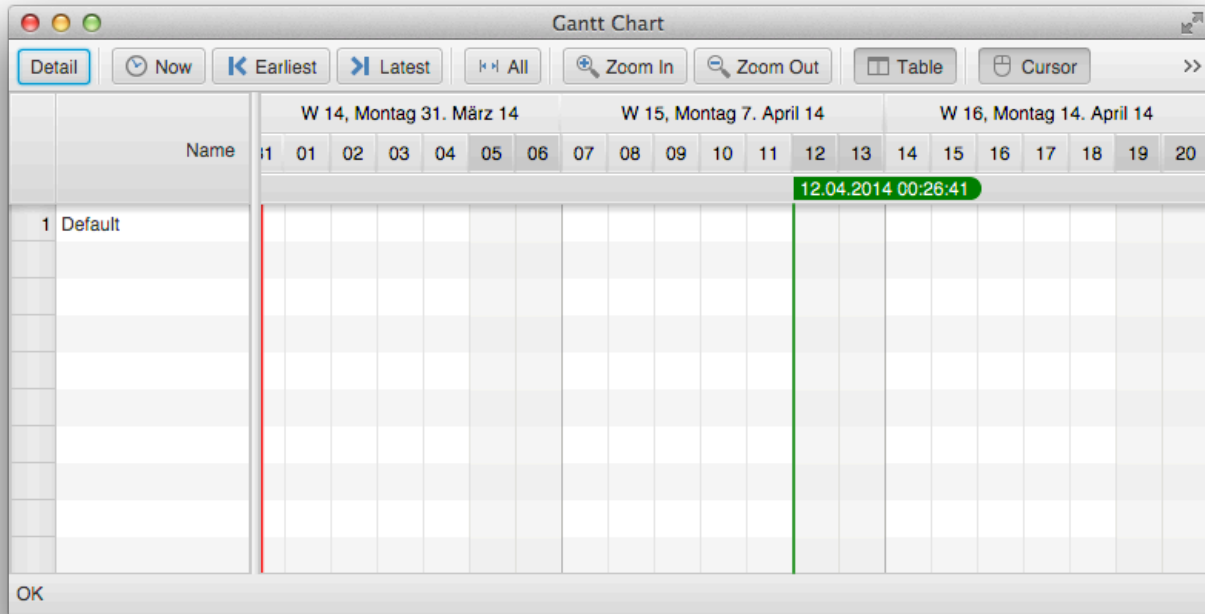


1. FlexGanttFX Developer Manual	2
1.1 1. Installation	2
1.2 2. Tutorial	3
1.3 3. Controls	11
1.3.1 3.1 GanttChart	11
1.3.1.1 3.1.1 Model	12
1.3.1.2 3.1.2 Detail Node	13
1.3.1.3 3.1.3 Display Mode	13
1.3.1.4 3.1.4 Graphics Header	15
1.3.1.5 3.1.5 Row Header	15
1.3.1.6 3.1.6 Property Sheet	17
1.3.1.7 3.1.7 Other Features	18
1.3.2 3.2 MultiGanttChartContainer	20
1.3.3 3.3 DualGanttChartContainer	21
1.3.4 3.4 QuadGanttChartContainer	21
1.3.5 3.5 GraphicsBase	22
1.3.5.1 3.4.1 System Layers	24
1.3.5.2 3.4.2 Drag & Drop	25
1.3.5.3 3.4.3 Event Handling	27
1.3.5.4 3.4.4 Activity Editing	31
1.3.5.5 3.4.5 Row Editing	35
1.3.5.6 3.4.6 Activity Rendering	39
1.3.5.7 3.4.7 Row Rendering	42
1.3.5.8 3.4.8 Context Menu	43
1.3.6 3.6 Timeline	44
1.3.6.1 3.5.1 Timeline Model	46
1.3.6.2 3.5.2 Time Tracker	47
1.3.7 3.7 Dateline	51
1.3.7.1 3.6.1 Dateline Model	52
1.3.8 3.8 Eventline	58
1.4 4. Model	59
1.4.1 4.1 Activity	59
1.4.1.1 4.1.1 ChartActivity	60
1.4.1.2 4.1.2 CompletableActivity	61
1.4.1.3 4.1.3 HighLowChartActivity	61
1.4.2 4.2 ActivityRef	61
1.4.3 4.3 ActivityLink	61
1.4.4 4.4 ActivityRepository	61
1.4.4.1 4.4.1 IntervalTreeActivityRepository	62
1.4.4.2 4.4.2 ListActivityRepository	63
1.4.5 4.5 Row	63
1.4.6 4.6 Layer	65
1.4.7 4.7 LinesManager	65
1.4.8 4.8 Layout	69
1.4.8.1 4.8.1 Gantt Layout	69
1.4.8.2 4.8.2 Agenda Layout	70
1.4.8.3 4.8.3 Chart Layout	71
1.4.9 4.9 Calendar	72
1.5 5. Styling (CSS)	76
1.5.1 dateline.css	76
1.5.2 eventline.css	80
1.5.3 gantt.css	82
1.5.4 graphics.css	88
1.5.5 timeline.css	93
1.6 6. Logging	93

