IBM ILOG Gantt for .NET V4.0

Programming with

IBM ILOG Gantt for .NET

Silverlight 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

Preface **Programming with IBM ILOG Gantt for .NET Silverlight Controls 3**

Getting Started with Silverlight Gantt Components 5

Creating a Gantt Control 9

Connecting a GanttChart Control to Data 11

Using Predefined Data Sources in a GanttChart. 11

Using Your Own Data or Creating your own Data Source for the GanttChart 14

Creating Task Classes and Displaying them in a GanttChart 14

Establishing Parent-Child Relationship 17

Displaying Constraints between Tasks in a GanttChart Control 20

Connecting a ScheduleChart Control to Data 25

Using Predefined Data Sources in a ScheduleChart 26

Using Your Own Data or Creating your own Data Source for the ScheduleChart. 29

Using Predefined Data Models 35

The SimpleGanttModel Data Model 36

The ProjectSchedulingModel Data Model 39

The ProjectSchedulingModel Class 40

Activities in the ProjectSchedulingModel 41

How the Project Scheduling Model Computes the Schedule of a Project 43

Calendars in the Project Scheduling Model 47

Resource Leveling in the Project Scheduling Model 48

Specifying and Styling the Gantt Bars to Display 51
Using Default Bar Representations 54
Defining Start and End Time for the Bar 54
Defining the Shape of the Bar 55
Displaying Additional Information on the Right and on the Left of the Bar 57
Alignment Properties 57
Defining When a BarDefinition Applies for an Item 59
Connection of Constraint links in a GanttChart 59
Interactions on the Bar 59
Defining a Tooltip for the Bar 60
Example of Bar Definitions 61
Working with Tables in the Gantt Controls 65
Configuring Columns 66
Using Predefined Columns and Creating a Template Column 67
Styling Columns 69
Grouping, Sorting and Filtering Data 71
Controlling the Displayed Time Interval 75
Storing and Displaying Working and Nonworking Times 77
User Interaction on a Gantt Bar 81
Editing Process - Moving and Resizing a Bar 82
Using Bar Editing Events 82
Snapping Time when Moving or Resizing a Bar 83
Working with Editing Tooltip 83
Styling User Interaction Elements 85
Internationalization 87

Programming with IBM ILOG Gantt for .NET Silverlight Controls

The Silverlight controls of IBM® ILOG® Gantt for .NET are a set of Silverlight components that bring Gantt chart displays and scheduling algorithms to the Microsoft Silverlight platform.

In This Section

Getting Started with Silverlight Gantt Components

Introduces the Silverlight Gantt components.

Creating a Gantt Control

Explains how to create a Gantt control.

Connecting a GanttChart Control to Data

Describes how to connect a GanttChart to data.

Connecting a ScheduleChart Control to Data

Describes how to connect a ScheduleChart to data.

Using Predefined Data Models

Describes how to use predefined data models.

Specifying and Styling the Gantt Bars to Display

Describes how to style the Gantt bars.

Working with Tables in the Gantt Controls

Describes how to handle tables with the Gantt controls.

Grouping, Sorting and Filtering Data

Describes how to group, sort and filter data.

Controlling the Displayed Time Interval

Describes how to control the displayed time interval.

Storing and Displaying Working and Nonworking Times

Describes how to handle working and nonworking times.

User Interaction on a Gantt Bar

Describes the various user interactions on a Gantt bar.

Internationalization

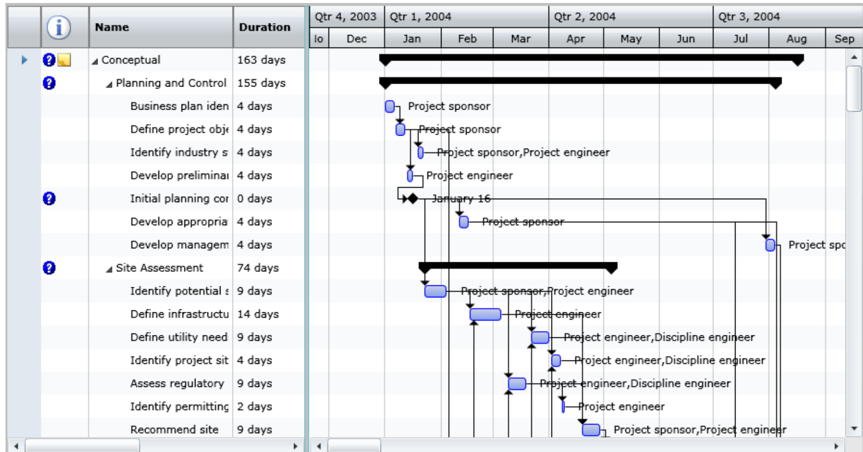
Describes the resource files and tools available with IBM ILOG Gantt for .NET to localize the library for a particular culture.

Getting Started with Silverlight Gantt Components

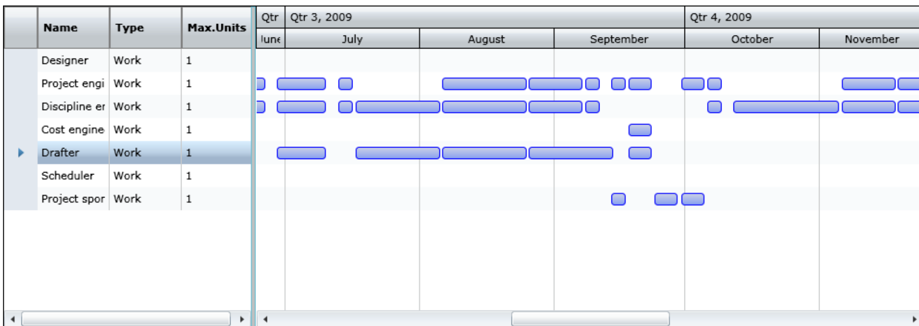
The Silverlight support in IBM® ILOG® Gantt for .NET consists of a library of classes used to display and edit scheduling data as a Gantt chart deployed in a Silverlight application.

It supports the two essential ways of displaying schedules: activity oriented and resource oriented. It can be used in a large variety of applications and industries: transportation scheduling and planning, logistics, supply chain management, production scheduling and planning, workforce planning, resource booking, project management, and many more.

The GanttChart control shows one task in each row. The hierarchy table on the left displays task information. The Gantt sheet on the right shows how the tasks are positioned on the time scale as well as constraints between tasks.



The ScheduleChart control shows one resource in each row. The table on the left displays resource information on the resource. The schedule sheet on the right shows the resource reservations. Each row in the schedule sheet contains 0, 1, or more bars to represent the activities for which the matching resource has been reserved.



Although the **GanttChart** control is mainly used to display tasks and the **ScheduleChart** to display the tasks assigned to a resource, both Gantt controls can be used with any type of data that has some time-based information. The main difference between the two controls is that the **GanttChart** control displays a unique information for each row (usually a task). The **ScheduleChart** control displays several time items in the sheet for each row in the table (usually a resource per row and all tasks assigned to this resource displayed in the sheet on the right).

In addition to the **GanttChart** and **ScheduleChart** controls, the library also contains some predefined data models that you can easily connect to the controls. The SimpleGanttModel is a simple data model that defines activities (or tasks), resources and reservations (assignments of resources to tasks). The ProjectSchedulingModel is a data model that also

defines the same type of information, but it uses this information to maintain the schedule of the project plan (computation of critical path(s) and resource leveling). Those data models are optional and require a separate assembly. You may create your own data model or even use a simple list of objects to be displayed in a Gantt control.

The **GanttChart** and **ScheduleChart** controls are located in the assembly named **ILOG.Controls.Gantt.dll** and in a namespace that has the same name. The predefined data models are located in the assembly named **ILOG.Controls.Gantt.Data.dll**.

Both the **GanttChart** and **ScheduleChart** inherit from the same base class HierarchyChart that concentrates properties and methods common to both representations.

Creating a Gantt Control

In order to create a GanttChart or ScheduleChart control, you need to add the reference to the assembly **ILOG.Controls.Gantt.dll** in your Silverlight project. In addition to **ILOG.Controls.Gantt.dll**, you can use the **ILOG.Controls.Gantt.Data.dll** assembly that contains the predefined data models such as the SimpleGanttModel class or the ProjectSchedulingModel class that are described in *Using Predefined Data Models*.

In your XAML file, you will have to define a new namespace to refer to the assembly:

```
xmlns:iloggantt="clr-
namespace:ILOG.Controls.Gantt;assembly=ILOG.Controls.Gantt"
```

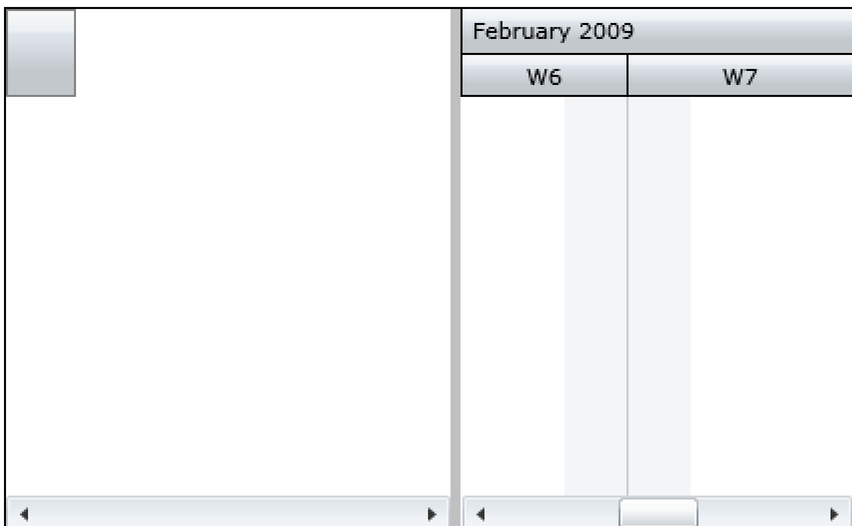
When you have defined the namespace, you can declare a **GanttChart** component in your XAML file:

```
<iloggantt:GanttChart x:Name="gantt"/>
```

Or a **ScheduleChart** component:

```
<iloggantt:ScheduleChart x:Name="gantt"/>
```

At this point you have the following graphical result:



The control is not yet connected to data. To connect the GanttChart to data, see *Connecting a GanttChart Control to Data*. Also, the controls do not define any columns or any definitions for the bars. To see how to define columns see *Working with Tables in the Gantt Controls*.

To see how to draw Gantt bars in a Gantt component see *Specifying and Styling the Gantt Bars to Display*.

Connecting a GanttChart Control to Data

A GanttChart control can display any kind of time-related items including instances of your own classes. Some predefined data sources are available so that you do not necessarily need to create your own data source.

You will learn how to use predefined data sources, how to create a new data source that can be used in a GanttChart control, and how to connect the data sources to a GanttChart.

In This Section:

Using Predefined Data Sources in a GanttChart

Explains how to use predefined data sources to display data in a GanttChart control.

Using Your Own Data or Creating your own Data Source for the GanttChart

Explains how to create a data source and connect it to a GanttChart control.

Using Predefined Data Sources in a GanttChart

To display data in a GanttChart control, you may use one of the following predefined data sources:

- SimpleGanttModel
- ProjectSchedulingModel

These data models can be easily connected to a **GanttChart** so that the **GanttChart** displays the list of activities in the model and the constraints between activities. For more details on the two data models, see *Using Predefined Data Models*.

To simply connect a **GanttChart** to a **SimpleGanttModel** use the following code:

```
GanttChart chart = new GanttChart();
SimpleGanttModel model = new SimpleGanttModel();
chart.InitializeFrom(model);
```

The InitializeFrom method is an extension method for the **GanttChart** class. This method does the following initializations:

- Creation of some predefined columns in the table: A column for Name, Duration, Start Time, End Time and Resources property.
- Creation of predefined bar definitions: A bar definition for normal activity, for activity completion, for summaries and for milestones.
- Connection of the data model to the control for displaying the activities and the constraints between activities.

Here is an extract of C# code that creates and populates a **SimpleGanttModel** with four activities and one constraint and displays the model in a **GanttChart**.

In XAML you simply add a **GanttChart** to the page:

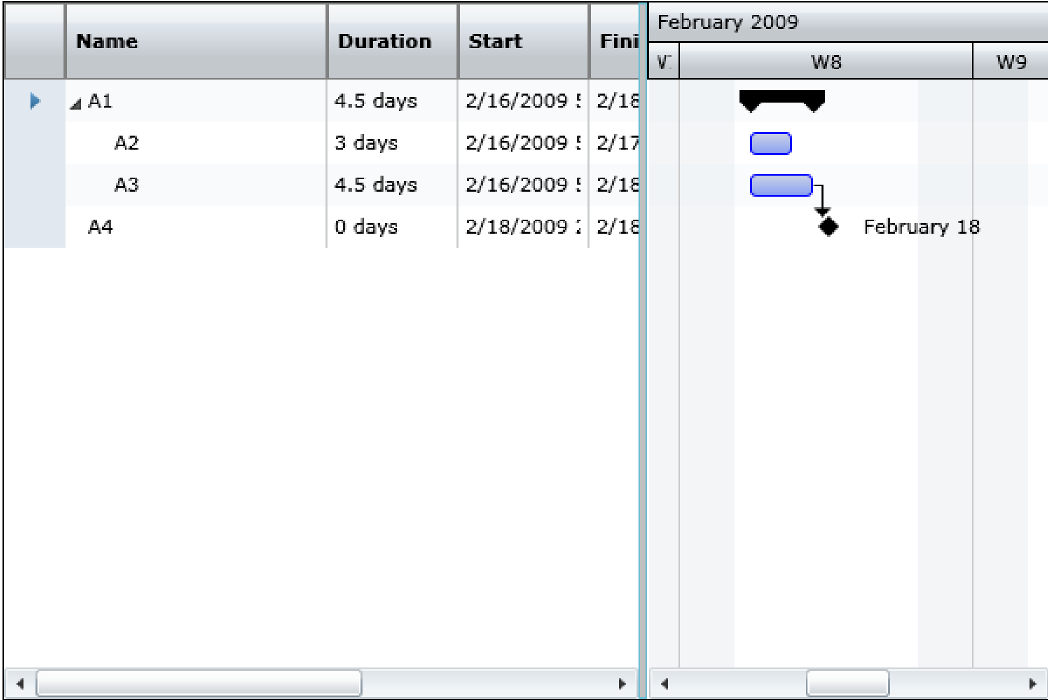
```
<iloggantt:GanttChart x:Name="gantt"/>
```

And in the code-behind:

```
SimpleGanttModel model = new SimpleGanttModel();

SimpleActivity activity1
    = new SimpleActivity() { Name = "A1",
                             StartTime = DateTime.Now,
                             Duration = TimeSpan.FromHours(8)};

SimpleActivity activity2
    = new SimpleActivity() { Name = "A2",
                             StartTime = DateTime.Now,
                             Duration = TimeSpan.FromHours(24),
                             Parent = activity1};
SimpleActivity activity3
    = new SimpleActivity() { Name = "A3",
                             StartTime = DateTime.Now,
                             Duration = TimeSpan.FromHours(36),
                             Parent = activity1};
SimpleActivity activity4
    = new SimpleActivity() { Name = "A4",
                             StartTime = DateTime.Now.AddHours(45),
                             IsMilestone = true };


model.Activities.Add(activity1);
model.Activities.Add(activity2);
```

```
model.Activities.Add(activity3);
model.Activities.Add(activity4);

SimpleConstraint constraint
    = new SimpleConstraint() { FromActivity = activity3,
                               ToActivity = activity4 };

model.Constraints.Add(constraint);

gantt.InitializeFrom(model);
```

This gives the following result:



The **InitializeFrom** extension method is very convenient for initializing the **GanttChart** component, but you can also create a **GanttChart** component and define yourself the columns. For more details on how to do this, see *Working with Tables in the Gantt Controls*. To define yourself the shape of the bars see *Specifying and Styling the Gantt Bars to Display*. Finally, to connect a **SimpleGanttModel** to display the activities you need to set the ItemsSource property of the **GanttChart** to the activities of the model and the ConstraintsSource property to the constraints of the model. To do this, use the following code extract:

```
chart.ItemsSource = model.Activities;
chart.ConstraintsSource = model.Constraints;
```

```
chart.ConstraintTypeBinding = new Binding("Type");
chart.ConstraintOriginPropertyName = "FromActivity";
chart.ConstraintDestinationPropertyName = "ToActivity";
chart.ParentBinding = new Binding("Parent");
```

For more details on how to connect a **GanttChart** control to data, see *Using Your Own Data or Creating your own Data Source for the GanttChart*.

