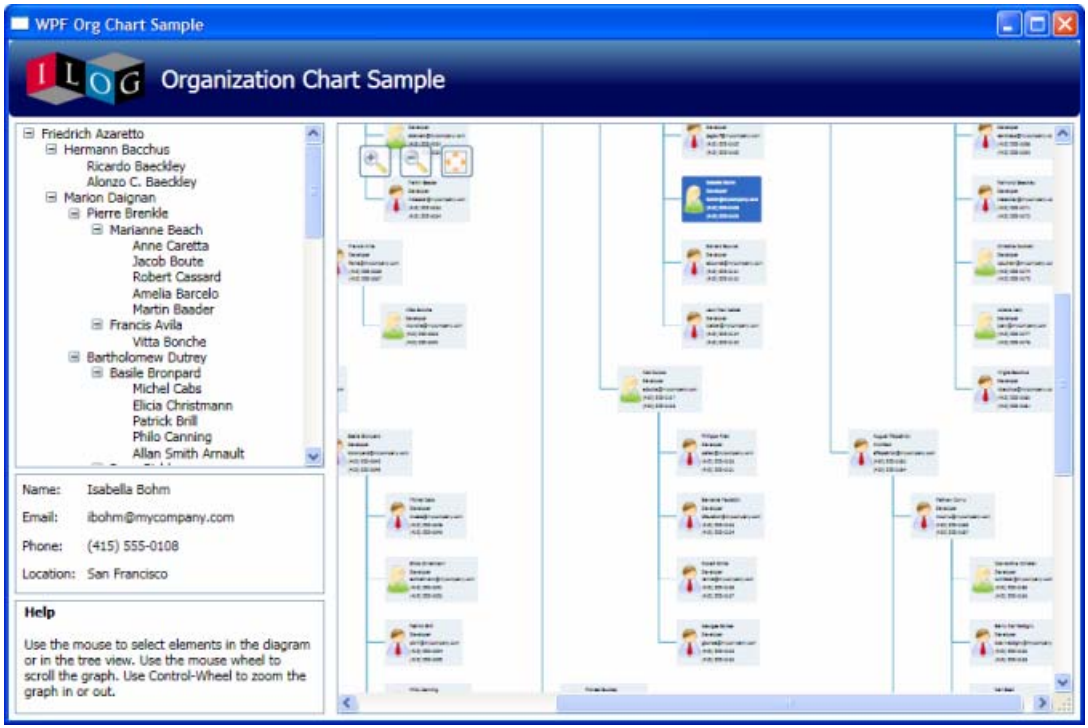
IBM ILOG Diagram for .NET V2.0 Programming with IBM ILOG Diagram for .NET and Windows Presentation Foundation

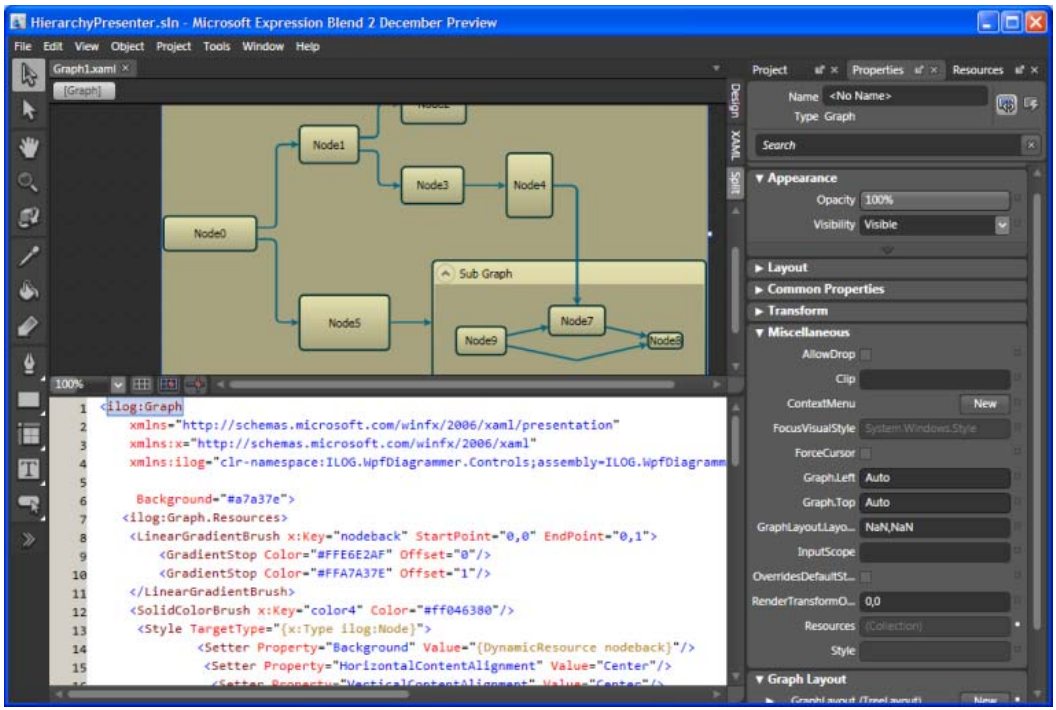
June 2009

Contents

Programming with IBM ILOG Diagram for .NET and Windows Presentation Foundation

IBM® ILOG® Diagram for .NET offers the ability to integrate graph displays inside your Window Presentation Foundation (WPF) application. IBM ILOG Diagram for .NET brings a dedicated set of WPF elements that ease the creation of graph representations. With these classes you may create your graph directly from a data source, by specifying the node and links that compose your graph in XAML (the XML language of WPF) or directly by code. Thanks to the power of WPF, it is possible to create very appealing graph representations through the styling and templating features of WPF combined with the power of the Graph Layout library of IBM ILOG Diagram for .NET.


WPF Org Chart Sample
Organization Chart Sample
Friedrich Azaretto
Hermann Bacchus
Ricardo Baeckley
Alonzo C. Baeckley
Marion Daignan
Pierre Brenkle
Marianne Beach
Anne Caretta
Jacob Boute
Robert Cassard
Amelia Barcelo
Martin Baader
Francis Avila
Vitta Bonche
Bartholomew Dutrey
Basile Bronpard
Michel Cabs
Elicia Christmann
Patrick Brill
Philo Canning
Allan Smith Arnault
Name: Isabella Bohm
Email: ibohm@mycompany.com
Phone: (415) 555-0108
Location: San Francisco
Help
Use the mouse to select elements in the diagram or in the tree view. Use the mouse wheel to scroll the graph. Use Control-Wheel to zoom the graph in or out.



Note that specific WPF classes require to use .NET 3.0 or .NET 3.5 and they are located in the DLL named **ILOG.Controls.Diagram**. The namespace for those classes is ILOG.Controls.Diagram.

In This Section

Overview

Provides a short overview of the main WPF controls and classes.

Creating a Graph Display from a Data Source Using the Diagram Control

Describes how to create a graph representation in WPF by using the Diagram class.

Creating a Graph by Specifying Nodes and Links

Explains how to create a graph through instances of the Node and Link classes and use them directly inside any WPF container.

Specifying Graph Layout

Briefly describes how to use the Graph Layout classes.

Overview

IBM® ILOG® Diagram for .NET defines a set of predefined WPF controls, elements and classes dedicated for displaying and interacting with graphs in the WPF platform. Here is an overview of the main classes and controls:

- The Diagram control

  The **Diagram** control allows you to present the information as a graph display while leveraging the most powerful and flexible WPF features available. The **Diagram** control can be populated in two ways: by using a data source or by adding nodes and links by code to create a graph. The control comes with a new variety of WPF data sources for your graph data. These include the **XmlDataProvider** and **ObjectDataProvider**, as well as more traditional data sources such as **DataSet**, **Array**, and **Collection** objects. The **Diagram** control provides visual designers with many advanced styling and layout capabilities and support for scrolling and zooming. The graph created from the data source can be automatically arranged by a large variety of graph layout algorithms that you can specify and tune both in XAML or by code. The **Diagram** control allows you to customize nodes and links in many different ways, through the styling and templating features it enables the creation of very aesthetic user interfaces by using the Windows Presentation Foundation.

- The Graph control

  The **Graph** control is used to display graph representations. The **Diagram** control uses a **Graph** internally to display the graph. Unlike the **Diagram** class, the **Graph** control is not bound to a data source. Instead, you can specify the graph data by adding nodes and

links (instances of the Node and Link classes) as the children of the **Graph** control. Elements in a **Graph** can be positioned through the Graph.Left and Graph.Top attached properties, but node and links can optionally be arranged automatically by a variety of graph layout algorithms. The **Graph** control does not support zooming or scrolling, so if you want to create a graph that supports zooming and scrolling, you should use a **Diagram** instead of a **Graph**. One use case of the **Graph** control is to create sub-graphs, that is, nodes that contain another graph.

- The Node and Link classes

  The **Node** and **Link** classes are used to compose a graph. The **Diagram** control generates instances of the **Node** and **Link** classes from its data source. You can also add nodes and links manually using the collections contained in the **Nodes** and **Links** properties of the **Diagram**. When you need to style a graph displayed by a **Diagram**, you define a specific style for the **Node** and **Link** classes. You can use the **Node** and **Link** elements in various places. When **Node** and **Link** instances are inside a **Graph** control, the **Graph** control is responsible for arranging the graph through graph layout algorithms, like a WPF **Grid** control arranges its children as a grid. **Node** and **Links** can also be used in any WPF containers. For example, you may use the **Node** and **Link** elements inside a WPF **Canvas**. Inside a **Canvas** or any WPF container the links will stay connected to the nodes when the nodes change location or size. Note that the **Node** class is a subclass of the WPF **ContentControl** class and therefore you can place any WPF element inside a **Node**. For example, it is possible to place a **Graph** control inside a **Node** to create the notion of a subgraph.

- The Graph Layout classes

  Graph Layout classes are also present in the ILOG.Controls.Diagram namespace so that you can specify both by code or in XAML how your graph should be arranged. The classes are providing the same algorithms as the classes provided for the Windows Forms and ASP.NET classes of IBM ILOG Diagram for .NET but are implemented as WPF dependency objects. See *Using Graph Layout Algorithms* to learn about the various options available for Graph Layout algorithms, such as Tree Layout or Hierarchical Layout algorithms. Note that some of the advanced functionalities of the Graph Layout library are not reported in WPF.