FlexGanttFX Developer Manual

This confluence space is the home of the **FlexGanttFX** documentation. Please feel free to comment on anything that you think might need improving.



Search this documentation

1. Installation

- Step 1: Download and Install Java 8
- Step 2: Download the FlexGanttFX distribution
- Step 3: Unpack the distribution
- Step 4: Add JAR files to classpath
- Step 5: Create application class

Step 1: Download and Install Java 8

Download [JDK 8](#) and run the installer. Java 8 includes JavaFX 8.

Step 2: Download the FlexGanttFX distribution

Go to the downloads section of <http://www.dlsc.com> and download the latest release of FlexGanttFX. The download file will be a ZIP archive containing the required JAR files, demos, tutorials, API documentation, etc...

Step 3: Unpack the distribution

Unzip the distribution to your local file system. Once unpacked you will see the following content:

The distribution contains the following subfolders:

- **css** - copies of the stylesheets used by FlexGanttFX (the originals are included in the JAR file)
- **demos** - several runnable jar files, simply double click to run or call "java -jar xxx-demo.jar" (make sure to use Java 8u60+)
- **docs** - the API documentation of FlexGanttFX
- **ext** - third-party JAR files required for running FlexGanttFX
- **legal** - the license agreements as PDF files
- **lib** - the FlexGanttFX libraries
- **tutorial** - files to get you started

Step 4. Add JAR files to classpath

Assuming that you downloaded release 1.6.0 then add the following files (located in the distribution's **lib** folder) to your classpath.

- **flexganttfx-core-1.6.0.jar** - contains various utility classes and the licensing support
- **flexganttfx-model-1.6.0.jar** - all classes related to the data model (activities, rows, repositories)
- **flexganttfx-view-1.6.0.jar** - the view classes, such as the actual Gantt Chart control
- **flexganttfx-extras-1.6.0.jar** - additional classes such as a toolbar and a statusbar

Add the files located in the **ext** folder to your classpath.

- **controlsfx.jar** - the distribution of the [ControlsFX](#) project
- **license4j.jar** - code for supporting the licensing concepts

Step 5. Create application class

The following listing shows the most basic setup that is required to launch a Gantt chart user interface.