Using Your Own Data or Creating your own Data Source for the GanttChart

A GanttChart control can display any kind of time related items including instances of your own classes. In this section you will see how to implement your data source and connect it to a **GanttChart** control.

In This Section

Creating Task Classes and Displaying them in a GanttChart

Explains how to create task ckasses and display them in a **GanttChart**.

Establishing Parent-Child Relationship

Explains how to establish parent-child relationship.

Displaying Constraints between Tasks in a GanttChart Control

Describes how to display constraints between tasks in a **GanttChart** control.

Creating Task Classes and Displaying them in a GanttChart

The GanttChart control takes as primary data source any kind of enumeration. For example, it can be a collection, an array, or the result of a LinQ query. The items in this collection must define a start and end date since the primary goal of the **GanttChart** is to represent and edit the time interval of each item. The items to display in the **GanttChart** are then specified through the ItemsSource property.

The minimum implementation for an item to display in a **GanttChart** could be:

```
public class MyTask
{
   public DateTime Start { get; set; }
   public DateTime End { get; set; }
}
```

The **Start** and **End** property could be named differently. The important point is that there are two properties of type `DateTime`. Instances of `MyTask` can then be placed in a collection and displayed in a **GanttChart**:

In XAML

```
<iloggantt:GanttChart x:Name="gantt"/>
```

In the code behind:

```
List<MyTask> list = new List<MyTask>();
list.Add(new MyTask() { Start = DateTime.Now,
                        End = DateTime.Now.AddDays(1)});
list.Add(new MyTask() { Start = DateTime.Now,
                        End = DateTime.Now.AddDays(2) });
gantt.ItemsSource = list;
```

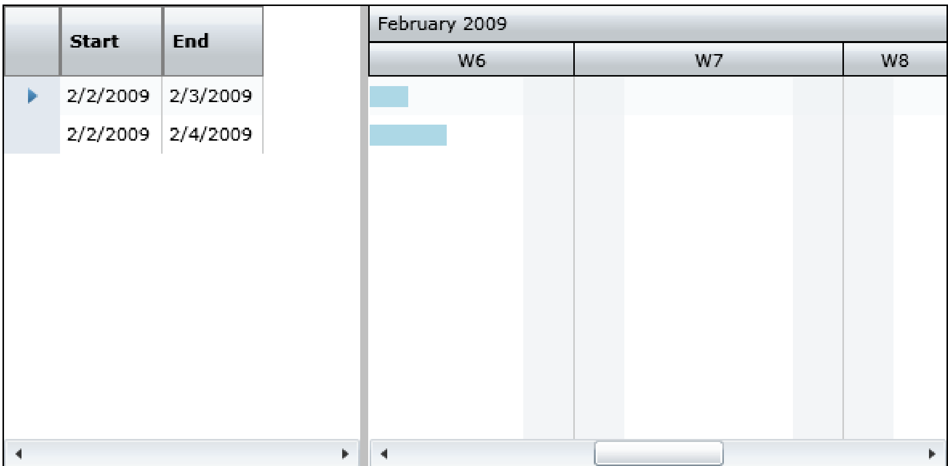
Note that even though the Gantt control is connected to data, by default a **GanttChart** has no column defined and no representation for bars. Therefore the code above will not display anything. Before improving the data source, you will modify the XAML so that the data can be displayed.

To start displaying the items you need to add columns to the table part of the Gantt and define a BarDefinition object that specifies the shape of bars on the time line.

In the following XAML definition two columns are added: one is bound to the `Start` property, the other one is bound to the `End` property. In this XAML you also add a **BarDefinition** object which indicates that a bar should be displayed from the `Start` to the `End` property.

```
<iloggantt:GanttChart x:Name="gantt">

  <!-- Specifies Columns -->
  <iloggantt:GanttChart.Columns>
    <iloggantt:TreeTableDateTimeColumn
       Header="Start"
       Binding="{Binding Start}" />
    <iloggantt:TreeTableDateTimeColumn
       Header="End"
       Binding="{Binding End}" />
  </iloggantt:GanttChart.Columns>

  <!-- Specifies bars to display -->
  <iloggantt:GanttChart.BarDefinitions>
    <iloggantt:BarDefinition
       StartBinding="{Binding Start}"
       FinishBinding="{Binding End}">
      <iloggantt:BarDefinition.BarTemplate>
        <DataTemplate>
          <Rectangle Height="13" Fill="LightBlue" />
        </DataTemplate>
      </iloggantt:BarDefinition.BarTemplate>
    </iloggantt:BarDefinition>
  </iloggantt:GanttChart.BarDefinitions>

</iloggantt:GanttChart>
```

With this specification, you will get the following result:



For each element in the **ItemsSource**, a row is created in the **GanttChart** and a bar appears displayed from the Start time to the End time.

For more information on columns, see *Working with Tables in the Gantt Controls*.

For more information on the **BarDefinition** class, see *Specifying and Styling the Gantt Bars to Display*.

Let's come back to the definition of the data source. The source of data has been defined as a `List` of `MyTask`. With such a definition, the **GanttChart** control is not able to react when an item is added or removed from the list. To see the items that are added or removed from the list, the collection specified in the **ItemsSource** must implement the **INotifyCollectionChanged** interface. Silverlight proposes the generic class **ObservableCollection<T>** as a default implementation of **INotifyCollectionChanged**. You can rewrite the code as follows:

```
ObservableCollection<MyTask> list = new ObservableCollection<MyTask>();
list.Add(new MyTask() { Start = DateTime.Now,
                        End = DateTime.Now.AddDays(1) });
list.Add(new MyTask() { Start = DateTime.Now,
                        End = DateTime.Now.AddDays(2) });
gantt.ItemsSource = list;
```

When tasks are added or removed from the list, the **GanttChart** control will reflect the changes.

Similarly, if the `Start` and `End` properties of `MyTask` change and the changes are reflected in the **GanttChart**, you must implement the interface **INotifyPropertyChanged**. The code for `MyTask` would then become the following:

```
public class MyTask : INotifyPropertyChanged
{
```

```
  private DateTime _start, _end;

  public DateTime Start
  {
    get {
      return _start;
    }
    set {
      if (_start != value) {
        _start = value;
        OnPropertyChanged("Start");
      }
    }
  }

  public DateTime End
  {
    get {
      return _end;
    }

    set {
      if (_end != value) {
        _end = value;
        OnPropertyChanged("End");
      }
    }
  }

  public event PropertyChangedEventHandler PropertyChanged;

  private void OnPropertyChanged(string propertyName)
  {
    if (PropertyChanged != null)
      PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
  }

}
```

Now that `Start` and `End` properties are notifying their changes, the **GanttChart** control can react to these changes and modify the display accordingly. It is also possible to edit the `Start` and `End` properties in the table and move the bars.

Establishing Parent-Child Relationship

Instead of a flat list of tasks, the GanttChart can display a tree of tasks to handle the case of a parent task that is defined by a set of sub-tasks. This can be done by specifying a parent-child relationship in the **GanttChart** through the ParentBinding property of the **GanttChart**.

In this example, the `ParentTask` property is added to define the relation from a task to its parent. The **Name** property is also added and will be displayed in the Gantt.

```
public class MyTask : INotifyPropertyChanged
{
```

```
private DateTime _start, _end;
private string _name;
private MyTask _parent;

public string Name
{
  get {
    return _name;
  }
  set {
    if (_name!= value) {
      _ name = value;
      OnPropertyChanged("Name");
    }
  }
}

public MyTask ParentTask
{
  get {
    return _parent;
  }
  set {
    if (_parent != value) {
      _parent = value;
      OnPropertyChanged("ParentTask");
    }
  }
}

public DateTime Start
{
  get {
    return _start;
  }
  set {
    if (_start != value) {
      _start = value;
      OnPropertyChanged("Start");
    }
  }
}

public DateTime End
{
  get {
    return _end;
  }

  set {
    if (_end != value) {
      _end = value;
      OnPropertyChanged("End");
    }
  }
}

public event PropertyChangedEventHandler PropertyChanged;
```

```
  private void OnPropertyChanged(string propertyName)
  {
    if (PropertyChanged != null)
      PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
  }

}
```

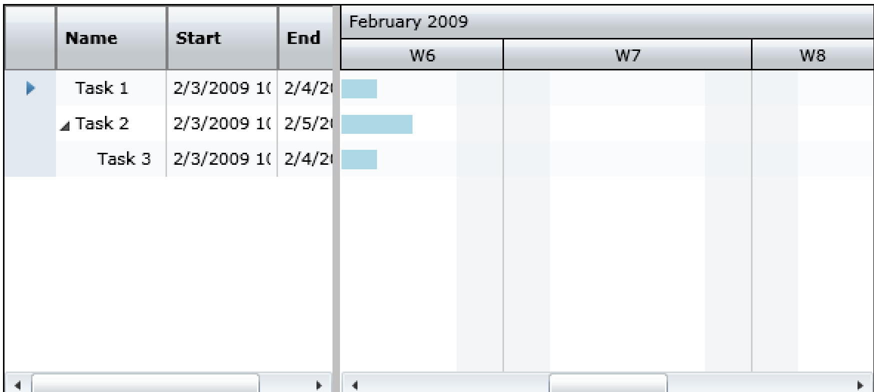
Modify the XAML definition to specify the parent-child relationship through the **ParentBinding** property and add a new column that displays the name of tasks. This new column has the IsTreeColumn property set to `true`. This transforms the column in a tree that enables the expanding and collapsing of elements in the table.

```
<iloggantt:GanttChart x:Name="gantt" ParentBinding="{Binding ParentTask}">
  <iloggantt:GanttChart.Columns>
    <iloggantt:TreeTableTextColumn IsTreeColumn="True"
                                   Header="Name"
                                   Binding="{Binding Name}" />
    <iloggantt:TreeTableDateTimeColumn Header="Start"
                                       Binding="{Binding Start}" />
    <iloggantt:TreeTableDateTimeColumn Header="End"
                                       Binding="{Binding End}" />

  </iloggantt:GanttChart.Columns>

  <iloggantt:GanttChart.BarDefinitions>
    <iloggantt:BarDefinition StartBinding="{Binding Start}"
                             FinishBinding="{Binding End}">
      <iloggantt:BarDefinition.BarTemplate>
        <DataTemplate>
          <Rectangle Height="13" Fill="LightBlue" />
        </DataTemplate>
      </iloggantt:BarDefinition.BarTemplate>
    </iloggantt:BarDefinition>
  </iloggantt:GanttChart.BarDefinitions>
</iloggantt:GanttChart>
```

Change the code-behind to use the new `ParentTask` property:

```
ObservableCollection<MyTask> list = new ObservableCollection<MyTask>();
MyTask task1 = new MyTask() { Name = "Task 1",
                              Start = DateTime.Now,
                              End = DateTime.Now.AddDays(1) };
MyTask task2 = new MyTask() { Name = "Task 2",
                              Start = DateTime.Now,
                              End = DateTime.Now.AddDays(2) };
MyTask task3 = new MyTask() { Name = "Task 3",
                              ParentTask = task2,
                              Start = DateTime.Now,
                              End = DateTime.Now.AddDays(1) };

list.Add(task1);
list.Add(task2);
list.Add(task3);
gantt.ItemsSource = list;
```

This code will give the following result:



The task named `Task2` has now an arrow button that enables the expanding or collapsing of its children.

Note that the bar of the parent task is represented in the same manner as the child tasks. This is due to the fact that you have a single **BarDefinition** defined in the **GanttChart**. To learn how to specify different representations for various kinds of bars see *Specifying and Styling the Gantt Bars to Display*.

Displaying Constraints between Tasks in a GanttChart Control

The GanttChart control can also display links between tasks. Usually, these links are used to display predecessor constraints between tasks. The following table shows the types of constraint defined by the ConstraintType enumeration.

Constraint Type	Example	Description
EndToStart		The activity B cannot start until the activity A is finished.
StartToStart		The activity B cannot start until the activity A is started.
StartToEnd		The activity B cannot finish until the activity A starts.
EndToEnd		The activity B cannot finish until the activity A finishes.

To display constraint links in the **GanttChart** you must specify the list of constraints in the ConstraintsSource property of the **GanttChart**. This property should contain a collection of objects that represent the constraint links between objects. A link will be created for each element in the **ConstraintsSource** collection. In addition to the **ConstraintsSource** property, you must specify the properties ConstraintOriginPropertyName and ConstraintDestinationPropertyName. Through these properties, the **GanttChart** finds the origin and destination tasks for the links. These properties usually refer to members of the constraint objects in the constraint source. Finally a binding must be specified to tell the **GanttChart** what is the type of the constraint link. The ConstraintTypeBinding property of the **GanttChart** provides a binding from a constraint object in the **ConstraintsSource** to a **ConstraintType**.

Just like the ItemsSource property, if the collection specified in the **ConstraintsSource** property implements **INotifyCollectionChanged**, the **GanttChart** is able to reflect changes in the collection when constraints are added or removed. The constraint also should

implement **INotifyPropertyChanged** if you want the **GanttChart** to react to changes of the constraint properties.

Create a constraint object named `MyConstraint`, with a property named `Origin` for the origin of the constraint, `Destination` for the destination of the constraint link and `Type` for the type of the constraint.

```
public class MyConstraint : INotifyPropertyChanged
{
  private MyTask _origin, _destination;
  private ConstraintType _type;

  public MyTask Origin
  {
    get {
      return _origin;
    }

    set {
      if (_origin != value) {
        _origin = value;
        OnPropertyChanged("Origin");
      }
    }
  }

  public MyTask Destination
  {
    get {
      return _destination;
    }

    set {
      if (_destination != value)
      {
        _destination = value;
        OnPropertyChanged("Destination");
      }
    }
  }

  public ConstraintType Type
  {
    get {
      return _type;
    }

    set {
      if (_type != value) {
        _type = value;
        OnPropertyChanged("Type");
      }
    }
  }

  public event PropertyChangedEventHandler PropertyChanged;
```

```
  private void OnPropertyChanged(string propertyName)
  {
    if (PropertyChanged != null)
      PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
  }
}
```

You can now modify the XAML definition of the **GanttChart**:

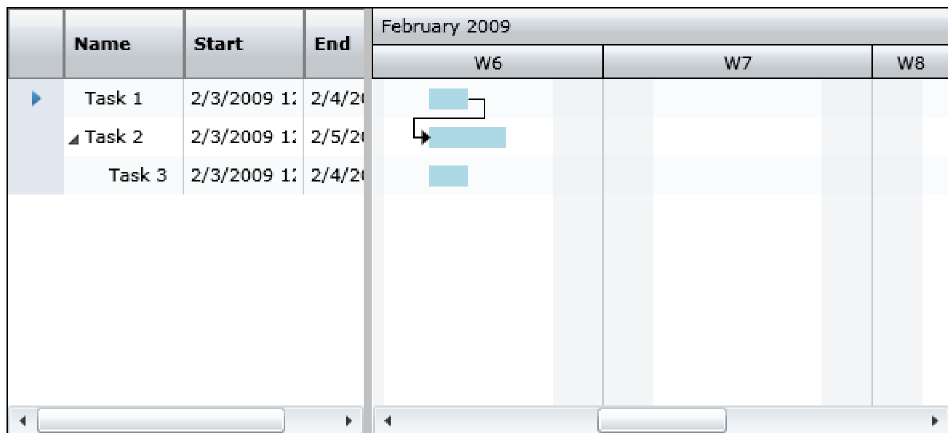
```
<iloggantt:GanttChart x:Name="gantt"
       ParentBinding="{Binding ParentTask}"
       ConstraintOriginPropertyName="Origin"
       ConstraintDestinationPropertyName="Destination"
       ConstraintTypeBinding="{Binding Type}" >
  <iloggantt:GanttChart.Columns>
    <iloggantt:TreeTableTextColumn IsTreeColumn="True"
                                   Header="Name"
                                   Binding="{Binding Name}" />
    <iloggantt:TreeTableDateTimeColumn Header="Start"
                                       Binding="{Binding Start}" />
    <iloggantt:TreeTableDateTimeColumn Header="End"
                                       Binding="{Binding End}" />

  </iloggantt:GanttChart.Columns>

  <iloggantt:GanttChart.BarDefinitions>
    <iloggantt:BarDefinition StartBinding="{Binding Start}"
                             FinishBinding="{Binding End}">
      <iloggantt:BarDefinition.BarTemplate>
        <DataTemplate>
          <Rectangle Height="13" Fill="LightBlue" />
        </DataTemplate>
      </iloggantt:BarDefinition.BarTemplate>
    </iloggantt:BarDefinition>
  </iloggantt:GanttChart.BarDefinitions>
</iloggantt:GanttChart>
```

In the code-behind, create a constraint and specify the constraint source:

```
ObservableCollection<MyTask> list = new ObservableCollection<MyTask>();
MyTask task1 = new MyTask() { Name = "Task 1",
                              Start = DateTime.Now,
                              End = DateTime.Now.AddDays(1) };
MyTask task2 = new MyTask() { Name = "Task 2",
                              Start = DateTime.Now,
                              End = DateTime.Now.AddDays(2) };
MyTask task3 = new MyTask() { Name = "Task 3",
                              ParentTask = task2,
                              Start = DateTime.Now,
                              End = DateTime.Now.AddDays(1) };

list.Add(task1);
list.Add(task2);
list.Add(task3);
gantt.ItemsSource = list;

ObservableCollection<MyConstraint> constraints =
        new ObservableCollection<MyConstraint>();
```

```
MyConstraint constraint = new MyConstraint() {
              Origin = task1,
              Destination = task2,
              Type = ConstraintType.EndToStart };

constraints.Add(constraint);

gantt.ConstraintsSource = constraints;
```

This gives the following graphical result:



Note that the **GanttChart** control does not do any validation or scheduling logic on your data. For example, it will not enforce that a task with a predecessor constraint of type End-to-Start starts after the end of its predecessor. Also, there is no specific logic for the duration of a parent task relatively to its child tasks. Such logic should be added to the data themselves. Alternatively, you can use the predefined data sources like SimpleGanttModel or ProjectSchedulingModel.