Creating a Graph Display from a Data Source Using the Diagram Control

An easy way to create a graph representation in WPF is to use the Diagram class. The **Diagram** class is a WPF control that can automatically create a graph composed of nodes and links from any source of data. If you are familiar with WPF then you must have already used a WPF ListBox bound to a data source. A WPF ListBox bound to a data source will automatically be filled with items that are instances of ListBoxItem. Then, it is possible to provide a data template to specify how a ListBoxItem is rendered depending on the data. For the **Diagram** class, the concept is very similar: once the diagram is bound to a data source, the **Diagram** control is automatically filled with Node objects that represent the nodes of the graph. For each item in the data source a **Node** instance is created to represent the item. Of course, it is also necessary to specify the relations between items in the data source so that links between the nodes (instance of the Link class) can be created and rendered. The relations between items will also be expressed using the WPF binding system.

In This Section

A First Example Using the Diagram Control

Provides an example that shows how to use the Diagram control.

Specifying the Data Sources of a Diagram

Explains how to specify the data sources of a diagram.

Making the Diagram Reflect the Changes in my Data Source

Explains how to make the diagram reflect the changes in a data source.

Populating a Diagram Using the Nodes and Links Collections

Explains how to populate a diagram using nodes and links collections.

Styling Nodes and Links in a Diagram

Describes how to customize the representation of the nodes and links to your particular needs.

Selection in a Diagram

Explains how to select nodes in a diagram.

Zoom in a Diagram

Explains how to zoom in and out the graph displayed.

Specifying Graph Layout Algorithms in a Diagram

Describes the graph layout algorithms in a diagram.

A First Example Using the Diagram Control

This basic example uses an XML data source. In this example, you will render a graph from data stored in XML in the `data.xml` file. Let's assume that the XML data represents the organization of a company as a hierarchy of employees. The XML looks like this:

```
<employee name="employee 1">
  <employee name="employee 2"/>
  <employee name="employee 3">
    <employee name="employee 5"/>
    <employee name="employee 6"/>
  </employee>
  <employee name="employee 7"/>
</employee>
```

Employee 1 manages employees 2, 3 and 7; employee 3 manages employees 5 and 6.

The following extract of XAML shows a Diagram object bound to this XML data source.

```
<Window x:Class="Sample"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:ilog="http://www.ibm.com/diagrammer.net/2008"
    Title="WPF Diagram Example" Height="300" Width="300">
    <Window.Resources>
        <XmlDataProvider x:Key="xmldata"  Source="data.xml"/>
    </Window.Resources>
    <Grid>
        <ilog:Diagram
              NodesSource="{Binding Source={StaticResource xmldata},
XPath=//employee}"
          SuccessorsBinding="{Binding XPath=employee}"
```

```
            DisplayMemberBinding="{Binding XPath=@name}"/>
    </Grid>
</Window>
```

As shown in this code extract, the first thing to do is to declare the XML namespace for the WPF controls of IBM® ILOG® Diagram for .NET. The following line declares a new XML namespace named `ilog` that will be used later with the IBM ILOG Diagram for .NET WPF classes:

```
xmlns:ilog="http://www.ibm.com/diagrammer.net/2008"
```

The following lines declare an `XmlDataProvider` (stored into the resources of the Window) that reads the `data.xml` file.

```
<Window.Resources>
    <XmlDataProvider x:Key="xmldata" Source="data.xml"/>
</Window.Resources>
```

The last part of the code extract declares the **Diagram** control:

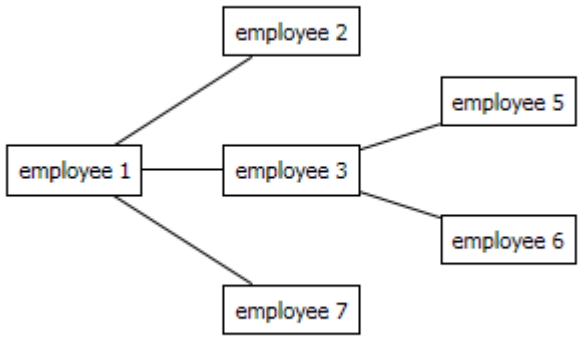
```
<ilog:Diagram
  NodesSource="{Binding Source={StaticResource xmldata}, XPath=//employee}"
    SuccessorsBinding="{Binding XPath=employee}"
    DisplayMemberBinding="{Binding XPath=@name}"/>
```

The NodesSource property should be a collection of items that will be represented as nodes in your graph. Here, the **NodesSource** property refers to the `XmlDataProvider` through a WPF binding declaration with an `XPath` specified as '`//employee`'. The `XPath` specification indicates that the nodes in the graph will be all the XML elements with the tag name `'employee'` in the document whatever the level of the element in the XML document is.

The SuccessorsBinding property is used to indicate the relations (the links) between the nodes. There are several ways to indicate the relations between the nodes; the **SuccessorsBinding** property represents a binding from one item in the data source to the collection of items that are successors of this item. Here, the **SuccessorsBinding** specifies a binding with an XPath of '`employee`'. This tells the diagram that the successors of an employee should be the children XML elements with tag name '`employee`'. For more information on how to specify the relations between nodes, see *Specifying the Data Sources of a Diagram*.

The DisplayMemberBinding specifies what you want to display inside a node. Here, it specifies that you want to display the '`name`' attribute of the XML element. The **DisplayMemberBinding** specifies a simple representation in the nodes of the graph. For a more complex representation you would use styling and templating as described in *Styling Nodes and Links in a Diagram.*

This XAML declaration will give the following result:



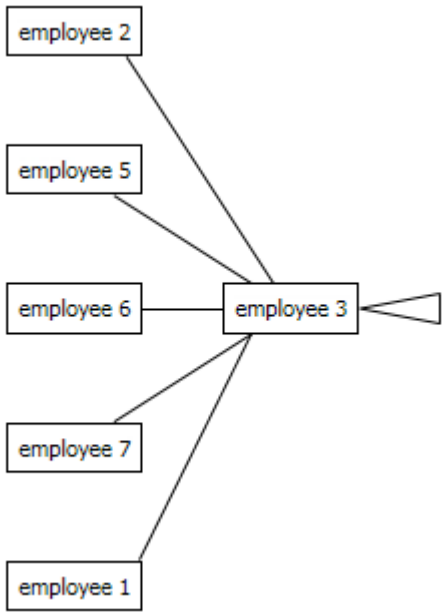
By default, the **Diagram** control will automatically arrange the resulting graph with a Hierarchical Layout algorithm. To change the layout algorithm use the GraphLayout properties and choose between the various algorithms such as TreeLayout, HierarchicalLayout, ForceDirectedLayout and more. Each algorithm has many options that allow you to tailor the placement of nodes and the shape and connection points of the links.

By slightly modifying the **SuccessorsBinding** property you can get different graphs. Here are some examples:

The following binding specifies that each employee has for successors the employee named '`employee 3`':

```
SuccessorsBinding="{Binding XPath=//employee[@name\='employee 3\']}"
```

With this binding, you can get the following graph:

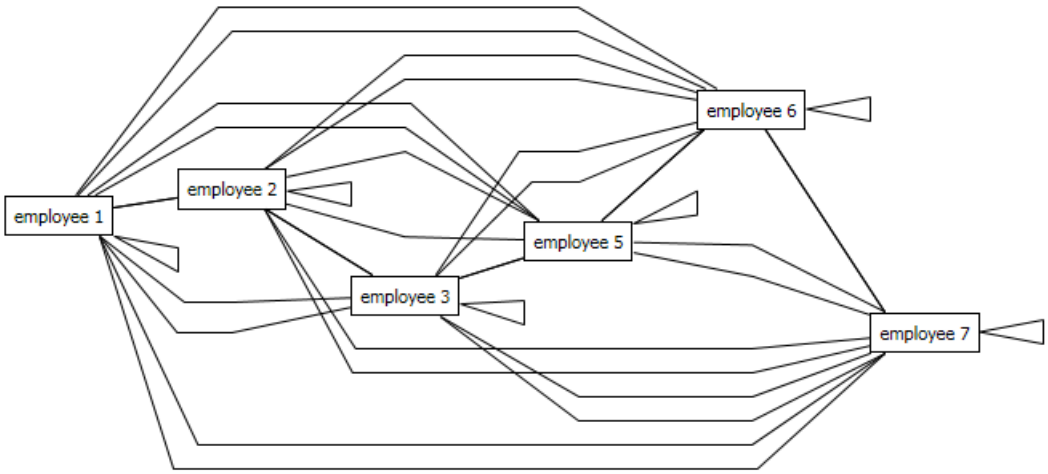


You have noted the 'self link' from ‘`employee 3`’ to itself.

The following **SuccessorsBinding** specifies that each employee has for successors the list of all employees:

```
SuccessorsBinding="{Binding XPath=//employee}"
```

With this binding, you can get a much more complex graph:



To learn all the possibilities about data binding to graph data, see *Specifying the Data Sources of a Diagram*.

Specifying the Data Sources of a Diagram

Within a Diagram control, the data source that represents all the nodes of the graph can be specified through the NodesSource property of the **Diagram** class. This property is of type **IEnumerable** so anything that can be enumerated can be used as the source of data, in particular the data source provider classes that are part of WPF: **ObjectDataProvider** and **XMLDataProvider** that will allow you to do things like sorting or filtering but also simple arrays or collections of objects.

The **NodesSource** property is not sufficient to represent graph data, it is also necessary to specify the relations between the items in the data source.You can do this by using bindings that define implicit relationships between nodes, or using a second data source, LinksSource, which contains explicit relationships between nodes. Note that these two ways of defining links are not exclusive; you can use both in the same diagram.

In This Section

Defining Links Using Implicit Relations

Describes the properties that allow you to specify the relations between the items.

Defining Links Using Explicit Relations

Explains how to specify explicit relations between nodes.

Defining Links Using Implicit Relations

The **Diagram** class contains three properties that allow you to specify the relations between the items. You only need to use one of these three properties and choose the one that best corresponds to the way your data source is organized. The three binding properties are SuccessorsBinding, PredecessorsBinding and ParentBinding.

The **SuccessorsBinding** property allows you to specify a WPF binding from an item in the data source to its "successors". The source of this binding is one item in the data source and the result must be the list of the successors of this item. Each successor should also be part of the data source that you have specified through the **NodesSource** property. Through this binding specification you notify the **Diagram** that there is a relation between an item in the data source and other items (successors) in the data source. By knowing these relations, the **Diagram** can render directed links from an item to its successors. For each relation a Link instance is created.