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;

import com.flexganttfx.view.GanttChart;

public class MyFirstGanttChart extends Application {

    @Override
    public void start(Stage stage) throws Exception {

        // <- Our Gantt chart
        GanttChart<> gantt = new GanttChart<>();

        Scene scene = new Scene(gantt);

        stage.setScene(scene);
        stage.centerOnScreen();
        stage.sizeToScene();
        stage.show();
    }

    public static void main(String[] args) {
        Application.launch(args);
    }
}
```

2. Tutorial

In this tutorial we are creating a very simple solution for displaying the schedule of an aircraft fleet. To install **FlexGanttFX** please follow the instructions found in [1. Installation](#).



View Model

Let's start by creating a view model for the Gantt chart. Our objects are **Fleet**, **Aircraft**, **Crew**, and **Flight**. Instances of **Flight** will be shown as a horizontal bar in the graphics area of the Gantt chart while the first three will be displayed in the rows of the tree table area. **Fleet**, **Aircraft**, and **Crew** share a common superclass called **ModelObject**, an extension of **Row**.

The **Row** class is being used to define a hierarchical data structure by the help of three type arguments: the first one specifies the type of the parent row, the second one the type of the children rows, and the third one the type of activities that will be shown on the right-hand side of the Gantt chart.

Model Object

```
class ModelObject<
    P extends Row<?,?,?>, // Type of parent row
    C extends Row<?,?,?>, // Type of child rows
    A extends Activity> extends Row<P, C, A> { }
```

We can now pass **ModelObject** as a type argument when creating an instance of a **GanttChart** control. This informs the control that all rows will have this common supertype.

Typed Gantt Chart

```
GanttChart<ModelObject<?,?,?> gantt = new GanttChart<>()
```

The **Aircraft** model class can be implemented as shown in the following code fragment, assuming that a fleet consists of several aircrafts, each aircraft having a crew, and flights being assigned to aircrafts and crews.

Row Type: Aircraft

```
public class Aircraft extends ModelObject<Fleet, Crew, Flight> {
    public Aircraft(String name) {
        super(name);
    }
}
```

The **Flight** class extends **MutableActivityBase** and might look like this:

Activity Type: Flight

```
public class Flight extends MutableActivityBase<FlightData> {
    public Flight(FlightData data) {
        super(data.getFlightName()); // the activity name
        setStartTime(data.getFlightDepartureTime()); // start / end times as
        java.time.Instant
        setEndTime(data.getFlightArrivalTime());
        setUserObject(data); // a user object according to the type argument
        above
    }
}
```

This class defines a flight as a mutable activity, which means that the flight can be edited by the user. We can also see that the activity gets its information from a domain object of type **FlightData**. Supporting a user object allows us to create a bridge between the domain model and the view model. All that is left to do now is to add the activities / the flights to the rows / the aircrafts. For this we can simply call the method **Row.addActivity(Layer, Activity)**.

Activity Repositories

Rows do not store activities themselves, instead they are delegating all activity-related functionality to a repository of type **ActivityRepository**. The default repository is of type **IntervalTreeRepository**. Applications can implement their own repositories and register them by calling **Row.setRepository()**.

Layers

Layers are used to create groups of activities so that they can be shown / hidden together. In our example we want to group flights based on their service type (cargo, charter, training, etc...).

Layers

```
Layer cargoLayer = new Layer("Cargo");
Layer trainingLayer = new Layer("Training");
Layer charterLayer = new Layer("Charter");
ganttt.getLayers().addAll(cargoLayer, trainingLayer, charterLayer); // make
layers known to Gantt
```

Now the Gantt chart knows which layers it needs to render and we can create the link between the layers and the activities. This is done when we add the activities to the rows (here: add flights to aircrafts).

Adding Activities / Flights

```
Flight flight1 = new Flight(); // a cargo flight
Flight flight2 = new Flight(); // a training flight
Flight flight3 = new Flight(); // a charter flight

