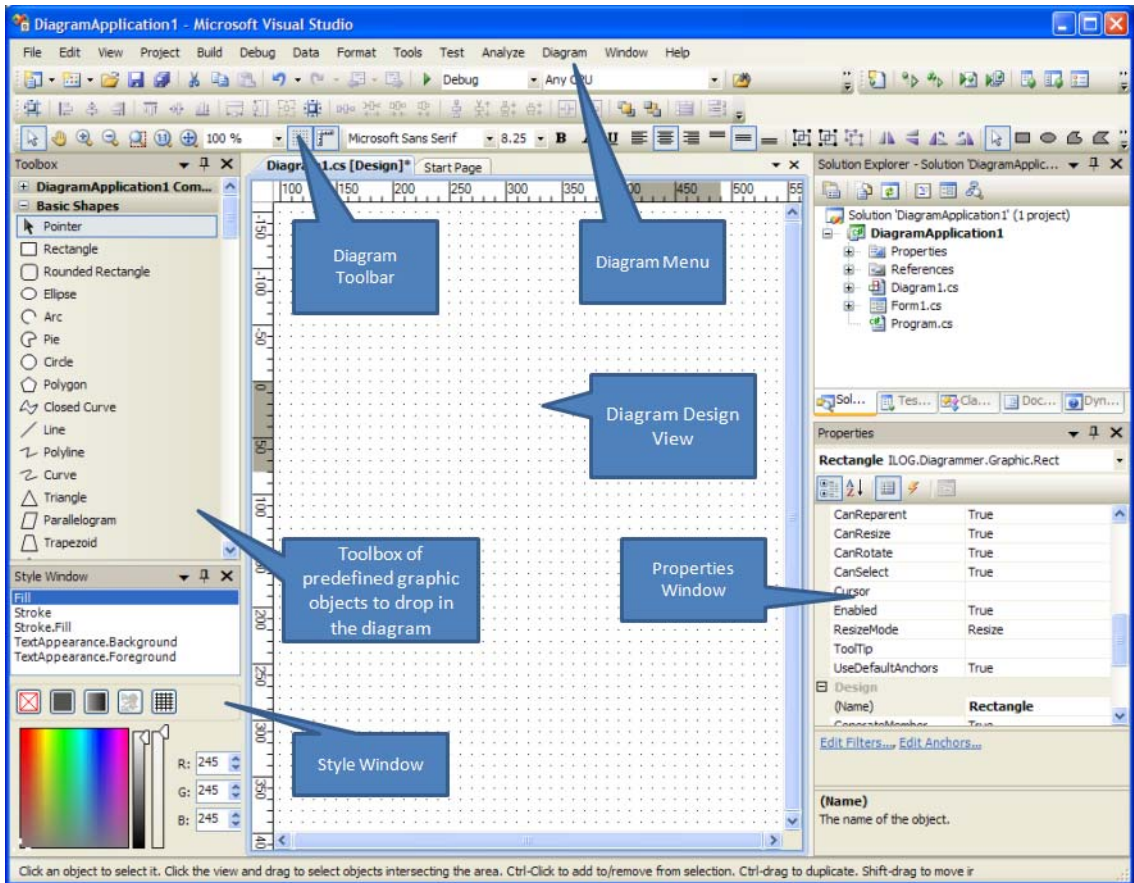
Presents several commands like the zoom commands, or commands to import a file in your diagram.
- ◆ **Style Window**

Presents the Fill and Stroke properties of the selected graphic objects and allows you to change the style of graphic objects. To open the Style Window select View>Other Windows>Diagram Style Window from the menu bar.



## Creating Diagrams and User Symbols Using Visual Studio

The following illustration shows the different elements you can use inside Visual Studio for creating diagrams or symbols.



A number of Visual Studio projects or item templates are installed when you install IBM® ILOG® Diagram for .NET. Through these templates you accelerate the development process and you do not need to create new IBM ILOG Diagram for .NET projects from scratch.

The project templates provide the basic files needed to create a new IBM ILOG Diagram for .NET project. To use these templates choose File>New>Project from the menu bar and select:

- ◆ IBM ILOG Diagram for .NET Application (WinForms) template, to create a Windows® Forms application that contains the diagram displayed in a Diagram View.

or

- ◆ IBM ILOG Diagram for .NET Symbol Library template, to create a library (DLL) of user symbols that you can reuse later in the diagrams of your Windows Forms or Web Forms applications.

Use the project item templates to add new elements to an existing project. To use these templates, right-click an existing project in the Solution Explorer, choose Add>New Item and select:

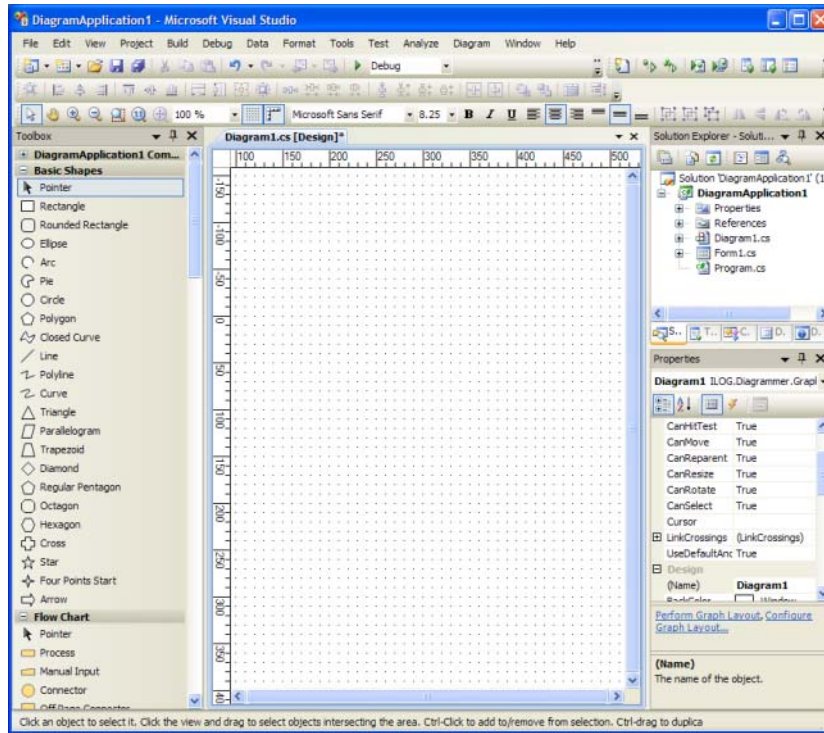
- ◆ Diagram (IBM ILOG Diagram for .NET) template, to add a new diagram in you project, or
- ◆ UserSymbol (IBM ILOG Diagram for .NET) template, to add a new user symbol to your project.

When you create a project with a diagram or a user symbol, Visual Studio opens the diagram or the symbol in design mode and allows you to create the graphical representation by dragging graphic objects from the Toolbox to the Design view.

When you create a diagram inside Visual Studio a new subclass of the **Group** class is created. When you create a new user symbol, a new subclass of the **UserSymbol** class is created.

The concepts of the Diagram Designer are similar to the concept of the Winforms Designer of Visual Studio. In the Winforms Designer you design your interface by creating a new **Form** or a new **UserControl**. In the Diagram Designer you create a new **Group** or a new **UserSymbol**. As you drag graphic objects to the Design view, the code that creates the graphic objects that compose the diagram or user symbol is generated by Visual Studio in the new **Group** or **UserSymbol** class in their **InitializeComponent** method.

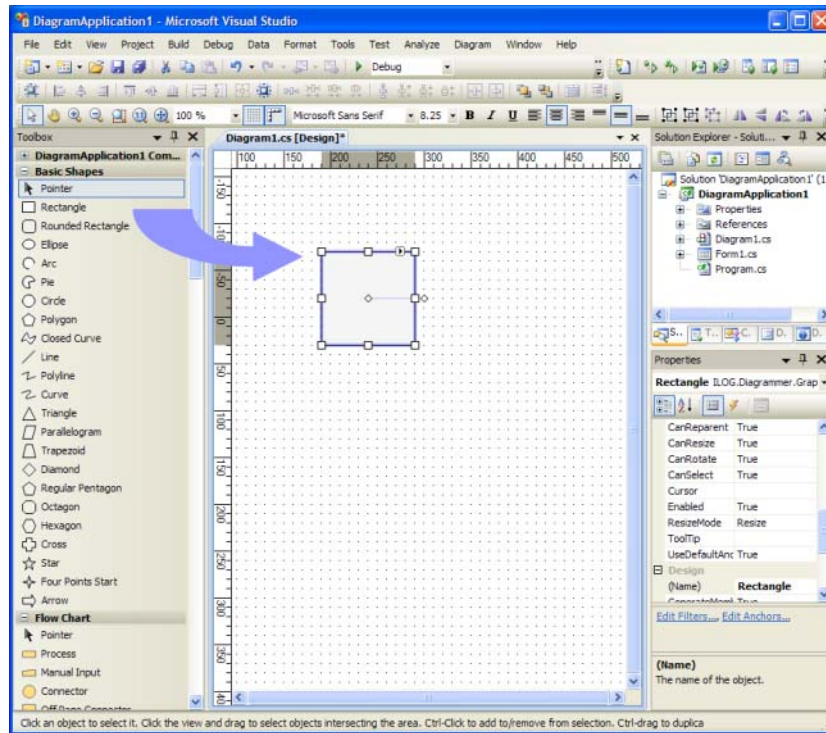
The following illustration shows an empty diagram in Design mode inside Visual Studio.



## Adding Graphic Objects to Diagram or User Symbol

You can add graphic objects to your diagram by dragging objects from the Toolbox. The following illustration shows a rectangle dragged to the Design view.

## Building Diagrams and User Symbols Inside Visual Studio



















As you drag graphic objects to the Design view, the corresponding code is generated in the **InitializeComponent** method of the new class. Here the **InitializeComponent** method that is generated when a single Rect object named rect1 is dragged to a diagram of a C# project.

```
private void InitializeComponent()
{
    this.rect1 = new ILOG.Diagrammer.Graphic.Rect();
    ((System.ComponentModel.ISupportInitialize)(this)).BeginInit();
    //
    // rect1
    //
    this.rect1.Fill = new
ILOG.Diagrammer.SolidFill(System.Drawing.Color.WhiteSmoke);
    this.rect1.Name = "rect1";
    this.rect1.Rectangle = new ILOG.Diagrammer.Rectangle2D(-170F, -200F, 100F,
100F);
    //
    // Diagram1
    //
    this.Objects.Add(this.rect1);
    ((System.ComponentModel.ISupportInitialize)(this)).EndInit();
}
```

## Adding Graphic Objects to Diagram or User Symbol

To add graphic objects to a diagram or a user symbol the following commands are available on the IBM ILOG Diagram for .NET toolbar.

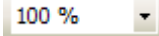





Create Arc		Creates an Arc object.
Create Closed Curve		Creates a ClosedCurve object.
Create Curve		Creates a Curve object.
Create Ellipse		Creates an Ellipse object.
Create Filled Closed Curve		Creates a filled <b>ClosedCurve</b> object.
Create Filled Ellipse		Creates a filled Ellipse object.
Create Filled Polygon		Creates a filled Polygon object.
Create Filled Rectangle		Creates a filled Rect object.
Create FreeHand Shape		Creates a <b>Curve</b> object.
Create Image		Creates an Image object.
Create Link		Creates a Link object.
Create Pie		Creates a Pie object.
Create Polygon		Creates a <b>Polygon</b> object.
Create Polyline		Creates a Polyline object.
Create Rectangle		Creates a <b>Rect</b> object.
Create Text		Creates a Text object.

As you start creating your own graphic object classes, you will be able to place them in the Toolbox and reuse them for creating diagrams inside Visual Studio.

---

### Controlling the Zoom Level

The following options are available for controlling the zoom level of the Design view:

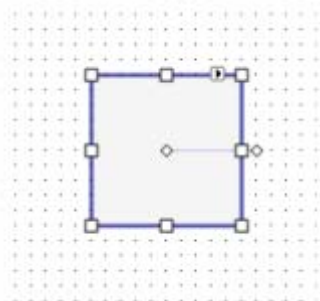
- ◆  to change the zoom level.
- ◆ Mouse wheel + CTRL key to zoom in and out.
- ◆  to zoom in.
- ◆  to zoom out.
- ◆  to adjust the zoom level so that all the graphic objects are visible.
- ◆  to reset the zoom level to 100% (diagram original size).
- ◆  to select the area to zoom in the diagram.

---

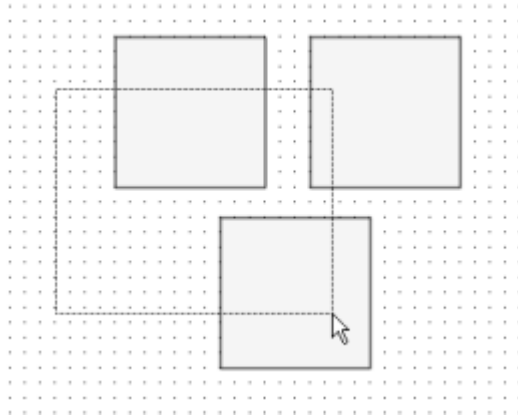
### Selecting Graphic Objects

To select a graphic object in the Design view, click the graphic object. Simultaneously, all the other graphic objects are automatically deselected. To add new graphic objects to the selection, hold down the CTRL or Shift key while clicking the new graphic object. If you click an object which is already selected while pressing the CTRL or Shift key, this object will be deselected.

When a graphic object is selected, the selection handles appear around the graphic object, as shown in the following illustration.



The easiest way to select several graphic objects is to use the lasso. Right-click an empty area of the view and drag the lasso so that it intersects the graphic objects you want to select. When you release the mouse, the elements that intersect the lasso will be selected.



Hold down the CTRL or Shift key while you drag the lasso to invert the selection. Objects already selected will then be deselected.

If you click a selected graphic object and drag the mouse instead of clicking in the background of the view, the selected object will be moved. To avoid this and force the lasso mode, hold down the ALT key before you click.

When several objects are selected, one of them has the selection handles with a different color. This object is the primary selection. For example, the primary selection is used by alignment actions (aligning several objects means to align them on the primary selection).

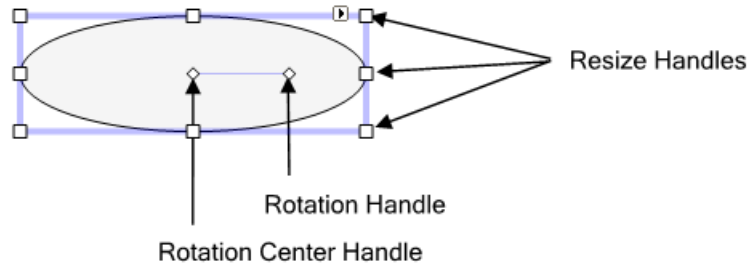
You can also select a graphic object by clicking its name in the drop-down list at the top of the Properties Window or by selecting the object name in the Document Outline view. To open the Document Outline view choose View>Other Windows>Document Outline.

---

## Moving, Resizing, Rotating and Aligning Graphic Objects

To move an object, simply right-click it and drag the mouse.


When a graphic object is selected, the selection handles appear around the object. These handles allow you to resize and rotate the object.



By default a magnetic grid is installed on the Design view so that graphic objects are automatically aligned on the grid when you move or resize them. To prevent grid snapping when moving or resizing a graphic object, you can turn off the grid or press the ALT key while moving the mouse.

You can also move and resize the selected graphic objects using the following keyboard combinations:

- ◆ Arrow keys to move the selected object by the grid spacing in the arrow direction.
- ◆ Shift+Arrow keys to resize the selected object by the grid spacing.
- ◆ CTRL+Arrow keys to move the objects by one pixel on the screen.
- ◆ CTRL+Shift+Arrow keys to resize the selected objects by one pixel on the screen.

The grid can be turned on or off using the Grid button  on the IBM ILOG Diagram for .NET toolbar.
















To modify the options of the grid, click in the background of a Design view and modify the following properties in the Properties Window:





- ◆ **GridActive:** turn on or off the magnetic grid.
- ◆ **GridSpacing:** change the grid spacing.
- ◆ **GridColor:** change the grid color.
- ◆ **GridVisible:** show or hide the grid.

Use the Visual Studio Layout toolbar to access the alignment commands. The following table lists the commands that are available to align the graphic objects.



## Moving, Resizing, Rotating and Aligning Graphic Objects

Command	Toolbar icon	Description
Align to Grid		Moves the graphic object so that the top left corner is aligned on the grid.
Align Lefts		Aligns the left border of the selected graphic objects to the left border of the primary selection.
Align Centers		Aligns the center of the selected graphic objects to the center of the primary selection.
Align Rights		Aligns the right border of the selected graphic objects to the right border of the primary selection.
Align Tops		Aligns the top of the selected graphic objects to the top of the primary selection.
Align Middles		Aligns the middle of the selected graphic objects to the middle of the primary selection.
Align Bottoms		Aligns the bottom of the selected graphic objects to the bottom of the primary selection.
Make Same Width		Resizes the selected graphic objects to the same width of the primary selection.
Make Same Height		Resizes the selected graphic objects to the same height of the primary selection.
Make Same Size		Resizes the selected graphic objects to the same size of the primary selection.
Size to Grid		Resizes the selected graphic objects so that the size fits the grid.
Make Horizontal Spacing Equal		Moves the selected graphic objects so that the horizontal spacing between them is the same.
Increase Horizontal Spacing		Moves the selected graphic objects in order to increase the horizontal spacing between them.
Decrease Horizontal Spacing		Moves the selected graphic objects in order to decrease the horizontal spacing between them.
Remove Horizontal Spacing		Moves the selected graphic objects in order to remove the horizontal spacing between them.

Command	Toolbar icon	Description
Make Vertical Spacing Equal		Moves the selected graphic objects so that the vertical spacing between them is the same.
Increase Vertical Spacing		Moves the selected graphic objects in order to increase the vertical spacing between them.
Decrease Vertical Spacing		Moves the selected graphic objects in order to decrease the vertical spacing between them.
Remove Vertical Spacing		Moves the selected graphic objects in order to remove the vertical spacing between them.

The alignment commands are also available in the Format menu of Visual Studio.

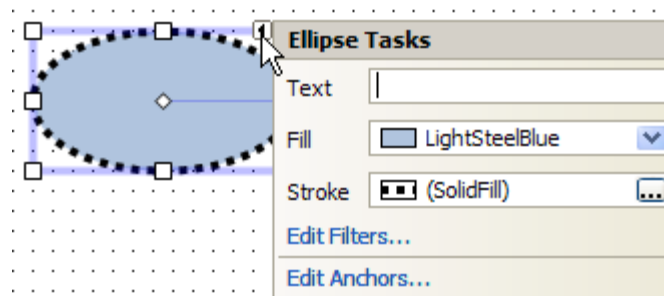
## Changing Properties of Graphic Objects

The Properties Window gives access to the properties and events of a graphic object. To open the Properties Window, click a graphic object to select it, then right-click the graphic object to open the contextual menu and choose Properties Window in the menu. You can also open it through the option View>Properties Window on the menu bar.

### The Smart Tags

Like in the Winforms Designer of Visual Studio, the Diagram Designer presents the notion of smart tags. A smart tag panel proposes a quick access to the most important properties and actions on the graphic object.

The following illustration shows the smart tag panel for an Ellipse object:



To open the smart tag panel, simply click on the arrow on the top of the selected graphic object.

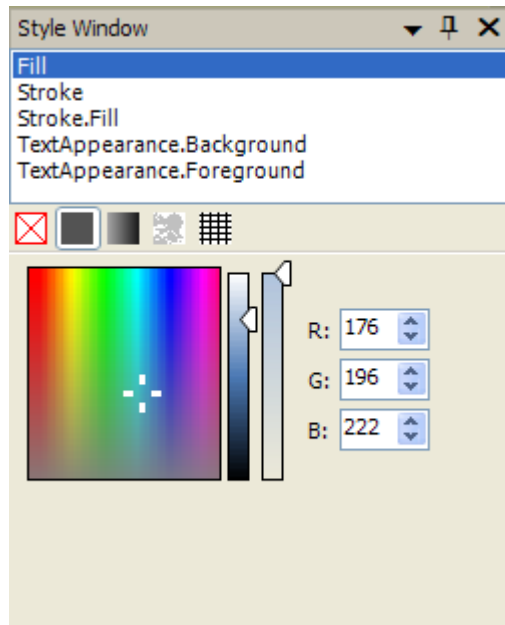
***Note:** It is possible to customize the smart tag of your own graphic objects. To learn more about customizing the design-time behavior of a graphic object see [Improving the Design-Time Behavior of Your Graphic Object](#).*

## Changing the Style with the Style Window

You can modify the appearance of graphic objects by changing the properties from the Properties Window. ILOG Diagram for .NET offers another quick access to the styling properties of a graphic object: the Style Window.