Connecting a ScheduleChart Control to Data

A ScheduleChart control can display any kind of time related items including instances of your own classes. Some predefined data sources are available so that you do not need to create your own data source.

You will learn how to use predefined data sources, how to create a new data source that can be used in a **ScheduleChart** control and how to connect the data sources to a **ScheduleChart**.

In This Section

Using Predefined Data Sources in a ScheduleChart

Explains how to use predefined data sources to display data in a **ScheduleChart** control.

Using Your Own Data or Creating your own Data Source for the ScheduleChart

Explains how to create a data source and connect it to a **ScheduleChart** control.

Using Predefined Data Sources in a ScheduleChart

To display data in a ScheduleChart control, you may use one of the predefined data sources, the SimpleGanttModel class or the ProjectSchedulingModel classes. These two data models are described in more details in *Using Predefined Data Models*.

These data models can be easily connected to a **ScheduleChart** so that the **ScheduleChart** displays the resources in the model and the activities assigned to each resource (also know as reservations).

To connect a **ScheduleChart** to a **SimpleGanttModel** use the following code:

```
ScheduleChart chart = new ScheduleChart();
SimpleGanttModel model = new SimpleGanttModel();
chart.InitializeFrom(model);
```

The InitializeFrom method is an extension method for the **ScheduleChart** class and does the following initializations:

- Creation of some predefined columns in the table: A column for Name.
- Creation of predefined bar definitions: A bar definition for normal activity, for activity completion, for summaries and for milestones.
- Connection of the data model to the control for displaying the resources in the table and the activities assigned to each resource.

Here is an extract of C# code that creates and populates a **SimpleGanttModel** with four activities and one constraint and displays the model in a **ScheduleChart**.

In XAML you simply add a **ScheduleChart** to the page:

```
<iloggantt: ScheduleChart x:Name="gantt"/>
```

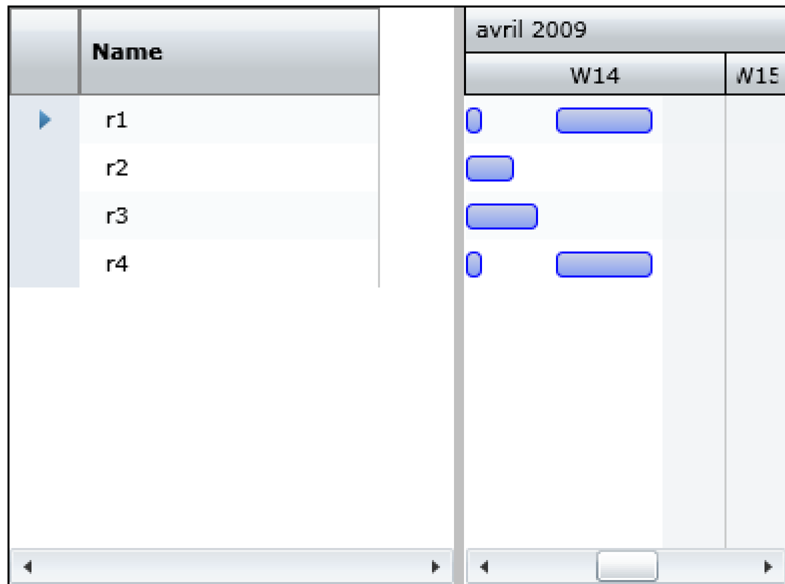
And in the code-behind:

```
SimpleGanttModel model = new SimpleGanttModel();


// create the resources

SimpleResource resource1 = new SimpleResource()
{
    Name = "r1",
};

SimpleResource resource2 = new SimpleResource()
{
     Name = "r2",

};

SimpleResource resource3 = new SimpleResource()
{
```

```
    Name = "r3",
};

SimpleResource resource4 = new SimpleResource()
{
    Name = "r4",
};

model.Resources.Add(resource1);
model.Resources.Add(resource2);
model.Resources.Add(resource3);
model.Resources.Add(resource4);

// create the activities. and assign them to resources.

SimpleActivity activity1
                = new SimpleActivity()
                {
                    Name = "A1",
                    StartTime = DateTime.Now,
                    Duration = TimeSpan.FromHours(8)
                };

SimpleActivity activity2
                = new SimpleActivity()
                {
                    Name = "A2",
                    StartTime = DateTime.Now,
                    Duration = TimeSpan.FromHours(24),
                };
SimpleActivity activity3
                = new SimpleActivity()
                {
                    Name = "A3",
                    StartTime = DateTime.Now,
                    Duration = TimeSpan.FromHours(36),

                };
SimpleActivity activity4
                = new SimpleActivity()
                {
                    Name = "A4",
                    StartTime = DateTime.Now.AddHours(45),
                    Duration = TimeSpan.FromHours(48),
                };


model.Activities.Add(activity1);
model.Activities.Add(activity2);
model.Activities.Add(activity3);
model.Activities.Add(activity4);

SimpleReservation reservation1 = new SimpleReservation()
            {
                Resource = resource1,
                Activity = activity1
            };
SimpleReservation reservation2 = new SimpleReservation()
```

```
            {
                Resource = resource2,
                Activity = activity2
            };
SimpleReservation reservation3 = new SimpleReservation()
            {
                Resource = resource3,
                Activity = activity3
            };
SimpleReservation reservation4 = new SimpleReservation()
            {
                Resource = resource4,
                Activity = activity4
            };
SimpleReservation reservation5 = new SimpleReservation()
            {
                Resource = resource1,
                Activity = activity4
            };
SimpleReservation reservation6 = new SimpleReservation()
            {
                Resource = resource4,
                Activity = activity1
            };
model.Reservations.Add(reservation1);
model.Reservations.Add(reservation2);
model.Reservations.Add(reservation3);
model.Reservations.Add(reservation4);
model.Reservations.Add(reservation5);
model.Reservations.Add(reservation6);

scheduleChart.InitializeFrom(model);
```

This gives the following result:



The **InitializeFrom** extension method is very convenient for initializing the **ScheduleChart** component, but you can also create a **ScheduleChart** component and define yourself the columns. For more details refer to *Working with Tables in the Gantt Controls*. To define yourself the shape of the bars see *Specifying and Styling the Gantt Bars to Display*. Finally, to connect a **SimpleGanttModel** to display the resources you need to set the ItemsSource property of the **ScheduleChart** to the resources of the model and the ConstraintsSource property to the constraints of the model. To do this, use the following code:

```
chart.ItemsSource = model.Resources;
chart.TimeItemsBinding = new Binding("Reservations");
```

For more details on how to connect a **ScheduleChart** control to data, see *Using Your Own Data or Creating your own Data Source for the ScheduleChart*.

Using Your Own Data or Creating your own Data Source for the ScheduleChart

In this section you will see how to implement a data source so that it can be connected to a ScheduleChart control.

Creating Resource Classes and Displaying them in the ScheduleChart

The **ScheduleChart** control takes as primary data source any kind of enumeration. For example, it can be a collection, an array or the result of a LinQ query. The items in this

collection will be represented as a row in the **ScheduleChart**, for each item (that is, for each row) the **ScheduleChart** control can display several items over the time. For example, the main collection can be a list of persons and for each person, you may display the various tasks assigned to this person. Another example could be a list of TV channel and for each channel the various programs along the time. Here is a minimal implementation showing persons and tasks assigned to persons. The class `MyPerson` and `MyTask` may look like this:

```
public class MyTask
{
   public DateTime Start { get; set; }
   public DateTime End { get; set; }
}

public class MyPerson
{
   List<MyTask> _tasks = new List<MyTask>();
   public string Name { get; set; }
   public List<MyTask> Tasks { get { return _tasks; } }
}
```

The `Start` and `End` property of the class `MyTask` could be named differently. The important point is that there are two properties of type `DateTime` so that the task can be displayed along the time line. You can now create several persons with several tasks assigned.

In XAML

```
<iloggantt:ScheduleChart x:Name="scheduleChart"/>
```

In the code behind:

```
List<MyPerson> persons = new List<MyPerson>();
MyPerson person1 = new MyPerson() { Name = "person 1" };
person1.Tasks.Add(new MyTask() { Start = DateTime.Now,
                                 End = DateTime.Now.AddDays(2) });
person1.Tasks.Add(new MyTask() { Start = DateTime.Now.AddDays(3),
                                 End = DateTime.Now.AddDays(5) });

MyPerson person2 = new MyPerson() { Name = "person 2" };
person2.Tasks.Add(new MyTask() { Start = DateTime.Now.AddDays(1),
                                 End = DateTime.Now.AddDays(2) });
person2.Tasks.Add(new MyTask() { Start = DateTime.Now.AddDays(4),
                                 End = DateTime.Now.AddDays(7) });

MyPerson person3 = new MyPerson() { Name = "person 3" };
person3.Tasks.Add(new MyTask() { Start = DateTime.Now,
                                 End = DateTime.Now.AddDays(2) });
person3.Tasks.Add(new MyTask() { Start = DateTime.Now.AddDays(3),
                                 End = DateTime.Now.AddDays(4) });


persons.Add(person1);
persons.Add(person2);
persons.Add(person3);

scheduleChart.ItemsSource = persons;
scheduleChart.TimeItemsBinding = new Binding("Tasks");
```

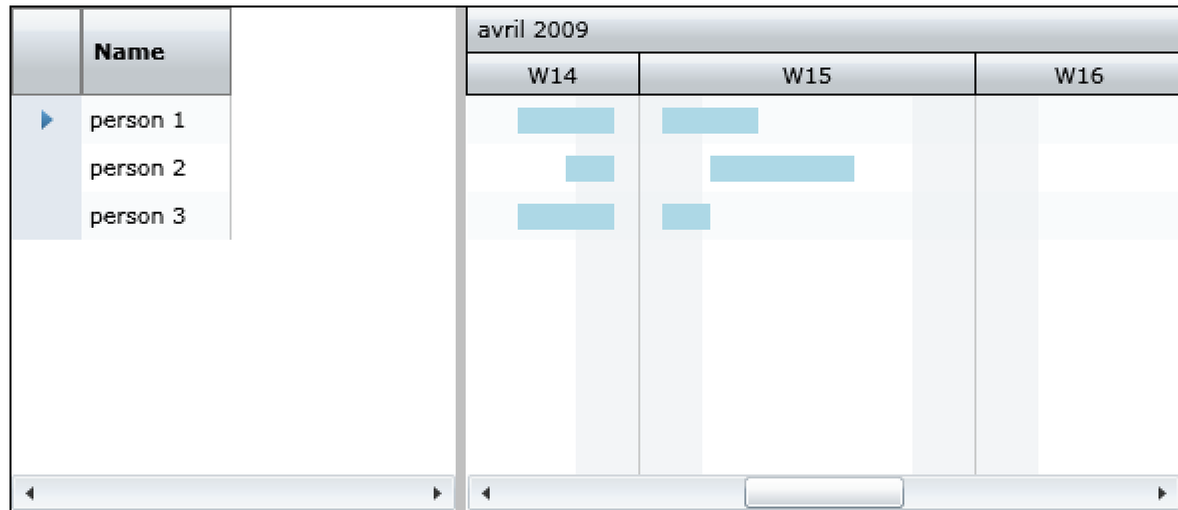
Note that even though the **ScheduleChart** control is connected to data, by default it has no column defined and no representation for bars. So the code above will not display anything. Before improving the data source, you will modify the XAML so that the data can be displayed.

To start displaying the items you need to add columns to the table part of the Gantt and define a BarDefinition object that specifies the shape of bars on the time line.

In the following XAML definition one column is added and is bound to the `Name` property. In this XAML you also add a **BarDefinition** object which indicates that a bar should be displayed from the `Start` to the `End` property of a task.

```
<iloggantt:ScheduleChart x:Name="scheduleChart">

  <!-- Specifies Columns for the Gantt table-->
  <iloggantt:ScheduleChart.Columns>
    <iloggantt:TreeTableTextColumn Header="Name" Binding="{Binding Name}"/>
  </ iloggantt:ScheduleChart.Columns>

  <!-- bar definitions-->
  <iloggantt:ScheduleChart.BarDefinitions>
    <iloggantt:BarDefinition
       StartBinding="{Binding Start}"
       FinishBinding="{Binding End}">
      <iloggantt:BarDefinition.BarTemplate>
        <DataTemplate>
          <Rectangle Height="13" Fill="LightBlue" />
        </DataTemplate>
      </iloggantt:BarDefinition.BarTemplate>
    </iloggantt:BarDefinition>
  </iloggantt:ScheduleChart.BarDefinitions>
</iloggantt:ScheduleChart>
```

With this specification, you will get the following result:



For each person in the **ItemsSource**, a row is created in the **ScheduleChart**. For each person the tasks (specified by the **Tasks** properties) are drawn along the time line.

For more information on columns, see Working with Gantt chart columns. For more information on the **BarDefinition** class, see *Specifying and Styling the Gantt Bars to Display*.

Let's come back to the definition of the data source. The source of data has been defined as a `List` of `MyPerson`. With such a definition, the **ScheduleChart** control is not able to react when a person is added or removed from the list. To see the items that are added or removed from the list, the collection specified in the **ItemsSource** must implement the **INotifyCollectionChanged** interface. Silverlight proposes the generic class **ObservableCollection<T>** as a default implementation of **INotifyCollectionChanged**.

Similarly the `Name` and `Tasks` properties of `MyPerson` may change and in order to see the changes in the **ScheduleChart**, the class `MyPerson` should implement **INotifyPropertyChanged**, and the collection of tasks should implement **INotifyCollectionChanged**.