aircraft1.addActivity(cargoLayer, flight1);
aircraft1.addActivity(trainingLayer, flight2);
aircraft2.addActivity(charterLayer, flight3);
```

Intermediate Result

In the following code sample we are combining all of the steps from above.

Aircraft Gantt Chart

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;

import com.flexganttfx.model.Activity;
import com.flexganttfx.model.Layer;
import com.flexganttfx.model.Row;
import com.flexganttfx.model.activity.MutableActivityBase;
import com.flexganttfx.view.GanttChart;

public class MyFirstGanttChart extends Application {

    /*
     * Common superclass of Fleet, Aircraft, and Crew.
     */
    class ModelObject<
        P extends Row<?,?,?>, // Type of parent row
        C extends Row<?,?,?>, // Type of child rows
        A extends Activity> extends Row<P, C, A> { }

    class Fleet extends ModelObject<Row<?,?,?>, Aircraft, Activity> { }

    class Aircraft extends ModelObject<Fleet, Crew, Flight> { }

    class Flight extends MutableActivityBase<Object> { }

    @Override
    public void start(Stage stage) throws Exception {

        // Our root object.
        Fleet fleet = new Fleet();
        fleet.setExpanded(true);

        // Create the control.
        GanttChart<ModelObject<?,?,?>> gantt = new GanttChart<>(fleet);
```

```
// Layers based on service type.
Layer cargoLayer = new Layer("Cargo");
Layer trainingLayer = new Layer("Training");
Layer charterLayer = new Layer("Charter");
ganttt.getLayers().addAll(cargoLayer, trainingLayer, charterLayer);

// Create the aircrafts.
Aircraft aircraft1 = new Aircraft();
Aircraft aircraft2 = new Aircraft();

// Add the aircrafts to the fleet.
fleet.getChildren().addAll(aircraft1, aircraft2);

// Create the flights
Flight flight1 = new Flight(); // a cargo flight
Flight flight2 = new Flight(); // a training flight
Flight flight3 = new Flight(); // a charter flight

aircraft1.addActivity(cargoLayer, flight1);
aircraft1.addActivity(trainingLayer, flight2);
aircraft2.addActivity(charterLayer, flight3);

Scene scene = new Scene(ganttt);

stage.setTitle("Fleet Schedule");
stage.setScene(scene);
stage.centerOnScreen();
stage.sizeToScene();
stage.show();
}

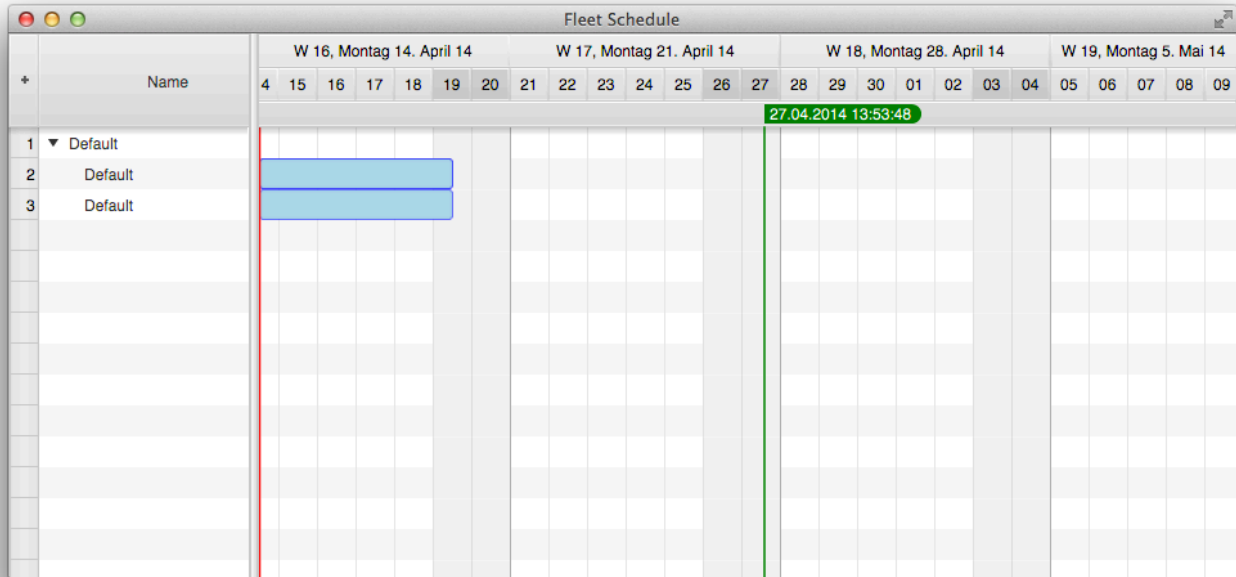
public static void main(String[] args) {
```

```

    Application.launch(args);
}
}

```

The image below shows what we will see when we run this code.