The **PredecessorsBinding** is similar to the **SuccessorsBinding**. The source of the binding is also an item in the data source and the result must be a list of items (also in the data source) that represents the predecessors of the item. By knowing these relations, the **Diagram** will generate a directed link from each predecessor to its successor. Again, a **Link** instance will be created for each relation.

The **ParentBinding** is similar to the **PredecessorsBinding**. The source of the binding is an item in the data source but the result is another item of the data source that represents the parent of the item. Only one single link will be created from the parent to the child. Therefore, the **ParentBinding** is equivalent to a **PredecessorsBinding** with a single predecessor. The **ParentBinding** is used to simplify the binding when each item has only a single parent item.

For example, suppose that you need to represent an organization chart. The data source will be a list of employees represented by the `Employee` class.

The `Employee` class provides a property that represents the manager of this employee and a property that represents the list of employees managed by this employee. We will obtain a C# class with the following skeleton:

```
public class Employee {
  public Employee Manager {get; set;}
  public List<Employee> ResponsibleFor {get;}
}
```

If your data source is a list of all the employees of the company, you can specify the relation in two different ways:

In XAML (assuming that `allEmployees` is a resource defined earlier in the XAML representing a collection of all the employees):

```
<ilog:Diagram
  NodesSource="{Binding Source={StaticResource allEmployees}}"
```

```
  SuccessorsBinding="{Binding ResponsibleFor}"
/>
```

or

```
<ilog:Diagram
  NodesSource="{Binding Source={StaticResource allEmployees}}"
  ParentBinding="{Binding Manager}"
/>
```

In C#:

```
List<Employee> allEmployees = …
Diagram diagram = new Diagram();
diagram.NodesSource = allEmployees;
diagram.SuccessorsBinding = new Binding("ResponsibleFor");
```

or

```
List<Employee> allEmployees = …
Diagram diagram = new Diagram();
diagram.NodesSource = allEmployees;
diagram.ParentBinding = new Binding("Manager");
```

> ***Note:*** *The resulting type of the binding specified in the* ***SuccessorsBinding*** *or* ***PredecessorsBinding*** *properties must be* ***IEnumerable*** *which is the case in the example, where the* `ResponsibleFor` *property is defined with the type* `List<Employee>`.

Through the **PredecessorsBinding** and **SuccessorsBinding** properties you can create any type of graph. There is no constraint in the successors or predecessors of an item, except that they must be part of the data source. For example, the same item may appear several times in the successors or predecessors of an item. An item can be the successor of himself and this will generate a 'self link' (a link from the item to itself). You can easily create cyclic graphs or a graph composed of several disconnected parts. As you are specifying only one relation from an item to its parent item, the **ParentBinding** property is more suitable for defining a simple hierarchy that can be displayed as a tree.

Note that the example is rather simple and this is mainly because our `Employee` class provides properties that are easy to bind to. It may happen that the relations between items are more complex and require some computation. In this case it is still possible to use a binding and the **Converter** property of the binding. By specifying a converter in the binding you will be able to code more complex relations between items of the data source.

When using a converter in the binding, the value to convert is the item in the data source from which you want to find the successors or predecessors and the result should be the collection (**IEnumerable**) of the successors or predecessors of this item. An example of this technique can be found in **Samples/Applications/WPFCriticalPath**.

Defining Links Using Explicit Relations

Instead of (or in addition to) defining implicit relations between nodes using the **SuccessorsBinding**, **PredecessorsBinding** or **ParentBinding** properties, you can use the **LinksSource** property to specify a collection of explicit relations between nodes.

Like the **NodesSource** property, the **LinksSource** property must contain an **IEnumerable** object. The items in the **LinksSource** are enumerated, and each item will be represented by a **Link** in the **Diagram**.

To tell the **Diagram** which nodes a link will be connected to, you need to specify two more bindings through the StartBinding and EndBinding properties. These properties specify the items in the **NodesSource** that represent the start and end of the link.

For example, look at the XML data source example in *A First Example Using the Diagram Control.* Sometimes, you need to define "dotted-line" relationships between employees (for example, an employee may have a direct manager and another supervisor). Such relations can be represented by additional elements of tag `dottedline` in the XML data file:

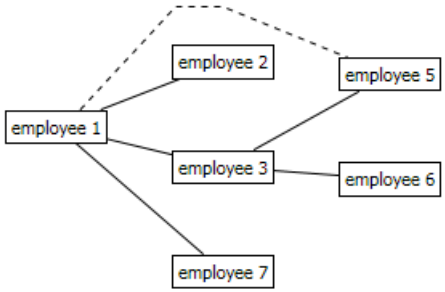
```
<employee name="employee 1">
  <employee name="employee 2"/>
  <employee name="employee 3">
    <employee name="employee 5"/>
    <employee name="employee 6"/>
  </employee>
  <employee name="employee 7"/>
  <dottedline dotted="true" from="employee 1" to="employee 5"/>
</employee>
```

To display the new dotted-line links, you must modify the XAML as follows:

```
<ilog:Diagram
          NodesSource="{Binding Source={StaticResource xmldata},
XPath=//employee}"
          LinksSource="{Binding Source={StaticResource xmldata},
XPath=//dottedline}"
          SuccessorsBinding="{Binding XPath=employee}"
          StartBinding="{Binding XPath=@from}"
          EndBinding="{Binding XPath=@to}"
          NodeNameBinding="{Binding XPath=@name}"
          DisplayMemberBinding="{Binding XPath='@name'}">
</ilog:Diagram>
```

We added three declarations: **StartBinding** and **EndBinding** specify the names of the start and end nodes, while **NodeNameBinding** specifies that the names correspond to the **name** attribute of each node.

The resulting graph looks like this:



You also added the following style to make the new link appear as a dotted line:

```
<Window.Resources>
      ...
      <Style TargetType="ilog:Link">
          <Style.Triggers>
              <DataTrigger Binding="{Binding XPath='@dotted'}" Value="true">
                  <Setter Property="StrokeDashArray" Value="4,4" />
              </DataTrigger>
          </Style.Triggers>
      </Style>
 </Window.Resources>
```

Instead of specifying node names as in this example, the **StartBinding** and **EndBinding** properties can also specify the start or end nodes directly (that is, the bindings can specify properties whose value are the data items that correspond to the start/end nodes). In that case, it is not necessary to specify the **NodeNameBinding**.

Making the Diagram Reflect the Changes in my Data Source

You can create a graph over any collection of items that implements **IEnumerable**. However, to set up dynamic bindings so that insertions or deletions in the collection of nodes or in the predecessors or successors collection update the graph automatically, the collections must implement the **INotifyCollectionChanged** interface. This interface exposes the **CollectionChanged** event that is raised whenever the underlying collection changes.

WPF provides the **ObservableCollection< T>** class, which is a built-in implementation of a data collection that exposes the **INotifyCollectionChanged** interface.

When using the ParentBinding property, the modification of the property that represents the parent must be notified to the Diagram. This must be done by implementing the **INotifyPropertyChanged** interface of the .NET platform.

If we need dynamic binding in the above example, we would implement the `Employee` class as follow:

```
public class Employee : INotifyPropertyChange {
  public Employee Manager {get; set;}
  public ObservableCollection<Employee> ResponsibleFor {get;}
   …
}
```

A full example using dynamic binding can be found in **QuickStart/WPFGraphDynamicBinding**. This example shows also how to use the **ListCollectionView** class of the WPF platform to enable filtering in the graph.

Populating a Diagram Using the Nodes and Links Collections

Instead of using the **NodesSource** and/or **LinksSource** properties, you can add nodes and links directly to a **Diagram** using the **Nodes** and **Links** properties. The initial values of these properties are empty collections. You can add either **Node** or **Link** objects that will be added directly to the underlying **Graph**, or you can add data items that will be translated to **Node** or **Link** objects, using the same process and properties as explained in *Specifying the Data Sources of a Diagram.*

Note that, even if you specified a data source using the **NodesSource** and/or **LinksSource** properties, you can also use the **Nodes** and **Links** properties to add more nodes or links to the **Diagram**, provided that the specified data source(s) are collections and are not read-only.

Styling Nodes and Links in a Diagram

The Diagram component generates instances of the Node class to represent the items in the data source and instances of the Link class to represent the relations between items. By using the WPF styling and templating features, it is possible to tailor the representation of the nodes and links to your particular needs. This section introduces the various ways to style the nodes and the links. For more information on WPF styling and templating features see the WPF documentation.

In This Section

Styling Nodes and Links Using Basic Style Features

Explains how to use basic features to style nodes and links.

Styling Nodes with Data Templates and Data Template Selectors

Explains how to use data templates to style nodes.

Changing the Control Template of the Node Class

Describes how to change the Control template of the Node class.

Creating a Style of Links that Depends on the Data

Explains how to create a style of links that depends on the data.

Styling Nodes and Links Using Basic Style Features

In a **Diagram** control, you style the nodes of your diagram by using the styling and templating features of WPF. In this section, you will find some techniques that can be applied to the nodes of a diagram.