Similarly if the `Start` and `End` properties of `MyTask` change and the changes are reflected in the **ScheduleChart**, you must implement the interface **INotifyPropertyChanged**. The code for `MyTask` and `MyPerson` become

```
public class MyTask : INotifyPropertyChanged
{
  private DateTime _start, _end;

  public DateTime Start
  {
    get {
```

```
      return _start;
    }
    set {
      if (_start != value) {
        _start = value;
        OnPropertyChanged("Start");
      }
    }
  }

  public DateTime End
  {
    get {
      return _end;
    }

    set {
      if (_end != value) {
        _end = value;
        OnPropertyChanged("End");
      }
    }
  }

  public event PropertyChangedEventHandler PropertyChanged;

  private void OnPropertyChanged(string propertyName)
  {
    if (PropertyChanged != null)
      PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
  }

}

public class MyPerson : INotifyPropertyChanged
{
  private ObservableCollection<MyTask> _tasks
              = new ObservableCollection<MyTask>();
  private string _name;

  public string Name
  {
    get
    {
      return _name;
    }
    set
    {
      if (_name != value)
      {
        _name = value;
        OnPropertyChanged("Name");
      }
    }
  }

  public ICollection<MyTask> Tasks { get { return _tasks; } }
```

```
  public event PropertyChangedEventHandler PropertyChanged;

  private void OnPropertyChanged(string propertyName)
  {
    if (PropertyChanged != null)
      PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
  }
}
```

Now that the properties `Name`, `Start` and `End` are notifying their changes, the **ScheduleChart** control can react to these changes and modify the display accordingly. It is now also possible to move the bars with the mouse pointer.

Using Predefined Data Models

Although the GanttChart and ScheduleChart controls can be connected to any kind of data, IBM® ILOG® Gantt for .NET provides two predefined data model that can be easily connected to the Silverlight Gantt controls:

The SimpleGanttModel contains definitions for the activities (or tasks) in the project, for the resources that will be used to perform the activities, for constraints between tasks and for reservations (an assignment of a resource to an activity).

The ProjectSchedulingModel defines the same type of information as the **SimpleGanttModel** but adds project scheduling algorithms that will automatically maintain the schedule of the project by computing the critical path and performing resource leveling to remove resource overallocations. The two data models are located in the **ILOG.Controls.Gantt.Data.dll** assembly.

In This Section

The SimpleGanttModel Data Model

Explains how to use the SimpleGanttModel class to manipulate scheduling data.

The ProjectSchedulingModel Data Model

Describes how to use the ProjectSchedulingModel to create Silverlight applications with project scheduling capabilities.

The SimpleGanttModel Data Model

IBM® ILOG® Gantt for .NET provides a ready-to-use implementation of the Gantt data model through the SimpleGanttModel class. This Gantt data model can be used to manipulate standard scheduling data directly or can be extended to meet specific requirements. It can be easily connected to the Silverlight Gantt controls.

The SimpleGanttModel class

The **SimpleGanttModel** gathers the Gantt model entities in a single object. Its main role is to maintain consistency in the model and to ease initialization of the Silverlight Gantt controls.

The in-memory data model implementation is defined by the following classes:

Class	Description
SimpleGanttModel	Gathers the activities, resources, constraints, and reservations in the same class.
SimpleActivity	A basic activity implementation.
SimpleResource	A basic resource implementation.
SimpleConstraint	A basic constraint implementation.
SimpleReservation	A basic reservation implementation.

All these classes are located in the ILOG.Controls.Gantt.Data namespace, in the **ILOG.Controls.Gantt.Data.dll** assembly.

Activities in the SimpleGanttModel

Activities in the **SimpleGanttModel** are defined by the **SimpleActivity** class. A **SimpleGanttModel** handles a flat collection of **SimpleActivity** objects. This collection can be accessed through the SimpleGanttModel.Activities property. To add or remove activities, use the standard collection methods as shown below:

```
SimpleGanttModel model = new SimpleGanttModel();
SimpleActivity activity = new SimpleActivity() {
                              Name = "Task 1",
                              StartTime = DateTime.Now,
                              Duration = TimeSpan.FromHours(8)
                              };
model.Activities.Add(activity);
```

To change the time interval of a **SimpleActivity**, use the StartTime, EndTime, and Duration properties.

An activity with child activities is called *summary activity*. A summary activity is created by setting the Parent property of the children activities to this activity. Note that it is not possible to modify the time interval or summary activities, because it is automatically computed from their children.

The following code shows how to create a summary activity, that is, an activity whose time interval is defined by its children:

```
SimpleGanttModel model = new SimpleGanttModel();
SimpleActivity summary = new SimpleActivity() { Name = "Summary" };
SimpleActivity child1 = new SimpleActivity() {
                         Name = "Sub Task 1",
                         StartTime = DateTime.Now,
                         Duration = TimeSpan.FromHours(8),
                         Parent = summary
                         };
SimpleActivity child2 = new SimpleActivity() {
                         Name = "Sub Task 2",
                         StartTime = DateTime.Now,
                         Duration = TimeSpan.FromHours(8),
                         Parent = summary
                         };
model.Activities.Add(summary);
model.Activities.Add(child1);
model.Activities.Add(child2);
```

The children need to be explicitly added to the activity collection.

Note that setting up an activity hierarchy by setting the **SimpleActivity.Parent** property does not mean that the Silverlight Gantt controls will automatically reflect this hierarchy. To see how to configure the Silverlight Gantt controls to display a hierarchical view, see *Grouping, Sorting and Filtering Data*.

When you remove the activities from the model, the associated reservations and constraints are also removed.

Resources in the SimpleGanttModel

Resources in the **SimpleGanttModel** are defined by the **SimpleResource** class. A **SimpleGanttModel** handles a flat collection of **SimpleResource** objects. This collection can be accessed through the SimpleGanttModel.Resources property. To add or remove resources, use the standard collection methods as shown below:

```
SimpleGanttModel model = new SimpleGanttModel();
SimpleResource resource = new SimpleResource() { Name = "Resource 1" };
model.Resources.Add(resource);
```

To create a hierarchy between resources, use the SimpleResource.Parent property. By default, this property is set to null, which means the resource has no parent. The following code shows how to create a hierarchy of resources:

```
SimpleGanttModel model = new SimpleGanttModel();
SimpleResource parent = new SimpleResource() { Name = "Parent Resource" };
SimpleResource child1 = new SimpleResource() {
                            Name = "Sub Resource 1",
                            Parent = parent
                            };
SimpleResource child2 = new SimpleResource() {
                            Name = "Sub Resource 2",
                            Parent = parent
                            };
model.Resources.Add(parent);
model.Resources.Add(child1);
model.Resources.Add(child2);
```

The children need to be explicitly added to the resource collection.

Note that setting up a resource hierarchy by setting the **SimpleResource.Parent** property does not mean that the Silverlight Gantt controls will automatically reflect this hierarchy. To see how to configure the Silverlight Gantt controls to display a hierarchical view, see *Grouping Data*.

When you remove the resources from the model, the associated reservations are also removed.

Constraints in the SimpleGanttModel

Constraints in the **SimpleGanttModel** are defined by the **SimpleConstraint** class. A **SimpleGanttModel** handles a collection of **SimpleConstraint** objects. This collection can be accessed through the SimpleGanttModel.Constraints property. To add or remove constraints, use the standard collection methods as shown below:

```
SimpleGanttModel model = new SimpleGanttModel();
SimpleActivity activity1 = new SimpleActivity() {
                        Name = "Task 1",
                        StartTime = DateTime.Now,
                        Duration = TimeSpan.FromHours(8)
                        };
model.Activities.Add(activity1);
SimpleActivity activity2 = new SimpleActivity() {
                        Name = "Task 2",
                        StartTime = DateTime.Now,
                        Duration = TimeSpan.FromHours(8)
                        };
model.Activities.Add(activity2);
SimpleConstraint constraint = new SimpleConstraint() {
                        FromActivity = activity1,
                        ToActivity = activity2
                        };
model.Constraints.Add(constraint);
```

Reservations in the SimpleGanttModel

Reservations in the **SimpleGanttModel** are defined by the **SimpleReservation** class. A **SimpleGanttModel** handles a collection of **SimpleReservation** objects. This collection can be accessed through the SimpleGanttModel.Reservations property. To add or remove reservations, use the standard collection methods as shown below:

```
SimpleGanttModel model = new SimpleGanttModel();
SimpleActivity activity = new SimpleActivity() {
                         Name = "Task",
                         StartTime = DateTime.Now,
                         Duration = TimeSpan.FromHours(8)
                         };
model.Activities.Add(activity);
SimpleResource resource = new SimpleResource() { Name = "Resource" };
model.Resources.Add(resource);
SimpleReservation reservation = new SimpleReservation() {
                         Activity = activity,
                         Resource = resource
                         };
Model.Reservations.Add(reservation);
```

The ProjectSchedulingModel Data Model

IBM® ILOG® Gantt for .NET allows you to create Silverlight applications that require project scheduling capabilities through a specific Gantt Data Model class: the ProjectSchedulingModel.

The **ProjectSchedulingModel** stores information about your project and uses this information to calculate and maintain the schedule of your project. The **ProjectSchedulingModel** computes the schedule immediately. As soon as you have entered information about your project, you can learn about the scheduled start date of activities and the project target date.

In the `<installdir>\Samples\Applications\SilverlightProjectEditor` directory, you will find a fully-featured application sample that shows how to use the **ProjectSchedulingModel** in a real project scheduling application.

In This Section

The ProjectSchedulingModel Class

Describes the **ProjectSchedulingModel** class.

Activities in the ProjectSchedulingModel

Describes the SchedulingActivity class.

How the Project Scheduling Model Computes the Schedule of a Project

Describes how the ProjectSchedulingModel class calculates and creates the schedule of a project.

Calendars in the Project Scheduling Model

Explains how to use calendars to define working and nonworking periods.

Resource Leveling in the Project Scheduling Model

Explains the resource leveling process in the Project Scheduling Model.

The ProjectSchedulingModel Class

The ProjectSchedulingModel class contains algorithms for scheduling projects, leveling resources, and computing the critical paths. This option is particularly interesting for developing rapidly project management solutions that can be deployed over the Web.

The **ProjectSchedulingModel** class is a Gantt data model class similar to the SimpleGanttModel that adds project scheduling capabilities. Just like the **SimpleGanttModel**, the **ProjectSchedulingModel** defines the scheduling information that can be edited and displayed by the Silverlight controls of the IBM ILOG Gantt for .NET library.

For adding, removing and accessing activities, resources, constraints and reservations, the **ProjectSchedulingModel** operates as the **SimpleGanttModel**, but for each modification of the model, the **ProjectSchedulingModel** re-computes a working schedule of the project. For example, when you change the duration of an activity in the model, the model re-computes a schedule and this may have an impact on the start time of all the successors activities and on the project scheduled end date.

The following types are involved in the Project Scheduling Data model:

Type	Description
ProjectSchedulingModel	The project scheduling data model that uses the classes listed in this table.
SchedulingActivity	Represents an activity or a task that must be completed in the project.
SchedulingResource	Represents a resource that can be allocated to an activity to make its completion possible.
SchedulingConstraint	Represents an activity-to-activity scheduling constraint.
SchedulingReservation	Represents the allocation of a resource to an activity.

When creating a **ProjectSchedulingModel** you must specify a starting date for the project by using the StartDate property. The project is scheduled from the start date, so activities

that do not have predecessors or other scheduling constraints will be scheduled at the project start date. The project end date will be automatically computed and is available through the EndDate property.

By default, the **ProjectSchedulingModel** schedules activities by using the working times on the project calendar (see WorkCalendar class). The project calendar can be specified using the Calendar property. When creating a **ProjectSchedulingModel**, the model has a standard calendar that defines Saturday and Sunday as nonworking days and working times from 8AM to 12AM and from 1PM to 5PM. Specific calendars can also be defined for resources and activities. For more details see *Calendars in the Project Scheduling Model*.

The following C# code fragment creates a **ProjectSchedulingModel** and adds one activity. Finally, the scheduled start time of the activity and the project target date are displayed.

```
ProjectSchedulingModel project = new ProjectSchedulingModel();
project.StartDate = new DateTime(2005, 1, 1);
SchedulingActivity activity = new SchedulingActivity();
activity.Duration = TimeSpan.FromHours(8);
project.Activities.Add(activity);
```

This C# code schedules the activity to start on: 1/3/2005 8:00:00 AM (`activity.StartTime`) and the project end date (`project.EndDate`) will be 1/3/2005 5:00:00 PM.

The activity has been scheduled to start on January 3, 2005 at 8 AM, even though the project start date is January 1, this is because January 1, 2005 is Saturday.

Activities in the ProjectSchedulingModel

Activities in the ProjectSchedulingModel are defined by the SchedulingActivity class. Since the **ProjectSchedulingModel** automatically computes the schedule of the activity you should not specify a start date for activities. For an activity, it is only mandatory to specify a duration. The schedule of activities will be computed by the model.

The Duration property of the activity represents the amount of work needed to complete the activity. For example, if you are using the default standard calendar, a duration of 8 hours represents in fact one day of elapsed duration since in the default calendar each working day has 8 hour of work (The default calendar has working times from 8 AM to 12PM and from 1PM to 5PM).

The following C# code creates a **ProjectSchedulingModel** with a start date of January 5, 2005 and adds an activity with a duration of 8 hours.

```
ProjectSchedulingModel model = new ProjectSchedulingModel();
model.StartDate = new DateTime(2005,1,5);
SchedulingActivity activity = new SchedulingActivity();
activity.Duration = TimeSpan.FromHours(8);
model.Activities.Add(activity);
```

Once the activity is added to the model, the following properties of the **SchedulingActivity** class are computed by the **ProjectSchedulingModel**:

Property Name	Property Type	Description
StartTime	DateTime	The scheduled start time of the activity.
EndTime	DateTime	The scheduled end time of the activity.
EarlyStart	DateTime	The earliest date the activity can start based on the predecessor and successors of the activity and other scheduling constraints.
EarlyFinish	DateTime	The earliest date the activity can finish based on the predecessors and successors of the activity and other scheduling constraints.
LateStart	DateTime	The latest date the activity can start based on the predecessors and successors of the activity and other scheduling constraints.
LateFinish	DateTime	The latest date the activity can finish based on the predecessors and successors of the activity and other scheduling constraints.
TotalSlack	TimeSpan	The amount of time the activity can be delayed without delaying the project's end date.
FreeSlack	TimeSpan	The amount of time the activity can be delayed without delaying any successor activity.
IsCritical	Boolean	Indicates whether the activity is critical or not.

You should not specify the **StartTime** or **EndTime** properties of a **SchedulingActivity**. Those properties are automatically computed. If you set the value of the **StartTime** property, a constraint of type "Start No Earlier Than" will be set on the activity. If you specify the **EndTime** property, a constraint of type "Finish No Earlier Than" will be set on the activity.

To learn more about constraints and other properties of **SchedulingActivity** that allow you to control the schedule of an activity and how the schedule of the project is computed, see *How the Project Scheduling Model Computes the Schedule of a Project*.

When the automatic resource leveling is turned on, the following properties of the **SchedulingActivity** are computed:

Property Name	Property Type	Description
PreleveledStart	DateTime	Represents the start time of the activity before the resource leveling was computed.
PreleveledEnd	DateTime	Represents the end time of the activity before the resource leveling was computed.
LevelingDelay	TimeSpan	The amount of time the activity is delayed from its early start date (**EarlyStart**) in order to remove resource overallocations.

To learn more about resource leveling, see *Resource Leveling in the Project Scheduling Model*.

You can also specify that an activity has already started by defining and actual start time (SchedulingActivity.ActualStartTime property) and the percentage of completion (SchedulingActivity.WorkComplete property). This information is taken into account by the model when scheduling the activity. Activities that have already started are always scheduled at their actual start time and will not be delayed by the resource leveling algorithm.

How the Project Scheduling Model Computes the Schedule of a Project

As you build a project plan, the ProjectSchedulingModel class calculates and creates a working schedule based on information you provide about the activities to be done and the people assigned to those activities.

The **ProjectSchedulingModel** schedules the project from the information you specify about the project itself: the individual activities (class SchedulingActivity) required to complete the project and, if necessary, the resources (class SchedulingResource) needed to complete those activities. If anything about your project changes after you create your schedule, you can update the activities or resources and the **ProjectSchedulingModel** adjusts the schedule for you.