Activity Renderers

This result is not bad for just a few lines of code, however the rendering of the flights is not attractive at all. We can customize their appearance by registering a different **ActivityRenderer** for the activity type **Flight**. This is done by calling the method **GraphicsBase.setActivityRenderer()** where the graphics view is the control on the right-hand side of the Gantt chart. It is responsible for rendering all activities. We can add the following lines to our example from above.

Registering an Activity Renderer

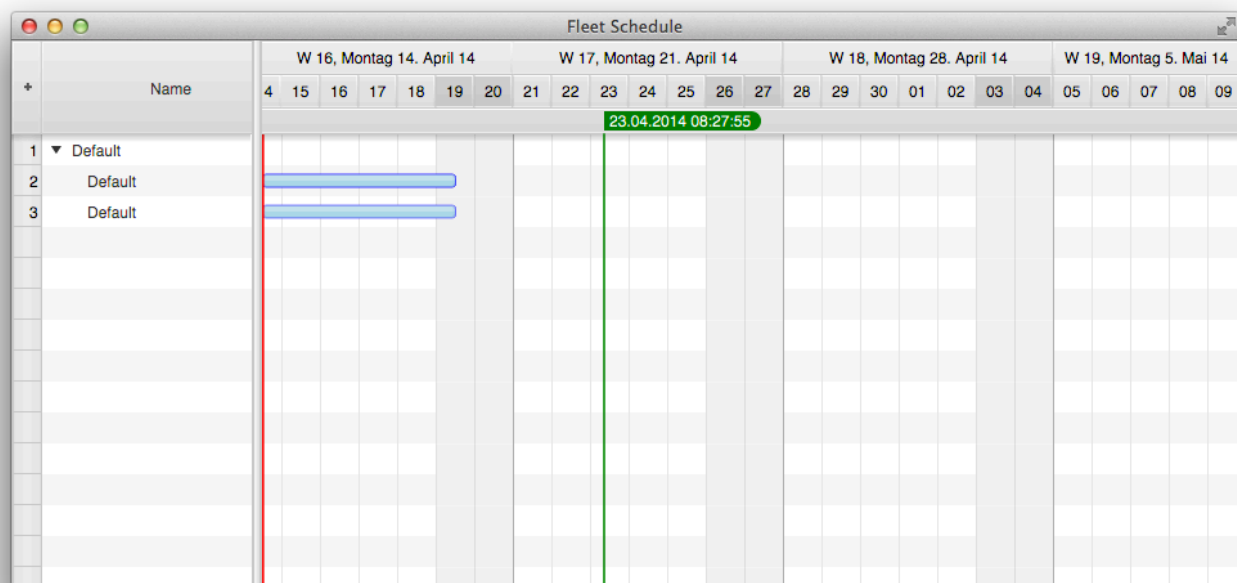
```

GraphicsView <ModelObject<?, ?, ?>> graphics = gantt.getGraphics();
graphics.setActivityRenderer(
    Flight.class,
    GanttLayout.class,
    new ActivityBarRenderer<>(graphics, "FlightRenderer"));

```

This replaces the default activity renderer with a renderer that draws a fixed-height bar. Interesting about this code is that we are not only passing the activity type and the renderer instance but also a layout type. We don't want to spend too much time on layouts in the context of this quick start guide but let's just say that **FlexGanttFX** is capable of displaying activities in several different ways (as time bars, as chart entries, as agenda entries).