To open the Style Window choose View>Other Windows>Diagram Style Window from the menu bar.

The following illustration shows the Style Window displaying the style properties of an **Ellipse** object:

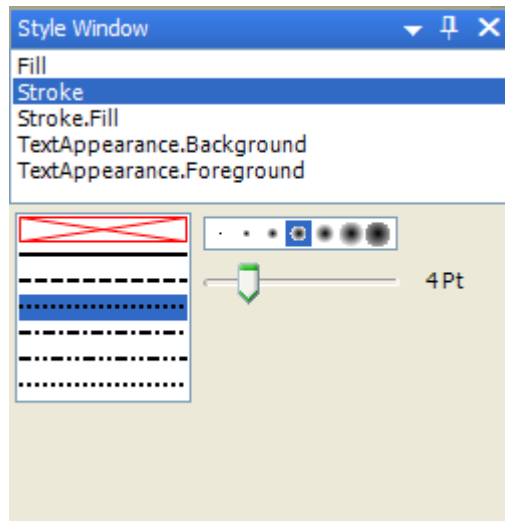


The Style Window shows the list of properties that you can modify. This list displays the properties of the selected graphic object that are of type Fill or Stroke. A **Fill** object defines

how the interior of a shape is filled and a **Stroke** object defines how the outline of the shape is drawn.

For a property of type **Fill**, the Style Window allows you to set the **Fill** to Nothing (the read cross), to a solid color (like in the picture above), a gradient, a texture or a predefined pattern.

For a property of type **Stroke**, you can specify the width of the line as well as the dash pattern. The following illustration shows the Style Window editing a **Stroke** property:





To learn more about the **Stroke** and **Fill** objects see *Filling and Stroking Graphic Objects*.

---

### Copy and Apply Style Commands


Two additional commands are available in Visual Studio to simplify the task of copying the style of a graphic object to other objects.

Use the Copy Style command (  ) to change the style of the selected objects. First select the graphic objects you want to modify, then choose the Copy Style command and click the graphic object from which you want to take the style from.

Use the Apply Style command (  ) to change the style of some objects. First select the graphic object that has the style you want to copy, then choose the Apply Style command and click all the graphic objects to which you want to apply the style.

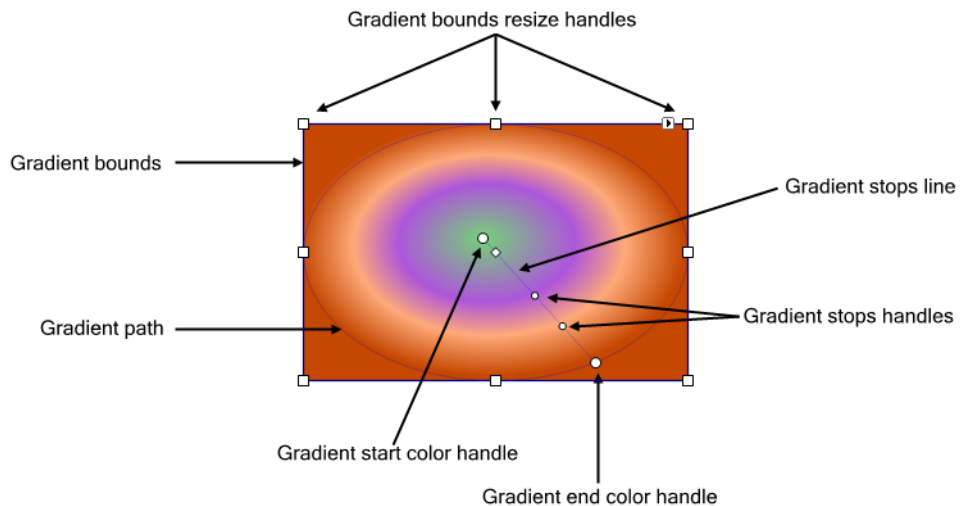
### In Place Gradient Brush Editor

The following commands are available in Visual Studio to simplify the editing of gradient brushes:

Use the Edit Gradient Fill command (  ) to edit the gradient of a graphic object fill,

Use the Edit Gradient Stroke command (  ) to edit the gradient of graphic object stroke.

To edit a gradient, a gradient editor is displayed on top of selected objects. The following picture shows a rectangle object with a circular gradient being edited:



Both the Edit Gradient Fill and Edit Gradient Stroke commands share the same kind of interaction. The table below lists the possible interactions during the editing of a gradient:

Interaction	Description
Click a graphic object	Shows the gradient editor on top of the graphic object.
Double-click a graphic object	Shows the modal fill editor dialog. This allows you to change the type of fill (pattern, gradient, solid, and so on) used by the graphic object.
Drag the gradient stops line or the gradient bounds	Moves the gradient inside the graphic object.

Interaction	Description
Click the gradient stops line with the CTRL key pressed	Adds a new gradient stop at the clicked location.
Click a gradient stop with the CTRL key pressed	Removes the clicked gradient stop.
Double-click a gradient stop	Shows the modal color editor dialog to change the color of the gradient stop.
Drag a gradient stop	Moves the gradient stop along the gradient stops line.
Drag a gradient stop with the CTRL key pressed	Duplicate the gradient stop.

## Setting Text to a Graphic Object

Every graphic object has a property named Text. This property represents a string that can be displayed on top of the graphic object. Some graphic objects, such as Text or TextOnPath, are dedicated to display text. The text displayed by the graphic objects can be formatted by using the commands available on the Diagram toolbar, as illustrated in the following figure:



The following table describes the options available on the Diagram toolbar

Align Text Bottom		Aligns the text to the bottom.
Align Text Center		Aligns the text horizontally in the center of the graphic object.
Align Text Left		Aligns the text to the left.
Align Text Middle		Aligns the text vertically in the middle of the graphic object.
Align Text Right		Aligns the text to the right.
Align Text Top		Aligns the text to the top.

Bold	<b>B</b>	Sets the text in bold.
Italic	<i>I</i>	Sets the text in italic.
Underline	<u>U</u>	Underlines the text.

To learn more about text in a diagram see *Displaying Text in a Diagram*.

---

## Grouping Graphic Objects

Graphic objects can be grouped together inside a **Group** object or inside a **GraphicSymbol** object.

To group objects you need to create a **Group** or a **GraphicSymbol** object and place the selected objects as children of this new **Group** or **GraphicSymbol**.

Use the **Group as GraphicSymbol** command to create a new **GraphicSymbol**, and the **Group as Group** command to create a new **Group**.

**Group** and **GraphicSymbol** are similar, they both contain child objects and they have no particular rendering apart their children, but are used for different purposes. You can use the **Group** object to create a logical group of graphic objects, while the **GraphicSymbol** can be used to create a new graphic symbol that you will manipulate as a whole. For example, in a **GraphicSymbol** you will not be able to select the graphic objects that compose the symbol.

To see the structure of the diagram with groups and symbols use the Document Outline view.

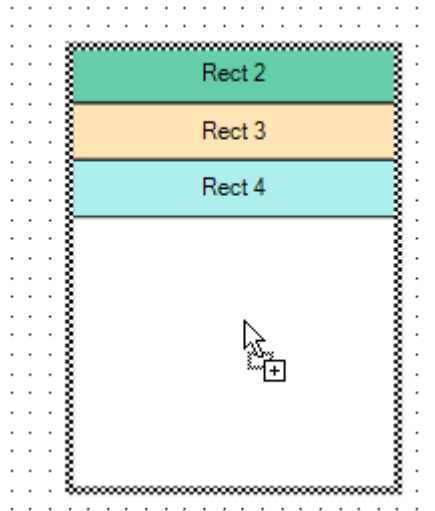
---

## Manipulating Panels and Other Containers

Containers are graphic objects that contain other objects. In the Diagram Designer several functionalities allow you to manipulate the children in a container object. You can drag several panel objects from the Panels tab of the Toolbox.

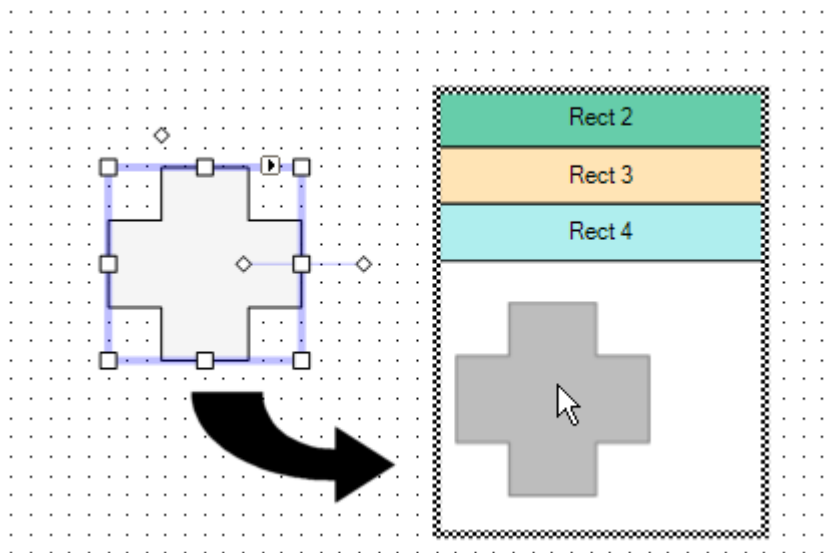
Those graphic objects are container that automatically control the size and the location of their children. For example, a **GridPanel** layouts its children in rows and columns, the **StackPanel** layouts its children vertically or horizontally.

To create a Panel object simply drag it from the Toolbox. Once a panel is created, you may drag a new graphic object to it from the Toolbox. As you drop a new graphic object in a container, the bounding area of the container is highlighted to indicate the drop target as shown in the following illustration.



You can also drag a graphic object which is already in the Design view to place it in a panel. To do so simply drag the graphic object to the destination panel. As you drag the graphic object, the destination panel will be highlighted. If you simply need to move the graphic object without changing its parent container, hold down the Shift key during the interaction.

The following illustration shows a graphic object representing a cross being dropped on a **StackPanel**:





The same operation can be used to remove a graphic object from a panel.

When a graphic object is inserted in a panel, additional properties are added to the graphic object so that you can control some alignments and attachment options.

For example, when a graphic object is inserted in a **GridPanel**, you can control the row and column in which the graphic object will be placed by changing the Rows and Columns properties in the Properties Window.

**Panel** objects are particularly useful when you create a new **UserSymbol** composed of graphic objects that must be laid out in some way. For example, a **UserSymbol** that represents a UML class would use a vertical **StackPanel** where graphic objects representing methods and properties will be added.

To learn more about the Panel objects see *Panels*.

---

## Controlling the Drawing Order of Graphic Objects

The drawing order of graphic objects that compose a diagram or a symbol is determined by the order of the graphic objects in their parent. Right-click a graphic object and choose one of the following commands available in the Order menu:

- ◆ Bring to Front, to place the object in front of all other objects in the same parent.
- ◆ Send to Back, to send the object behind other objects in the same parent.
- ◆ Send Backward, to place the object behind his first sibling.
- ◆ Send Forward, to place the object in front of his first sibling.

The commands Bring to Front and Send to Back are also available in the Layout toolbar as well as in the Format menu.

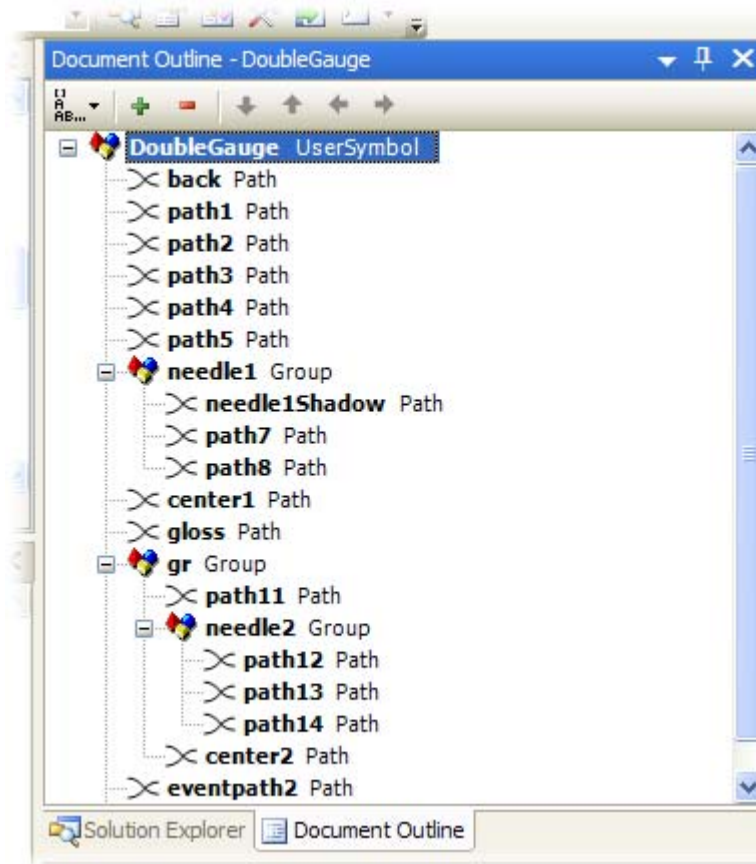
You can also have a look at the drawing order using the Document Outline view.

---

## Inspecting the Structure of a Diagram

Through the Document Outline view you can look at the structure of the diagram. To open the Document Outline view choose Document Outline from the View>Other Windows>Document Outline from the menu bar.

The following illustration shows the document outline:



From this view you can also rename the objects that compose your diagram or symbols.

---

## Showing and Hiding Objects

By default when you hide a graphic object by setting its Visibility property to **Hidden** or **Collapsed**, the graphic object stays visible in the Diagram Designer, although the object will not be visible at runtime. To hide or show the invisible objects use the option **Diagram>Show Invisible Objects** available on the menu bar.

---

## Creating Complex Path Objects

A Path object is a graphic object that can display almost any kind of shape. The geometry of this object is defined by a collection of segments. The segments can be lines, arcs, Bezier or quadratic Bezier segments. It is also possible to create compound paths (a path with multiple subpaths) to enable effects such as "donut holes".

The following illustration shows a **Path** object:



You can create a **Path** object by converting a basic shape into a **Path**.

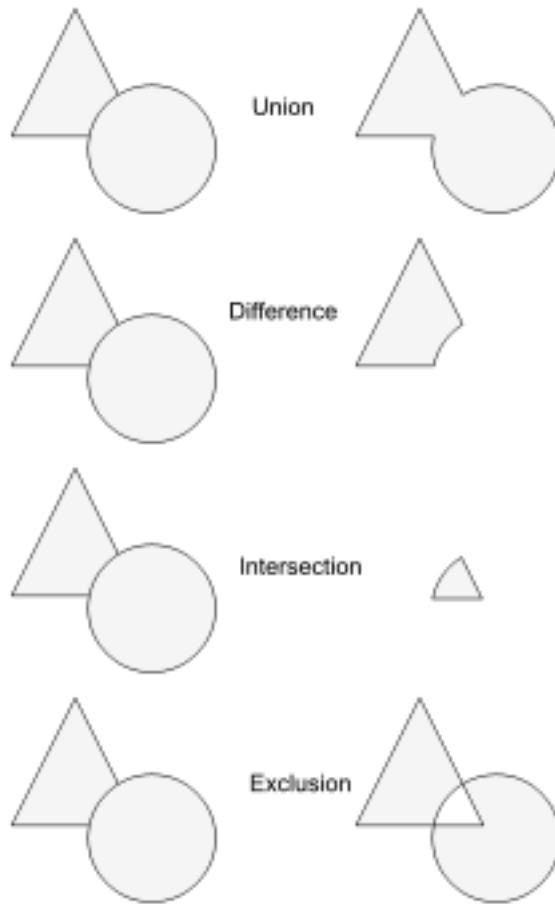
Any basic shapes (rectangle, ellipse, arc, polygons, polylines, curves, and so on) can be converted into a **Path** object. The Text object as well as the TextOnPath object can also be converted into a **Path**.

To convert an object into a **Path**, select the command Path>Object to Path in the contextual menu that opens when you right-click a graphic object.

You can also create a path by combining basic shapes. In this case you have to select several objects and right click to open the contextual menu which will display the following commands:

- ◆ Union: replaces the selected objects by a single Path that is the union of all the selected paths.
- ◆ Difference: replaces the selected objects by a single **Path** with a geometry that corresponds to the shape of the primary selected object from which all the intersection with other objects have been removed.
- ◆ Intersection: replaces the selected objects by a single **Path** with a shape that corresponds to the intersection of all the selected objects.
- ◆ Exclusion: replaces the selected objects by a single **Path** with a geometry that corresponds to the shape of all the selected objects from which all the intersection have been removed.

The following illustration shows the result of these commands:



A **Path** object can also be made of several disconnected figures. Use the following commands to manipulate such paths.

- ◆ Merge: creates a **Path** object composed of several disconnected figures representing the geometry of the selected objects.
- ◆ Break: converts a **Path** object composed of several disconnected figures into several Path objects.

For more information see *Paths*.

---

## Cut, Copy and Paste

At any time you can cut, copy and paste graphic objects in the Design view by selecting objects and using the commands Cut/Copy and Paste available in the contextual menu that is displayed when you right click a graphic object.

You can also duplicate the selected graphic objects by dragging the objects while holding down the CTRL key.

---

## Graph, Link and Anchors

Link objects are used to connect graphic objects together and are useful to build graphs. Examples of graphs are flow charts, UML diagrams or business process diagrams.

To create a link, select the desired type of link in the Toolbox under the Connectors category. Move the mouse cursor over a graphic object to connect the start of the link. The anchors of the graphic object are highlighted (they are drawn as small red circles). Move the mouse cursor over one of the anchors and press the left mouse button. Then, drag the link to another graphic object to connect the end of the link and choose the end anchor in the same way.

Once a link is connected to its start and end nodes, the link will be kept connected to the nodes when they are moved or resized.

Links can have several shapes: straight, orthogonal, oblique or free. The link shape is controlled by the ShapeType property. See *Link Objects* for a complete description of the Link class.

Links are connected to graphic objects through anchors. Anchors are represented by the class Anchor and its subclasses. By default, most graphic objects have anchors on the top, left, right and bottom sides, in addition to one floating anchor that follows the shape of the node.

See *Introducing Link and Anchor Classes* and *Creating Diagrams with Nodes and Links* for more information on links and anchors.

---

## Graph Layout

When you create complex graphs with nodes and links, it is often useful to arrange them automatically to obtain a clear display. To do this, IBM® ILOG® Diagram for .NET provides a powerful set of graph layout algorithms.

To choose a graph layout algorithm, click the background of the Designer view to select the Group that contains the diagram.

In the Property Window, edit the **GraphLayout** property to show the Graph Layout dialog box. This dialog box lets you choose the type of layout that you want to apply and allows you to customize the parameters of the layout. Once you have chosen and customized the layout, select Set Graph Layout if you want to apply the layout later, or Set and Execute Layout if you want to apply the layout immediately.

You can re-execute the graph layout algorithm at any time using the command Format>Execute Graph Layout available on the Diagram toolbar.

In addition to the **GraphLayout** property, you can set the **LinkLayout** property to choose and customize a second layout algorithm that will only route the links of the diagram so as to avoid crossings with nodes and other links, without moving the nodes. As for the graph layout, you can re-execute the link layout at any time using the Execute Link Layout command.

See *Graph Layout Algorithms* for more information on graph layout algorithms.