For each activity, you enter duration, dependencies between activities (class SchedulingConstraint), and activity constraints (type ActivityConstraintType), then the **ProjectSchedulingModel** calculates the start and end date for each activity. You can enter resources in your project and then assign them to activities to indicate which resource is responsible for completing each activity (class SchedulingReservation). If you enter resources, task schedules are further refined according to resource availability and working times entered on calendars (class WorkCalendar). Other elements, such as lead time and lag time on dependencies can affect the scheduling. Understanding the effects of these elements can help you maintain and adjust the project schedule.

Controlling When the Project Schedule is Recomputed

Since every modification of the model leads to a re-computation of the schedule of the project, it is important to control when the schedule is re-computed in order to avoid unnecessary re-computations.

For example, if you want to change the duration of several activities, you do not want the model to re-compute the schedule several times.

The **ProjectSchedulingModel** provides the BeginScheduleSession and EndScheduleSession methods that allow you to group several modifications so that only one single schedule will be computed. Here is the typical C# code that you would write:

```
try {
    model.BeginScheduleSession();

    // change several things in the model
} finally {
    model.EndScheduleSession();
}
```

The **ProjectSchedulingModel** class also provides the BeginSchedule and EndSchedule events that are fired when the scheduling starts and finishes.

How Project Start Date Affects the Schedule

The **ProjectSchedulingModel** schedules your project according to the project start date. The start date of the project can be specified by the StartDate property of the **ProjectSchedulingModel**. When an activity is added to the model, it is initially scheduled at the project start date. Later on, if you add predecessors to the activity or if you set other constraints on the activity, the schedule start time of the activity will be computed accordingly. The project end date is computed like the latest end date of all activities in the model and can be retrieved using the EndDate property of the **ProjectSchedulingModel** class.

You may change the project start date at any time and the **ProjectSchedulingModel** will re-compute a new schedule based on this new start date.

How Constraint Links Affect the Schedule

When a new activity is inserted in the **ProjectSchedulingModel**, the activity has no predecessor constraint and will be scheduled to start at the project start date.

Later on, predecessor constraints can be added between activities using the **SchedulingConstraint** class that defines precedence constraints between activities.

See *Displaying Constraints between Tasks in a GanttChart Control* for details on the different types of precedence constraints defined by the **ConstraintType** enumeration.

Lead and Lag Time

The **SchedulingConstraint** class also allows you to specify a lead or a lag time for the constraint.

A lag time is a delay between the end of an activity and the start of its successor. A lead time is an overlap between the two activities, so that the successor starts before the end of the predecessor.

To specify a lag or lead time in the **SchedulingConstraint** use the Lag and LagFormat property. The lead and lag time can be expressed in various formats: it can be defined as a work duration or an elapsed duration. The duration may be directly specified or defined as a percentage of the duration of the predecessor activity. For example, to create an **EndToStart** constraint between activity A and B with a lag expressed as 50% of the elapsed duration of A (assuming that the variable `a` and `b` are instances of the **SchedulingActivity** class), you would do:

```
SchedulingConstraint c = new SchedulingConstraint()
       {FromActivity = a,
        ToActivity = b,
        Type = ConstraintType.EndToStart};
c.Lag = 50;
c.LagFormat = LagFormat.EllapsedPercentage;
model.Constraints.Add(c);
```

To express a lead time, you would give a negative value to the **Lag** property.

How Constraints on Activities Affect the Schedule

The schedule start date of an activity is mainly controlled by the dependencies between activities, as explained in *How Constraint Links Affect the Schedule*.

However, the scheduled start date of an activity can also be controlled using constraints on activity. Constraints on activities are set and retrieved through the Constraint property of the **SchedulingActivity** class. The different types of constraints that can be specified on an activity are defined by the ActivityConstraintType enumeration. By default, when creating an activity, the constraint on the activity is set to **AsSoonAsPossible**. This means that the

activity will be scheduled as soon as possible. For example, if the activity has several end-to-start predecessors, the task will be scheduled as soon as the predecessors are finished.

For the **AsSoonAsPossible** and **AsLateAsPossible** constraints, it is not necessary to specify a constraint date. For the other types of constraints, a constraint date must be specified in the ConstraintDate property of the **SchedulingActivity** class.

Here are the different types of constraints and how they affect the schedule of an activity:

Constraint type	Description
AsSoonAsPossible	The **ProjectSchedulingModel** schedules the activity to start as soon as it can. This is the default constraint type. No specific date constraint is added to the activity. Activities with this constraint that have no predecessors will be scheduled at the project start date.
AsLateAsPossible	The **ProjectSchedulingModel** schedules the task to start as late as it can, based on the constraints on predecessors. No specific date constraint is added to the activity.
FinishNoLaterThan	The **ProjectSchedulingModel** schedules the task to finish no later than the date specified by the **ConstraintDate** property of the activity. The activity can be scheduled to finish before or at the specified date. With such a constraint, the activity may be scheduled to start before the date allowed by predecessor constraints and thus violate the constraint imposed by predecessors.
StartNoLaterThan	The **ProjectSchedulingModel** schedules the task to start no later than the date specified by the **ConstraintDate** property of the activity. The activity can be scheduled to start before or at the specified date. With such a constraint, the activity may be scheduled to start before the date allowed by predecessor constraints and thus violate the constraint imposed by predecessors.
StartNoEarlierThan	The **ProjectSchedulingModel** schedules the task to start no earlier than the date specified by the **ConstraintDate** property of the activity. The activity can be scheduled to start after or at the specified date. Such a constraint will be automatically set on the activity, when you modify the **StartTime** property of the activity.

Constraint type	Description
FinishNoEarlierThan	The **ProjectSchedulingModel** schedules the task to finish no earlier than the date specified by the **ConstraintDate** property of the activity. The activity can be scheduled to finish after or at the specified date. Such a constraint will be automatically set on the activity, when you modify the **EndTime** property of the activity.
StartOn	The **ProjectSchedulingModel** schedules the task to start at the date specified by the **ConstraintDate** property of the activity without taking into account other scheduling constraints.
FinishOn	The **ProjectSchedulingModel** schedules the task to end at the date specified by the **ConstraintDate** property of the activity without taking into account other scheduling constraints.

Calendars in the Project Scheduling Model

The ProjectSchedulingModel uses calendars to define the working and nonworking periods of the project such as holidays and weekends. Calendars are defined by the WorkCalendar class.

The **ProjectSchedulingModel** defines different types of calendars:

- project calendar

  Defines the default working and nonworking times for the project and can be set using the Calendar property of the **ProjectSchedulingModel**. If no calendar is defined for resources working on an activity or for the activity itself, an activity will be scheduled within the working times defined by the project calendar.

- resource calendar

  Defines specific working and nonworking periods for a resource and can be retrieved using the Calendar property of the SchedulingResource class. The work assigned to the resource will be scheduled within the working time of the resource calendar. Note that the resource calendar only applies to work resource and not to material resource (see the SchedulingResource.Type property).

- activity calendar

  Defines a specific calendar for an activity. This calendar can be retrieved using the Calendar property of the SchedulingActivity class. When a calendar is assigned to an activity, the activity will be scheduled within the working times of this calendar without taking into consideration the calendar of resources that may be assigned to the activity.

When creating a **ProjectSchedulingModel**, the model has a standard calendar that defines Saturday and Sunday as nonworking days and working times from 8AM to 12AM and from 1PM to 5 PM. The **ProjectSchedulingModel** holds a list of base calendars in its BaseCalendars property. This collection contains the base calendars that can be used for the project calendar or for activity calendars. The resource calendars must be a subcalendar of one of the calendars in the base calendars collection.

To learn more about Calendars see *Storing and Displaying Working and Nonworking Times*.

Resource Leveling in the Project Scheduling Model

The resource leveling is the process of removing overallocation of resources. A resource is overallocated when too much work is assigned to it.

The ProjectSchedulingModel can remove overallocations automatically by delaying activities so that resources have enough time to work on the activity. While delaying, the **ProjectSchedulingModel** ensures that all the scheduling constraints are still valid. The resource leveling process may delay the project end date.

Overallocations of resource can be removed automatically by the resource leveling algorithm of the **ProjectSchedulingModel** and can also be executed manually by specifying delays for each activity. To turn on or off the automatic resource leveling of the model use the AutomaticResourceLeveling property of the **ProjectSchedulingModel** class.

When the resource leveling is automatic, the **ProjectSchedulingModel** uses a heuristic to determine which activity should be delayed first. This heuristic examines the following properties of the activity:

- late start
- total slack
- duration
- priority
- ID

The delays computed automatically are stored in the LevelingDelay property of the SchedulingActivity class.

The automatic resource leveling can be started using no leveling delays or can use the current values stored in the **LevelingDelay** property of each activity. This is controlled by the ClearDelaysBeforeLeveling property of the **ProjectSchedulingModel** class. When the **ClearDelaysBeforeLeveling** property is set to **true**, all the delays will be reset to zero before the resource leveling starts.

The resource leveling process may be time consuming, that is why the leveling algorithm fires events about the progress of the algorithm, so that some feedback can be given to a

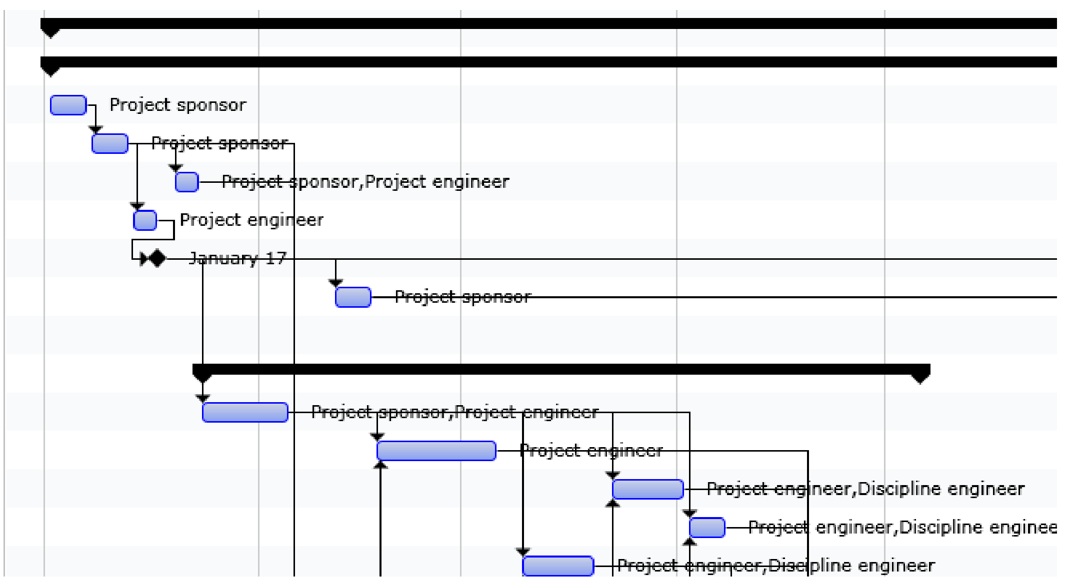
final user. This is done through the LevelingProgress event of the **ProjectSchedulingModel** class.

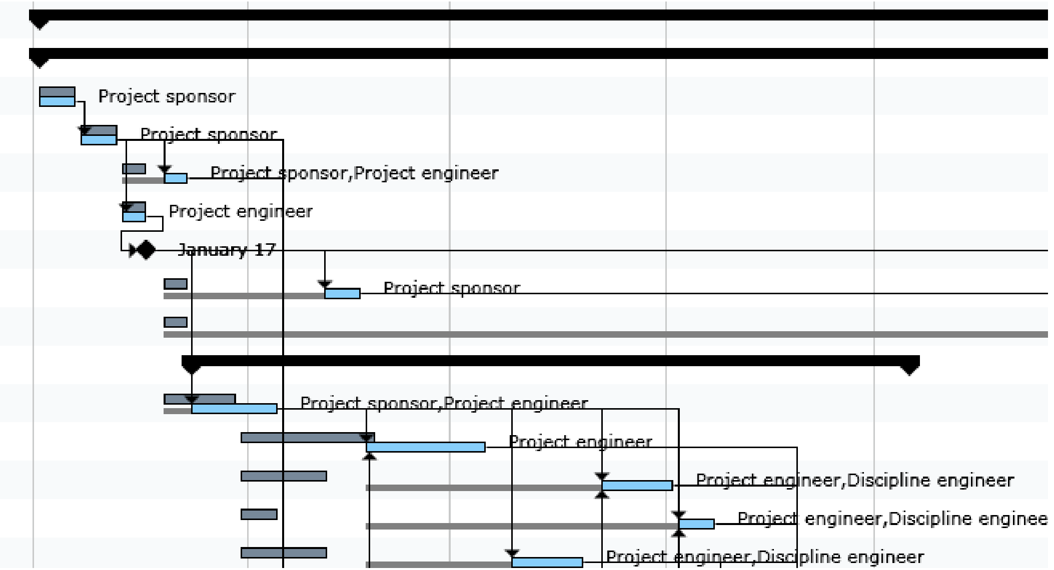
In some cases the resource leveling algorithm may not be able to find a solution that removes all the resource overallocation. For example, if two activities are causing an overallocation for a resource and have a "StartOn" constraint, they cannot be delayed and the overallocation cannot be removed. In this case, the **LevelingProgress** event gives information on which resource is problematic, at which date, and decides wether the algorithm can continue or needs to be stopped.

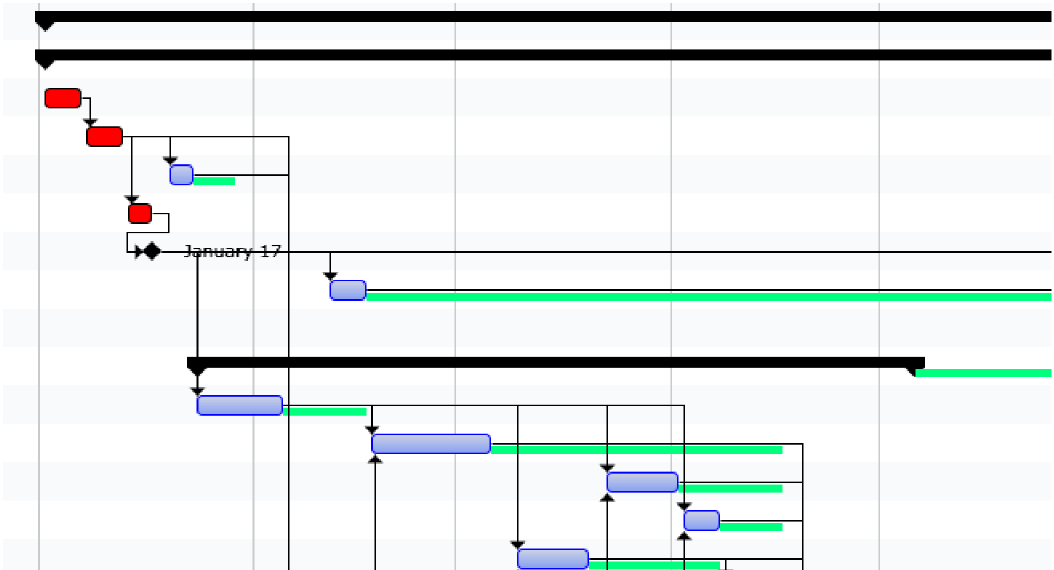
Specifying and Styling the Gantt Bars to Display

The GanttChart and ScheduleChart controls display time bars to represent a task along a time scale. By default, a **GanttChart** or a **ScheduleChart** control has no default representation for time bars so you have to specify the appearance of bars to fit your needs and your data model.

The following illustrations show some representations of time bars in a **GanttChart**.


Project sponsor
Project sponsor
Project sponsor,Project engineer
Project engineer
January 17
Project sponsor
Project sponsor,Project engineer
Project engineer
Project engineer,Discipline engineer
Project engineer,Discipline enginee
Project engineer,Discipline engineer


Project sponsor
Project sponsor
Project sponsor,Project engineer
Project engineer
January 17
Project sponsor
Project sponsor,Project engineer
Project engineer
Project engineer,Discipline engineer
Project engineer,Discipline enginee
Project engineer,Discipline engineer



The representation of time bars on the screen is controlled by a collection of 'bar definitions' that can be accessed by the BarDefinitions property of the GanttSheet class.

The **GanttChart** and **ScheduleChart** classes also have a BarDefinitions property that refers to the **BarDefinitions** property of the internal **GanttSheet** class used by these controls.