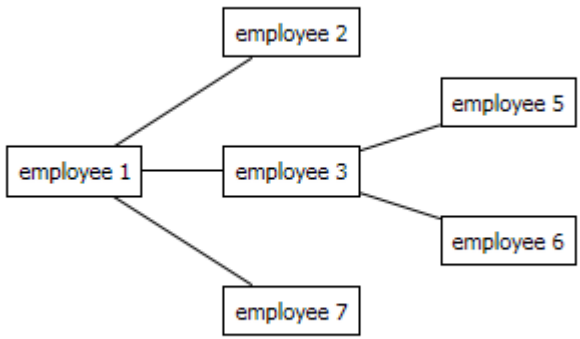
Imagine that you want to display the same XML file that represents a hierarchy of employee:

```
<employee name="employee 1">
  <employee name="employee 2"/>
  <employee name="employee 3">
    <employee name="employee 5"/>
    <employee name="employee 6"/>
  </employee>
  <employee name="employee 7"/>
</employee>
```

The **Diagram** control is defined with following markup:

```
<ilog:Diagram
      NodesSource="{Binding Source={StaticResource xmldata}, XPath=//employee}"
     SuccessorsBinding="{Binding XPath=employee}"
     DisplayMemberBinding="{Binding XPath=@name}"/>
```

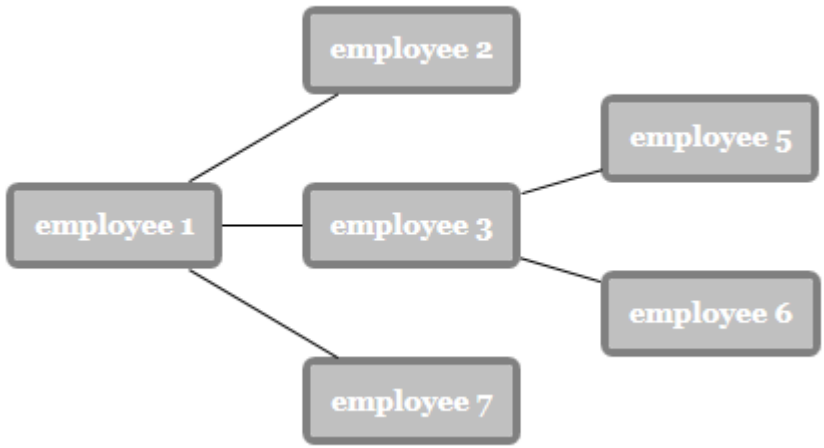
Without any styling the diagram appears like this:



The first thing to do is to define a WPF Style that will apply to all the nodes. Here is a style definition that specifies some properties of the **Node** class:

```
<Window.Resources>
...
<!-A style that affects all nodes-->
<Style TargetType="ilog:Node">
    <Setter Property="Padding" Value="10"/>
    <Setter Property="FontFamily" Value="Georgia"/>
    <Setter Property="FontSize" Value="14"/>
    <Setter Property="FontWeight" Value="bold"/>
    <Setter Property="Background" Value="Silver"/>
    <Setter Property="Foreground" Value="White"/>
    <Setter Property="BorderBrush" Value="Gray"/>
    <Setter Property="BorderThickness" Value="4"/>
    <Setter Property="CornerRadius" Value="4"/>
</Style>
…
</Window.Resources>
```

With this style in place the diagram now appears like this:

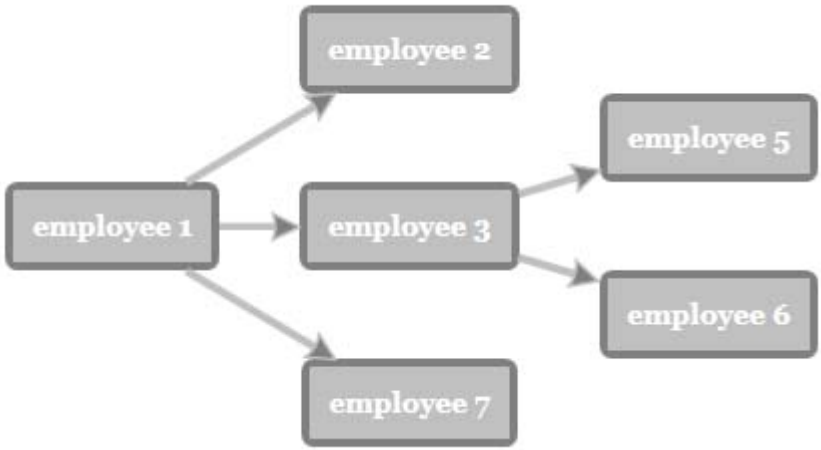


You can also define a style for the **Link** class to change the appearance of all links: the following example defines a specific thickness, a color and an end arrow for all links.

```
<Style TargetType="ilog:Link">
     <Setter Property="StrokeThickness" Value="4"/>
     <Setter Property="Stroke" Value="Silver"/>
       <Setter Property="EndArrow">
         <Setter.Value>
            <ilog:LinkArrow  Shape='curved'  Fill="gray"/>
         </Setter.Value>
       </Setter>
```

```
</Style>
```

The diagram now looks like this:



Note that some properties of the **Link** class that control the connection points and bend points of the links may not be taken into account when the graph is automatically arranged by the graph layout algorithm specified in a **Diagram**. Some graph layout algorithm will completely define the shape and connection points of the links and in this case the properties such as Points, StartAnchorPosition, EndAnchorPosition, StartAnchorOffset, EndAnchorOffset, and ShapeType of the **Link** class will not be taken into account. To change the shape of links or the way they are connected, you will have to change the properties on the Graph Layout instance used to layout the graph. For more information on this, see *Specifying Graph Layout Algorithms in a Diagram*.

Styling Nodes with Data Templates and Data Template Selectors

In a **Diagram** control, you can decide what to display in a **Node** by specifying a WPF **DataTemplate** for the nodes generated by the **Diagram**. The **Diagram** control defines the NodeTemplate property (of type **DataTemplate**) that lets you define the appearance of the nodes and the bindings between the properties of the node and the item in the data source.

Each XML element in the data source has a name attribute that you may want to display in the nodes. The following data template displays the name in a WPF **TextBlock** element with a small ellipse on the left using the WPF **Ellipse** element.

```
<DataTemplate x:Key="nodeTemplate">
      <StackPanel Orientation="Horizontal">
          <Ellipse Fill="red" Width="10" Height="10"/>
          <TextBlock Margin="5,0,0,0" Text="{Binding XPath=@name}"/>
```

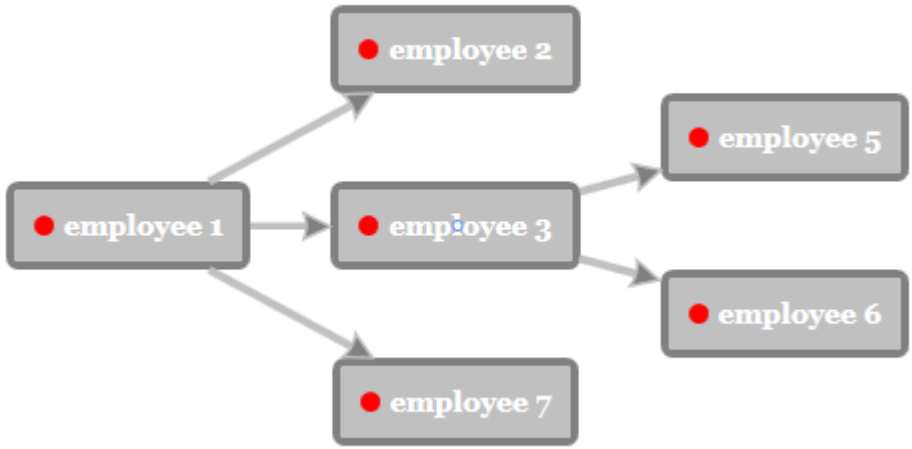
```
      </StackPanel>
</DataTemplate>
```

While specifying a data template for nodes, you can bind to the data that this node represents. The data context of the node is the item in the data source that this node represents, in this case an XML element.

The full markup file would be as follows:

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:ilog="http://www.ibm.com/diagrammer.net/2008"
    Title="WPF Diagram Example" Height="333" Width="455">
    <Window.Resources>
        <XmlDataProvider x:Key="xmldata"  Source="data.xml"/>
        <Style TargetType="ilog:Node">
            <Setter Property="Padding" Value="10"/>
            <Setter Property="FontFamily" Value="Georgia"/>
            <Setter Property="FontSize" Value="14"/>
            <Setter Property="FontWeight" Value="bold"/>
            <Setter Property="Background" Value="Silver"/>
            <Setter Property="Foreground" Value="White"/>
            <Setter Property="BorderBrush" Value="Gray"/>
            <Setter Property="BorderThickness" Value="4"/>
            <Setter Property="CornerRadius" Value="4"/>
        </Style>

        <Style TargetType="ilog:Link">
            <Setter Property="StrokeThickness" Value="4"/>
            <Setter Property="Stroke" Value="Silver"/>
            <Setter Property="EndArrow">
                <Setter.Value>
                    <ilog:LinkArrow  Shape='curved'  Fill="gray"/>
                </Setter.Value>
            </Setter>
        </Style>

        <DataTemplate x:Key="nodeTemplate">
            <StackPanel Orientation="Horizontal">
                <Ellipse Fill="red" Width="10" Height="10"/>
                <TextBlock Margin="5,0,0,0" Text="{Binding XPath=@name}"/>
            </StackPanel>
        </DataTemplate>
    </Window.Resources>
    <Grid>
        <ilog:Diagram
            NodeTemplate="{StaticResource nodeTemplate}"
            SuccessorsBinding="{Binding XPath=employee}"
            NodesSource="{Binding Source={StaticResource xmldata},
XPath='//employee'}"/>
    </Grid>
</Window>
```

And the resulting graph looks like this:



Note that the **DataTemplate** class also allows you to specify triggers that are used to set values in the template depending on some conditions. In the markup below, the data template has been modified so that an employee with name equal to '`employee 1`' has a blue ellipse instead of a red one.

```
<DataTemplate x:Key="nodeTemplate">
    <StackPanel Orientation="Horizontal">
        <Ellipse x:Name="ellipse" Fill="red" Width="10" Height="10"/>
        <TextBlock Margin="5,0,0,0" Text="{Binding XPath=@name}"/>
    </StackPanel>
    <DataTemplate.Triggers>
        <DataTrigger Binding="{Binding XPath=@name}" Value="employee 1">
            <Setter Property="Fill" TargetName="ellipse" Value="blue"/>
        </DataTrigger>
    </DataTemplate.Triggers>
</DataTemplate>
```

You may also decide to have different data templates depending on some logic that you want to code. In this case, create a subclass of the **DataTemplateSelector** class and override the **SelectTemplate** method.

For example, define a different template for managers and non-managers:

```
<Window.Resources>
    …
    <DataTemplate x:Key="managerNodeTemplate">
        <StackPanel>
             <Label
               HorizontalAlignment='center'
```

```
                FontSize='10'
                FontStyle="Italic"
                Content="Manager"/>
             <TextBlock Text="{Binding XPath=@name}"/>
        </StackPanel>
    </DataTemplate>

    <DataTemplate x:Key="nodeTemplate">
            <StackPanel Orientation="Horizontal">
                <Ellipse Fill="red" Width="10" Height="10"/>
                <TextBlock Margin="5,0,0,0" Text="{Binding XPath=@name}"/>
            </StackPanel>
    </DataTemplate>
    …
</Window.Resources>
```

In the following example, the **SelectTemplate** method provides logic to return the appropriate template wether the XML employee element has subemployee elements or not. The template to return is found in the resources of the enveloping **Window** element.