---

## Importing Vector Graphics in SVG or IVN Format

When creating a diagram or a user symbol in the Diagram Designer, you can import files in the SVG or IVN format (the XML format for storing diagrams in IBM® ILOG® Diagram for .NET). To do this, select Diagram>Import File from the menu bar. You can import a file in two different ways:

- ◆ by creating a single Image object that displays the content of the file. This is useful when you have to create a background for your diagram on which you will not interact,
- or
- ◆ in a mode where each graphic object becomes available as a member of your class, as if you had dragged them from the Toolbox.

To learn more about the SVG support in IBM ILOG Diagram for .NET see *Importing SVG Files*. For more information on the IVN format see *Serializing and Deserializing a Diagram in XML*.

---







## Printing a Diagram

It is possible to print the diagrams or symbols that you create with the Diagram Designer using the commands File>Print, File>Print Preview and File>Page Settings available from the menu bar.







For more information on the printing dialog box see *Using Predefined Printing Dialog Boxes*.














## Diagram Designer Commands

### Zoom Commands



Command	Toolbar icon	Description
Pan		Scrolls the diagram design view by panning.
Zoom Area		Zooms the diagram design view by dragging a rectangle to select the zoom area.
Fit to Contents		Zooms the diagram design view to make all the graphic objects visible.
Reset Zoom		Resets the zoom of the diagram design view to 100%.
Zoom In		Zooms in the diagram design view.
Zoom Out		Zooms out the diagram design view.

### Grid and Alignment Commands










Command	Toolbar icon	Description
Show Grid		Shows or hides the magnetic grid.
Align to Grid		Moves the graphic object so that the top left corner is aligned to the grid.
Align Lefts		Aligns the left border of the selected graphic objects to the left border of the primary selection.
Align Centers		Aligns the center of the selected graphic objects to the center of the primary selection.
Align Rights		Aligns the right border of the selected graphic objects to the right border of the primary selection.
Align Tops		Aligns the top of the selected graphic objects to the top of the primary selection.

Command	Toolbar icon	Description
Align Middles		Aligns the middle of the selected graphic objects to the middle of the primary selection.
Align Bottoms		Aligns the bottom of the selected graphic objects to the bottom of the primary selection.
Make Same Width		Resizes the selected graphic objects to the same width of the primary selection.
Make Same Height		Resizes the selected graphic objects to the same height of the primary selection.
Make Same Size		Resizes the selected graphic objects to the same size of the primary selection.
Size to Grid		Resizes the selected graphic objects to make the size fit the grid.
Make Horizontal Spacing Equal		Moves the selected graphic objects so that the horizontal spacing between them is the same.
Increase Horizontal Spacing		Moves the selected graphic objects to increase the horizontal spacing between them.
Decrease Horizontal Spacing		Moves the selected graphic objects to decrease the horizontal spacing between them.
Remove Horizontal Spacing		Moves the selected graphic objects to remove the horizontal spacing between them.
Make Vertical Spacing Equal		Moves the selected graphic objects so that the vertical spacing between them is the same.
Increase Vertical Spacing		Moves the selected graphic objects to increase the vertical spacing between them.
Decrease Vertical Spacing		Moves the selected graphic objects to decrease the vertical spacing between them.
Remove Vertical Spacing		Moves the selected graphic objects to remove the vertical spacing between them.
Flip Horizontal		Flips the selected graphic objects horizontally.
Flip Vertical		Flips the selected graphic objects horizontally.









Command	Toolbar icon	Description
Rotate Left		Rotates the selected graphic objects by 45 degrees to the left.
Rotate Right		Rotates the selected graphic objects by 45 degrees to the right.

## Text Commands







Command	Toolbar icon	Description
Align Text Bottom		Aligns the text in a graphic object to the bottom.
Align Text Center		Aligns the text horizontally to the center of the graphic object.
Align Text Left		Aligns the text to the left of the graphic object.
Align Text Middle		Aligns the text vertically in the middle of the graphic object.
Align Text Right		Aligns the text to the right of a graphic object.
Align Text Top		Aligns the text to the top of a graphic object.
Bold		Sets the text of the graphic object in bold.
Italic		Sets the text of the graphic object in italic.
Underline		Underlines the text of the graphic object.











## Path Commands

Command	Toolbar icon	Description
Object to Path		Converts a graphic object to a <b>Path</b> object.
Stroke to Path		Converts the outline of a graphic object to a <b>Path</b> object.

Command	Toolbar icon	Description
Union		Creates a <b>Path</b> object union of several graphic objects.
Difference		Replaces the selected objects by a single Path with a geometry that corresponds to the shape of the primary selected object from which all the intersection with other objects have been removed.
Intersection		Replaces the selected objects by a single <b>Path</b> with a shape that corresponds to the intersection of all the selected objects.
Exclusion		Replaces the selected objects by a single <b>Path</b> with a geometry that corresponds to the shape of all the selected objects from which all the intersection have been removed.
Merge		Creates a <b>Path</b> object composed of several disconnected figures representing the geometry of the selected objects.
Break		Converts a <b>Path</b> object composed of several disconnected figures into several <b>Path</b> objects.

Object Creation Commands




Command	Toolbar icon	Description
Create Arc		Creates an Arc object.
Create Closed Curve		Creates a ClosedCurve object.
Create Curve		Creates a Curve object.
Create Ellipse		Creates an Ellipse object.
Create Filled Closed Curve		Creates a filled <b>ClosedCurve</b> object.
Create Filled Ellipse		Creates a filled <b>Ellipse</b> object.

Command	Toolbar icon	Description
Create Filled Polygon		Creates a filled Polygon object.
Create Filled Rectangle		Creates a filled Rect object.
Create FreeHand Shape		Creates a <b>Curve</b> object.
Create Image		Creates an Image object.
Create Link		Creates a Link object.
Create Pie		Creates a Pie object.
Create Polygon		Creates a <b>Polygon</b> object.
Create Polyline		Creates a Polyline object.
Create Rectangle		Creates a <b>Rect</b> object.
Create Text		Creates a Text object.

## Printing Commands

Command	Toolbar icon	Description
Print		Prints the diagram.
Print Preview		Opens the print preview dialog box.
Page Setup		Opens the page setup dialog box.




Editing Commands

Command	Toolbar icon	Description
Cut		Cuts the selected graphic objects to the clipboard.
Copy		Copies the selected graphic objects to the clipboard.
Paste		Pastes graphic objects from the clipboard.
Delete		Deletes the selected graphic objects.





Order Commands

Command	Toolbar icon	Description
Bring To Front		Brings the selected graphic object to the front.
Send To Back		Sends the selected graphic object to the back.
Send Forward		Sends the selected graphic object forward.
Send Backward		Sends the selected graphic object backward.

Group Commands







Command	Toolbar icon	Description
Group As Group		Groups the selected graphic objects in a Group object.
Group As GraphicSymbol		Groups the selected graphic objects in a GraphicSymbol object.
Ungroup		Ungroups the selected graphic objects.

## Nudge Commands

Command	Icon	Keyboard binding	Description
Key Move Left		Left	Moves the selected graphic objects to the left of the grid size.
Key Move Right		Right	Moves the selected graphic objects to the right of the grid size.
Key Move Up		Up	Moves the selected graphic objects on top of the grid size.
Key Move Down		Down	Moves the selected graphic objects down to the grid size.
Key Nudge Left		CTRL+Left	Moves the selected graphic objects to the left by 1 pixel.
Key Nudge Right		CTRL+Right	Moves the selected graphic objects to the right by 1 pixel.
Key Nudge Top		CTRL+Up	Moves the selected graphic objects up by 1 pixel.
Key Nudge Bottom		CTRL+Down	Moves the selected graphic objects down by 1 pixel.
Key Nudge Width Increase		CTRL+Shift+Right	Increases the width of the selected graphic objects by 1 pixel.
Key Nudge Width Decrease		CTRL+Shift+Left	Decreases the width of the selected graphic objects by 1 pixel.
Key Nudge Height Increase		CTRL+Shift+Down	Increases the height of the selected graphic objects by 1 pixel.
Key Nudge Height Decrease		CTRL+Shift+Up	Decreases the height of the selected graphic objects by 1 pixel.

Command	Icon	Keyboard binding	Description
Key Size Width Increase		Shift+Right	Increases the width of the selected graphic objects by the grid size.
Key Size Width Decrease		Shift+Left	Decreases the width of the selected graphic objects by the grid size.
Key Size Height Increase		Shift+Down	Increases the height of the selected graphic objects by the grid size.
Key Size Height Decrease		Shift+Up	Decreases the height of the selected graphic objects by the grid size.

Miscellaneous

Command	Toolbar icon	Description
Apply Style		Applies the style of the graphic object that you clicked on to the selected graphic objects.
Copy Style		Copies the style of the selected graphic object to all the graphic objects you click on.
Edit Anchors		Sets the anchors edition mode.
Export As...		Saves the diagram to the IVN format.
Import File...		Imports an SVG or IVN file in the diagram.
Select Shape		Sets the selection mode
Show Invisible Objects		Shows or hides all the objects marked as invisible.

# XML Serialization

In IBM® ILOG® Diagram for .NET the persistence of a diagram is implemented as an XML serialization. For this purpose, IBM ILOG Diagram for .NET provides a generic, fully customizable XML serialization framework to support shared references resolution, custom types and several levels of customization during the serialization process.

In IBM ILOG Diagram for .NET the XML documents are standard XML files that have the `.ivn` filename extension. GZIP compressed files are also natively supported and the filename extension is `.ivnz` (for GZIP compressed file).

## In This Section

### *Serializing and Deserializing a Diagram in XML*

Describes how to serialize and deserialize a diagram to the IBM ILOG Diagram for .NET XML format.

### *Understanding the XML Serialization Mechanism*

Goes through the low-level serialization API and explains the XML serialization mechanism

### *Customizing the XML Serialization*

Introduces the several levels of customization during the XML serialization process.

---

## Serializing and Deserializing a Diagram in XML

This section describes how to serialize and deserialize a diagram to the IBM® ILOG® Diagram for .NET XML format.

In IBM ILOG Diagram for .NET the API supports the XML serialization through two levels:

- ◆ high-level API: allows you to serialize and deserialize a diagram quickly and easily by means of the `SaveAsXml(Stream stream)` and `FromXmlStream(Stream stream)` methods of the `GraphicObject`. In most cases these methods will be used to serialize or deserialize a diagram.
- ◆ low-level API: see *Understanding the XML Serialization Mechanism*.

---

### Serializing a Diagram

The following example illustrates the required steps to serialize a diagram.

```
Group group = new Group();
group.Objects.Add(new Rect(0, 0, 75, 110));
group.Objects.Add(new Ellipse(10, 30, 70, 50));
FileStream output = null;
try {
    output = new FileStream(@"c:\tmp\contents.ivn", FileMode.Create);
    group.SaveAsXml(output);
} catch (Exception e) {
    // TODO: handle exception
} finally {
    if (output != null) output.Close();
}
Dim group As Group = New Group
group.Objects.Add(New Rect(0, 0, 75, 110))
group.Objects.Add(New Ellipse(10, 30, 70, 50))
Dim output As FileStream = Nothing
Try
    output = New FileStream("c:\tmp\contents.ivn", FileMode.Create)
    group.SaveAsXml(output)
Catch e As Exception
Finally
    If Not (output Is Nothing) Then
        output.Close
    End If
End Try
```

In this example, the graphical objects are created programmatically and added to a `Group` instance. Then, a file stream is opened on the destination file and the group is serialized in this stream by means of the **SaveAsXml** method.



---

## Deserializing a Diagram

The following example illustrates the required steps to deserialize a diagram.

```
FileStream input = null;
GraphicObject container = null;
try {
    input = new FileStream(@"c:\tmp\contents.ivn",
        FileMode.Open, FileAccess.Read);
    container = GraphicObject.FromXmlStream(input);
} catch (Exception e) {
    // TODO: handle exception
} finally {
    if (input != null) input.Close();
}
Dim input As FileStream = Nothing
Dim container As GraphicObject = Nothing
Try
    input = New FileStream("c:\tmp\contents.ivn", FileMode.Open,
        FileAccess.Read)
    container = GraphicObject.FromXmlStream(input)
Catch e As Exception
Finally
    If Not (input Is Nothing) Then
        input.Close
    End If
End Try
```

In this example, an input stream is opened on a file and a container is deserialized from the stream by means of the **FromXmlStream** static method.

In the following sections, you will go through the low-level XML serialization API and see how the serialization process can be controlled and customized.

---

## Understanding the XML Serialization Mechanism

The serialization of a graphical objects hierarchy is performed by means of the XML serialization API available in IBM® ILOG® Diagram for .NET.

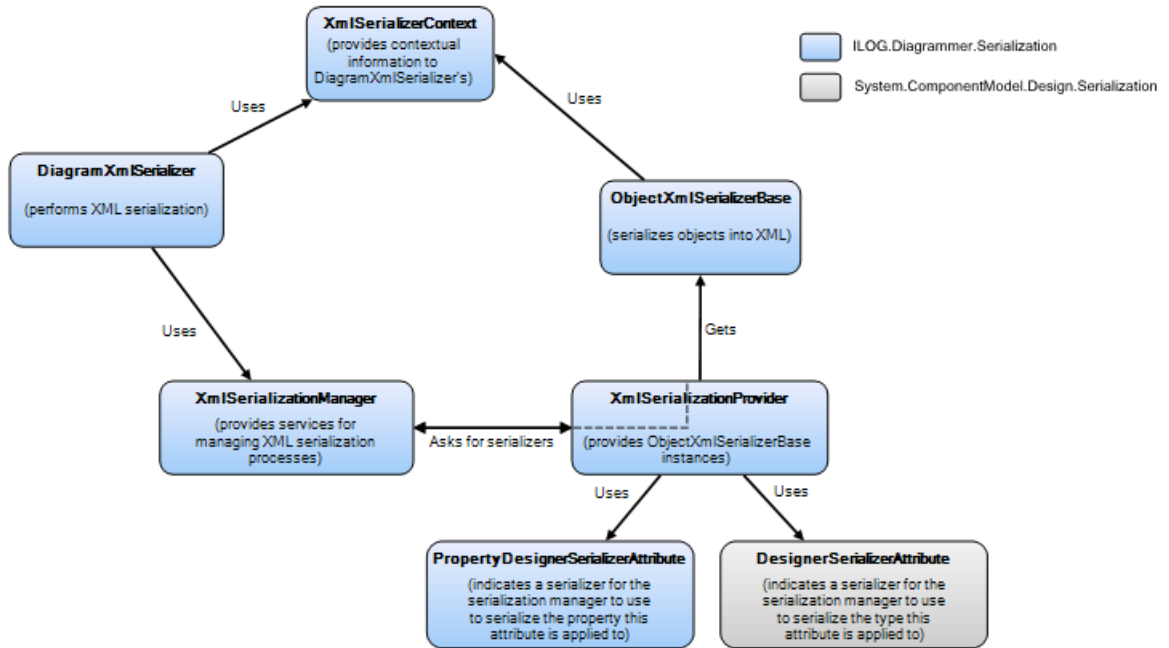
This section goes through the low-level serialization API and explains the XML serialization mechanism.

The XML serialization framework in IBM ILOG Diagram for .NET has been designed as a generic XML serialization framework. As such, it supports several levels of customization that allow you to control the serialization process. For this purpose, the API provides the `DiagramXmlSerializer` class which is the low-level entry point of the serialization process. This class supports both the serialization (output) and the deserialization (input) and allows you to customize and control the serialization process.

The XML serialization framework is composed of the following types:

- ◆ **DiagramXmlSerializer**: represents the entry point of the serialization process. It manages and performs the serialization of a diagram. It handles shared references, custom types and assemblies, and extender-provided properties resolution.
- ◆ **XmlSerializationManager**: provides services for managing the XML serialization process: type resolution, CLR versus XML namespace mapping, XML element creation, naming, context stack, and so on.
- ◆ **ObjectXmlSerializerBase**: base class for classes that effectively performs XML serialization of a type. Several concrete implementations are provided for arrays, lists, value types and objects.
- ◆ **XmlSerializerContext**: provides contextual information to **ObjectXmlSerializerBase** instances. These instances are handled by means of a stack by the **XmlSerializationManager** and they are typically used to pass contextual information between serializers.
- ◆ **XmlSerializationProvider**: acts as an **ObjectXmlSerializerBase** provider to the **XmlSerializationManager**. It manages the search of an **ObjectXmlSerializerBase** for a specified type depending on the current context.
- ◆ **PropertyDesignerSerializerAttribute**: indicates which serializer the serialization manager should use in order to serialize the property to which this attribute is applied.

The following illustration shows the relationship between the classes and the interfaces that are involved in the XML serialization.



## Introducing the DiagramXmlSerializer Class

The **DiagramXmlSerializer** class is the entry point of the XML serialization framework. It manages the whole process of serialization, handling shared references, **IExtenderProvider** and type resolution.

## Serializing a Diagram Using the DiagramXmlSerializer

The following example shows the required steps to serialize a diagram using the **DiagramXmlSerializer** API.

```

Group group = new Group();
group.Objects.Add(new Rect(0, 0, 75, 110));
group.Objects.Add(new Ellipse(10, 30, 70, 50));
DiagramXmlSerializer serializer = new DiagramXmlSerializer();
FileStream output = null;
try {
    output = new FileStream(@"c:\tmp\contents.ivn", FileMode.Create);
    serializer.Serialize(group, output);
} catch (Exception e) {
    // TODO: handle exception
} finally {
    if (output != null) output.Close();
}
Dim group As Group = New Group
group.Objects.Add(New Rect(0, 0, 75, 110))
    
```

```

group.Objects.Add(New Ellipse(10, 30, 70, 50))
Dim serializer As DiagramXmlSerializer = New DiagramXmlSerializer
Dim output As FileStream = Nothing
Try
    output = New FileStream("c:\tmp\contents.ivn", FileMode.Create)
    serializer.Serialize(group, output)
Catch e As Exception
Finally
    If Not (output Is Nothing) Then
        output.Close
    End If
End Try

```

In this example, the graphical objects are created programmatically and added to a Group instance. Then, a file stream is opened on the destination file and the group is serialized in this stream by a **DiagramXmlSerializer** instance by means of the Serialize method.

### Deserializing a Diagram Using the DiagramXmlSerializer

The following example shows the required steps to deserialize a diagram using the **DiagramXmlSerializer** API.

```

DiagramXmlSerializer serializer = new DiagramXmlSerializer();
FileStream input = null;
GraphicObject container = null;
try {
    input = new FileStream(@"c:\tmp\contents.ivn",
        FileMode.Open, FileAccess.Read);
    container = serializer.Deserialize(input) as GraphicObject;
} catch (Exception e) {
    // TODO: handle exception
} finally {
    if (input != null) input.Close();
}
Dim serializer As DiagramXmlSerializer = New DiagramXmlSerializer
Dim input As FileStream = Nothing
Dim container As GraphicObject = Nothing
Try
    input = New FileStream("c:\tmp\contents.ivn", _
        FileMode.Open, FileAccess.Read)
    container = CType(serializer.Deserialize(input), GraphicObject)
Catch e As Exception
Finally
    If Not (input Is Nothing) Then
        input.Close()
    End If
End Try

```

In this example, an input stream is opened on a file, and the input stream contents are deserialized by means of the Deserialize method.

---

## Structure of an IBM ILOG Diagram for .NET XML Document

The IBM ILOG Diagram for .NET XML format follows a declarative programming model. In this model, an object is serialized as an XML element that represents the instance class, which contains child nodes for every properties that need to be serialized. The child nodes contain themselves other child nodes that represent their value. This gives origin to the following sequence:

```
Class
  Property
    Class...
```

For example, the serialization of an `Rect` instance whose `Fill` property is a `LinearGradientFill` instance gives the following result:

```
1<Rect>
2  <Fill>
3   <LinearGradientFill>...
4  </LinearGradientFill>
5  </Fill>
6 </Rect>
```

On line 1, the element `<Rect>` represents the class declaration.

On line 2, the element `<Fill>` represents the `Fill` property.

On line 3, the element `<LinearGradientFill>` represents the class declaration of the value of the `Fill` property (that is, the value of the `Fill` property is an instance of the `LinearGradientFill` class).