When the **GanttSheet** class needs to render an item in the data source, it looks inside its collection of bar definition to find all the definitions that are relevant to this item. Matching definitions are then used to render the item. Applying several bar definitions to render a single item can be used to give several pieces of information about the item. For example, a first bar can be used to display a rectangle that shows the duration of an activity; a second style can superimpose a rectangle that shows the percentage completion of this activity.

The order of the bar definitions in the collection is significant. Since the **GanttSheet** can use several bars to render a single activity, the bar definitions that appear first in the collection are rendered before, thus underneath, the bar definitions that appear later in the collection.

The collection of bar definitions holds a collection of instances of the BarDefinition class. The **BarDefinition** class defines the way the bar will be rendered on the screen, as well as the kind of item to which this definition applies.

In This Section

Using Default Bar Representations

Explains how to use default bar representations.

Defining Start and End Time for the Bar

Explains how to define Start and End time for the bar.

Defining the Shape of the Bar

Explains how to define the shape of the bar.

Displaying Additional Information on the Right and on the Left of the Bar

Explains how to display additional information on the bar.

Alignment Properties

Explains how to control the alignment of the bar.

Defining When a BarDefinition Applies for an Item

Explains how to define when a bar definition applies for an item.

Connection of Constraint links in a GanttChart

Explains how to draw constraint links between tasks.

Interactions on the Bar

Describes the various interactions on the bar.

Defining a Tooltip for the Bar

Explains how to define a tooltip for the bar.

Example of Bar Definitions

Provides an example of bar definition.

Using Default Bar Representations

If you are using the predefined data model (SimpleGanttModel or ProjectSchedulingModel) to store your data, you can use the InitializeFrom method (an extension method for the GanttChart class). This method connects the data source and creates a default set of BarDefinition instances for the Gantt control as well as a default set of columns for the table.

```
GanttChart chart = new GanttChart();
ProjectSchedulingModel model = new ProjectSchedulingModel();
chart.InitializeFrom(model);
```

Defining Start and End Time for the Bar

The BarDefinition class defines many visual attributes used to render an item on the screen. The first thing to specify is the time interval that will be displayed. This interval is defined by two properties: StartBinding and FinishBinding. These two properties are Silverlight Binding instances that describe the path to find the start time and the end time of the bar

from the item in the data source. These two bindings should represent dates and thus refer to properties that are of type `DateTime`.

For example, if the items to display in a **GanttChart** (items in the **ItemsSource** collection) are of type **SimpleActivity** (a base implementation of an activity provided in the namespace ILOG.Controls.Gantt.Data), you can use the StartTime and EndTime properties that represent the start and end time of the activity in the bindings. In XAML you would specify:

```
<iloggantt:BarDefinition
     StartBinding="{Binding StartTime}"
     FinishBinding="{Binding EndTime}" .../>
```

You can also display the portion of the activity that is completed by using the **StartTime** and CompletedThrough properties defined by the SimpleActivity class:

```
<iloggantt:BarDefinition
      StartBinding="{Binding StartTime}"
      FinishBinding="{Binding CompletedThrough }" .../>
```

Defining the Shape of the Bar

In order to give a shape to the bar, you need to specify a data template that defines the graphical representation of the bar. This data template is specified through the BarTemplate property of the **BarDefinition** class. For normal tasks, you would probably specify a data template that is mainly a rectangle. By default, the bar and thus the data template that you specify will be displayed from the start date to the end date that you have specified through the **StartBinding** and **FinishBinding** properties.

Here is XAML example of a data template that is simply drawing a blue rectangle with a black border:

```
<iloggantt:BarDefinition ...>
  <iloggantt:BarDefinition.BarTemplate>
    <DataTemplate>
      <Rectangle Height="13" Fill="Blue"
                 Stroke="Black" StrokeThickness="1"/>
    </DataTemplate>
  </iloggantt:BarDefinition.BarTemplate>
</iloggantt:BarDefinition>
```

As expected, this will display a blue bar as shown in the following picture:

You have noticed that the Height is specified for the rectangle but not the Width. The width of the bar is computed by default by the start and end time of the bar.

Since the shape of the bar is given through a `DataTemplate`, you can use the full power of Silverlight bindings. The bar that is created has for data context the item in the data source that this bar displays, so you might have the color of the bar bound to some property of the item. For example, the bar can be filled in red if an activity is critical.

Milestones are another type of information that you might want to display. They are often represented by a diamond shape. In case of a milestone or any other information represented by a single date, the shape does not need to be displayed from start date to end date, as those might be the same. In this case use the HorizontalAlignment property of the **BarDefinition**, by default the value **HorizontalBarAlignment.Stretch** that corresponds to the alignment from start date to end date. Set this value to **HorizontalBarAlignment.Start** or **HorizontalBarAlignment.End**. This places the bar centered on the start or end date.

Here is a **BarDefinition** that draws a black diamond centered on the start date of the item.

```
<iloggantt:BarDefinition HorizontalAlignment="Start" ...>
  <iloggantt:BarDefinition.BarTemplate>
    <DataTemplate>
      <Path Width="12" Data="M6 0 L0 6 L 6 12 L 12 6z" Fill="Black"/>
    </DataTemplate>
  </iloggantt:BarDefinition.BarTemplate>
</iloggantt:BarDefinition>
```

This will give the following result:

This time you have specified a width to the Path representing the diamond. The width is no longer computed from start and end date, the size of the bar is defined by the data template.

The following example shows how to place some symbols at the beginning or at the end of a bar. It shows how to use the BarMargin property of the **BarDefinition** class.

```
<iloggantt:BarDefinition
    HorizontalAlignment="Stretch"
    BarMargin="-6.5,0,-6.5,0"
    ...>
  <iloggantt:BarDefinition.BarTemplate>
    <DataTemplate>
      <Grid>
        <Rectangle VerticalAlignment="Top" Height="7" Fill="Black"/>
        <Path Height="13" HorizontalAlignment="Left" Fill="Black"
             Data="M0 0L0 6.5L6.5 13L13 6.5L13 0z"/>
        <Path  Height="13" HorizontalAlignment="Right" Fill="Black"
             Data="M0 0L0 6.5L6.5 13L13 6.5L13 0z"/>
      </Grid>
    </DataTemplate>
  </iloggantt:BarDefinition.BarTemplate>
</iloggantt:BarDefinition>
```

This bar definition defines a thin black bar and two down pentagons on each side of the bar. This gives the following graphical result:

The **HorizontalAlignment** is set to `Stretch` so the bar it displayed from the start time to the end time.

The **BarMargin** adds additional fixed size on the left of the start date and on the right of the end date. Each pentagon has a width of 13, therefore setting the **BarMargin** to -6.5,0,-6.5,0 (6.5 is half of 13) will ensure that the pentagons point to the start and end dates.

Displaying Additional Information on the Right and on the Left of the Bar

In addition to the bar itself, information can be displayed on the right and on the left of the bar. This information is usually a property of the item that is displayed, for example the date for a milestone.

The additional information is specified through data templates. The **BarDefinition** class provides the properties RightTemplate and LeftTemplate to define the information to be displayed.

Assume that `Name` is a property of the item displayed by the bar. You can define a `DataTemplate` that displays the name, as follows:

```
<iloggantt:BarDefinition ...>
  <iloggantt:BarDefinition.RightTemplate>
    <DataTemplate>
      <TextBlock Text="{Binding Name}"/>
    </DataTemplate>
  </iloggantt:BarDefinition.RightTemplate>
<iloggantt:BarDefinition>
```

Note how the Silverlight binding can be used here to access properties of the displayed item, just like in the **BarTemplate** property.

Alignment Properties

By default, inside a GanttChart all the bars are centered in the middle of the row where they are drawn, but this behavior can also be changed through the VerticalAlignment property of the BarDefinition. This allows you to align the bar on the top or at the bottom of the row.

The **VerticalAlignment** property applies to all elements that are displayed, that is, the bar and the additional information on the right and on the left of the bar. For more details on how

to add information on the left and on the right of the bar, see *Displaying Additional Information on the Right and on the Left of the Bar*. Other properties control the alignment of each individual element.

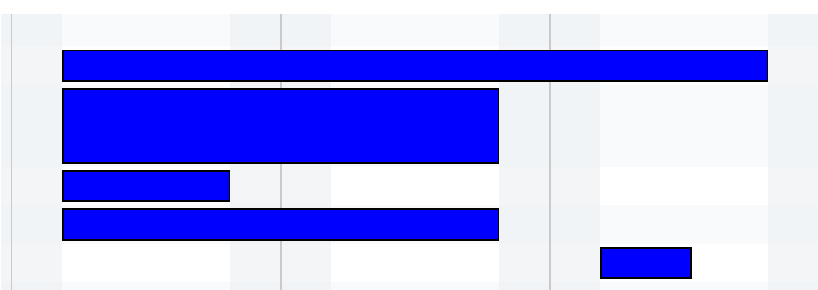
- BarPresenterVerticalAlignment: controls the vertical alignment of the bar.
- LeftPresenterVerticalAlignment: controls the vertical alignment of the information on the left of the bar.
- RightPresenterVerticalAlignment: controls the vertical alignment of the information on the right of the bar.

You can also define a bar that follows the height of a row by setting the **BarPresenterVerticalAlignment** to **Stretch**. In this case, you would need to add a margin on the top and at the bottom of the bar to separate two bars in two consecutive rows. You can do this by means of the BarMargin property of the **BarDefinition** class.

Here is an example:

```
<iloggantt:BarDefinition BarMargin="0,2,0,2"
       BarPresenterVerticalAlignment="Stretch" ...>
  <iloggantt:BarDefinition.BarTemplate>
    <DataTemplate>
      <Rectangle Fill="Blue" Stroke="Black" StrokeThickness="1"/>
    </DataTemplate>
  </iloggantt:BarDefinition.BarTemplate>
</iloggantt:BarDefinition>
```

This will give the following graphical result:



Note that in this case the height of the rectangle is not specified so that it can be stretched to fit the row height.

Defining When a BarDefinition Applies for an Item

You have seen that several BarDefinition can be used to display a single item in the data source. Now you need to specify how the Gantt components will choose the **BarDefinition** to be used for a particular item in the data source.

When the Gantt sheet needs to render an item on the screen, first it looks in its collection of **BarDefinition** (BarDefinitions property) to collect all the bar definitions that are relevant. It uses the Conditions property of the **BarDefinition** class to know which instances of the **BarDefinition** class are relevant to the item that needs to be displayed.

The **Conditions** property represents a collection of the class Condition. Each **Condition** should be **true** in order for the bar definition to be used by an item. In the following example the items in the **ItemsSource** are of type **SimpleActivity** which defines a property named `IsMilestone`.

```
<iloggantt:BarDefinition ...>
  <iloggantt:BarDefinition.Conditions>
     <iloggantt:Condition Property="IsMilestone" Value="true"/>
  </iloggantt:BarDefinition.Conditions>
</iloggantt:BarDefinition>
```

The evaluation of conditions is dynamic, as soon as a property defined in a condition changes, the conditions are re-evaluated and items may be displayed using a different set of bar definitions.

Connection of Constraint links in a GanttChart

Inside a GanttChart, the constraint links between tasks are drawn from a bar representing a task to another bar. When a task is represented by several bars, the **GanttChart** needs to choose the bar at which the link should be connected. For example, if you have a bar representing the start and the end of a task and another bar representing the deadline date for the task, you will have the links connected to the first bar and not to the second one. This information is also defined through the **BarDefinition** and its AllowConstraintLinkConnection property.

Interactions on the Bar

By default, each bar displayed in a Gantt component can be moved and resized using the mouse pointer. To move the bar, change its start date (the date defined by the StartBinding property); to resize the bar, change its end date (the date defined by the FinishBinding property).

You can easily disable this behavior for each bar definition through the CanMoveBar and CanResizeBar properties of the **BarDefinition**. For example, if a bar definition is used to display a milestone you usually do not want to resize it so you would set the **CanResizeBar** property to **false**.

Note that there are other ways to disable the editing of a bar. For example, you can set the CanEditBars property of the **GanttChart** or **ScheduleChart** to **false**. This will completely disable the editing of the bars. Additionally, the QueryGanttBarEditable event of the HierarchyChart class (base class for **GanttChart** and **ScheduleChart**) can allow you to code a more complex logic to decide if a bar can be edited or not.

By default, when you move a bar displayed between a `start` property and a `finish` property, both the `start` and `finish` property change. The `finish` property is changed so that the duration between `start` and `finish` stays the same. In some cases (and in particular in the ProjectSchedulingModel), changing the start time will automatically change the end time of the displayed item, because the logic is built in the implementation of the displayed item (for example, changing the StartTime in a SchedulingActivity will automatically change the EndTime property). In this case you do not want the Gantt control to change the `finish` property, but only the `start`. To do so, you simply set the BarMovingMode property to **BarMovingMode.ChangeStartOnly**.

A specific cursor is used when moving or resizing the bars. Since Silverlight 2 does not enable the creation of a custom cursor, the cursors are replaced by Silverlight UI Elements that you can specify through the MoveCursor and ResizeCursor property of the **BarDefinition**.

For more information on the editing process of the bar see *Editing Process - Moving and Resizing a Bar*.

Defining a Tooltip for the Bar

The BarDefinition class allows you to specify a tooltip that is displayed when moving or resizing the bar through the EditingTooltip property. If you want to have a regular tooltip that is displayed when the mouse hovers the bar, you can simply use a Silverlight Tooltip inside the data template that defines the shape of the bar. The tooltip can also use Silverlight binding to reflect some properties of the item displayed by the bar.

The following example shows how to specify a tooltip displaying the name, the start and end time of a task using the **Name**, **StartTime** and **EndTime** properties of the task.

```
<iloggantt:BarDefinition
  StartBinding="{Binding StartTime}" FinishBinding="{Binding EndTime}">
...
  <iloggantt:BarDefinition.BarTemplate>
    <DataTemplate>
      <Rectangle Fill="Blue" Height="13">
        <ToolTipService.ToolTip>
```

```
            <Border>
              <Border.Resources>
                <iloggantt:DateTimeConverter x:Key="DateConverter"/>
              </Border.Resources>
              <StackPanel Orientation="Vertical">
                <TextBlock FontWeight="Bold"
                   HorizontalAlignment="Center" Text="Task"/>
                <TextBlock Text="{Binding Name}"/>
                <StackPanel Orientation="Horizontal">
                  <TextBlock Text="Start:"/>
                  <TextBlock Text="{Binding Start,
                                  Converter={StaticResource DateConverter},
                                  ConverterParameter=d}"/>
                </StackPanel>
                <StackPanel Orientation="Horizontal">
                  <TextBlock Text="Finish:"/>
                  <TextBlock Text="{Binding Finish,
                                  Converter={StaticResource DateConverter},
                                  ConverterParameter=d}"/>
                </StackPanel>
              </StackPanel>
            </Border>
          </ToolTipService.ToolTip>
        </Rectangle>
      </DataTemplate>
    </iloggantt:BarDefinition.BarTemplate>
</iloggantt:BarDefinition>
```

This example also uses a utility converter class named DateTimeConverter that can format dates using standard .NET date formats passed in the **ConverterParameter** of the converter.

Example of Bar Definitions

Here is a full example of bar definitions that applies to a GanttChart displaying instance of the SimpleActivity class. This example defines four bar definitions: one for normal bars, one for summary task, one for milestones and another one to display the task completion.