```
public class EmployeeDataTemplateSelector : DataTemplateSelector
{
    public override DataTemplate SelectTemplate(object item, DependencyObject
container)
    {
        XmlElement element = item as XmlElement;
        if (item != null)
        {
            Window window = Application.Current.MainWindow;
            if (element.GetElementsByTagName("employee").Count != 0)
                return
                    window.FindResource("managerNodeTemplate") as DataTemplate;
            else
                return
                    window.FindResource("nodeTemplate") as DataTemplate
        }
        return null;
    }
}
```

Then, you can declare the data template selector as a resource:

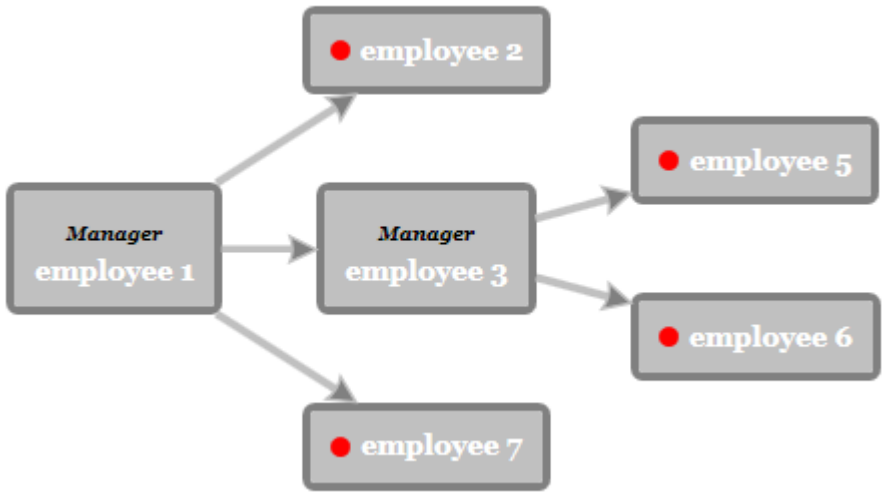
```
<Window.Resources>
…
<local: EmployeeDataTemplateSelector x:Key="employeeDataTemplateSelector"/>
…
</Window.Resources>
```

To use the template selector resource, assign it to the NodeTemplateSelector property of the **Diagram** class. The **Diagram** calls the **SelectTemplate** method of the **EmployeeDataTemplateSelector** for each of the items in the underlying collection. The call passes the data object as the item parameter. The **DataTemplate** returned by the method is applied to that data object.

```
<ilog:Diagram
     NodeTemplateSelector="{StaticResource employeeDataTemplateSelector}"
```

```
    SuccessorsBinding="{Binding XPath=employee}"
    NodesSource="{Binding Source={StaticResource xmldata} ,
XPath='//employee'}"/>
```

With the template selector in place the graph looks like this:



Through the NodeTemplate property, you can combine all the WPF elements to create any kind of representation for the nodes.

Changing the Control Template of the Node Class

The Node class is a subclass of the **ContentControl** class, thus a **Node** is a WPF control and has a **ControlTemplate** that you can modify. By default, the control template of the node represents a node by a WPF Border that encapsulates a **ContentPresenter** (remember that the content presenter is there for representing the content of a **ContentControl**, in this case the content of the node). One way to change the appearance of nodes is to change the control template of the **Node** class. Here is the default control template:

```
<ControlTemplate TargetType="{x:Type local:Node}">
        <Border
            SnapsToDevicePixels="true" x:Name="Border"
            Background="{TemplateBinding Background}"
            BorderBrush="{TemplateBinding BorderBrush}"
            BorderThickness="{TemplateBinding BorderThickness}"
            CornerRadius="{TemplateBinding CornerRadius}"
            Padding="{TemplateBinding Padding}">
          <ContentPresenter
            SnapsToDevicePixels="{TemplateBinding SnapsToDevicePixels}"
            HorizontalAlignment="{TemplateBinding
```

```
HorizontalContentAlignment}"
              VerticalAlignment="{TemplateBinding VerticalContentAlignment}"/>
          </Border>

          <ControlTemplate.Triggers>
            <Trigger Property="IsSelected" Value="true">
              <Setter Property="Background"   TargetName="Border"
                  Value="{DynamicResource {x:Static
SystemColors.HighlightBrushKey}}"/>
              <Setter Property="Foreground"
                  Value="{DynamicResource {x:Static
SystemColors.HighlightTextBrushKey}}"/>
            </Trigger>
            <MultiTrigger>
                <MultiTrigger.Conditions>
                  <Condition Property="IsSelected" Value="true"/>
                  <Condition Property="local:Diagram.IsSelectionActive"
Value="false"/>
                </MultiTrigger.Conditions>
                <Setter Property="Background" TargetName="Border"
                   Value="{DynamicResource {x:Static
SystemColors.ControlBrushKey}}"/>
                <Setter Property="Foreground"
                   Value="{DynamicResource {x:Static
SystemColors.ControlTextBrushKey}}"/>
              </MultiTrigger>
              <Trigger Property="IsEnabled" Value="false">
                <Setter Property="Foreground"
                   Value="{DynamicResource {x:Static
SystemColors.GrayTextBrushKey}}"/>
              </Trigger>
            </ControlTemplate.Triggers>
        </ControlTemplate>
```

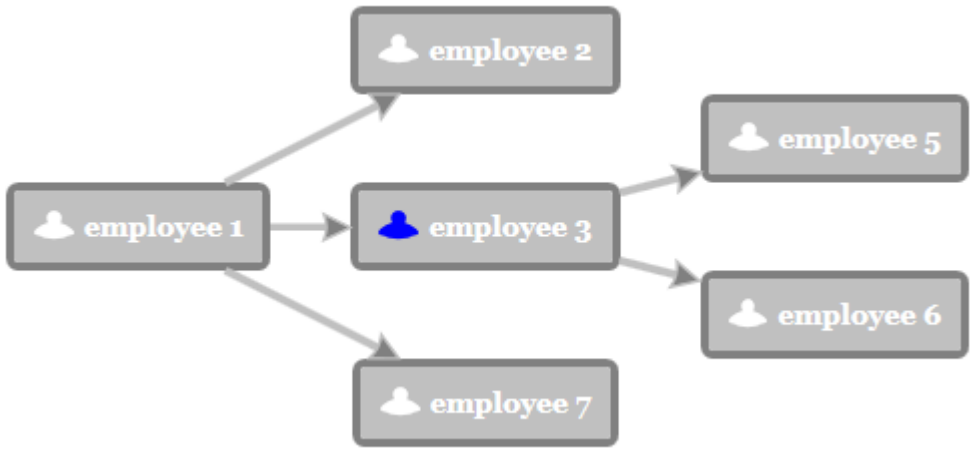
Let's change the control template for a new one that adds a **Path** element representing an employee on the left of the **ContentPresenter**.

The full markup file would be as follows:

```
<Window    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:ilog="http://www.ibm.com/diagrammer.net/2008"
    xmlns:local="clr-namespace:Sample"
    Title="WPF Diagram Example" Height="336" Width="475">
    <Window.Resources>
        <ControlTemplate x:Key="myControlTemplate" TargetType="ilog:Node">
            <Border
              SnapsToDevicePixels="true" x:Name="Border"
              Background="{TemplateBinding Background}"
              BorderBrush="{TemplateBinding BorderBrush}"
              BorderThickness="{TemplateBinding BorderThickness}"
              CornerRadius="{TemplateBinding CornerRadius}"
              Padding="{TemplateBinding Padding}">
                 <StackPanel Orientation="Horizontal">
                 <Path x:Name="user" Fill="{TemplateBinding Foreground}"
                    Data="M9.93 0C7.82 0 6.12 1.70 6.12 3.80C6.12 4.62 6.38
5.38 6.82
                      6.01C3.14 6.88 -1.91 10.58 0.62 11.85Q9.79 16.35 19.24
```

```
                        11.85C21.83 10.67 16.54 7.01 13 6.07C13.46 5.44 13.74
                        4.66 13.74 3.80C13.74 1.70 12.03 0 9.93 0z"/>
                     <ContentPresenter />
                    </StackPanel>
                    </Border>
                <ControlTemplate.Triggers>
                    <Trigger Property="IsSelected" Value="true">
                        <Setter Property="Fill"   TargetName="user" Value="blue"/>
                    </Trigger>
                </ControlTemplate.Triggers>
            </ControlTemplate>

            <Style TargetType="ilog:Node">
                <Setter Property="Template"
Value="{StaticResourcemyControlTemplate}"/>
                <Setter Property="Padding" Value="10"/>
                <Setter Property="FontFamily" Value="Georgia"/>
                <Setter Property="FontSize" Value="14"/>
                <Setter Property="FontWeight" Value="bold"/>
                <Setter Property="Background" Value="Silver"/>
                <Setter Property="Foreground" Value="White"/>
                <Setter Property="BorderBrush" Value="Gray"/>
                <Setter Property="BorderThickness" Value="4"/>
                <Setter Property="CornerRadius" Value="4"/>

            </Style>

            <Style TargetType="ilog:Link">
                <Setter Property="StrokeThickness" Value="4"/>
                <Setter Property="Stroke" Value="Silver"/>
                       <Setter Property="EndArrow">
                             <Setter.Value>
                                 <ilog:LinkArrow  Shape='curved'  Fill="gray"/>
                             </Setter.Value>
                       </Setter>
            </Style>
            <XmlDataProvider x:Key="xmldata"  Source="data.xml"/>
            <DataTemplate x:Key="nodeTemplate">
               <StackPanel Orientation="Horizontal">
                <TextBlock Margin="5,0,0,0" x:Name="text"
Text="{BindingXPath=@name}"/>
                </StackPanel>
            </DataTemplate>
        </Window.Resources>
        <Grid>
            <ilog:Diagram
                NodeTemplate="{DynamicResource nodeTemplate}"
                SuccessorsBinding="{Binding XPath=employee}"
                NodesSource="{Binding Source={StaticResource xmldata},
XPath='//employee'}" />
        </Grid>
</Window>
```

The new **ControlTemplate** for the **Node** is specified in the style defined for the **Node** class through the **Template** property. A trigger in the template changes the **Fill** property of the **Path** to blue when the node is selected (the **IsSelected** property of the **Node** is **true**). This control template gives the following result when `employee 3` is selected.



Creating a Style of Links that Depends on the Data

In *Styling Nodes with Data Templates and Data Template Selectors* you have seen that the NodeTemplate property can be used to create a representation for the nodes that depends on the data. The data context of a node is the item in the data source that this node represents, that is why in a node data template you can bind to the properties of the item in the data source. However, the situation is different for the links. A link represents the relation between two nodes in the data source. For this reason, links created by the **Diagram** control have a specific data context to enable specific bindings. This context is an instance of the Diagram.LinkContext class. This class has two properties: StartItem and EndItem. These two properties give access to the two items in the data source that corresponds to the **Link** instance.

The following example defines a style that will change the color of the link to red, when the start item and end item in the data source are critical. This example assumes that the items in the data source are instances of a class that defines a Boolean property named "Critical".