However, the standard Class/Property/Class pattern presents the following exceptions: property as XML attribute and arrays/list.

When a property is being serialized and there is an `TypeConverter` associated to the property value type supporting string conversion, the property is serialized as an XML attribute of the class declaration element. For example, the `EndColor` and `StartColor` colors properties of a `LinearGradientFill` are serialized as:

```
<LinearGradientFill EndColor="242, 242, 249" StartColor="153, 151, 181" />
```

Arrays and `IList` are also exceptions to the standard pattern. When contents of a collection are serialized, the `Item` property (or the equivalent) is skipped and therefore the structure is Class/Class. For example, here is the serialization of an `ArrayList` of strings:

```
<ArrayList>
  <String>hello</ String>
```

```
<String>world</ String>  
<String>!</String>  
</ArrayList>
```

Because class elements are named according to the short name of the corresponding type, the XML serialization framework needs a way to properly identify a type.

To avoid naming conflict, a mapping between the CLR (Common Language Runtime) namespaces and the XML namespaces is built: each CLR namespace has a corresponding XML namespace, and every type from a given namespace will have an XML element created in the corresponding XML namespace.

The following example illustrates how the CLR namespace `System` is mapped to the XML namespace `"xmlns:sys"` and how any XML element matching a type for the `System` namespace is created in the corresponding namespace:

```
<?xml version="1.0" encoding="utf-8"?>  
<idn:grapher ... xmlns:sys="System, mscorlib, Version=2.0.0.0, Culture=neutral,  
  PublicKeyToken=b77a5c561934e089" xmlns:idn="www.ibm.com/diagramnet">  
  <sys:String>Hello World</sys:String>  
</idn:grapher>
```

The **`XmlSerializationManager`** supports a default assembly which is mapped to the default XML namespace. When a default assembly is specified, all types from this assembly (except possible naming collision) are mapped to the default XML namespace and therefore the XML elements are created in the default XML namespace.

---

### What is serialized

In order to be serialized, a class must fulfill the following conditions:

- ◆ it must be public,
- ◆ it must have a default constructor (see exceptions below),
- ◆ the assembly where the type is defined must be accessible.

***Note:** The default constructor limitation might be raised by providing a custom serializer for the specified type. See [Customizing the XML Serialization](#) for more information about customizing serialization.*

By default, the following items can be serialized using the **`DiagramXmlSerializer`**:

- ◆ Public read/write properties of public classes
- ◆ Value types, primitive types,
- ◆ Arrays, `IList`,

- ◆ public reference types via generic serialization.

Since it is an extensible framework, types that are not supported by default can be serialized by means of a custom **ObjectXmlSerializerBase**.

In order to reduce the amount of data to serialize, the serializer may skip some properties which are useless to represent the current state of an instance. In particular, if one the following conditions occurs, the property will not be serialized:

- ◆ the property is not public,
- ◆ the property is not read/write,
- ◆ the property value has the default value,
- ◆ and/or the ShouldSerialize method for this property returns false,
- ◆ the DesignerSerializationVisibility attribute is set to Hidden.

## Customizing the XML Serialization

The XML serialization framework in IBM® ILOG® Diagram for .NET supports several levels of customization to control how a diagram is serialized during the XML serialization process.

### Introducing the ObjectXmlSerializerBase Abstract Class

When a particular class cannot be serialized because it does not fulfill the requirements listed in *What is serialized*, it is necessary to implement a custom serializer that will handle the class serialization.

You can write a custom serializer by extending the ObjectXmlSerializerBase abstract class, or by extending one of the concrete implementations provided in the IBM ILOG Diagram for .NET assembly.

The **ObjectXmlSerializerBase** abstract class defines the following methods:

```
public abstract void Serialize(XmlSerializationManager manager,
                              object instance,
                              XmlDocument document,
                              XmlElement parent);

public abstract object Deserialize(XmlSerializationManager manager,
                                 XmlDocument document,
                                 object instance,
                                 XmlElement element);
```

The Serialize method is called during the serialization process to dump the current state of the specified instance into its XML representation.

The parameters are:

- ◆ **manager**: the `XmlSerializationManager` instance that manages the serialization process. It provides services that might be needed to perform the XML serialization (like type resolution, CLR namespace versus XML namespace mapping resolution, shared reference resolution, and so on).
- ◆ **instance**: the instance to serialize in XML.
- ◆ **document**: the XML document used for the XML serialization. It might be used to create XML attributes or elements depending on the XML representation.
- ◆ **parent**: The parent XML element in the DOM (Document Object Model). Depending on the current context, this element represents either the class declaration or the property. A serializer will typically append a new child element to it for a class declaration serialization or append a new attribute for a to-string representation.

The `Deserialize` method is called during the serialization process to create an instance from an XML representation.

The parameters are:

- ◆ **manager**: the `XmlSerializationManager` instance that manages the deserialization process. It provides services that might be needed to perform the XML deserialization (like type resolution, CLR namespace versus XML namespace mapping resolution, shared reference resolution, and so on).
- ◆ **instance**: the instance to deserialize. This parameter is not **null** when deserializing an instance whose properties have been serialized. For example, this is the case of properties with a `SerializationVisibility.Hidden` attribute.
- ◆ **document**: the XML document containing the XML representation of the instance to deserialize.
- ◆ **element**: The root XML element of the instance XML representation. Depending on the current context, this element represents either the class declaration or the property.

---

### Writing a Custom Serializer

This section provides step-by-step examples that illustrate how to write a custom serializer.

The first example, *Extending the `ObjectXmlSerializerBase` Abstract Class*, shows how to write a serializer for instances that support string conversion through their `TypeConverter` by extending directly the `ObjectXmlSerializerBase` abstract class.

*Note: This is a simpler version of the string XML serializer provided in the IBM ILOG Diagram for .NET assembly.*



The second example, *Extending the ObjectXmlSerializer Class*, shows how to extend the `ObjectXmlSerializer` class in order to bypass the default constructor limitation.

### Extending the `ObjectXmlSerializerBase` Abstract Class

This example illustrates how to extend the `ObjectXmlSerializerBase` abstract class.

The new serializer has the following requirements:

- ◆ applies for types that support string conversion (both ways) through their associated **TypeConverter**.
- ◆ must write the "value as string" as an XML attribute when serializing a property, or as an XML element. In other words, assuming a class called `Shape` defines a property `Fill` of type `Color`, and an instance of this class is serialized, the expected result is:

```
<Shape Fill="Red"/>
```

On the other hand, if a `Color` instance is serialized but not in a context of a property, the expected result is:

```
<Color>Red</Color>
```

Now that the requirements have been defined, proceed as follows:

#### 1. Import the Serialization namespace.

```
using ILOG.Diagrammer.Serialization;
Imports ILOG.Diagrammer.Serialization
```

#### 2. Declare your new class and extend the `ObjectXmlSerializerBase` abstract class.

```
class StringXmlSerializer : ObjectXmlSerializerBase {
    public override void Serialize(XmlSerializationManager manager,
                                   object instance,
                                   XmlDocument document,
                                   XmlElement parent) {
    }

    public override object Deserialize(XmlSerializationManager manager,
                                       object instance,
                                       XmlDocument document,
                                       XmlElement objectElement) {
    }
}
Class StringXmlSerializer Inherits ObjectXmlSerializerBase

    Public Overrides Sub Serialize(ByVal manager As _
                                   XmlSerializationManager, _
                                   ByVal instance As Object, _
                                   ByVal document As XmlDocument, _
                                   ByVal parent As XmlElement)

    End Sub

    Public Overrides Function Deserialize(ByVal manager As _
                                       XmlSerializationManager, _
```

```

        ByVal instance As Object, _
        ByVal document As _
        XmlDocument, _
        ByVal objectElement As _
        XmlElement) As Object

    End Function
End Class

```

### 3. Implement the Serialize method as follows:

- ◆ Get the current `XmlSerializerContext` from the manager context stack. The **`XmlSerializerContext`** provides information about the current context of the serialization. It will be used in the next steps.

```

public override void Serialize(XmlSerializationManager manager,
                               object instance,
                               XmlDocument document,
                               XmlElement parent) {

    string val = null;
    IDesignerSerializationManager mgr =
        (IDesignerSerializationManager)manager;
    XmlSerializerContext context =
        mgr.Context[typeof(XmlSerializerContext)] as XmlSerializerContext;
    Public Overrides Sub Serialize(ByVal manager As XmlSerializationManager, _
                                   ByVal instance As Object, _
                                   ByVal document As XmlDocument, _
                                   ByVal parent As XmlElement)

        Dim val As String = Nothing
        Dim mgr As IDesignerSerializationManager = _
            CType(manager, IDesignerSerializationManager)
        Dim context As XmlSerializerContext = _
            CType(mgr.Context(GetType(XmlSerializerContext)), XmlSerializerContext)

```

- ◆ Check whether the value it is passed is **null**. If this is the case, then it calls the `SerializeNullInstance` method that handles the serialization of **null** value.

```

        if (instance == null) {
            XmlUtilities.SerializeNullInstance(manager,
                                               document,
                                               context,
                                               parent);

            return;
        }
        If (instance = Nothing) Then
            XmlUtilities.SerializeNullInstance(manager, _
                                               document, _
                                               context, _
                                               parent)

            Return
        End If

```

- ◆ Check whether there is an **TypeConverter** associated with the instance type and whether it supports conversion to string. If such a converter exists, the value is converted to a string.

```

Type objectType = instance.GetType();
TypeConverter conv = TypeDescriptor.GetConverter(objectType);
if (conv == null || !conv.CanConvertTo(context, typeof(String))) {
    throw new ArgumentException("No converter found.");
}

val = (string)conv.ConvertTo(context,
                            CultureInfo.InvariantCulture,
                            instance,
                            typeof(string));

if (val == null) {
    throw new ArgumentException("null conversion");
}
Dim objectType As Type = instance.GetType
Dim conv As TypeConverter = TypeDescriptor.GetConverter(objectType)
If ((conv = Nothing) _
    OrElse Not conv.CanConvertTo(context, GetType(String))) Then
    Throw New ArgumentException("No converter found.")
End If
val = CType(conv.ConvertTo(context, CultureInfo.InvariantCulture, _
                            instance, GetType(System.String)), String)

If (val = Nothing) Then
    Throw New ArgumentException("null conversion")
End If

```

The value is now converted into a string.

To fulfil the second requirement, the serialization context is checked to know whether a property is being serialized. If this is the case, the value is serialized as an attribute of the parent element (the attribute name is the property name and the attribute value is the string value). Otherwise, the value is serialized in the form of a class declaration element and the value is represented by a child text node.

**Note:** To create a class declaration element, the *XmlSerializationManager* class provides the *CreateClassElement* method that returns an *XmlElement* instance properly initialized for the specified type. This method handles internally the mapping between the CLR namespace of the type and the corresponding XML namespace.

```

string name;
PropertyDescriptor pdesc = context != null ?
    context.PropertyDescriptor :
    null;

if (pdesc != null) {
    name = context.PropertyDescriptor.Name;
    XmlAttribute attribute = document.CreateAttribute(name);
    attribute.Value = val;
    parent.Attributes.Append(attribute);
}

```

```

} else { // ... as a TextNode
    XmlElement elt = manager.CreateClassElement(document,
                                                objectType);

    parent.AppendChild(elt);
    XmlText valElt = document.CreateTextNode(val);
    elt.AppendChild(valElt);
}
Dim name As String
Dim pdesc As PropertyDescriptor
If (Not (context) Is Nothing) Then
    pdesc = context.PropertyDescriptor
Else
    pdesc = Nothing;
End If
If (Not (pdesc) Is Nothing) Then
    name = context.PropertyDescriptor.Name
    Dim attribute As XmlAttribute = document.CreateAttribute(name)
    attribute.Value = val
    parent.Attributes.Append(attribute)
Else
    ' ... as a TextNode
    Dim elt As XmlElement = manager.CreateClassElement(document, _
                                                        objectType)

    parent.AppendChild(elt)
    Dim valElt As XmlText = document.CreateTextNode(val)
    elt.AppendChild(valElt)
End If

```

#### 4. Implement the Deserialize method.

This method implements the reverse logic of the **Serialize** implementation. To implement this method proceed as follows:

- ◆ Determine in which context the serialization happened: a property or a class declaration. To do so, the **XmlSerializerContext** is retrieved and is checked whether there is a **PropertyDescriptor**.

```

public override object Deserialize(XmlSerializationManager manager,
                                object instance,
                                XmlDocument document,
                                XmlElement objectElement) {
    XmlSerializerContext context =
        ((IDesignerSerializationManager)manager).
            Context[typeof(XmlSerializerContext)] as XmlSerializerContext;
    object val = null;
    string pvalue = null;
    string pname = null;
    PropertyDescriptor pdesc = context != null ?
        context.PropertyDescriptor : null;
    Public Overrides Function Deserialize(ByVal manager As
        XmlSerializationManager, _
                                        ByVal instance As Object, _
                                        ByVal document As XmlDocument, _
                                        ByVal objectElement As XmlElement) As
        Object
        Dim context As XmlSerializerContext =
            CType(CType(manager, IDesignerSerializationManager). _
                Context(GetType(XmlSerializerContext)), XmlSerializerContext)

```

```

Dim val As Object = Nothing
Dim pvalue As String = Nothing
Dim pname As String = Nothing
Dim pdesc As PropertyDescriptor
If (Not (context) Is Nothing) Then
    pdesc = context.PropertyDescriptor
Else
    pdesc = Nothing
End If

```

- ◆ Get the value as **string** in order to convert it. If it is a property, it means that it has been serialized as an attribute of the **objectElement** parameter. In this case, the XML attribute corresponding to the context **PropertyDescriptor** is retrieved, and if such an attribute exists, then the property value is the attribute value. Otherwise, it checks whether it is a **null** value.

If the current serialization context does not specify a **PropertyDescriptor**, the string value is serialized as a class declaration element, in a form like `<Color>Red</Color>`.

```

if (pdesc != null) {
    pname = context.PropertyDescriptor.Name;
    if (objectElement.HasAttribute(pname))
        pvalue = objectElement.GetAttribute(pname);
    if (pvalue == null) {
        XmlNode child = objectElement.FirstChild;
        while (child != null && child.NodeType != XmlNodeType.Element)
            child = child.NextSibling;
        if (child != null &&
            XmlSerializationManager.IsNullElement((XmlElement)child))
            return null;
    }
} else {
    // In this case, objectElement == the class element
    if (XmlSerializationManager.IsNullElement(objectElement))
        return null;
    // We look for the value in a text element
    // <objectElement>text_value</objectElement>
    XmlNode child = objectElement.FirstChild;
    while (child != null && child.NodeType != XmlNodeType.Text)
        child = child.NextSibling;
    if (child != null) { // the value element has been found
        pvalue = child.Value;
    }
}
}
If (Not (pdesc) Is Nothing) Then
    pname = context.PropertyDescriptor.Name
    If objectElement.HasAttribute(pname) Then
        pvalue = objectElement.GetAttribute(pname)
    End If
    If (pvalue = Nothing) Then
        Dim child As XmlNode = objectElement.FirstChild
        While ((Not (child) Is Nothing) _
            AndAlso (child.NodeType <> XmlNodeType.Element))
            child = child.NextSibling
        End While
        If ((Not (child) Is Nothing) AndAlso _
            XmlSerializationManager.IsNullElement(CType(child,XmlElement))) Then

```

```

        Return Nothing
    End If
End If
Else
    ' In this case, objectElement == the class element
    If XmlSerializationManager.IsNullElement(objectElement) Then
        Return Nothing
    End If
    ' We look for the value in a text element
    ' <objectElement>text_value</objectElement>
    Dim child As XmlNode = objectElement.FirstChild
    While ((Not (child) Is Nothing) _
        AndAlso (child.NodeType <> XmlNodeType.Text))
        child = child.NextSibling
    End While
    If (Not (child) Is Nothing) Then
        ' the value element has been found
        pvalue = child.Value
    End If
End If

```

5. Finally, the string value is converted by means of the **TypeConverter** associated with the type of the deserialized object. Determining the type of the object being deserialized depends on the serialization context. If a property is being deserialized, then the type of the value is the property type given by the **PropertyDescriptor**. Otherwise (it is not in a property context), the type is given by the **XmlElement** itself and the type resolution is performed by the **XmlSerializationManager**.

The value is eventually converted into the expected type and the method returns.

```

if (pvalue != null ) {
    Type ptype;
    if (pdesc != null)
        ptype = context.PropertyDescriptor.PropertyType;
    else
        ptype = manager.ResolveType(objectElement);
    if (ptype == null) {
        string msg = context != null &&
            context.PropertyDescriptor != null ?
            context.PropertyDescriptor.PropertyType.ToString() :
            objectElement.LocalName;
        throw new ArgumentException("Cannot resolve type: " + msg);
    }
    TypeConverter conv = TypeDescriptor.GetConverter(ptype);
    if (conv == null || !conv.CanConvertFrom(context, typeof(string)))
        throw new InvalidOperationException("No string converter for type: " +
ptype);
    val = conv.ConvertFrom(context, CultureInfo.InvariantCulture,
        pvalue);
}
if (val == null)
    throw new InvalidOperationException("Conversion failed");
return val;
}
If (Not (pvalue) Is Nothing) Then
    Dim ptype As Type
    If (Not (pdesc) Is Nothing) Then

```

```

        ptype = context.PropertyDescriptor.PropertyType
    Else
        ptype = manager.ResolveType(objectElement)
    End If
    If (ptype = Nothing) Then
        Dim msg As String
        If ((Not (context) Is Nothing) AndAlso _
            (Not (context.PropertyDescriptor) Is Nothing)) Then
            msg = context.PropertyDescriptor.PropertyType.ToString()
        Else
            msg = objectElement.LocalName
        End If
        Throw New ArgumentException(("Cannot resolve type: " + msg))
    End If
    Dim conv As TypeConverter = TypeDescriptor.GetConverter(ptype)
    If ((conv = Nothing) OrElse _
        Not conv.CanConvertFrom(context, GetType(System.String))) Then
        Throw New InvalidOperationException(("No string converter for type: " +
        ptype))
    End If
    val = conv.ConvertFrom(context,
        CultureInfo.InvariantCulture, pvalue)
End If
If (val = Nothing) Then
    Throw New InvalidOperationException("Conversion failed.")
End If
Return val

```

In order to be recognized by the `XmlSerializationProvider` as a valid `ObjectXmlSerializerBase`, the custom class must be associated with a given type. For more information see *Associating an ObjectXmlSerializerBase Implementation with a Given Custom Type*.

### Extending the ObjectXmlSerializer Class

The `ObjectXmlSerializer` class is a concrete implementation of the `ObjectXmlSerializerBase` that manages the serialization of generic reference type, provided the reference type has a default constructor. This example shows how to extend the `ObjectXmlSerializerBase` to bypass this limitation and be able to serialize **Cursor** instances.

This sample is a simplified version of the `Cursor` serializer provided in the IBM ILOG Diagram for .NET assembly.

Looking at the interface of the `ObjectXmlSerializer` class, the following methods must be overridden:

- ◆ `ShouldSerialize`  
Specifies whether the serializer can write the instance.
- ◆ `SerializeDeclaration`  
Performs the XML serialization of the class declaration.

## ◆ CreateInstance

Creates an instance from an XML class declaration.

Since the **CursorConverter**, which is the **TypeConverter** implementation associated with the **Cursor** class, supports the conversion from byte array to **Cursor**, the easiest solution is to serialize the byte array that defines the bitmap as a base64 encoded string. However, the serializer should also handle the case of the predefined cursors declared as **Cursors** static fields. If the cursor being serialized is a predefined cursor, it is expected to get the reference to the corresponding static field once deserialized, and not a new instance with the same bitmap.

To fulfill this requirement, the name of the static field is stored in an XML attribute.