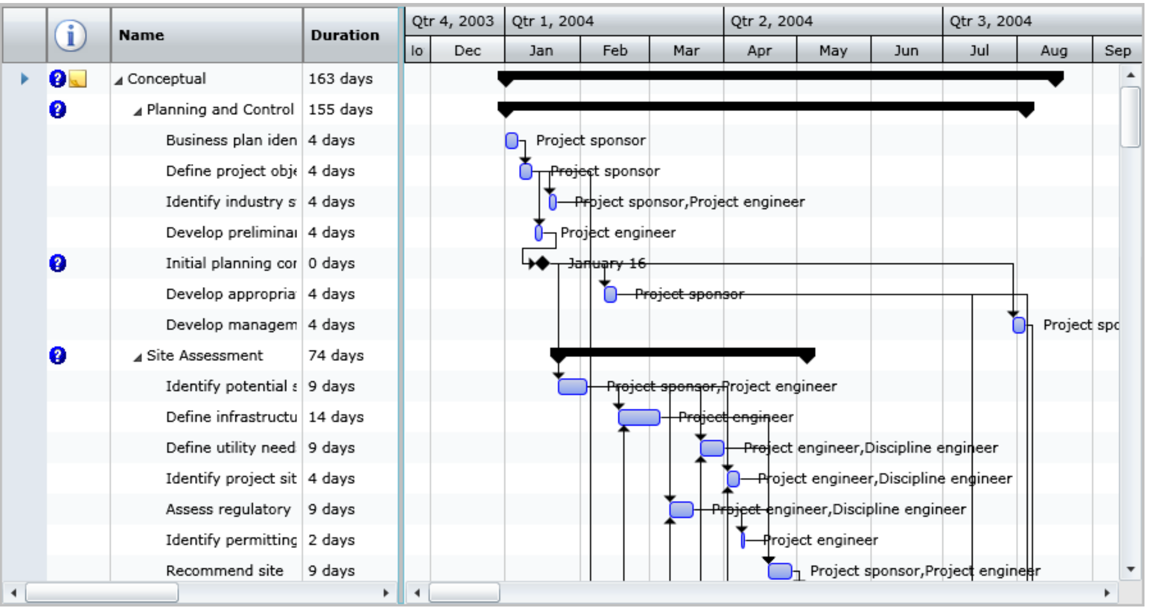
```
<ilog:GanttChart.BarDefinitions>

  <!-- Bar Definition for normal bars-->
  <ilog:BarDefinition StartBinding="{Binding StartTime}"
                      FinishBinding="{Binding EndTime}"
                      BarMovingMode="ChangeStartOnly">

    <ilog:BarDefinition.Conditions>
      <ilog:Condition Property="IsMilestone" Value="false" />
      <ilog:Condition Property="IsSummary" Value="false" />
    </ilog:BarDefinition.Conditions>

    <ilog:BarDefinition.RightTemplate>
      <DataTemplate>
        <TextBlock Text="{Binding Resources}" />
      </DataTemplate>
    </ilog:BarDefinition.RightTemplate>
```

```
  <ilog:BarDefinition.BarTemplate>
    <DataTemplate>
      <Rectangle
                 StrokeThickness="1"
                 Stroke="Blue"
                 RadiusX="3"
                 RadiusY="3"
                 Height="13">
        <Rectangle.Fill>
          <LinearGradientBrush StartPoint="0,0"
                               EndPoint="0,1">
            <GradientStop Color="#FFCED3E6" />
            <GradientStop Color="#FF88A0F2" Offset="1" />
          </LinearGradientBrush>
        </Rectangle.Fill>
      </Rectangle>
    </DataTemplate>
  </ilog:BarDefinition.BarTemplate>
</ilog:BarDefinition>

<!-- Bar Definition for task completion bars-->
<ilog:BarDefinition CanMoveBar="False"
                    StartBinding="{Binding StartTime}"
                    FinishBinding="{Binding CompletedThrough}">

  <ilog:BarDefinition.Conditions>
    <ilog:Condition Property="IsMilestone" Value="false" />
    <ilog:Condition Property="IsSummary"  Value="false" />
  </ilog:BarDefinition.Conditions>

  <ilog:BarDefinition.BarTemplate>
    <DataTemplate>
      <Rectangle
                 StrokeThickness="0"
                 Fill="Green"
                 RadiusX="2"
                 RadiusY="2"
                 Height="6"/>
    </DataTemplate>
  </ilog:BarDefinition.BarTemplate>
</ilog:BarDefinition>

<!-- Bar Definition for summary tasks -->

<ilog:BarDefinition BarMovingMode="ChangeStartOnly"
                    StartBinding="{Binding StartTime}"
                    FinishBinding="{Binding EndTime}"
                    BarMarging="-6.5,0,-6.5,0">
  <ilog:BarDefinition.Conditions>
    <ilog:Condition Property="IsSummary"  Value="true" />
  </ilog:BarDefinition.Conditions>

  <ilog:BarDefinition.BarTemplate>
    <DataTemplate>
      <Grid>
        <Rectangle VerticalAlignment="Top" Height="7" Fill="Black" />
        <Path Height="13" HorizontalAlignment="Left"
```

```
                Fill="Black" Data="M0 0L0 6.5L6.5 13L13 6.5L13 0z" />
          <Path Height="13" HorizontalAlignment="Right" Fill="Black"
                Data="M0 0L0 6.5L6.5 13L13 6.5L13 0z" />
        </Grid>
      </DataTemplate>
    </ilog:BarDefinition.BarTemplate>
  </ilog:BarDefinition>

  <!-- Bar Definition for milestones -->

  <ilog:BarDefinition BarMovingMode="ChangeStartOnly"
                      CanResizeBar="False"
                      HorizontalAlignment="Start"
                      StartBinding="{Binding StartTime}"
                      FinishBinding="{Binding StartTime}">

    <ilog:BarDefinition.Conditions>
      <ilog:Condition Property="IsMilestone"  Value="true" />
    </ilog:BarDefinition.Conditions>

    <ilog:BarDefinition.RightTemplate>
      <DataTemplate>
        <TextBlock HorizontalAlignment="Left"
                   Text="{Binding StartTime, Converter={StaticResource
DateConverter}, ConverterParameter=m}" />
      </DataTemplate>
    </ilog:BarDefinition.RightTemplate>

    <ilog:BarDefinition.BarTemplate>
      <DataTemplate >
        <Path Width="12" Data="M6 0 L0 6 L 6 12 L 12 6z" Fill="Black"/>
      </DataTemplate>
    </ilog:BarDefinition.BarTemplate>
  </ilog:BarDefinition>

</ilog:GanttChart.BarDefinitions>
```

Working with Tables in the Gantt Controls

The left part of a GanttChart or a ScheduleChart displays a table, as shown in the following picture:



A table is composed of rows and columns. Each row represents an object, and each column displays a property of the object represented by a row. To describe a column use the TreeTableColumn class.The columns of a **GanttChart** or **ScheduleChart** instance can be accessed through the Columns property, which holds the collection of columns displayed by the table.

The table used by a **GanttChart** or a **ScheduleChart** is an instance of the TreeTable class. It can be retrieved using the Table property.

Most of the methods and properties used in this section are defined in the **TreeTable** class, and also in the **GanttChart** or **ScheduleChart** through the HierarchyChart class, which is their common base class.

In This Section

Configuring Columns

Explains how to configure columns.

Using Predefined Columns and Creating a Template Column

Explains how to use predefined columns and how to create a template column.

Styling Columns

Describes the properties used to style columns.

Configuring Columns

A table column can be configured to display any kind of data using any kind of representation. It can also be used to edit or sort the data.

The following XAML code shows how to create three columns in a GanttChart that displays a SimpleGanttModel.

```
<ilog:GanttChart>
    ...
    <ilog:GanttChart.Columns>

        <ilog:TreeTableTextColumn
                  IsTreeColumn="true"
                  Width="150"
                  Header="Name"
                  Binding="{Binding Name}" />

        <ilog:TreeTableTimeSpanColumn
                  Header="Duration"
                  Width="75"
                  Binding="{Binding Duration}" />

        <ilog:TreeTableDateTimeColumn
                  Header="Start"
                  Binding="{Binding StartTime}"
```

```
                        SortDirection="Ascending" />

        </ilog:GanttChart.Columns>
...
</ilog:GanttChart>
```

- The first column is created through the TreeTableTextColumn class and displays the activity name. This column has the IsTreeColumn property set to **true** to reflect the hierarchy of activities. The data binding is set on the Name property of the row data which is, in this example, a SimpleActivity instance.
- The second column is created through TreeTableTimeSpanColumn class and displays the activity duration. The duration of a **SimpleActivity** is represented by a **TimeSpan** structure. The **TreeTableTimeSpanColumn** class uses a predefined value converter to convert **TimeSpan** instances to strings. Note that you can also provide your own value converter when specifying the data binding.
- The third column is created by means of the TreeTableDateTimeColumn class and displays the activity start time. The start time of a **SimpleActivity** is represented by a **DateTime** structure. The **TreeTableDateTimeColumn** class uses a predefined value converter to convert **DateTime** instances to strings and a **DatePicker** control to edit dates. Note that you can also provide your own value converter when specifying the data binding.

Using Predefined Columns and Creating a Template Column

There are several predefined columns that can be used to display data. These predefined table columns can be divided into two families:

- Table columns with a predefined visual representation

  For example, the TreeTableTextColumn displays a **TextBlock**. Data binding is used to connect data to the graphic element displayed by the column. The base class for these columns is the TreeTableBoundColumn class. Use the Binding property to specify the data binding. These columns are useful in many scenarios, where basic data needs to be displayed.

- Table columns with a custom visual representation

  If you want to display a visual representation in a table column that cannot be achieved with the table columns described above, you can provide your own cell template. The base class for these columns is the TreeTableTemplateColumn class. Use the TreeTableTemplateColumn.CellTemplate to specify the data template. Use a **TreeTableTemplateColumn** when you need a custom data representation.

The following table lists the existing table columns type:

Class	Description	Representation	Editor
TreeTableTextColumn	A table column that displays text.	TextBlock	TextBox
TreeTableDateTimeColumn	A table column that displays DateTime values.	TextBlock	DatePicker
TreeTableTimeSpanColumn	A table column that displays TimeSpan values.	TextBlock	TextBox
TreeTableCheckBoxColumn	A table column that displays Boolean values.	CheckBox	None
TreeTableComboBoxColumn	A table column that displays text values.	TextBlock	ComboBox
TreeTableTemplateColumn	A table column that displays a custom cell template.	Custom Template	Custom Template

The **TreeTableBoundColumn** class allows you to bind data on the row data context through the **TreeTableBoundColumn.Binding** property. The following XAML code shows how to create a **TreeTableTextColumn** (a subclass of **TreeTableBoundColumn**) column that displays the **Name** property of an activity:

```
<ilog:GanttChart>
    ...
    <ilog:GanttChart.Columns>

        <ilog:TreeTableTextColumn
                  IsTreeColumn="true"
                  Width="150"
                  Header="Name"
                  Binding="{Binding Name}" />

    </ilog:GanttChart.Columns>
                                                                                ...
</ilog:GanttChart>
```

See TreeTableBoundColumn for details.

The **TreeTableComboBoxColumn** class displays a text that can be edited using a **ComboBox**. If the object being edited is an enumeration or a Boolean value, the combo box will be automatically populated with the appropriate values. Otherwise, the **ComboBox** items can be set using the TreeTableComboBoxColumn.ItemsSource property.

The **TreeTableTemplateColumn** class can be used to display data using a custom cell template, making it possible to display anything in a table cell. The custom cell template can be set by using the TreeTableTemplateColumn.CellTemplate property. A cell editing template can also be provided to enable the editing of the column. The following XAML code shows a **TreeTableTemplateColumn** displaying a collection of images laid out in a horizontal **StackPanel**:

```
<ilog:GanttChart>
    ...
    <ilog:GanttChart.Columns>
      ...
        <ilog:TreeTableTemplateColumn
           Width="50"
           HorizontalAlignment="Center">

            <ilog:TreeTableTemplateColumn.Header>
                <Image Source="Images/info.png"
                Stretch="None" VerticalAlignment="Center" />
            </ilog:TreeTableTemplateColumn.Header>

          <ilog:TreeTableTemplateColumn.CellTemplate>
            <DataTemplate>
               <StackPanel Height="20" Orientation="Horizontal">
                    <Image
                          Stretch="None"
                          Source="Images/image1.png" />
                    <Image
                          Stretch="None"
                          Source="Images/image1.png" />
               </StackPanel>
            </DataTemplate>
          </ilog:TreeTableTemplateColumn.CellTemplate>

        </ilog:TreeTableTemplateColumn>
      ...
    </ilog:GanttChart.Columns>
    ...
</ilog:GanttChart>
```

Styling Columns

Table columns can be styled using the following properties:

Property	Description
Header	The object displayed in the table column header.
HeaderStyle	The style applied to the table column header.
Width	The width of the column.

Property	Description
HorizontalAlignment	The horizontal alignment of data in a cell.
VerticalAlignment	The vertical alignment of data in a cell.
DisplayIndex	The display index of the column in the table.
SortDirection	The direction in which the column will be sorted.
IsTreeColumn	Specifies whether the table column should display the hierarchy of the table data source, if any.

All those properties are located in the TreeTableColumn class. In addition, each specific table column offers additional properties.

Grouping, Sorting and Filtering Data

Grouping Data

Rows in a table can be grouped together, either to reflect a hierarchy in the data, or to create a specific categorized view. Grouping is automatically re-evaluated each time the data changes, provided that data implements the **INotifyPropertyChanged** interface.

To group rows in a table you can adopt one of the following alternatives:

1. use the Group property,
2. use the ParentBinding property,
3. use the GroupDescriptions collection.

If you need to group your data to reflect a hierarchy in the data source, you should only use option 1 or 2. For example, if the data source is a collection of SimpleActivity objects, and if you want to display the hierarchy of activities, you can set the **ParentBinding** property to tell the GanttChart to group rows using the Parent property of a **SimpleActivity**.

```
<ilog:GanttChart
     ...
     ParentBinding = "{Binding Parent}">
     ...
</ilog:GanttChart>
```

If you want to categorize your data according to specific criteria, fill the **GroupDescriptions** collection using the appropriate **GroupDescription** instances. The following XAML code shows how to categorize **SimpleActivity** objects. The first category level will split activities using the IsMilestone property, and the second category level will split activities using the IsSummary property:

```
<ilog:GanttChart>
   ...
     <ilog:GanttChart.GroupDescriptions>
          <ilog:GroupDescription PropertyName="IsMilestone"/>
          <ilog:GroupDescription PropertyName="IsSummary"/>
     </ilog:GanttChart.GroupDescriptions>
   ...
</ilog:GanttChart>
```

Sorting Data

Rows in a table can be sorted. Sorting can be automatic or manual. The sorting is automatic when the sort order is recomputed each time the data changes, provided that data implements the **INotifyPropertyChanged** interface. The sorting is manual when a specific action from the user is required.

To use the automatic sorting, you can adopt one of the following alternatives:

1. use the Sort property, if it already set,
2. if there are sorted columns (TreeTableColumn.SortDirection) in the table, sort the table according to these columns,
3. use the HierarchyChart.SortDescriptions collection.

To apply the manual sorting, you can either sort at the column level by calling the TreeTableColumn.Sort method, or use the HierarchyChart.SortRows method.

> ***Note:*** Not all columns support the sorting feature. To know if a column supports it, use the TreeTableColumn.SupportsSorting property.

As there is no predefined binding in the TreeTableTemplateColumn class, the sorting is achieved by providing a property name through the TreeTableTemplateColumn.SortProperty property.

The automatic sorting can be set interactively by clicking in a column header. When the column is not sorted, the first click sorts the column in ascending order, the second in descending order, and the third click restores the data source order. To disable this functionality at the table level, set the TreeTable.CanSortColumns property to false. To

disable this functionality at the table column level, set the TreeTableColumn.CanSort property to false.

The following XAML code shows how to sort a **GanttChart** according to the activity **StartTime** property using the **HierarchyChart.SortDescriptions** collection:

```
<ilog:GanttChart>
    ...
    <ilog:GanttChart.SortDescriptions>
        <ilog:SortDescription   PropertyName="StartTime"
                                Direction="Ascending"/>
    </ilog:GanttChart.SortDescriptions>
    ...
</ilog:GanttChart>
```

Filtering Data

Rows in a table can be filtered. Filtering is applied at the row level, meaning that each row can be visible or not depending on the filter set on the table. To set a filter, use the HierarchyChart.Filter property. Filtering is automatically re-evaluated each time the data changes, provided that data implements the **INotifyPropertyChanged** interface. The following C# code shows how to set a filter on a **GanttChart** that contains SimpleActivity objects to displays only activities whose name begins with 'A':

```
GanttChart chart = new GanttChart();
List<SimpleActivity> tasks = new List<SimpleActivity>()
{
    new SimpleActivity() { Name = "Activity 1" },
    new SimpleActivity() { Name = "Activity 2" ),
    new SimpleActivity() { Name = "Task 3") }
};
chart.ItemsSource = tasks;
chart.Filter = task => (task as SimpleActivity).Name.StartsWith("A");
```

Controlling the Displayed Time Interval

In GanttChart or ScheduleChart controls, the displayed time range is specified by the following properties:

◆ FirstVisibleTime

  This property specifies the displayed date and time on the left of the chart.