```
<Style TargetType="ilog:Link">
    <Setter Property="Stroke" Value="Silver"/>
    <Style.Triggers>
        <MultiDataTrigger>
          <MultiDataTrigger.Conditions>
               <Condition Binding="{Binding Path=StartItem.Critical}" Value="true" />
                <Condition Binding="{Binding Path=EndItem.Critical}" Value="true" />
            </MultiDataTrigger.Conditions>
            <Setter Property="Stroke" Value="Red" />
        </MultiDataTrigger>
    </Style.Triggers>
 </Style>
```

Selection in a Diagram

The Diagram control has a built-in selection system that allows you to select the nodes in a graph. The node gets selected when you click it. The selection system is similar to the system in the WPF ListBox: for example, you can use the SelectionMode property to specify a single or multiple selection mode.

You can also select the links of the graph, provided that they were created from items contained in the **LinksSource** data source or that they have been added to the **Links** collection. In other terms, you cannot select a link that was created implicitly by the **SuccessorsBinding**, **PredecessorsBinding** or **ParentBinding** properties, because in that case there is no data item associated with the **Link**.

When the **SelectionMode** property is Single, you can select only one single node and you may get or set the item selected in the data source through the SelectedItem property. When the **SelectionMode** is Multiple or Extended, you may select several nodes and the SelectedItems property reflects the items that are selected in the data source. You may add or remove an item from the selection by adding or removing items in the **SelectedItems** collection.

When items get selected or deselected, the **Diagram** control fires events that you can listen through the SelectionChanged event.

The **SelectedItem** and **SelectedItems** property are dependency properties. For example, you can use these properties with the WPF binding system to keep some parts of your interface synchronized with the selection in the diagram.

Note that the **Node** and **Link** classes has a dependency property named IsSelected that is **true** when the node or link is selected and **false** otherwise. This property is very useful when styling a node; you may have a WPF trigger on the style that changes the look of the node based on the value of the **IsSelected** property. You can see an example of this technique in *Changing the Control Template of the Node Class*.

Zoom in a Diagram

The Diagram control allows you to zoom in and out the graph displayed through the Zoom property. A Zoom value of 100 represents the normal size of the diagram. You can control the minimum and maximum zoom level through the MinZoom and MaxZoom properties.

The **Diagram** class has the following built-in commands that allow you to easily control the zoom level:

- IncreaseZoomCommand: to zoom in,
- DecreaseZoomCommand: to zoom out,
- ZoomTo100PercentCommand: to set the zoom to 100,
- ZoomToFitCommand: to zoom so that the graph fits in the diagram control.

The graph itself is displayed inside a WPF scroll viewer. As you zoom in, scroll bars appear to allow you to scroll in the graph. The HorizontalScrollBarVisibility and VerticalScrollBarVisibility allows you to control the visibility of the scroll bars.

Finally, the HorizontalDiagramAlignment and VerticalDiagramAlignment properties allow you to control the alignment of the graph inside the scroll viewer.

Specifying Graph Layout Algorithms in a Diagram

Inside a Diagram control, the nodes and links that are generated from the data source are automatically arranged by a Graph Layout algorithm that places the nodes and the links. You can specify the algorithm through the GraphLayout property of the **Diagram** class. By default, a HierarchicalLayout instance is specified in this property, but a set of algorithms are present to cover various types of graph data.

The following example specifies a TreeLayout algorithm on a **Diagram** object:

```
<ilog:Diagram …>
    <ilog:Diagram.GraphLayout>
        <ilog:TreeLayout FlowDirection="Bottom" LinkStyle="Orthogonal"/>
    </ilog:Diagram.GraphLayout>
</ilog:Diagram>
```

Note that most of the Graph Layout algorithms have per-node and per-link properties, they are implemented as attached properties that you can specify on the Node or Link instance. For example, in the **TreeLayout** algorithm you can specify an attached property named Alignment for each **Node** instance. This property defines the alignment of a node with respect to its children. In a **Diagram** object you may use these attached properties when defining a style for the **Link** class or a style, a control template or a data template for a **Node**.

The following code extract provides an example of style for nodes. All the nodes have the **Alignment** property set to **East** except the nodes with the **Critical** property to **true**, that will have the **Alignment** set to **Center**. (It is assumed that items in the data source have a Boolean property named **Critical**).

```
<Style TargetType="{x:Type ilog:Node}">
    <Setter Property="ilog:TreeLayout.Alignment" Value="East"/>
    <Style.Triggers>
        <!-- All nodes are aligned with East mode except the root that is Centered" -->
        <DataTrigger Binding="{Binding Path=Critical}" Value="true" >
            <Setter Property="ilog:TreeLayout.Alignment" Value="Center"/>
        </DataTrigger>
     </Style.Triggers>
</Style>
```

Creating a Graph by Specifying Nodes and Links

The Diagram control (as explained in *Creating a Graph Display from a Data Source Using the Diagram Control*) creates a graph from a data source and generates nodes and links instances of the Node and Link classes to render the data inside the data source. However, it is also possible to create a graph by creating instances of the **Node** and **Link** classes and use them directly inside any WPF container.

There are various ways to create a graph using the **Node** and **Link** classes.

You can use **Node** and **Link** instances in any WPF container, for example a **Canvas**. In this case you create **Node** instances connected by **Link** instances and you add them as children of the **Canvas**. Then, you are responsible for placing the nodes in the **Canvas** and specifying the link shape. The **Canvas** does not do an automatic layout of your graph, but you may execute a Graph Layout algorithm as needed.

You may also use **Node** and **Link** instances inside a Graph control. This control is very similar to a WPF **Canvas** and allows you to specify the position of nodes through the Graph.Left and Graph.Top attached properties, as when you specify the positions in a **Canvas** using the **Canvas.Left**, **Canvas.Top**, **Canvas.Bottom** and **Canvas.Right** properties. The difference is that the **Graph** control has a property named GraphLayout and a property named LinkLayout. You may use these properties to specify a Graph Layout algorithm to layout the graph automatically, for example you may specify a **TreeLayout** instance in the **GraphLayout** properties and the nodes and links will be automatically

arranged as a tree when you add/remove nodes and links or as the desired size of node changes.

Finally, a third possibility is to create **Node** and **Link** instances and add them to the **Nodes** and **Links** properties of a **Diagram**. This technique is similar to the previous one (adding nodes and links in a **Graph** control), but the advantage is that the **Diagram** control offers zooming and scrolling.

In This Section

Creating Nodes and Links

Explains how to create nodes and links.

Example of Nodes and Links in a Canvas

Provides examples of nodes and links in a Canvas.

Example of Nodes and Links in a Graph with TreeLayout

rovides examples of nodes and links in a graph with TreeLayout.

Controlling the Connection Points of Links

Explains how the link is connected to the start and end nodes.

Specifying the Link Shape

Explains how to control the shape of the link.

Color, Thickness and Other Appearance Properties of a Link

Describes how to control appearance properties of a link.

Customizing the Arrows of a Link

Describes how to customize the arrows of a link.

Creating Nodes and Links

Using the Node and Link classes you can create a graph representation. The **Node** class is a WPF 'content control'. This means that you may add any kind of content inside a node, thus you may combine any WPF elements to create any kind of representation inside a node.

The following example shows how to create a node represented by an ellipse:

```
<ilog:Node BorderThickness="0" Padding="0">
      <Ellipse Width="50" Height="50" Fill="red"/>
</ilog:Node>
```

You connect two nodes by an instance of the **Link** class. The **Link** class defines two properties, Start and End, that specify where the link should visually start and end. Alternatively, you can use the StartName and EndName property of the **Link** class to specify

the start and end node. Then, you will use the **StartName** and **EndName** properties to define the names of the start and end nodes respectively.

Note that it is not necessary to have the link, the start and the end nodes inside the same container. For example, two nodes can be placed inside two different **Canvas** and the link in another one. The only restriction is that the start and end nodes should have a common ancestor with the link.

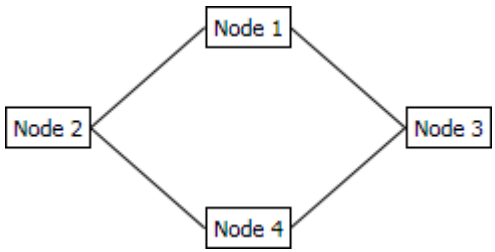
The links that are starting at a node can be retrieved through the FromLinks property of the **Node** class, while the links that are ending at a node can be retrieved through the ToLinks property.

Example of Nodes and Links in a Canvas

Here is a small markup declaration for four nodes and four links in a **Canvas**:

```
<Canvas>
    <ilog:Node Canvas.Top="0"   Canvas.Left="100" x:Name="node1">Node 1</ilog:Node>
    <ilog:Node Canvas.Top="50"  Canvas.Left="0"   x:Name="node2">Node 2</ilog:Node>
    <ilog:Node Canvas.Top="50"  Canvas.Left="200" x:Name="node3">Node 3</ilog:Node>
    <ilog:Node Canvas.Top="100" Canvas.Left="100" x:Name="node4">Node 4</ilog:Node>
    <ilog:Link StartName="node2" EndName="node1"></ilog:Link>
    <ilog:Link StartName="node1" EndName="node3"></ilog:Link>
    <ilog:Link StartName="node3" EndName="node4"></ilog:Link>
    <ilog:Link StartName="node4" EndName="node2"></ilog:Link>
</Canvas>
```

This piece of markup will produce the following graph:



The same graph can be created in C# using the following code:

```
public void CreateNodeAndLinks(Canvas canvas)
{
    Node node1 = new Node();
    node1.Content = "node 1";
    Canvas.SetTop(node1, 0);
    Canvas.SetLeft(node1, 100);
```