## 1. Import the Serialization namespace.

```
using ILOG.Diagrammer.Serialization;
Imports ILOG.Diagrammer.Serialization
```

2. Extend the **ObjectXmlSerializer** class.

```
public class CursorSerializer : ObjectXmlSerializer {
}
Public Class CursorSerializer
    Inherits ObjectXmlSerializer
End Class
```

Because the default implementation of the **ShouldSerialize** method ensures there is a default constructor, the method must be overridden to remove this condition. Therefore, it is limited to the check of the serialization scope.

```
protected override bool ShouldSerialize(object obj,
                                         XmlSerializationManager manager,
                                         XmlDocument document,
                                         XmlSerializerContext context) {
    return (obj != null) && (obj.GetType() == typeof(Cursor));
}
Protected Overloads Overrides Function ShouldSerialize(ByVal obj As Object, _
                                                       ByVal manager As
XmlSerializationManager, _
                                                       ByVal document As
XmlDocument, _
                                                       ByVal context As
XmlSerializerContext) _
    As Boolean
    Return ((Not (obj) Is Nothing) AndAlso (Object.ReferenceEquals(obj.GetType, _
        GetType(Cursor))))
End Function
```

## 3. Implement the class declaration serialization.

This is the purpose of the `SerializeDeclaration` method. As explained in the introduction of this example, a reference to a predefined cursor (like **Cursors.Hand**) must be serialized so that the same reference is returned at deserialization. This is performed by adding an XML attribute named **Cursors** with the cursor name as value. For example,



here is the result of the serialization of a **Cursor** instance referencing the **Cursors.Hand** static field:

```
<Cursor Cursors="Hand" />
```

To benefit from the services provided by the **ObjectXmlSerializer** class like shared reference resolution and type resolution, the class declaration element is created by calling the base implementation.

```
XmlElement objectElt =
    base.SerializeDeclaration(manager, obj, document, parent, refElt);
// Determine whether it is one of the predefined Cursors
PropertyInfo[] cursors = typeof(Cursors).GetProperties(BindingFlags.Public |
BindingFlags.Static);
for (int i = 0; i < cursors.Length; ++i) {
    Cursor cursor = (Cursor)cursors[i].GetValue(null, null);
    if (cursor == ((Cursor)obj)) {
        // Serialize the cursor name as an attribute
        string field = cursors[i].Name;
        XmlAttribute attribute = document.CreateAttribute("Cursors");
        attribute.Value = field;
        objectElt.Attributes.Append(attribute);
        return objectElt;
    }
}
Dim objectElt As XmlElement = _
    MyBase.SerializeDeclaration(manager, obj, document, parent, refElt)
Dim cursors As PropertyInfo() = _
    GetType(Cursors).GetProperties(BindingFlags.Public Or BindingFlags.Static)
Dim i As Integer = 0
While i < cursors.Length
    Dim cursor As Cursor = _
        CType(cursors(i).GetValue(Nothing, Nothing), Cursor)
    If cursor = CType(obj, Cursor) Then
        Dim field As String = cursors(i).Name
        Dim attribute As XmlAttribute = _
            document.CreateAttribute("Cursors")
        attribute.Value = field
        objectElt.Attributes.Append(attribute)
        Return objectElt
    End If
    i++
End While
```

If it is a custom cursor, the bitmap data are encoded into a base64 string and stored in a text node of a child node named data. For example, here is the result of the serialization of a custom cursor (note that the base64 encoded string is partially shown below):

```
<Cursor>
  <data>AAACAAEI...</data>
</Cursor>
```

```
XmlElement propElt = document.CreateElement(objectElt.Prefix, "data",
    objectElt.NamespaceURI);
byte[] data = (byte[])Converter.ConvertTo(obj, typeof(byte[]));
string base64 = Convert.ToBase64String(data);
```

```

XmlText textNode = document.CreateTextNode(base64);
propElt.AppendChild(textNode);
objectElt.AppendChild(propElt);
return objectElt;
Dim propElt As XmlElement = document.CreateElement(objectElt.Prefix, _
                                                    "data", objectElt.NamespaceURI)
Dim data As Byte() = CType(Converter.ConvertTo(obj, GetType(Byte())), _
                           Byte())
Dim base64 As String = Convert.ToBase64String(data)
Dim textNode As XmlText = document.CreateTextNode(base64)
propElt.AppendChild(textNode)
objectElt.AppendChild(propElt)
Return objectElt

```

4. Finally, the last step is to write the deserialization code. Since the **Cursor** class does not define a default constructor, the way an instance is created by the **ObjectXmlSerializer** class needs to be changed to take into account the new declaration element written during the serialization. To do so, the method `CreateInstance` must be overridden. The purpose of this method is to return an instance of the specified type for the **XmlElement** given as parameter.

The overridden implementation first checks whether there is a **Cursors** XML attribute and try to get the corresponding predefined cursor if it is the case. Otherwise, it looks for a `<data>` child node and convert the base64 string to a byte array and creates a **Cursor** from it thanks to the **CursorConverter** class:

```

protected override object
    CreateInstance(IDesignerSerializationManager manager,
                  XmlDocument document,
                  Type objType,
                  string name,
                  XmlElement objectElt) {
    // Determine whether it is one of the predefined Cursors
    if (objectElt.HasAttribute("Cursors")) {
        string field = objectElt.GetAttribute("Cursors");
        PropertyInfo pinfo = typeof(Cursors).GetProperty(field,
                                                            BindingFlags.Public | BindingFlags.Static);
        if (pinfo != null)
            return pinfo.GetValue(null, null);
        else
            throw new InvalidOperationException("Cannot resolve cursor " + field);
    }
    // else deserialize the bitmap data
    XmlNode node = objectElt.FirstChild;
    while (node != null && !node.LocalName.Equals("data"))
        node = node.NextSibling;
    if (node != null) {
        XmlElement dataElt = (XmlElement)node;
        string value = (string)((XmlText)dataElt.FirstChild).Value;
        value = XmlUtilities.FixBase64ForImage(value);
        if (value.Length > 0) {
            byte[] bitmapData = new byte[value.Length];
            bitmapData = Convert.FromBase64String(value);
            Cursor cursor = (Cursor)Converter.ConvertFrom(bitmapData);
            return cursor;
        }
    }
}

```

```

    }
    throw new ArgumentException("Cannot create Cursor from XmlElement: " +
objectElt + ".");
}
Protected Overloads _
Overrides Function CreateInstance(ByVal manager As
IDesignerSerializationManager, _
                                ByVal document As XmlDocument, _
                                ByVal objType As Type, _
                                ByVal name As String, _
                                ByVal objectElt As XmlElement) As Object
If objectElt.HasAttribute("Cursors") Then
    Dim field As String = objectElt.GetAttribute("Cursors")
    Dim pinfo As PropertyInfo _
        = GetType(Cursors).GetProperty(field, BindingFlags.Public Or
BindingFlags.Static)
    If Not (pinfo Is Nothing) Then
        Return pinfo.GetValue(Nothing, Nothing)
    Else
        Throw New InvalidOperationException("Cannot resolve cursor " + field)
    End If
End If
Dim node As XmlNode = objectElt.FirstChild
While Not (node Is Nothing) AndAlso Not node.LocalName.Equals("data")
    node = node.NextSibling
End While
If Not (node Is Nothing) Then
    Dim dataElt As XmlElement = CType(node, XmlElement)
    Dim value As String = CType(CType(dataElt.FirstChild, XmlText).Value,
String)
    value = XmlUtilities.FixBase64ForImage(value)
    If value.Length > 0 Then
        Dim bitmapData(value.Length) As Byte
        bitmapData = Convert.FromBase64String(value)
        Dim cursor As Cursor = CType(Converter.ConvertFrom(bitmapData), Cursor)
        Return cursor
    End If
End If
Throw New ArgumentException("Cannot create Cursor from XmlElement: " + _
objectElt + ".")
End Function

```

Once the new serializer class is written, it needs to be associated with a type in order to be taken into account during the search of the **ObjectXmlSerializerBase**.

Depending on the context, the following solutions are available:

- ◆ *Associating an ObjectXmlSerializerBase Implementation with a Given Custom Type*
- ◆ *Associating an ObjectXmlSerializerBase Implementation with a Given Type at Property Level*
- ◆ *Providing an ObjectXmlSerializerBase Implementation Through a Custom ISerializationProvider*

### **Associating an ObjectXmlSerializerBase Implementation with a Given Custom**

## Type

When an instance is serialized, the `XmlSerializationProvider` tries to find an **ObjectXmlSerializerBase** implementation suitable for the given type. You can specify the **ObjectXmlSerializerBase** to use for a specific type by means of the **System.ComponentModel.Design.Serialization.DesignerSerializerAttribute** class.

For example, the following example shows how to set **MyXmlSerializer** class as the **ObjectXmlSerializerBase** implementation to be used to serialize **MyGraphicObject** class.

```
[DesignerSerializerAttribute(typeof(MyXmlSerializer),
                             typeof(ObjectXmlSerializerBase))]
public class MyGraphicObject : GraphicObject {
    ...
}
```

This is the simplest solution when you have access to the source code of the class. It requires no operation other than tagging the class with the attribute.

## Associating an ObjectXmlSerializerBase Implementation with a Given Type at Property Level

In some cases it might be necessary to change the **ObjectXmlSerializerBase** implementation to use at property level without impacting the default association. For example, when the value of a property refers to predefined constant fields and must be referenced again after deserialization. In this case, the XML serialization framework provides the `PropertyDesignerSerializerAttribute` that acts as the **DesignerSerializerAttribute** but at property level.

The following code extract sets the **CursorSerializer** class implemented in the example *Extending the ObjectXmlSerializer Class* as the **ObjectXmlSerializerBase** implementation to be used to serialize the **Cursor** property of **MyGraphicObject** class:

```
public class MyGraphicObject : GraphicObject {
    ...
    [PropertyDesignerSerializer(typeof(CursorSerializer),
                               typeof(ObjectXmlSerializerBase))]
    public Cursor Cursor {
        ...
    }
}
```

This solution requires no operation other than tagging the property of your class with the attribute.

## Providing an ObjectXmlSerializerBase Implementation Through a Custom ISerializationProvider

If the logic of the default **XmlSerializationProvider** has to be changed, a new **ISerializationProvider** implementation can be provided to the `XmlSerializationManager`. This is performed by means of the **IDesignerSerializationProvider.AddSerializationProvider** method.

For example, this can be the case of a custom **ObjectXmlSerializerBase** implementation that must be associated with a type for which the source code is not available (therefore the first solution cannot be applied).

The following example shows how to use the string serializer implemented in the example *Extending the ObjectXmlSerializerBase Abstract Class*.

```
public class MySerializationProvider : ISerializationProvider {
    public object GetSerializer(IDesignerSerializationManager manager,
                               object currentSerializer,
                               Type objectType,
                               Type serializerType) {
        if (serializerType == typeof(ObjectXmlSerializerBase)) {
            // implements the logic depending on objectType.
            if (objectType == ...) {
                return new StringXmlSerializer ();
            }
        }
        return null;
    }
}

...
XmlSerializationManager mgr = new XmlSerializationManager();
((IDesignerSerializationManager)mgr).AddSerializationProvider(new
MySerializationProvider());
DiagramXmlSerializer s = new DiagramXmlSerializer(mgr);
MemoryStream output = new MemoryStream();
s.Serialize(instance, output);
...
```



# *Animating Graphic Objects*

IBM® ILOG® Diagram for .NET provides a rich set of graphic objects that can be used to build attractive user interfaces. Animation can be used to improve the user experience by creating smooth transitions between states or by providing helpful visual cues.

## **In This Section**

### *Animation Overview*

Introduces the animation process.

### *Controlling Animation Execution*

Introduces the animation engine.

### *Animating a Property*

Introduces the PropertyAnimation class.

### *Grouping Animations*

Explains how to group animations.

### *Using Animation as a Timer*

Introduces the TimerAnimation class.

### *Animating a Graphic Object Along a Path*

Introduces the MotionPathAnimation class.

### *Animation Types*

Describes all the animations available in IBM ILOG Diagram for .NET.

### *Creating a Custom Animation*

Explains how to write your own animation class.

---

## Animation Overview

Animation is the process of displaying quickly a series of images. Each image is slightly different from the last one to give the illusion of a moving scene.

Animation can be used to improve the user experience, by providing transitions between states. For example, if you want to make an object disappear, instead of simply hiding it, you can make it fade out by changing its opacity.

IBM® ILOG® Diagram for .NET provides a built-in animation framework that eases the creation of animations. It is composed of an animation engine, and a set of predefined animations. The animation engine is represented by the **Animator** class. It allows you to control and monitor animations at the application level. In most cases, users do not have to access the animation engine. An **Animation** is a subclass of the **Animation** class.

It can be started, stopped, suspended. When an animation is started, the animation engine animates it by calling its animation method repeatedly.

**Note:** *The animation framework can also be used to animate custom objects, not only graphic objects.*

---

## Controlling Animation Execution

Animations are scheduled by the animation engine. There is one animation engine per application. The animation engine class is represented by the **Animator** class which can be used to monitor animations or to fix the animations frame rate at the application level.

To fix the animations frame rate, use the `FramesPerSeconds` property.

To monitor animations, use the `BeforeAnimation` and `AfterAnimation` events.



The animation engine schedules animations depending on their running status. The animation status is given by the Status property and can have the following values:

<b>Started</b>	The animation has been started and is running.
<b>Suspended</b>	The animation has been suspended.
<b>Stopped</b>	The animation has been stopped.

To start an animation, use the Start method.

To suspend a running animation, use the Suspend method. A suspended animation can be resumed by calling the **Start** method.

To stop a running animation, use the Stop method.

Each animation can be monitored by using the animation events: Started, Stopped, Suspended, Resumed, Finished, BeforeAnimating and AfterAnimating.

There are two different types of animations:

- ◆ Animations whose duration is specified.

This type of animation is represented by the BoundedAnimation class. The following table lists several properties of the **BoundedAnimation** class that can be used to control the animation execution.

<b>Duration</b>	Gets or sets the animation duration.
<b>AutoReverse</b>	Indicates if the animation should play in reverse at the end.
<b>RepeatCount</b>	The number of times the animation will be run before ending. A negative value can be used to specify that the animation will never end.
<b>HoldEnd</b>	Indicates whether the animation should go back to the state before playing the animation.

- ◆ Animation whose duration is infinite.

This type of animation is represented by the TimerAnimation class. For more details see *Using Animation as a Timer*.

## Animating a Property

In most cases, animating a graphic object means changing one or several of its properties to give the illusion of animation. The base class for animating properties is the PropertyAnimation class. This class can be used to animate any kind of property of any object. For example, you can use it to move a graphic object by modifying its location or to

change the color of a `SolidFill` object. The **PropertyAnimation** class has several subclasses dedicated to the animation of properties of a specific type. For a complete list of **PropertyAnimation** subclasses, see *Animation Types*.

In this section you are going to see how to animate the opacity of a graphic object to make it fade out.

The opacity of a graphic object is given through the `Opacity` property, whose type is **float**. The animation class responsible for animating float values is the `FloatAnimation` class. The **FloatAnimation** class produces float values between a starting value and an ending value in a given duration. In this case, the starting value is 1, which means that the object is opaque, and the ending value is 0, which means that the object is transparent. The animation duration is fixed to 1 second: the animation interpolates float values from 0 to 1 within this duration and then is automatically stopped.

```
public void StartFadeOutAnimation(GraphicObject obj)
{
    TimeSpan duration = TimeSpan.FromSeconds(0.5f);
    FloatAnimation animation = new FloatAnimation(obj, "Opacity", 1f, 0f,
duration);
    animation.Start();
}
Public Sub StartFadeOutAnimation(ByVal obj As GraphicObject)
    Dim duration As TimeSpan = TimeSpan.FromSeconds(0.5!)
    Dim animation As FloatAnimation = New FloatAnimation(obj, Opacity, 1!,
0!, duration)
    animation.Start
End Sub
```

For some property types, like enumeration or string, it is not possible to interpolate values. If you need to animate a property that cannot be interpolated, you can use the `SetAnimation` class. The **SetAnimation** class allows you to specify a collection of pairs (duration, value) that is used to animate the property instead of using interpolation.

The following example shows how to animate the `Text` property of a `GraphicObject` using the **SetAnimation** class.

```
public void StartTextAnimation(GraphicObject obj)
{
    AnimationFrame[] frames = new AnimationFrame[] {
        new AnimationFrame(TimeSpan.FromSeconds(0), "Zero"),
        new AnimationFrame(TimeSpan.FromSeconds(1), "One"),
        new AnimationFrame(TimeSpan.FromSeconds(2), "Two"),
        new AnimationFrame(TimeSpan.FromSeconds(3), "Three")
    };
    SetAnimation animation = new SetAnimation(obj, "Text", frames);
    animation.Start();
}
Public Sub StartTextAnimation(ByVal obj As GraphicObject)
    Dim frames() As AnimationFrame = New AnimationFrame() {
        New AnimationFrame(TimeSpan.FromSeconds(0), "Zero"),
        New AnimationFrame(TimeSpan.FromSeconds(1), "One"),
        New AnimationFrame(TimeSpan.FromSeconds(2), "Two"),
    }
```

```

        New AnimationFrame(TimeSpan.FromSeconds(3), "Three")
    }
    Dim animation As SetAnimation = New SetAnimation(obj, "Text", frames)
    animation.Start
End Sub

```

---

## Grouping Animations

When building complex animations, it may be useful to group different animations into a single one to make the programming easier. For example, if you want to create an animation that rotates a graphic object and changes its fill color at the same time, you can group both animations into an **AnimationGroup**.

```

public void StartAnimation(Shape obj)
{
    TimeSpan duration = TimeSpan.FromSeconds(1);

    AnimationGroup animation = new AnimationGroup();

    animation.Add(new TransformAnimation(obj, "Transform",
                                         obj.Transform, 90f,
                                         obj.Location, duration));

    SolidFill fill = new SolidFill(Color.Red);
    obj.Fill = fill;
    animation.Add(new ColorAnimation(fill, "Color", fill.Color,
                                     Color.Blue, duration));

    animation.Start();
}
Public Sub StartAnimation(ByVal obj As Shape)
    Dim duration As TimeSpan = TimeSpan.FromSeconds(1)

    Dim animation As AnimationGroup = New AnimationGroup

    animation.Add(New TransformAnimation(obj, Transform, _
                                         obj.Transform, _
                                         90!, obj.Location, _
                                         duration))

    Dim fill As SolidFill = New SolidFill(Color.Red)
    obj.Fill = fill
    animation.Add(New ColorAnimation(fill, Color, _
                                     fill.Color, _
                                     Color.Blue, duration))

    animation.Start()
End Sub

```

Animations added to an **AnimationGroup** are executed in parallel. This means that when a group of animations starts, every child animation also starts. To simulate animations run in

series, you can use the `Begin` property to specify that an animation begins after a specified amount of time after it has been started.

The following example shows how to create an animation that will animate the X coordinate from 0 to 100 during 1 second, then the Y coordinate from 0 to 100 during 1 second.

```
public void StartAnimation(Shape obj)
{
    TimeSpan duration = TimeSpan.FromSeconds(1);

    AnimationGroup animation = new AnimationGroup();

    FloatAnimation xAnim = new FloatAnimation(obj, "X", 0f, 100f, duration);
    animation.Add(xAnim);

    FloatAnimation yAnim = new FloatAnimation(obj, "Y", 0f, 100f, duration);
    yAnim.Begin = xAnim.GetActualDuration();
    animation.Add(yAnim);

    animation.Start();
}
Public Sub StartAnimation(ByVal obj As Shape)
    Dim duration As TimeSpan = TimeSpan.FromSeconds(1)

    Dim animation As AnimationGroup = New AnimationGroup

    Dim xAnim As FloatAnimation = New FloatAnimation(obj, X, 0!, 100!, duration)
    animation.Add(xAnim)

    Dim yAnim As FloatAnimation = New FloatAnimation(obj, Y, 0!, 100!, duration)
    yAnim.Begin = xAnim.GetActualDuration
    animation.Add(yAnim)

    animation.Start()
End Sub
```

Grouping animations eases the control of the animation execution. To start, suspend or stop an animation, you have just to call the corresponding method on the group rather than on each animation.

---

## Using Animation as a Timer

The `TimerAnimation` class is an animation that can be used as a timer. An event is triggered at fixed intervals and the animation runs until it has been explicitly stopped by calling the `Stop` method.

Use the **TimerAnimation** class when you do not know the duration of the animation, or when you want to animate an object at regular intervals.