◆ ZoomFactor

  This property specifies the time zoom level.

By default the **ZoomFactor** is 1. This means that one hour of time is represented by one pixel. If you want to have a week represented by 100 pixels on the screen, set the **ZoomFactor** to 100 pixels divided by the number of hours in 7 days = 100d / 7*24.

The SetTimeInverval method of the **GanttChart** and **ScheduleChart** is convenient to specify the start and end displayed date. This method simply changes the **FirstVisibleTime** and **ZoomFactor** properties.

Note that the zoom factor is bounded between two values specified by the MinZoomFactor and MaxZoomFactor properties.

Storing and Displaying Working and Nonworking Times

The WorkCalendar class allows you to store working and nonworking periods and times. This class is also used to display working and non working periods in a Gantt chart.

Using a WorkCalendar to Store Working and Nonworking Periods

IBM® ILOG® Gantt for .NET defines two types of **WorkCalendar**: a base calendar or a subcalendar that inherits from a base calendar. A subcalendar inherits its specification from its base calendar and can modify the working and nonworking times specified by the base calendar. For example, a base calendar can be used to store the general working times on a project and a subcalendar can be used to specify the working times and vacations of a specific resource of the project.

To create a base calendar named `MyCalendar`, use the following C# code:

```
WorkCalendar myBaseCalendar = new WorkCalendar("MyCalendar", null);
```

To create a subcalendar of this calendar proceed as follow:

```
WorkCalendar mySubCalendar = new WorkCalendar("MySubCalendar", myBaseCalendar);
```

By default a **WorkCalendar** defines Saturday and Sunday as nonworking days, for the other days the working times are from 8 AM to 12 AM and from 1 PM to 5 PM defining 8 working hours per day.

The **WorkCalendar** class provides methods that allow you to modify these settings for a day of the week.

To set all Fridays as nonworking days do the following:

```
mySubCalendar.SetNonWorking(DayOfWeek.Friday);
```

To change the working times of all Mondays to be 7 AM - 1 PM and 3 PM - 8 PM do the following:

```
WorkingTime[] times = new WorkingTime[2];
times[0] = new WorkingTime(TimeSpan.FromHours(7), TimeSpan.FromHours(13));
times[1] = new WorkingTime(TimeSpan.FromHours(15), TimeSpan.FromHours(20));
mySubCalendar.SetWorkingTimes(DayOfWeek.Monday, times);
```

You can also specify exceptional periods:

To specify a nonworking period from 01/01/2005 to 01/07/2005 do the following:

```
mySubCalendar.SetNonWorking(new DateTime(2005,1,1), new DateTime(2005, 1,7));
```

To specify special working hours from 7 AM to 9 AM for the period 06/01/2005 to 06/06/2005 you would do:

```
WorkingTime[] times = new WorkingTime[1];
times[0] = new WorkingTime(TimeSpan.FromHours(7), TimeSpan.FromHours(9));
mySubCalendar.SetWorkingTimes(new DateTime(2005,6,1), new DateTime(2005, 6,6),
times);
```

Navigating a WorkCalendar

A **WorkCalendar** object not only allows you to store the working and nonworking times, but it also allows you to navigate in the working and nonworking times and perform several computation on working periods.

For example, you can get the next or previous working time from a date. In the following C# code the next and previous working time is computed from January 1 2005.

```
WorkCalendar calendar = new WorkCalendar("MyCalendar", null);
DateTime date = new DateTime(2005,1,1);
DateTime next = calendar.NextWorkingTime(date);
DateTime previous = calendar.PreviousWorkingTime(date);
Console.WriteLine("Next working time is " + next);
Console.WriteLine("Previous working time is " + previous);
```

This code gives you Monday January, 3, 2005 at 8 AM as the next working time and Friday December, 31 2004 at 5 PM as the previous working time.

Similarly, the **WorkCalendar** defines methods to navigate to next nonworking times (NextNonWorkingTime).

You may also compute the amount of work between two dates:

```
DateTime fromDate = new DateTime(2005,1,5);
DateTime toDate = new DateTime(2005,1,12);
TimeSpan work = calendar.WorkBetween(fromDate, toDate);
```

This fragment of C# code returns a duration of 40 hours. In the period from 05 to 12 January 2005, there are 5 working days of 8 hours, resulting in 40 hours of work.

Finally, the **WorkCalendar** object provides methods to add or remove a work duration from a date to compute another date. An example is in the following C# code:

```
TimeSpan work = TimeSpan.FromDays(88);
DateTime date = new DateTime(2004,1,1,8,0,0);
Console.WriteLine(date);
date = calendar.Add(date work);
Console.WriteLine(date);
date = calendar.Remove(date, work);
Console.WriteLine(date);
```

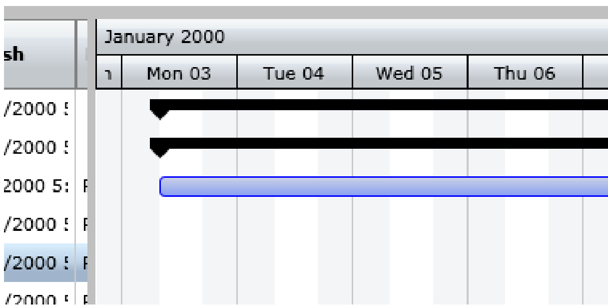
This C# code fragment will print:

```
1/1/2004 8:00:00 AM
1/4/2005 5:00:00 PM
1/1/2004 8:00:00 AM
```

Displaying Working and Nonworking Times in Gantt Controls

The nonworking periods defined by an instance of the **WorkCalendar** class can be displayed in the Gantt controls by means of a grid. By default the GanttChart control contains a grid that displays the non working time and the default calendar displayed is a standard calendar.

In the following figure, behind the Gantt bars you can see the nonworking period displayed in gray:



You may change the **WorkCalendar** displayed in the background of a **GanttChart** or ScheduleChart through its GridCalendar property. The grid is always up-to-date with the calendar that it displays. When the calendar changes, the grid is automatically refreshed.

The following C# code fragment shows how to change the calendar displayed by the grid of a **GanttChart** control.

```
WorkCalendar myCalendar = new WorkCalendar();
GanttChart gantt = new GanttChart();
Gantt.GridCalendar = myCalendar;
```

The non working area can be styled through the NonWorkingAreaStyle property of the control. Non working areas are implemented as Silverlight Rectangle objects. To change the color of those rectangles you must change the **Fill** property of the rectangle through the **NonWorkingAreaStyle**, as follows:

```
<iloggantt:GanttChart ...>
  <iloggantt:GanttChart.NonWorkingAreaStyle>
    <Style TargetType="Rectangle">
        <Setter Property="Fill" Value="yellow" />
    </Style>
  </iloggantt:GanttChart.NonWorkingAreaStyle>
</iloggantt:GanttChart>
```

User Interaction on a Gantt Bar

The GanttChart and ScheduleChart controls offer many possibilities to control the editing of the bars in the Gantt. While editing a bar, the user can change the behavior but also the style of elements that are used. This section explains the various elements involved in editing a bar of a Gantt control.

In This Section

Editing Process - Moving and Resizing a Bar

Describes how bars can be moved an resized on a Gantt control.

Using Bar Editing Events

Describes how to use bar editing events.

Snapping Time when Moving or Resizing a Bar

Describes the snapping time mechanism.

Working with Editing Tooltip

Explains how to work with editing tooltip.

Styling User Interaction Elements

Describes how to customize user interaction elements throw styling.

Editing Process - Moving and Resizing a Bar

By default, on a Gantt control each bar can be moved and resized. To move a bar drag it, to resize it click on the right side of the bar and drag it. It is possible to disable the editing of bars by setting the CanEditBars property of the **GanttChart** or **ScheduleChart** control to **false**, in this case all bars become non editable. The editing of a bar is controlled on a finer level by each BarDefinition instance that defines how the bar is displayed but also how the interaction on the bar works. Each bar in the Gantt controls are associated to a **BarDefinition** instance. For more details on how to specify the bar definitions see *Specifying and Styling the Gantt Bars to Display*.

The action of moving or resizing a bar does not really move or resize a bar: in fact, when you release the mouse pointer, the dates of the underlying data objects that are displayed by the bar are changed. When the dates change in the data object, the Gantt control is notified and the bars are drawn at their new location. The editing of the bar will then work only if the underlying edited object notifies the changes of the dates, therefore the underlying object should implement the **INotifyPropertyChanged** interface.

The bar is displayed from and to the dates that are specified by the StartBinding and FinishBinding of the **BarDefinition** class. These are two Silverlight bindings to date properties of the edited object. By default, when a bar is moved, the Gantt control gives a new value to the properties specified in the **StartBinding** and **FinishBinding**. When a bar is resized, only the property in the **FinishBinding** is modified.

While moving a bar, it is possible to change only the start date instead of changing both the start and end date. This may be useful when changing the start date in the data automatically changes the end date. To do so, you can set the BarMovingMode property of the **BarDefinition** to **ChangeStartOnly**.

By means of the **BarDefinition** class you can also completely disable the editing by setting the CanMoveBar and CanResizeBar to **false**.

Using Bar Editing Events

Every time a bar is moved or resized, an event is sent to the **GanttChart** or **ScheduleChart** controls to allow you to cancel or modify the way the edited item is going to be modified. This event is specified through the BeforeEditBar event.

In the following C# code, the **BeforeEditBar** event is used to ensure that the bar is not moved before January 1st 2009.

```
gantt.BeforeEditBar += delegate(object source, BeforeEditBarEventArgs e)
{
  if (e.Action == BarInteractionMode.Move &&
      e.NewValue < new DateTime(2009, 1, 1))
          e.Cancel = true;
```

```
};
```

Snapping Time when Moving or Resizing a Bar

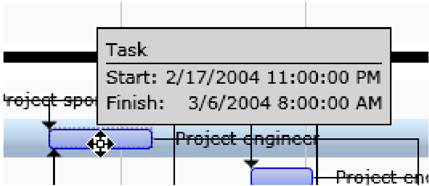
When moving or resizing a bar, a new date is computed for the start or end of the bar and the new date will be affected to the edited item (task/activity/reservation). The conversion from mouse inputs to dates could lead to non-rounded dates, that is why the date is snapped by default to the nearest hour. This mechanism can be changed in various ways.

First you can change the SnapUnit property of the HierarchyChart. This property is of type **TimeUnit**, an enumeration that defines the main unit of times from milliseconds to century. If you want to code the way the snapping is done, use the SnapTimeOnEditingBar event of the **HierarchyChart** and set the **SnapUnit** property to **TimeUnit.None** so that no snapping is done before the event is sent. The following code example makes sure that when a bar is moved, the bar will only be moved in hours from 8AM to 5PM.

```
gantt.SnapTimeOnEditingBar += delegate(object source, SnapTimeEventArgs e)
{
    if (e.Mode == BarInteractionMode.Move)
    {
        if (e.DateTime.TimeOfDay < TimeSpan.FromHours(8))
           e.DateTime = e.DateTime.Date.AddHours(8);
        else if (e.DateTime.TimeOfDay > TimeSpan.FromHours(17))
           e.DateTime = e.DateTime.Date.AddHours(17);
    }
};
```

Working with Editing Tooltip

When a bar is edited because you move or resize it, a tooltip appears to show the future new starting and ending dates of the bar. This tooltip is called the Editing Tooltip. The following image shows the default editing tooltip.

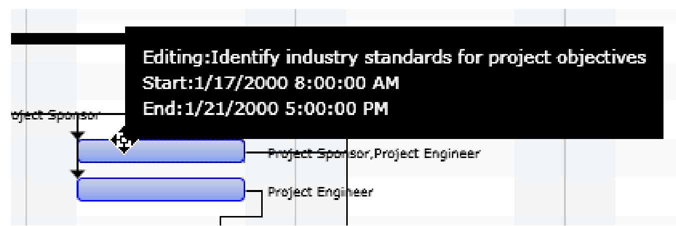


This tooltip can be disabled by setting the ShowEditingToolTips property to **false** on the ScheduleChart or GanttChart class. The tooltip can also be disabled on each individual bar by setting the ShowEditingToolTip property of the BarDefinition class that defines the bar. The Editing tooltip can also be modified at the level of the **BarDefinition** class.

To modify the editing tooltip, you may specify a new tooltip by setting the **EditingToolTip** property of the **BarDefinition** class. This property can be set with any user interface elements. The resulting tooltip will be displayed with a data context that is an instance of the EditingBarData class. This **EditingBarData** class defines three properties that you can use when redefining a new tooltip shape: the Item, Start and Finish properties. The **Item** is the object that is edited (the item in the **ItemsSource** of the Gantt control), the **Start** and **Finish** properties are the dates. The following example of XAML redefines a new **Editing** tooltip and uses bindings to display the start date, the end date and the name of the edited item:

```
<ilog:BarDefinition
    ...

  <ilog:BarDefinition.EditingToolTip>
    <Border  Background="Black">
      <StackPanel Margin="10" Orientation="Vertical">
        <StackPanel Orientation="Horizontal">
          <TextBlock Foreground="White" Text="Editing:" />
          <TextBlock Foreground="White"
                     Text="{Binding Item.Name}" />
        </StackPanel>
        <StackPanel Orientation="Horizontal">
          <TextBlock Foreground="White"
                     Text="Start:" />
          <TextBlock Foreground="White"
                     Text="{Binding Start}" />
        </StackPanel>
        <StackPanel Orientation="Horizontal">
          <TextBlock Foreground="White"
                     Text="End:" />
          <TextBlock Foreground="White"
                     Text="{Binding Finish}" />
        </StackPanel>
      </StackPanel>
    </Border>
  </ilog:BarDefinition.EditingToolTip>

</ilog:BarDefinition>
```

This gives the following graphical result:



Styling User Interaction Elements

While moving or resizing a bar a dashed rectangle appears to display the new time interval of the bar. This rectangle can be customized throw styling. To change the default dashed rectangle, use the EditedBarLocationIndicatorStyle property of the **GanttChart** or **ScheduleChart** classes.

The following XAML example specifies a transparent red rectangle instead of a dashed rectangle:

```
<iloggantt:ScheduleChart >
  <iloggantt:ScheduleChart.EditedBarLocationIndicatorStyle>
    <Style TargetType="Control">
      <Setter Property="Template">
        <Setter.Value>
          <ControlTemplate TargetType="Control">
            <Rectangle Fill="Red" Opacity=".5"/>
          </ControlTemplate>
        </Setter.Value>
      </Setter>
    </Style>
  </ iloggantt:ScheduleChart.EditedBarLocationIndicatorStyle>
</iloggantt:ScheduleChart>
```

You can do the same to change the style of the line that is drawn when the user creates a constraint with the mouse pointer in the **GanttChart** control. The style of this line can be changed using the ConstraintCreationIndicatorStyle property.

Here is a XAML example:

```
<iloggantt:GanttChart >
  <iloggantt:GanttChart.ConstraintCreationIndicatorStyle>
    <Style TargetType="Line">
      <Setter Property="Stroke"  Value="Black" />
      <Setter Property="StrokeThickness"   Value="1" />
      <Setter Property="StrokeDashArray" Value="1 1" />
    </Style>
  </iloggantt:GanttChart.ConstraintCreationIndicatorStyle >
```

Internationalization

IBM® ILOG® Gantt for .NET is internationalized. All messages, resources and dialog boxes of IBM ILOG Gantt for .NET are localized for English and French languages. If you need to localize for another language, IBM ILOG Gantt for .NET provides the resource files and tools that 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 Gantt for .NET you must create assemblies (dlls) that contain the culture-dependant resources of the library. Those assemblies are called satellite assemblies. IBM ILOG Gantt 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.

Index

E

Editing Tooltip **83**

R

resource leveling **48**

S

satellite assemblies **87**
summary activity **37**