```
    Node node2 = new Node();
    node2.Content = "node 2";
    Canvas.SetTop(node2, 50);
    Canvas.SetLeft(node2, 0);

    Node node3 = new Node();
    node3.Content = "node 3";
    Canvas.SetTop(node1, 50);
    Canvas.SetLeft(node1, 200);

    Node node4 = new Node();
    node4.Content = "node 4";
    Canvas.SetTop(node1, 100);
    Canvas.SetLeft(node1, 100);

    canvas.Children.Add(node1);
    canvas.Children.Add(node1);
    canvas.Children.Add(node1);
    canvas.Children.Add(node1);
    canvas.Children.Add(new Link(node2, node1));
    canvas.Children.Add(new Link(node1, node3));
    canvas.Children.Add(new Link(node3, node4));
    canvas.Children.Add(new Link(node4, node2));
}
```

Example of Nodes and Links in a Graph with TreeLayout

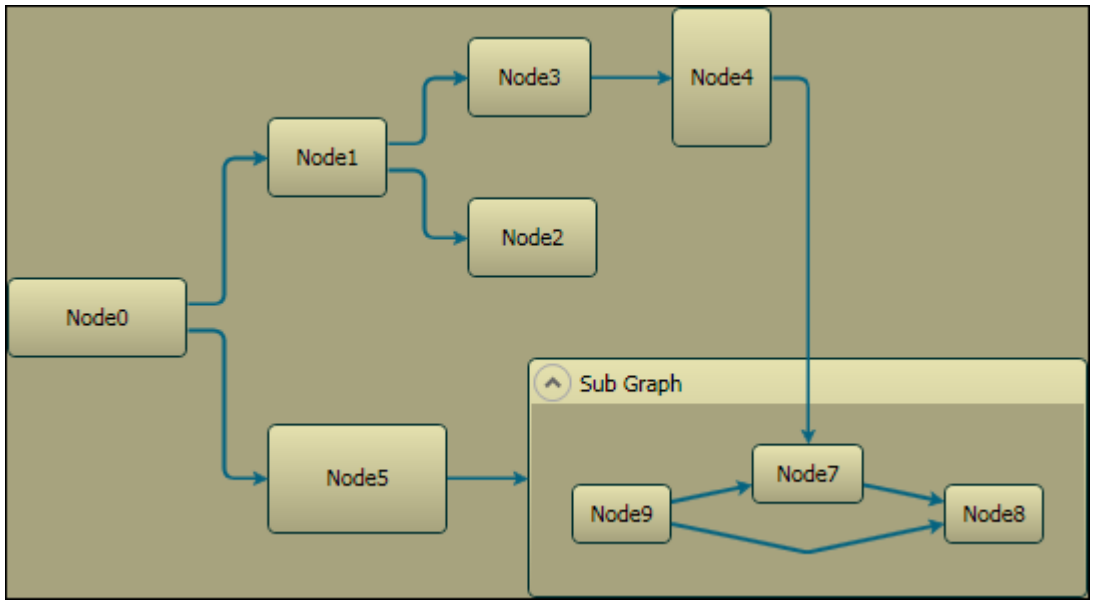
The following example shows how to use a Graph control in XAML and how to specify the nodes and links as children of the **Graph**:

```
<ilog:Graph
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:ilog="http://www.ibm.com/diagrammer.net/2008"
  Background="#a7a37e">

  <ilog:Graph.Resources>
    <LinearGradientBrush x:Key="nodeback" StartPoint="0,0" EndPoint="0,1">
      <GradientStop Color="#FFE6E2AF" Offset="0"/>
      <GradientStop Color="#FFA7A37E" Offset="1"/>
    </LinearGradientBrush>

    <SolidColorBrush x:Key="linkcolor" Color="#ff046380"/>

    <!-- The style for nodes-->
    <Style TargetType="{x:Type ilog:Node}">
      <Setter Property="Background" Value="{DynamicResource nodeback}"/>
      <Setter Property="HorizontalContentAlignment" Value="Center"/>
      <Setter Property="VerticalContentAlignment" Value="Center"/>
      <Setter Property="BorderBrush" Value="#002f2f"/>
      <Setter Property="CornerRadius" Value="4"/>
      <Setter Property="Padding" Value="0"/>
    </Style>

    <!-- The style for links-->
```

```
    <Style TargetType="{x:Type ilog:Link}">
      <Setter Property="EndArrow">
        <Setter.Value>
          <ilog:LinkArrow Shape="Sunken"/>
       </Setter.Value>
      </Setter>
      <Setter Property="Radius" Value="5"/>
      <Setter Property="Stroke" Value="{StaticResource linkcolor}"/>
      <Setter Property="StrokeThickness" Value="2"/>
    </Style>

  </ilog:Graph.Resources>

  <ilog:Graph.GraphLayout>
    <ilog:TreeLayout ConnectorStyle="EvenlySpaced"/>
  </ilog:Graph.GraphLayout>

  <ilog:Node x:Name="node0" Width="90" Height="40">Node0</ilog:Node>
  <ilog:Node x:Name="node1" Width="60" Height="40">Node1</ilog:Node>
  <ilog:Node x:Name="node3" Width="62" Height="40">Node3</ilog:Node>
  <ilog:Node x:Name="node2" Width="65" Height="40">Node2</ilog:Node>
  <ilog:Node x:Name="node4" Width="50" Height="70">Node4</ilog:Node>
  <ilog:Node x:Name="node5" Width="90" Height="55">Node5</ilog:Node>
  <ilog:Node x:Name="node6">
    <Expander IsExpanded="True"  Header="Sub Graph" ExpandDirection="Down">
      <Border Padding='20' Background='#a7a37e'>
        <ilog:Graph  Background="Transparent">
          <ilog:Graph.GraphLayout>
            <ilog:HierarchicalLayout/>
          </ilog:Graph.GraphLayout>
            <ilog:Link StartName="node9" EndName="node7"/>
            <ilog:Link StartName="node4" EndName="node7"
                       ShapeType="Orthogonal"
                       StartAnchorPosition="Right"
                       EndAnchorPosition="Top"/>
            <ilog:Link StartName="node7" EndName="node8"/>
            <ilog:Link StartName="node9" EndName="node8"/>
            <ilog:Node x:Name="node8" Width="50" Height="30">Node8</ilog:Node>
            <ilog:Node x:Name="node9" Width="50" Height="30">Node9</ilog:Node>
            <ilog:Node x:Name="node7" Width="56" Height="30">Node7</ilog:Node>
        </ilog:Graph>
      </Border>
    </Expander>
  </ilog:Node>
  <ilog:Link StartName="node1" EndName="node2"/>
  <ilog:Link StartName="node0" EndName="node1"/>
  <ilog:Link StartName="node0" EndName="node5"/>
  <ilog:Link StartName="node3" EndName="node4"/>
  <ilog:Link StartName="node5" EndName="node6"/>
  <ilog:Link StartName="node1" EndName="node3"/>
</ilog:Graph>
```

This XAML example defines a top graph that contains the nodes from node 0 to node 6. The top graph has its **GraphLayout** property set to a **TreeLayout** instance. The node number 6 contains a WPF expander that allows you to expand and collapse its content. This expander contains another **Graph** instance that contains node 7 to node 9. This second graph has a **GraphLayout** property set to an instance of the Hierarchical layout.

Here is the resulting graph:



Note that the link between node 4 and node 7 is crossing graph boundaries. This is an intergraph link (the start node is not in the same container than the link). Although this link is child of the subgraph, it is not taken into account by the hierarchical layout. However, as the link is part of the subgraph, the link will disappear when the subgraph collapses through the expander.

As you expand and collapse the subgraph, the top graph automatically re-arranges its content by applying the specified tree layout algorithms.

Example of Nodes and Links in a Diagram using the Nodes and Links Collections

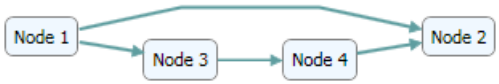
The following example shows how to add nodes and links by code to a **Diagram** using the **Nodes** and **Links** properties.

```
public void CreateNodeAndLinks(Diagram diagram)
{
    Node node1 = CreateNode("Node 1");
    Node node2 = CreateNode("Node 2");
    Node node3 = CreateNode("Node 3");
    Node node4 = CreateNode("Node 4");
    diagram.Nodes.Add(node1);
    diagram.Nodes.Add(node2);
    diagram.Nodes.Add(node3);
    diagram.Nodes.Add(node4);
    diagram.Links.Add(CreateLink(node1, node2));
```

```
            diagram.Links.Add(CreateLink(node1, node3));
            diagram.Links.Add(CreateLink(node3, node4));
            diagram.Links.Add(CreateLink(node4, node2));
        }

        private Node CreateNode(string content)
        {
            Node node = new Node();
            node.Content = content;
            node.Background = new SolidColorBrush(Color.FromArgb(255, 240, 248,
255));
            node.BorderBrush = new SolidColorBrush(Colors.Gray);
            node.CornerRadius = new CornerRadius(5);
            node.Padding = new Thickness(5);
            return node;
        }

        private Link CreateLink(Node start, Node end)
        {
            Link link = new Link(start, end);
            link.Stroke = new SolidColorBrush(Color.FromArgb(255, 95, 158,
160));
            link.StrokeThickness = 2;
            link.EndArrow = new LinkArrow();
            return link;
        }
```

Here, the appearance of the nodes and links is customized by code.

Here is the resulting graph:



Controlling the Connection Points of Links

You can control how the link is connected to the start and end nodes by means of several properties.

When a Link instance has a start and an end node specified by the Start and End properties the link will automatically connect to the start and end nodes. Some properties specify where the link connects on the shape of the start and end nodes.

The StartAnchorPosition and EndAnchorPosition properties represent the positions where the link is attached on the start and end node. The possible values are defined by the

AnchorPosition enumeration. This enumeration defines 9 points on the bounding box of the node:

- **Top**: the link is connected to the top center of the node.
- **TopLeft**: the link is connected to the top left corner of the node.
- **TopRight**: the link is connected to the top right corner of the node.
- **Left**: the link is centered to the left border of the node.
- **Right**: the link is centered to the right border of the node.
- **Bottom**: the link is centered to the bottom of the node.
- **BottomLeft**: the link is connected to the bottom left corner of the node.
- **BottomRight**: the link is connected to the bottom right of the node.
- **Center**: the link is connected to the center of the node.

Two additional values are possible:

- **Automatic**: the link will choose the position between the 9 positions above the one which best corresponds to its current points and opposite node.
- **Clip**: The link is going to the direction of the center of the node, but the connection point is clipped on the border of the node bounding box.

When the position is specified it is still possible to move the connection point by an offset from the position specified by the **StartAnchorPosition** and **EndAnchorPosition** properties. To do that, use the **StartAnchorOffset** and EndAnchorOffset properties. These offsets are defined as a **Size** object and are expressed as a percentage of the size of the node. Therefore, an offset of (0.25 0) will move the connection point to the right by half of the node size.

Here is a markup example:

```
<ilog:Link … StartAnchorPosition="Center" StartAnchorOffset="0.25,0" />
```

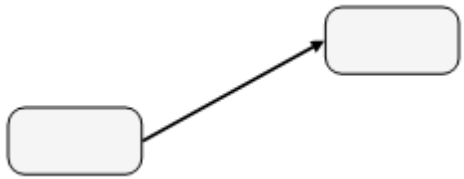
Notes:

1. *When a link is used inside a **Graph** or **Diagram** control and is automatically reshaped by a Graph Layout algorithm, the connection point of the link is controlled by the Graph Layout algorithm and not by the **StartAnchorPosition** and **EndAnchorPosition** properties. In this case, changing the properties has no effect.*
2. *A link may not be connected to a start node or to an end node. When a link is not connected to a start node (that is, when the **Start** property is **null**), the starting point of the link may be specified through the **StartPoint** property. The **Link** class also defines an **EndPoint** property when the **End** property is **null**.*

Specifying the Link Shape

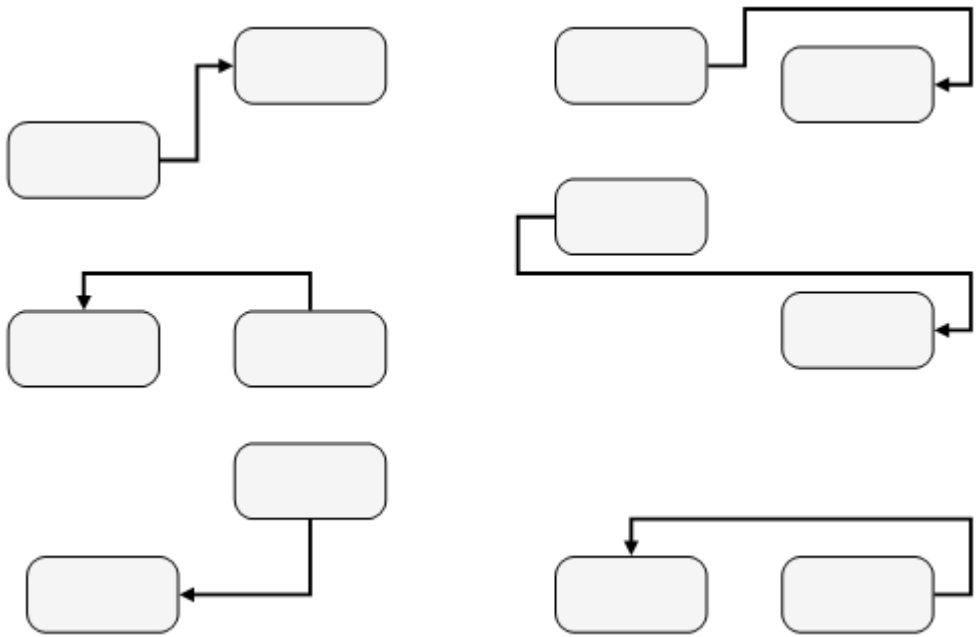
The ShapeType property controls the shape of the link. The type of this property is the LinkShapeType enumeration, and these are the possible values:

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



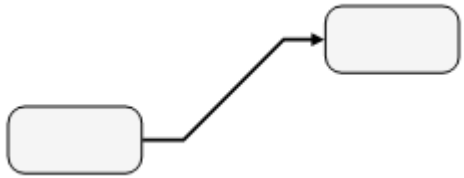
- **Orthogonal**: the link shape is computed automatically to a sequence of vertical or horizontal segments. There can be 2 to 5 segments, depending on the connection points and the bounds of the **Node** elements to which the link is connected.

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



> ***Note:*** *When* ***ShapeType*** *is* ***Orthogonal*** *and if the CanEditOrthogonalShape property is* ***true****, it is possible to modify the link shape using the Points property, while keeping the link orthogonal.*

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



- **Free**: the link shape is determined by the **Points** property.

> ***Note:*** *The* ***ShapeType*** *property may not be taken into account if the link is automatically arranged by a Graph Layout algorithm, that is, when the link is child of a* ***Graph*** *control that has a* ***GraphLayout*** *or* ***LinkLayout*** *property not null, or when the* ***Link*** *is created from a data source in a* ***Diagram*** *control. In these cases, shape and connection points of the links may be fully controlled by the Graph Layout algorithm. Some Graph Layout algorithms, such as the* ***GridLayout****, do not change the shape of links and in this case the* ***ShapeType*** *property will be respected.*

The shape of the link can be further customized using the Radius property.

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.

Color, Thickness and Other Appearance Properties of a Link

In the Link class you specify the brush used to render the link through the Stroke property. The thickness is specified through the StrokeThickness property.

Other properties such as StrokeDashArray, StrokeDashCap, StrokeDashOffset allows you to create dashed links.

Of course, the Link class can benefits from all the WPF features such as styling and animations: the following example shows a dashed link animated so that the dashes seem to move on the link:

```
<ilog:Link StartName="node2" EndName="node1">
    <ilog:Link.Style>
        <Style TargetType="{x:Type ilog:Link}">
            <Style.Triggers>
                <EventTrigger RoutedEvent="FrameworkElement.Loaded">
                    <BeginStoryboard>
                        <Storyboard  RepeatBehavior="Forever">
        <DoubleAnimation From="0" To="8" Duration="0:0:0.5"

Storyboard.TargetProperty="(ilog:Link.StrokeDashOffset)"/>
                        </Storyboard>
                    </BeginStoryboard>
                </EventTrigger>
            </Style.Triggers>
            <Setter Property="StrokeDashArray" Value="4"/>
            <Setter Property="Stroke" Value="blue"/>
            <Setter Property="StrokeThickness" Value="3"/>
        </Style>
    </ilog:Link.Style>
</ilog:Link>
```

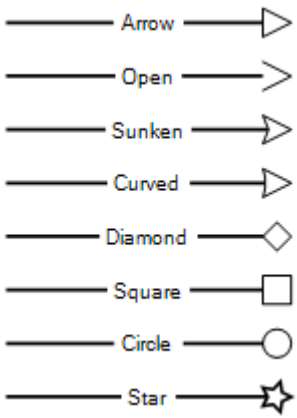
Customizing the Arrows of a Link

By default, a link has no 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 class.

The **LinkArrow** class represents an arrow that can have various shapes that you define through the Shape property. For example, to specify a red arrow at the end of the link that represents a star, in markup you would write:

```
<ilog:Link …>
   <ilog:Link.EndArrow>
       <ilog:LinkArrow Shape="Star" Size="20,20" Fill="red" Stroke="{x:Null}"/>
   </ilog:Link.EndArrow>
</ilog:Link>
```

The following example shows the predefined arrow shapes:



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

Specifying Graph Layout

Graph Layout classes are also present in the ILOG.Controls.Diagram namespace so that you can specify both by code or in XAML how your graph should be arranged. The classes are providing the same algorithms as the classes provided for the Windows Forms and ASP.NET classes of IBM® ILOG® Diagram for .NET but they are implemented as WPF dependency objects.

The Graph Layout classes are the following:

- HierarchicalLayout
- TreeLayout
- ForceDirectedLayout
- GridLayout
- ShortLinkLayout
- LongLinkLayout
- RandomLayout

For more details on each Graph Layout algorithm, see *Using Graph Layout Algorithms*. Note that this section is more oriented on using Graph Layout algorithms in a Windows Forms or ASP.NET context, therefore the code samples will not apply to the WPF case. The Graph Layout classes defined in the **ILOG.Controls.Diagram** namespace are different from the classes in ILOG.Diagrammer.GraphLayout namespace. Although the classes are not the

same, you will find that most of the properties are the same, only some advanced properties may not be present in the WPF versions.

How to Use the Graph Layout Classes

A Graph Layout instance can be directly specified through the GraphLayout property of the Diagram class or through the GraphLayout and LinkLayout properties of the Graph class. When used in this way, the graphs represented in the **Diagram** of the **Graph** instance are automatically arranged by the Graph Layout algorithm.

The following example specifies a TreeLayout algorithm on a **Diagram** object:

```
<ilog:Diagram …>
    <ilog:Diagram.GraphLayout>
        <ilog:TreeLayout FlowDirection="Bottom" LinkStyle="Orthogonal"/>
    </ilog:Diagram.GraphLayout>
</ilog:Diagram>
```

The **Diagram** control always requires a Graph Layout to layout the graph resulting from a data source. However, as you can specify the positions of nodes in a **Graph** control, you may or may not specify a Graph Layout in the **GraphLayout** property. In the same way, you may or may not specify a link layout in the **LinkLayout** property. All combinations are possible.

It is also possible to layout a graph composed of nodes and links that are contained in a WPF **Canvas**. Unlike the **Graph** control, the **Canvas** cannot automatically layout the graph as nodes and links are added, removed or reshaped. To layout a graph in a **Canvas** you will have to do it by code using the PerformLayout method. The following C# code shows how to layout a **Canvas** by a tree layout algorithm:

```
void ApplyTreeLayout(Canvas canvas) {
    TreeLayout layout = new TreeLayout();
    layout.Attach(canvas);
    layout.PerformLayout();
}
```

In this case, the **PerformLayout** method will change the position of nodes by changing the **Canvas.Left** and **Canvas.Top** properties on the nodes. It can also change the points of links (**Points** property), the shape of links (**ShapeType** property) and the connection properties of links (**StartAnchorPosition**/**EndAnchorPosition** properties).

When using a **Graph** control, you may also choose to keep the **GraphLayout** and **LinkLayout** to **null** and apply a layout algorithm by using the same code used for the **Canvas**.

```
void ApplyTreeLayout(Graph graph) {
    TreeLayout layout = new TreeLayout();
    layout.Attach(graph);
    layout.PerformLayout();
```

```
}
```

Index