## Animating a Graphic Object Along a Path

The `MotionPathAnimation` class allows you to animate a `GraphicObject` along a `PathData`. The following example moves a rectangle in order to describe an ellipse.

```
public void StartAnimation(GraphicObject obj)
{
    TimeSpan duration = TimeSpan.FromSeconds(10f);

    PathData path = new PathData();
    path.AddEllipse(new Rectangle2D(0, 0, 200, 200));

    MotionPathAnimation animation = new MotionPathAnimation(obj, path,
duration);
    animation.RepeatCount = -1;
    animation.Start();
}
Public Sub StartAnimation(ByVal obj As GraphicObject)
    Dim duration As TimeSpan = TimeSpan.FromSeconds(10!)

    Dim path As PathData = New PathData
    path.AddEllipse(New Rectangle2D(0, 0, 200, 200))

    Dim animation As MotionPathAnimation _
        = New MotionPathAnimation(obj, path, duration)
    animation.RepeatCount = -1

    animation.Start
End Sub
```

## Animation Types

This section describes all the animations that are provided in IBM® ILOG® Diagram for .NET.

Class	Type	Description
FloatAnimation	Bounded	Animates a <b>float</b> property by interpolating between two values.
DoubleAnimation	Bounded	Animates a <b>double</b> property by interpolating between two values.
IntAnimation	Bounded	Animates an <b>int</b> property by interpolating between two values.
Point2DAnimation	Bounded	Animates a <b>Point2D</b> property by interpolating between two values.

Class	Type	Description
Size2DAnimation	Bounded	Animates a <b>Size2D</b> property by interpolating between two values.
Rectangle2DAnimation	Bounded	Animates a <b>Rectangle2D</b> property by interpolating between two values.
ColorAnimation	Bounded	Animates a <b>Color</b> property by interpolating between two values.
SetAnimation	Bounded	Animates any property by specifying animation frames.
TransformAnimation	Bounded	Animates a <b>Transform</b> property by interpolating between two values.
MotionPathAnimation	Bounded	Animates a <b>GraphicObject</b> along a <b>PathData</b> .
TimerAnimation	Timer	Triggers an event at regular intervals.

---

## Creating a Custom Animation

By grouping and composing the existing animations classes, it is possible to create complex animations. However, you may have to write your own animation class. For example, if you want to animate a property whose type is a custom type, you may want to create an animation class dedicated to this type.

The following example shows how to implement an animation class that interpolates values between two **DateTime** objects.

```
public class DateTimeAnimation : PropertyAnimation
{
    private DateTime _from;
    private DateTime _to;

    public DateTimeAnimation(object instance, string propertyName,
        DateTime from, DateTime to, TimeSpan duration)
        : base(instance, propertyName, duration)
    {
        _from = from;
        _to = to;
    }

    protected override object GetValueAt(TimeSpan duration,
        TimeSpan relativeDuration,
        float completion,
        bool reversing,
        bool repeating)
    {
        long ticks = (long)(_from.Ticks + (_to.Ticks - _from.Ticks) *

```

```

completion);
        return new DateTime(ticks);
    }
}
Public Class DateTimeAnimation
    Inherits PropertyAnimation

    Private _from As DateTime
    Private _to As DateTime

    Public Sub New(ByVal instance As Object, ByVal propertyName As String, _
        ByVal from As DateTime, ByVal to As DateTime, _
        ByVal duration As TimeSpan)
        MyBase.New(instance, propertyName, duration)
        _from = from
        _to = to
    End Sub

    Protected Overrides Function GetValueAt(ByVal duration As TimeSpan, _
        ByVal relativeDuration As TimeSpan, _
        ByVal completion As Single, _
        ByVal reversing As Boolean, _
        ByVal repeating As Boolean) As Object

        Dim ticks As Long = CType((_from.Ticks -
            + ((_to.Ticks - _from.Ticks) *
            * completion)), Long)
        Return New DateTime(ticks)
    End Function
End Class

```





## *Printing a Diagram*

IBM® ILOG® Diagram for .NET provides the API that enables you to print diagrams. This API is composed of a print document, the `DiagramPrintDocument` class, and several dialog boxes to help you configure the print settings. As the printing API extends the classes of the .NET Framework, you are able to use all the printing support of .NET Framework to print your diagrams.

### **In This Section**

#### *Setting up a Print Document*

Introduces the **DiagramPrintDocument** class.

#### *Using Predefined Printing Dialog Boxes*

Introduces the predefined dialog boxes dedicated to printing.

---

## **Setting up a Print Document**

A print document contains all the information needed to print a diagram. It describes settings related to the printer such as the printer name, the paper size, or the paper orientation. It also contains data about the diagram that will be printed, such as the container or the number of pages on which the print document will be printed. The print document is defined by the **DiagramPrintDocument** class.

---

### Printing a Graphic Container

The **DiagramPrintDocument** class allows you to print the content of a **GraphicContainer**.

The following example shows how to print the content of a group that contains an **Ellipse**:

```
Group container = new Group();
Ellipse ellipse = new Ellipse(0, 0, 100, 100);
container.Objects.Add(ellipse);
DiagramPrintDocument document = new DiagramPrintDocument(container);
document.PrinterSettings.PrinterName = "MyPrinter";
document.Print();
Dim container As Group = New Group
Dim ellipse As Ellipse = New Ellipse(0, 0, 100, 100)
container.Objects.Add(ellipse)
Dim document As DiagramPrintDocument = New DiagramPrintDocument(container)
document.PrinterSettings.PrinterName = "MyPrinter"
document.Print()
```

In this example, the document content is set in the constructor of **DiagramPrintDocument**. The content can also be accessed by using the **Content** property.

**Note:** *MyPrinter should be replaced by a real printer name. You do not need to set this property when you use the predefined print dialog boxes. For more details, see Using Predefined Printing Dialog Boxes.*

---

### Specifying the Area to Print

By default, the document prints the whole content of its container. To change this, the **PrintAll** property of the document must be set to **false** and the **ContentPrintArea** should be set to the desired area. The **ContentPrintArea** property defines the rectangular area of the document container that will be printed.

The following example shows how to preview the partial content of a group that contains an **Ellipse** and a **Rect**, focusing on the **Ellipse**:

```
Group container = new Group();
Ellipse ellipse = new Ellipse(0, 0, 100, 100);
container.Objects.Add(ellipse);
Rect rect = new Rect(200, 200, 100, 100);
container.Objects.Add(rect);
DiagramPrintDocument document = new DiagramPrintDocument(container);
document.PrintAll = false;
document.ContentPrintArea = ellipse.Bounds;
DiagramPrintPreviewDialog dlg = new DiagramPrintPreviewDialog(document);
dlg.ShowDialog();
Dim container As Group = New Group
Dim ellipse As Ellipse = New Ellipse(0, 0, 100, 100)
container.Objects.Add(ellipse)
Dim rect As Rect = New Rect(200, 200, 100, 100)
```

```

container.Objects.Add(rect)
Dim document As DiagramPrintDocument = New DiagramPrintDocument(container)
document.PrintAll = false
document.ContentsPrintArea = ellipse.Bounds
Dim dlg As DiagramPrintPreviewDialog = New DiagramPrintPreviewDialog(document)
dlg.ShowDialog

```

---

## Printing on Several Pages

To adjust the size of the printing by specifying a zoom level on the document, the **AutoZoom** property of the document must be set to **false** and the **Zoom** property should be set to the desired zoom level. The **DiagramPrintDocument** class automatically computes the number of pages needed to draw the diagram at the specified zoom level.

To adjust the size of the printing by specifying a number of printed pages, the **AutoZoom** property of the document must be set to **true** and the **Rows** and **Columns** properties must be set to the desired values.

The following example shows how to preview the content of a group that contains an **Ellipse**, on two columns and two rows:

```

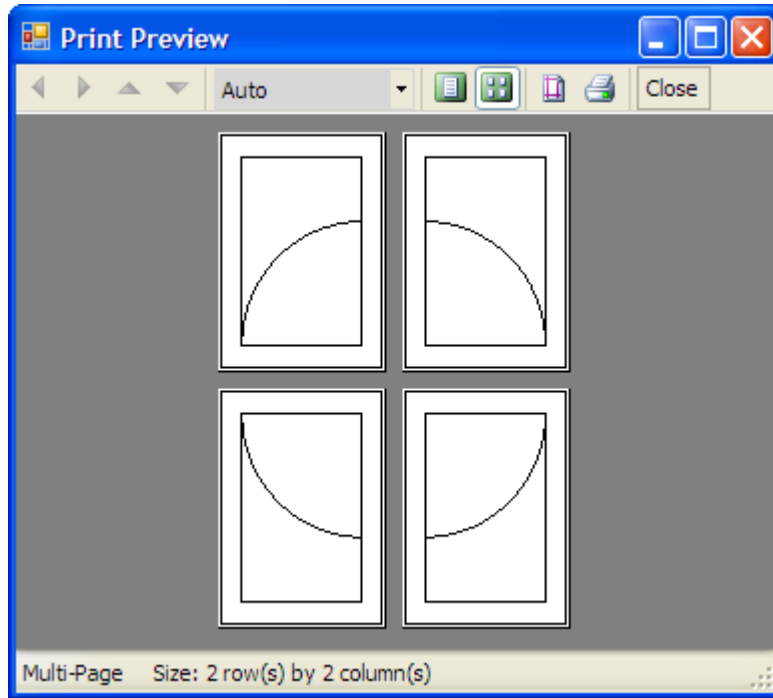
Group container = new Group();
Ellipse ellipse = new Ellipse(0, 0, 100, 100);
container.Objects.Add(ellipse);
DiagramPrintDocument document = new DiagramPrintDocument(container);
document.AutoZoom = true;
document.Rows = 2;
document.Columns = 2;
DiagramPrintPreviewDialog dlg = new DiagramPrintPreviewDialog(document);
dlg.ShowDialog();
Dim container As Group = New Group
Dim ellipse As Ellipse = New Ellipse(0, 0, 100, 100)
container.Objects.Add(ellipse)
Dim document As DiagramPrintDocument = New DiagramPrintDocument(container)
document.AutoZoom = true
document.Rows = 2
document.Columns = 2
Dim dlg As DiagramPrintPreviewDialog = New DiagramPrintPreviewDialog(document)
dlg.ShowDialog

```

**Note:** Although the *PrintPreviewDialog* of the .NET Framework® could have been used to preview the printing, IBM ILOG Diagram for .NET provides a more powerful print preview dialog box with the *DiagramPrintPreviewDialog* class.

## Printing a Diagram

The following illustration shows the print preview dialog you have just created:

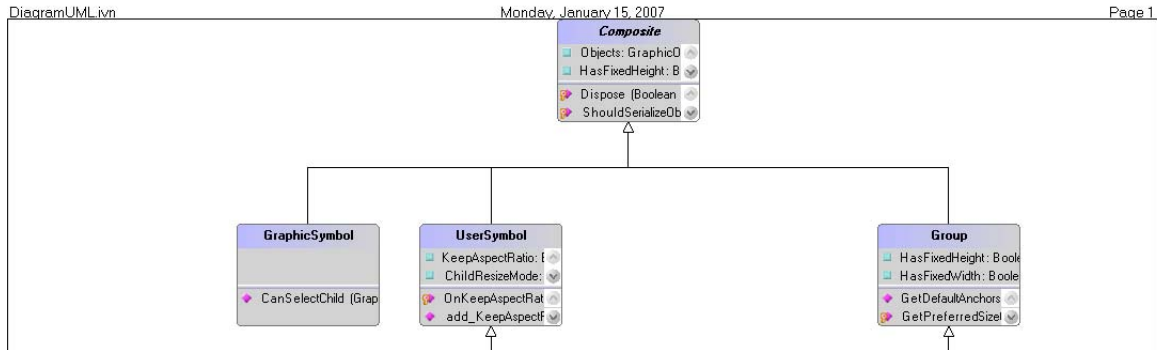


---

### Customizing the Printing

Decorations can be added to each printed page. Header and footer can be added by setting respectively the Header and Footer properties of `ExtendedPrintDocument`. A frame can also be painted around the printing area by setting the `MustPrintFrame` property of the `ExtendedPrintDocument`.

The following illustration shows an example of header.



Each of the three text sections of a header or footer can contain the text that you specify in the constructor of the object.

The following example shows how to obtain a header:

```
Header header =
    new Header("Monday, January 15, 2007", "DiagramUML.ivn", "Page 1");
Dim header as Header =
    New Header("Monday, January 15, 2007", " DiagramUML.ivn", "Page 1")
```

Since the header and footer are defined on the document, you should not specify the page number as in the previous example. The HeaderFooter class provides a few patterns that will be translated to values from the document when the document is printed. The following table lists the patterns that you can use:

Pattern	Description
PagePattern	The pattern for the page number.
PagesPattern	The pattern for the number of pages in the document.
DatePattern	The pattern for the printing date.
TimePattern	The pattern for the printing time.
FileNamePattern	The pattern for the name of the file associated with the document.
DocumentPattern	The pattern for the name of the document.
AuthorPattern	The pattern for the name of the author of the document.

## Printing a Diagram

To create the header in the figure above, use the following code:

```
Header header =  
    new Header(HeaderFooter.DatePattern,  
              HeaderFooter.DocumentPattern,  
              "Page " + HeaderFooter.PagePattern)  
Dim header as Header =  
    New Header(HeaderFooter.DatePattern, _  
              HeaderFooter.DocumentPattern, _  
              ("Page " + HeaderFooter.PagePattern))
```

The following table summarizes the properties of the DiagramPrintDocument class:

Property	Description	Default Value
DocumentName (inherited from PrintDocument)	The name of the document.	""
DefaultPageSettings (inherited from PrintDocument)	The paper size, margins, and orientation of pages.	
PrinterSettings (inherited from PrintDocument)	The printer that prints the document.	
Author (inherited from ExtendedPrintDocument)	The name of the author of the document.	""
File (inherited from ExtendedPrintDocument)	The name of the file that is printed.	""
Header (inherited from ExtendedPrintDocument)	The header of each page.	<b>null</b>
Footer (inherited from ExtendedPrintDocument)	The footer of each page.	<b>null</b>
AutoZoom	Indicates whether the documents automatically adjust the zoom level.	<b>false</b>
Zoom	The zoom level used when printing. Used only if <b>AutoZoom</b> is set to <b>false</b> .	1
Columns	The number of pages in the horizontal direction. Used only if <b>AutoZoom</b> is set to <b>true</b> .	1

Property	Description	Default Value
Rows	The number of pages in the vertical direction. Used only if <b>AutoZoom</b> is set to <b>true</b> .	1
ContentPrintArea	The area of the container to print. Used only if <b>PrintAll</b> is set to <b>false</b> .	The bounds of the document container.
PrintAll	Indicates whether the document chooses the printed area automatically to print all the contents of the container.	<b>true</b>
IsOverThenDownOrder	Indicates the page-numbering order.	<b>true</b>

---

## Using Predefined Printing Dialog Boxes

IBM® ILOG® Diagram for .NET provides two predefined dialog boxes dedicated to printing:

- ◆ A dialog box for setting up a print document.
- ◆ A dialog box for previewing a print document.

### In This Section

#### *Setting up a Print Document Using the Predefined Dialog Box*

Introduces the `DiagramPageSetupDialog` class.

#### *Previewing a Print Document Using the Predefined Dialog Box*

Introduces the `DiagramPrintPreviewDialog` class.

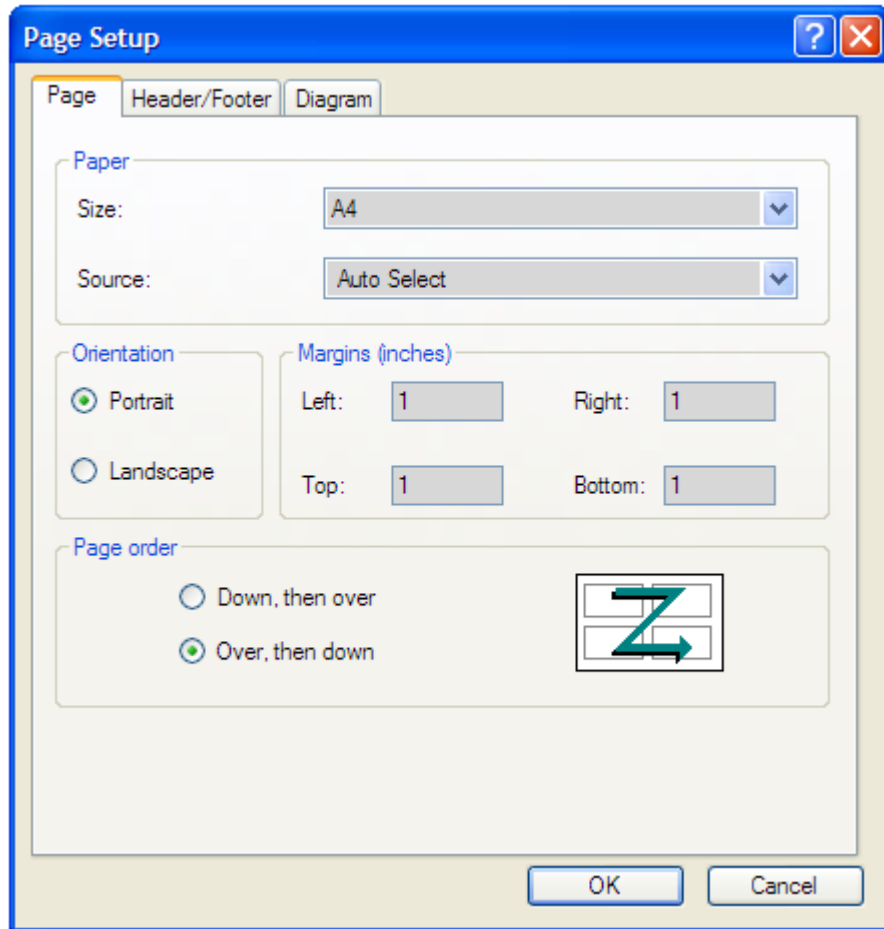
---

## Setting up a Print Document Using the Predefined Dialog Box

IBM ILOG Diagram for .NET provides a dialog box for setting up a print document by means of the **DiagramPageSetupDialog** class. This dialog box is used to edit the common printing properties, such as paper size, paper orientation, or margins, and also additional properties like header and footer or the area to print.

## Printing a Diagram

The following illustration shows the **DiagramPageSetupDialog** dialog box:



This dialog box has the following tab pages:

Tab Page	Description
Page	For editing page settings, such as paper size or paper orientation.
Header/Footer	For editing the header and footer of the print document.
Diagram	For editing diagram properties, such as the area to print or the zoom level.

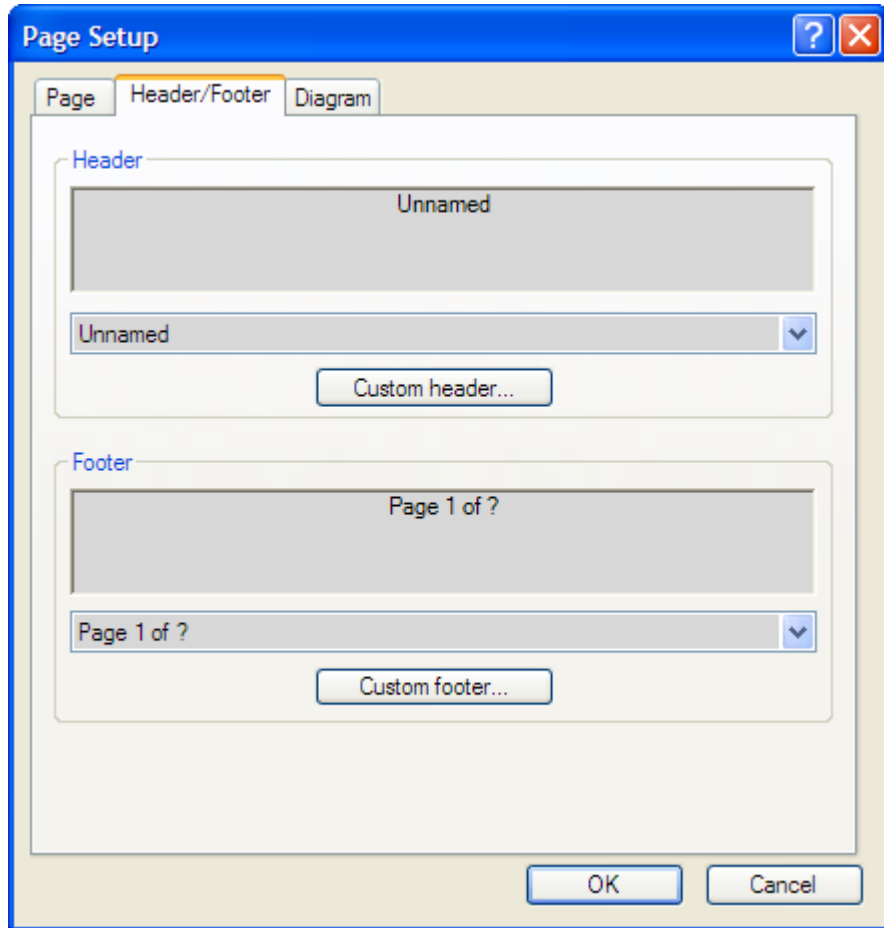


The following example shows how to use the **DiagramPageSetupDialog** dialog:

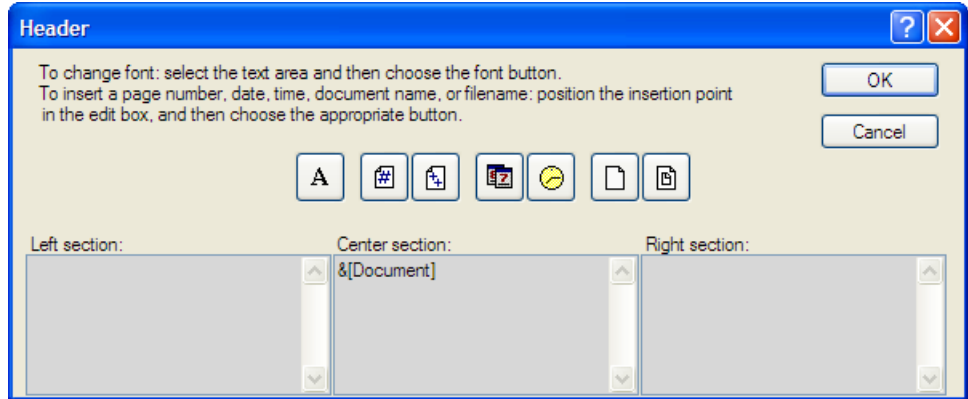
```
Group container = new Group();
Ellipse ellipse = new Ellipse(0, 0, 100, 100);
container.Objects.Add(ellipse);
DiagramPrintDocument document = new DiagramPrintDocument(container);
DiagramPageSetupDialog dialog = new DiagramPageSetupDialog();
dialog.Document = document;
dialog.ShowDialog();
Dim container As Group = New Group
Dim ellipse As Ellipse = New Ellipse(0, 0, 100, 100)
container.Objects.Add(ellipse)
Dim document As DiagramPrintDocument = New DiagramPrintDocument(container)
Dim dialog As DiagramPageSetupDialog = New DiagramPageSetupDialog
dialog.Document = document
dialog.ShowDialog
```

### Customizing the Header and Footer

The following illustration shows the Header/Footer tab of the Page Setup dialog box:

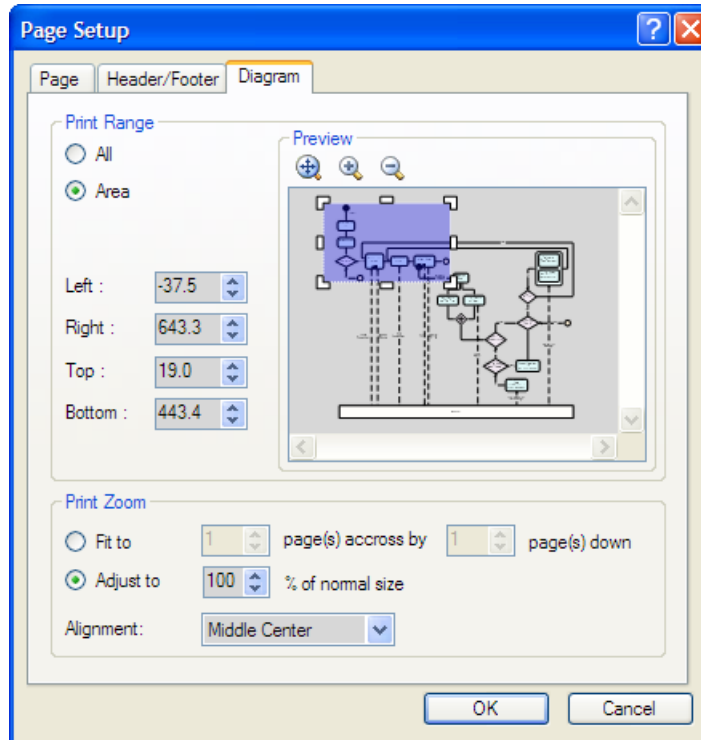


The combo boxes contain predefined header and footer settings. To use a custom header or footer, select the corresponding button. Then, the following dialog box is displayed to allow you to create a custom header or footer:



### Customizing the Diagram Print Settings

The following illustration shows the Diagram tab of the Page Setup dialog box:



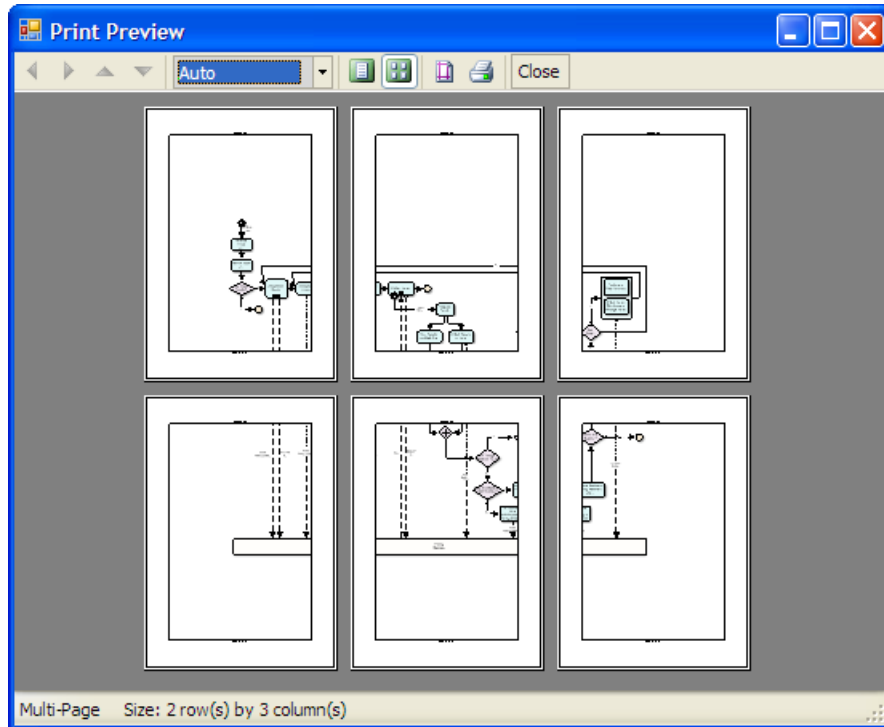
In Print Range, you can specify the area of the document that will be printed. In Print Zoom, you can select the size of the printing.

---

### Previewing a Print Document Using the Predefined Dialog Box

IBM ILOG Diagram for .NET provides a dialog box for previewing a print document by means of the **DiagramPrintPreviewDialog** class. This dialog box allows you to preview a print document, change the print settings through the Page Setup dialog box and print the document.

The following illustration shows the Print Preview dialog box:



The dialog box has two mode to visualize the pages:

- ◆ Single page mode: only one page is displayed at a time. In this mode, the arrow buttons can be used to navigate through the pages.
- ◆ Multiple pages mode: all the pages are visible.

The following example shows how to preview the content of a group that contains an Ellipse, on two columns and two rows:

```

Group container = new Group();
Ellipse ellipse = new Ellipse(0, 0, 100, 100);
container.Objects.Add(ellipse);
DiagramPrintDocument document = new DiagramPrintDocument(container);
document.AutoZoom = true;
document.Rows = 2;
document.Columns = 2;
DiagramPrintPreviewDialog dlg = new DiagramPrintPreviewDialog(document);
dlg.ShowDialog();
Dim container As Group = New Group
Dim ellipse As Ellipse = New Ellipse(0, 0, 100, 100)
container.Objects.Add(ellipse)
Dim document As DiagramPrintDocument = New DiagramPrintDocument(container)

```

## Printing a Diagram

```
document.AutoZoom = true
document.Rows = 2
document.Columns = 2
Dim dlg As DiagramPrintPreviewDialog = New DiagramPrintPreviewDialog(document)
dlg.ShowDialog
```

**Note:** To preview a print document, you can also use the **PrintPreviewDialog** class of the .NET Framework®.

## ***Importing SVG Files***

This section describes how to import a Scalable Vector Graphics (SVG) document in a diagram.

---

### **Scalable Vector Graphics**

IBM® ILOG® Diagram for .NET allows you to import Scalable Vector Graphics (SVG) in a diagram. SVG is a two-dimensional structure graphics format defined by the World Wide Web Consortium (W3C). The format is based on the eXtensible Markup Language (XML) which gives it a great interoperability.

An SVG file describes a set of two-dimensional graphics. For example, the following graphics could be found in an SVG file:

- ◆ Images through the image element.
- ◆ Rectangles through the rect element.
- ◆ Circles and ellipses through the circle and ellipse elements.
- ◆ Lines through the line and polyline elements.
- ◆ Polygons through the polygon element.
- ◆ Arbitrary paths (curves, arcs, lines, and so on) through the path element.

- ◆ Groups of other graphic elements through the `g` element.
- ◆ Text through the `text` element.

These elements can be styled using XML presentation attributes on them or Cascading Style Sheets (CSS) linked to the elements.

---

### SVG File Example

The following code shows a typical example of an SVG file:

```
<svg width="640" height="480">
  <defs>
    <!-- the style on path elements and element with id "myid" -->
    <!-- is defined through a style sheet -->
    <style type="text/css">
      path {stroke-width:3;stroke:blue;fill:none}
      .dash {stroke-dasharray:5 2}
      #myid {fill:rgb(205,5,5);fill-opacity:0.5}
    </style>
    <!-- style can be complex such as a gradient... -->
    <linearGradient id="grad" x1="0%" y1="0%" x2="100%"
      y2="100%">
      <stop offset="0" stop-color="yellow"/>
      <stop offset="0.2" stop-color="green"/>
      <stop offset="1" stop-color="red"/>
    </linearGradient>
  </defs>
  <!-- the style on the rectangle is defined through XML -->
  <!-- attributes -->
  <rect x="0" y="0" width="100%" height="100%"
    fill="url(#grad)"/>
  <!-- paths use a particular syntax to defined their shape -->
  <path d="M0 0L640 480"/>
  <path class="dash" d="M640 0L0 480"/>
  <!-- the style on the ellipse is defined through an inline -->
  <!-- style sheet -->
  <ellipse cx="320" cy="240" rx="40" ry="30"
    style="fill:rgb(180,10,10)"/>
  <circle id="myid" cx="320" cy="240" r="50"/>
</svg>
```

This SVG file will be rendered as a 640 x 480 rectangle filled with a linear green, yellow, and red gradient. On top of this gradient there will be two lines of thickness 3, and one of them is a dashed line. There is also an ellipse and a circle; the color on the circle is semi-transparent (fill-opacity:0.5) and lets you see the color of the ellipse underneath.

---

### For More Information

To better understand SVG and its possibilities, see the SVG specification at <http://www.w3.org/TR/SVG>. You will see that SVG provides many more features than those introduced here, such as transformations on graphic elements, filter effects, in-line animation, and scripting capabilities.



---

## Reading an SVG File

You can read an SVG file by means of the SVGReader class located in the ILOG.Diagrammer.SVG namespace.

The following code extract reads the SVG file in order to display the result in the Windows Forms component that displays diagrams (the DiagramView class).

```
void ReadSVGDocument(string fileName, DiagramView view)
{
    // Creates the Reader
    SVGReader svgReader = new SVGReader();

    // Specifies the initial viewport to the current size of the
    // diagram view
    Rectangle2D viewport = view.ViewRectangle;
    svgReader.InitialViewPort = new Rectangle2D(0, 0, viewport.Width,
viewport.Height);

    try {
        // read the document
        GraphicObject svgResult = svgReader.Load(fileName);
        Group top = new Group();
        top.Objects.Add(svgResult);
        // set the result in the view
        view.Contents = top;
        view.Transform = Transform.Identity;
    }
    catch (Exception e)
    {
        // error cannot open file
    }
}

Sub ReadSVGDocument(ByVal fileName As String, ByVal view As DiagramView)
' Creates the Reader
Dim svgReader As SVGReader = New SVGReader

' Specifies the initial viewport to the current size of the
' diagram view
Dim viewport As Rectangle2D = view.ViewRectangle
svgReader.InitialViewPort = New Rectangle2D(0, 0, viewport.Width,
viewport.Height)

Try
' reads the file
Dim svgResult As GraphicObject = svgReader.Load(fileName)
Dim top As Group = New Group
top.Objects.Add(svgResult)
' sets the result in the view
view.Contents = top
view.Transform = Transform.Identity

Catch e As Exception

' error cannot open file
```

## Importing SVG Files

```
End Try  
End Sub
```

For a more complete example, you may take a look at the SVG Viewer Sample located in Samples/Applications/SVGViewer.

---

## Using SVG to Create a Custom Graphic object

IBM ILOG Diagram for .NET allows you to create new graphical representations named user symbols.

A **UserSymbol** is a graphic object composed of other graphic objects. When creating a new **UserSymbol** through the Visual Studio Diagram Designer of IBM ILOG Diagram for .NET, you may import SVG documents to define the graphical representation of the symbol. For more information on how to create user symbols through the Visual Studio Designer, see *Creating Diagrams and User Symbols Using Visual Studio*.

---

## SVG Supported Features

To help you build SVG files that will be fully understood by the SVG reader of IBM ILOG Diagram for .NET, use the following tables for supported and unsupported SVG elements and CSS properties.

### Supported SVG Elements

Element Name	Attributes Not Supported on this Element
a	xlink:role, xlink:acrole, xlink:actuate
circle	N/A
clipPath	clipPathUnits
defs	N/A
desc	N/A
ellipse	N/A
feBlend	N/A
feColorMatrix	N/A
feComponentTransfer	N/A

Element Name	Attributes Not Supported on this Element
feComposite	N/A
feConvolveMatrix	N/A
feDiffuseLighting	N/A
feDisplacementMap	N/A
feDistantLight	N/A
feFlood	N/A
feFuncA	N/A
feFuncB	N/A
feFuncG	N/A
feFuncR	N/A
feGaussianBlur	N/A
feImage	N/A
feMerge	N/A
feMergeNode	N/A
feMorphology	N/A
feOffset	N/A
fePointLight	N/A
feSpecularLighting	N/A
feSpotLight	N/A
feTile	N/A
feTurbulence	N/A
filter	Background image Background alpha, Fill Paint and Stroke Paint not supported.
g	N/A
image	N/A
line	N/A
linearGradient	N/A

## Importing SVG Files

Element Name	Attributes Not Supported on this Element
metadata	N/A
path	N/A
pattern	patternUnits, patternTransform.
polygon	N/A
polyline	N/A
radialGradient	N/A
rect	N/A
stop	N/A
style	Important rules are not supported.
svg	zoomAndPan
switch	N/A
symbol	refX, refY, viewBox, preserveAspectRatio.
text	textLength, lengthAdjust.
textPath	textLength, lengthAdjust, method, spacing.
title	N/A
tspan	multiple values x, y, dx, dy (single values supported), rotate, textLength
use	N/A

### Supported CSS Properties

Property Name	Remark
clip-path	URI local to the file only.
color	N/A
fill	URI local to the file only, ICC colors are not supported.
fill-opacity	N/A
fill-rule	N/A
font-family	N/A

Property Name	Remark
font-size	Relative identifiers are not supported.
font-stretch	Relative identifiers are not supported.
font-style	N/A
font-weight	Relative identifiers are not supported.
opacity	N/A
stop-color	ICC colors are not supported.
stop-opacity	N/A
stroke	URI local to the file only, ICC colors are not supported.
stroke-dasharray	N/A
stroke-dashoffset	N/A
stroke-linecap	N/A
stroke-linejoin	N/A
stroke-miterlimit	N/A
stroke-opacity	N/A
stroke-width	N/A
text-anchor	N/A
visibility	N/A

## Importing SVG Files

## ***Improving the Design-Time Behavior of Your Graphic Object***

Any graphic object of IBM® ILOG® Diagram for .NET is a .NET component. You can improve the design-time behavior of the graphic objects that you create as you improve the design-time behavior of any .NET component, like for example when you create a Windows Forms **UserControl**.

See the .NET documentation on components in Visual Studio to know more about the design-time architecture of the .NET Framework®. This section describes the mostly used features.

The example presented in this section is based on the tutorial *Creating an IBM ILOG Diagram for .NET Windows Forms Application and User Symbol* which shows how to create a new graphic object (subclass of `UserSymbol`) that represents a traffic light.

This is the code of the new graphic object:

```
using System;
using ILOG.Diagrammer;
using ILOG.Diagrammer.Graphic;
using System.Drawing;

namespace DiagrammerApplication1
{
    public enum TrafficLightState
    {
        Stop,
```

## Improving the Design-Time Behavior of Your Graphic Object

```
        Go,  
        Amber  
    }  
  
    public partial class TrafficLight : UserSymbol  
    {  
        TrafficLightState state = TrafficLightState.Go;  
  
        public TrafficLight()  
        {  
            InitializeComponent();           // required by the designer  
  
            // Initialize the default value.  
            State = TrafficLightState.Stop;  
        }  
  
        public TrafficLightState State  
        {  
            get  
            {  
                return state;  
            }  
  
            set  
            {  
                if (state != value)  
                {  
                    state = value;  
  
                    stopLight.Fill = new SolidFill(  
                        value == TrafficLightState.Stop ? Color.Red :  
                        Color.LightGray);  
                    goLight.Fill = new SolidFill(  
                        value == TrafficLightState.Go ? Color.Green :  
                        Color.LightGray);  
                    amberLight.Fill = new SolidFill(  
                        value == TrafficLightState.Amber ? Color.Yellow :  
                        Color.LightGray);  
                }  
            }  
        }  
    }  
}
```

To improve the design-time experience for the traffic light object, you can add a description and a specific icon that will appear in the Visual Studio and Diagram Editor toolbox.

```
[ToolboxBitmap("c:\\trafficlight.bmp")]  
[Description("A traffic light object")]  
public partial class TrafficLight : UserSymbol  
{
```

As you see in the code, the traffic light object defines a property named **State**.



The design-time behavior of this property can be improved by adding various design-time attributes, such as information on the default value, a description or a category for the property.

```
[DefaultValue(TrafficLightState.Stop)]
[Category("Data")]
[Description("The state of the traffic light")]
public TrafficLightState State
{
    get { ...
```

The **DefaultValue** attribute is important for properties, because no code will be generated by Visual Studio and no XML node will be created by the XML serializer when the property has its default value.

For complex properties you can also specify a value editor by creating a specific **UITypeEditor** associated with the property. This editor will be used when editing the value in the Properties View of Visual Studio.

To further improve the design-time behavior of your new graphic object, you can also associate a designer object with your graphic object. The designer controls the appearance and behavior of your object at design-time and allows you to control the smart tags of your new graphic objects.

The following example shows how to code a designer for the traffic light object, so that the **State** property becomes available in the **SmartTags** window. This code creates a subclass of the `UserSymbolDesigner` class and overrides the `CreateActionList` method:

```
class TrafficLightDesigner : UserSymbolDesigner
{
    protected override DesignerActionList CreateActionList()
    {
        return new TrafficLightDesignerActionList(Component);
    }

    class TrafficLightDesignerActionList : DesignerActionList
    {
        public TrafficLightDesignerActionList(IComponent component)
            : base(component)
        {
        }

        public TrafficLightState State
        {
            get
            {
                return (this.Component as TrafficLight).State;
            }
            set
            {
                TypeDescriptor.GetProperties(
                    base.Component)["State"].SetValue(base.Component, value);
            }
        }
    }
}
```

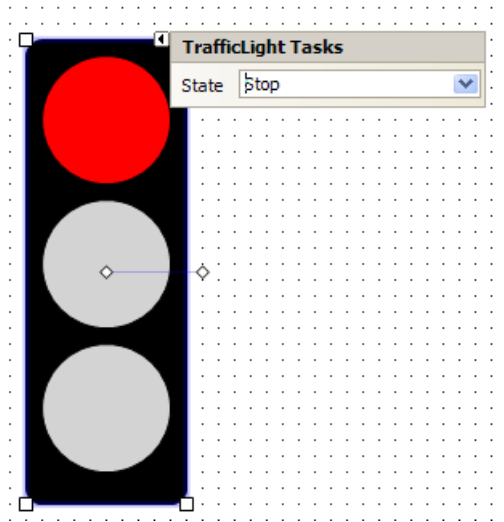
## Improving the Design-Time Behavior of Your Graphic Object

```
    }  
  
    public override DesignerActionItemCollection GetSortedActionItems()  
    {  
        DesignerActionItemCollection col =  
            new DesignerActionItemCollection();  
  
        PropertyDescriptor descr =  
            TypeDescriptor.GetProperties(Component) ["State"];  
        col.Add(new DesignerActionPropertyItem("State", "State",  
            descr.Category, descr.Description));  
  
        return col;  
    }  
}  
}
```

The designer is associated with the graphic object through the **Designer** attribute:

```
[Designer(typeof(TrafficLightDesigner))]  
public partial class TrafficLight : UserSymbol {
```

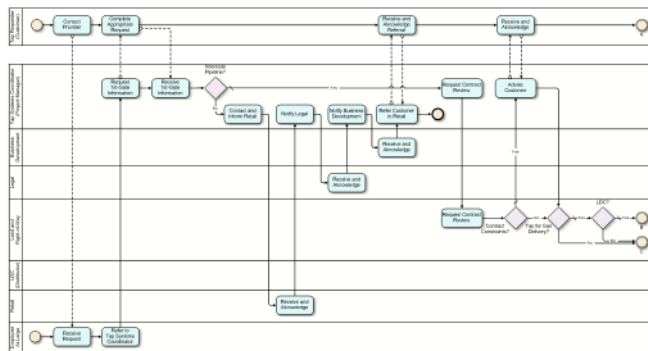
In this example, this allows you to get the following smart tag:



For more information about designers, designer action lists and smart tags, see the .NET Framework® documentation.

# Creating BPMN Diagrams

The Business Process Modeling Notation (BPMN) is a graphical notation that depicts the steps in a business process. Below is an example of a BPMN diagram:



The version 1.1 of the BPMN specification has been adopted by the Object Management Group (OMG), and is widely used by business process modelers. More information about BPMN can be found on the BPMN Web site (<http://www.bpmn.org>) and on the OMG Web site (<http://www.omg.org>).

IBM® ILOG® Diagram for .NET provides out-of-the-box support for creating BPMN diagrams, including:

- ◆ all the BPMN graphic objects available as predefined user symbols,
- ◆ a ready-to-use and customizable BPMN editor,
- ◆ support for both BPMN 1.0 and BPMN 1.1.

#### **In This Section**

##### *The BPMN Symbols*

Provides a short overview of the IBM ILOG Diagram for .NET BPMN symbols.

##### *The BPMN Editor*

Briefly introduces the BPMN Editor.

---

## **The BPMN Symbols**

IBM® ILOG® Diagram for .NET contains a library of symbols that implement all the graphic objects defined in the BPMN specification. These BPMN graphic objects are contained in the `ILOG.Diagrammer.BPMN` namespace.

The graphic appearance and also the properties attached to the graphic objects fully conform to the BPMN specification. Additional graphical properties have been added to some objects for a better usability, like the position of text relative to the shape of an object, and so on.

This section does not cover the BPMN symbols in details. Refer to the BPMN specification available on the BPMN Web site (<http://www.bpmn.org>) for a complete description of the BPMN objects and their semantics.

Here is a short overview of the IBM ILOG Diagram for .NET BPMN symbols.

---

### **Common Properties**

Most BPMN objects in IBM ILOG Diagram for .NET have a **BPMN\_Name** property that defines the text displayed in or around the object (the **BPMN\_** prefix is to differentiate from the design-time **Name** property).

BPMN objects also have an **Id** property that defines a unique identifier for the object.

Finally, most objects have a **Fill** property that defines the fill color, a **Border** property that defines the border color, and a **TextColor** property that defines the text color.

---

## Tasks

Task objects represent the basic work unit in a process. Tasks can have markers as defined by the BPMN specification.



---

## Events

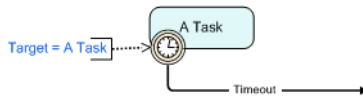
Event objects represent the events that occur during a process. There are three types of events: Start, Intermediate and End events.



Events can have triggers, represented by different icons inside the event shape.



Intermediate events can be attached to the boundary of an activity. This is done by setting the Target property of the event to a reference to a Task object.



---

## Gateways

Gateway objects represent the fork or join points in the process flow. The position of the text relative to the gateway is defined by the TextPosition property.



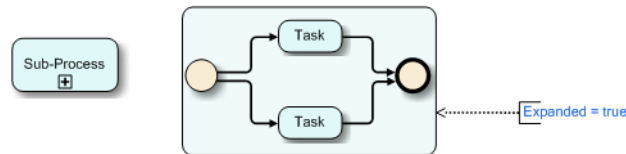
## Flow Objects

Flow objects are the links used to connect other objects in the process. There are two kinds of flow objects: SequenceFlow and MessageFlow.



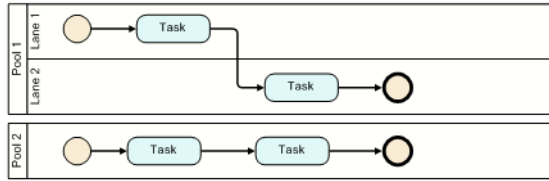
## Subprocesses

Subprocesses are activities that contain other activities. In IBM ILOG Diagram for .NET, embedded subprocesses can be dynamically expanded or collapsed, by setting the boolean Expanded property. When a subprocess is expanded, you can directly edit its content.



## Pools and Lanes

Pool objects are used to group the elements of a process into organization units. Pools can contain Lane objects that are used to further divide the process into smaller units. Lanes can contain sublanes at an arbitrary nesting level. In IBM ILOG Diagram for .NET, pools and lanes are graphic container, therefore you can just drop objects inside a pool or lane, drag the pool/lane to move all its contents.




---

## Artifacts

Artifacts are objects that are not part of the process flow but that define additional information about the process. There are three kinds of artifacts: DataObject, Annotation and Group



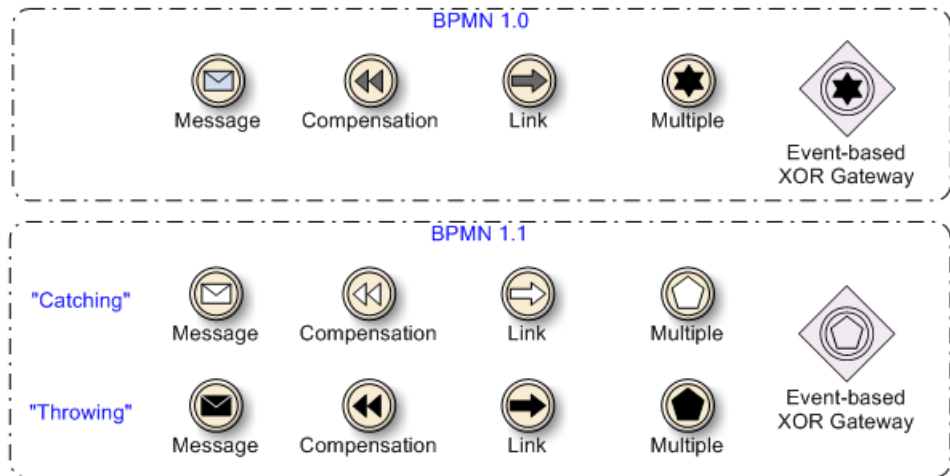

---

## BPMN 1.1 Support

There are a few differences in the graphic appearance of some symbols between the 1.0 and the 1.1 versions of the BPMN specification. IBM ILOG Diagram for .NET supports both versions.

To choose the right BPMN version for a diagram, set the `BPMNVersion` property of the `BusinessProcessDiagram` object, which is the top level container for BPMN diagrams. By default, BPMN 1.0 is used.

The following picture shows some differences between BPMN 1.0 and BPMN 1.1 symbols.



In BPMN 1.1, events can be considered as *catching* or *throwing*. This is reflected by the new `EventMode` property of the `IntermediateEvent` class. A new `Signal` trigger type is also available.

---

## The BPMN Editor

The BPMN editor is contained in the `Samples/Applications/BPMNEditor` subdirectory of the IBM® ILOG® Diagram for .NET installation directory.

The BPMN editor is a specialized version of the Diagram Editor application, where the toolbox is populated with the BPMN graphic objects. The source code of the editor is provided, and therefore you can easily customize it and integrate it in your own application.

All the standard diagram editing tools and commands are available: drag-and-drop from the toolbox, property grid and formatting commands.

Some examples of BPMN diagrams are available in the `Example` directory. Use the `File>Open` command to load them.



# ***Localizing an IBM ILOG Diagram for .NET Application***

IBM® ILOG® Diagram for .NET is internationalized. All messages, resources and dialog boxes of IBM ILOG Diagram for .NET are localized for English language. If you need to localize for another language, IBM ILOG Diagram for .NET provides the resource files and tools that will allow you to localize the library for a particular culture. The library uses the culture information that you have specified in the Control Panel to display dates and numbers.

In order to create a localized version of IBM ILOG Diagram for .NET you must create assemblies (DLLs) that contain the culture-dependant resources of the library. Those assemblies are called *satellite assemblies*. IBM ILOG Diagram for .NET provides the tools that will help you create satellite assemblies for a particular culture.

## **In This Section**

### *Creating a Localization Project*

Explains how to use the localization tool.

### *Translating the Resource Files*

Describes the different types of resource files and explains how to translate them.

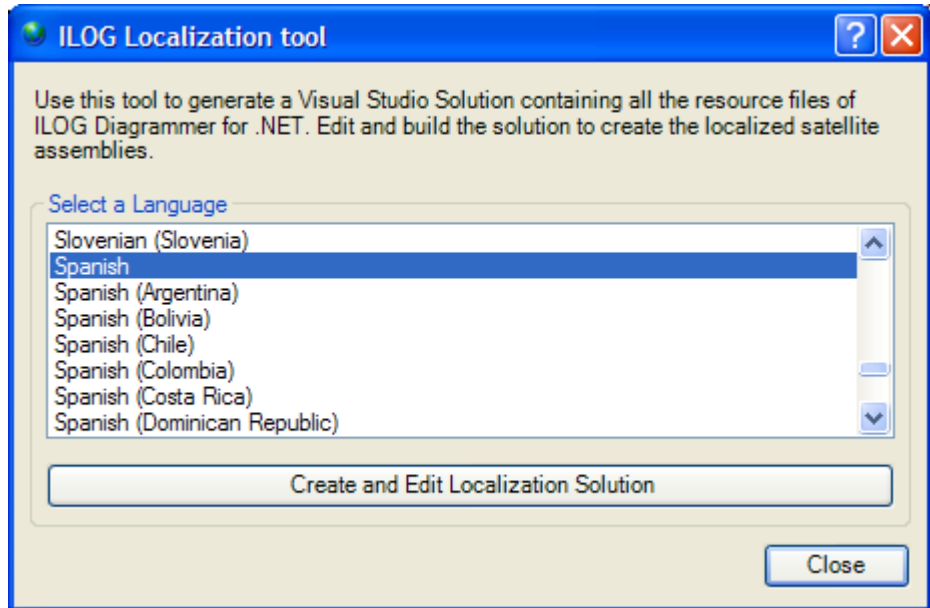
### *Creating the Satellite Assemblies*

Explains how to create the satellite assemblies.

## Creating a Localization Project

The IBM® ILOG® Diagram for .NET localization tool allows you to start localizing the library for a particular culture. The `LocalizationTool.exe` file can be found in the directory `<install-dir>\bin`.

Launch the `LocalizationTool.exe` and choose a language for the localization.



When you press the button `Create and Edit Localization Solution`, the tool opens a new Visual Studio solution.

The tool creates a new solution for building the satellite assemblies and creates also a new directory with a copy of all the resource files that you may need to localize. This new directory is called: `<install-dir>\localization\resources-<culture name>`.

For example, if you choose to localize for Spanish, a new directory named `resources-es` will be created. This new directory contains a copy of all the resource files of IBM ILOG Diagram for .NET and a solution named `Localization-es.sln`.

The solution created by the Localization tool contains a project for each of the IBM ILOG Diagram for .NET satellite assemblies. Each project contains the resource files (`.resx` files) that need to be translated for the culture you have chosen.

---

## Translating the Resource Files

The projects contain two types of resource files:

- ◆ resource files that contain global resource for the library. Those files are named for example `Resources.resx`.
- ◆ resource files that correspond to a predefined dialog box of IBM® ILOG® Diagram for .NET. For example, the `FilterDialog.resx` file corresponds to the FilterDialog dialog box.

You can use Visual Studio to open the resource files and do the translation, but you can also use the .NET Framework tool `winres.exe` to do the translation of the dialog box resources. Refer to the .NET Framework documentation for more information about `winres.exe`.

---

## Creating the Satellite Assemblies

When you have modified the resource files for your culture, you can build the solution created by the localization tool. When you build the solution, you create the satellite assemblies in the `<install-dir>\bin<culture name>` directory. It is important to note that these assemblies are not fully operational. They will not be recognized as satellite assemblies of IBM® ILOG® Diagram for .NET until they are signed with the IBM ILOG Diagram for .NET private key. In order to sign those assemblies you must send them to the technical support where they will be signed with the IBM ILOG Diagram for .NET private key. You can test your satellite assemblies before they are signed by registering the assemblies for "Verification skipping" using the .NET Framework tool named `SN.exe`.

In a Visual Studio command prompt do:

```
sn -Vr <assembly>
```

for each satellite assembly.



# Index

## A

- AJAX web application
  - creating a new interactor **49**
  - diagram view
    - displaying a diagram **38**
    - using predefined interaction tools **41**
  - graphic object creation interactor **46**
  - overview of the ILOG Diagram for .NET framework **38**
  - pan interactor **46**
  - selection interactor **42**
  - setting an interactor **41**
  - tiling mode **40**
  - zoom interactor **45**
- animation
  - controlling execution **294**
  - customizing **300**
  - graphic object along a path **299**
  - grouping **297**
  - overview **294**
  - property **295**
  - types **299**
  - using as a timer **298**

## B

- basic shapes
  - Basic2DShape **64**
  - geometry defined by a rectangle **62**
  - geometry defined by a set of points **63**

- path object **64**
- basic web application
  - diagram view
    - controlling the image generation **32**
    - controlling the zoom level **30**
    - displaying diagrams **28**
    - handling user input event **35**
    - showing the view content **32**
- BPMN **329**
  - editor **334**
  - symbols **330**
    - artifacts **333**
    - common properties **330**
    - events **331**
    - flow objects **332**
    - gateways **331**
    - pools and lanes **332**
    - subprocesses **332**
    - tasks **331**

## C

- controls
  - basic **84**
  - multiple content **84**
  - single content **84**
  - the Control class **82**
- coordinate systems
  - container **163**
  - conversion **165**

- geometry **162**
- overview **161**
- view **164**

creating

- graph **170**
- nodes **168**

## D

- deserializing **270**
- Design View **237**
- design-time behavior
  - improving **325**
- diagram
  - deserializing **271**
  - serializing **270**
- Diagram Designer **237**
  - commands **261**
    - editing **266**
    - grid and alignment **261**
    - group **266**
    - miscellaneous **268**
    - nudge **267**
    - object creation **264**
    - order **266**
    - path **263**
    - printing **265**
    - text **263**
    - zoom **261**
  - graphic objects
    - adding to diagram or user symbol **241**
    - aligning **245**
    - changing properties **248**
    - containers **253**
    - creating path objects **257**
    - cut, copy, paste **259**
    - drawing order **255**
    - graph layout **259**
    - links and anchors **259**
    - moving **245**
    - panels **253**
    - resizing **245**
    - rotating **245**
    - selecting **244**
    - setting text **252**

- showing/hiding **256**
- importing vector graphics **260**
- printing **260**

displaying diagram

- AJAX web application **37**
- basic web application **28**

## E

events

- capture **189**
- dispatching in a diagram view **181**
- dispatching to graphic objects **184**
- listening **155**
- raising **153**
- stopping propagation **188**

## G

- graphic containers
  - introducing **157**
  - using predefined **159**
- graphic objects
  - basic shape **60**
  - control **82**
  - displaying text **137**
  - editing fill objects **128**
  - filling and stroking **120**
  - filter
    - applying **129**
    - editing effects **134**
  - gauge **107**
  - graphic symbol **100**
  - image **69**
  - link object **71**
  - panel **85**
  - path **64**
  - preferred size **149**
  - scale object **102**
  - sub-diagram **97**
  - text object **70**
  - user symbol **110**
- grid
  - setting the geometry **22**
  - setting the style **21, 24**

snap coordinates **23**

## I

### interactor

- anchor editing **204**
- creating **204**
- link creation **201**
- pan **197**
- polypoints shape creation **199**
- rectangular shapes creation **197**
- rotate **196**
- selection **193**
- setting on a view **193**
- zoom **196**

## L

### link objects

- adding text **78**
- crossing **81**
- customizing arrows **76**
- customizing link appearance **77**
- free **75**
- oblique **74**
- specifying the connection points **72**
- specifying the link shape **73**

### link shape

- orthogonal **73**
- straight **73**

### localization

- creating a project **336**
- creating the satellite assemblies **337**
- translating the resource files **337**

## P

### painting

- interior of a shape **120**
- outline of a shape **126**
- with a linear gradient **121**
- with a path gradient **123**
- with a solid color **120**
- with a texture **124**
- with hatches **125**

### panels

#### predefined

- Canvas **89**
- DockPanel **95**
- GridPanel **93**
- StackPanel **90**

#### the Panel class **85**

- using predefined **89**

### predefined selection graphic objects

#### behavior

- common **226**
- editing path **229**
- editing polypoints **228**
- resizing **227**

### printing

- customizing **306**
- graphic container **304**
- on several pages **305**
- predefined dialog boxes **309**
- previewing **314**
- setting up a document **303**
- specifying the area **304**

## R

### rendering

- clipping **116**
- image filter **114**
- opacity **114**
- text **115**
- text appearance **115**
- transform **116**
- visibility **113**

## S

### satellite assemblies **335**

### scale objects

- circular **106**
- linear **106**
- the ScaleBase class **102**

### selection

- creating custom graphic objects **231**
- listening to events **219**
- managing selected objects **218**

- styling graphic objects **231**
- the SelectionGraphic class **221**
- using predefined graphic objects **225**

serializing **270**

SVG **317**

- creating a custom graphic object **320**
- file example **318**
- reading a file **319**
- supported features **320**

## T

text

- alignment **79**
- appearance **81**
- displaying in a graphic object **137**
- intersections **81**
- position **79**
- rotation **80**
- using Text object **141**
- using TextOnPath object **141**

## U

Using the Localization Tool **336**

## V

visibility

- of a graphic object **145**

## W

web application

- diagram view
- generating the image map **32**

WinForms

- creating a new interactor **204**
- dispatching events in a diagram view **181**
- using predefined interaction tools **192**

winforms application

- diagram view
- controlling the zoom level **17**
- displaying a grid **21, 23**
- displaying diagrams **16**

- modifying appearance **20**
- scrolling **19**
- showing view content **20**
- using predefined behavior **25**

## X

XML

- document **278**
- element **278**
- instance **278**
- manager **278**
- serialization
- customizing **277**
- mechanism **271**