Our example now looks like this:

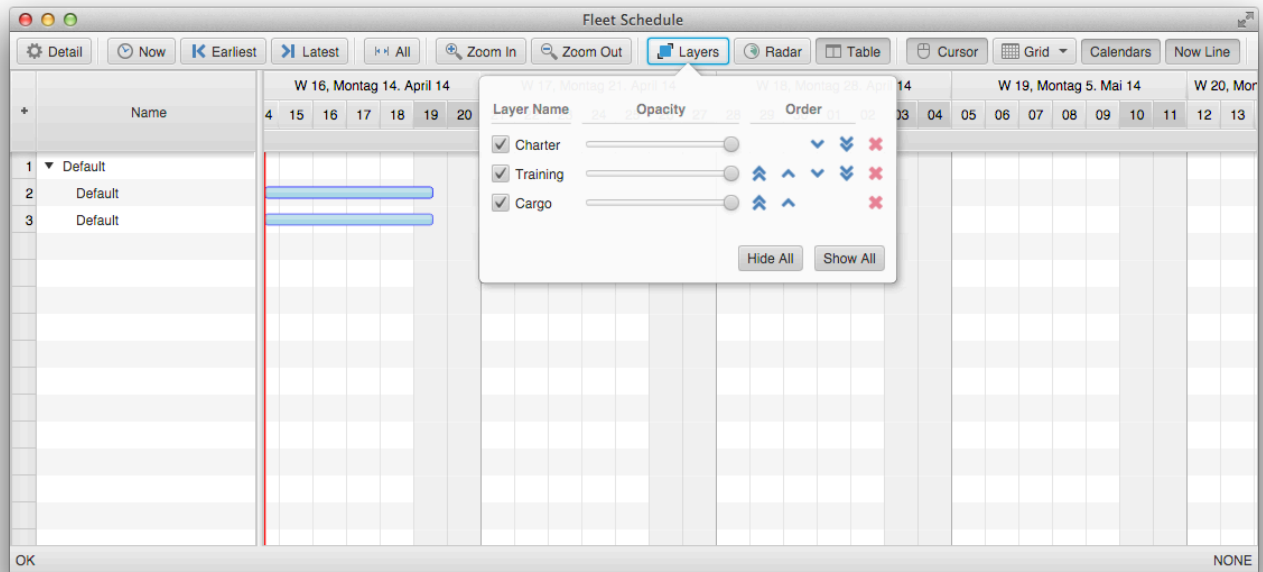


We can now add a **GanttChartToolBar** and a **GanttChartStatusBar** to the example. This allows us to perform actions on the chart and also to verify that the layers have been added properly. The following lines of code are needed for this.

Status- and Toolbar

```
BorderPane borderPane = new BorderPane();
borderPane.setTop(new GanttChartToolBar(gantt));
borderPane.setCenter(gantt);
borderPane.setBottom(new GanttChartStatusBar(gantt));
Scene scene = new Scene(borderPane);
```

Our example will now look like this after clicking on the layers button in the toolbar.



Listening to Change

Now that we have visualized our data we obviously want to interact with it and we want to be informed about the changes that we make. Our activities are, by default, editable. This means we can drag them horizontally or vertically. To receive events we only need to register an **ActivityEvent** handler with the graphics view control by calling **GraphicsBase.setOnActivityChanged()**.

Receiving Activity Events

```
graphics.setOnActivityChanged(evt -> System.out.println(evt));
```

When we run our application now we will see the following output in the console.

```

event type: DRAG_STARTED, time interval: 2014-04-17T21:45:00Z -
2014-04-22T21:30:00Z,
value (chart value / percentage complete): 0.0,
activity "null from 2014-04-18T12:15:00Z until 2014-04-23T12:00:00Z,
user object = null",
row = "Default",
layer = "Training"

event type: DRAG_ONGOING, time interval: 2014-04-17T21:45:00Z -
2014-04-22T21:30:00Z,
value (chart value / percentage complete): 0.0,
activity "null from 2014-04-18T12:15:00Z until 2014-04-23T12:00:00Z,
user object = null",
row = "Default",
layer = "Training"

event type: DRAG_FINISHED, time interval: 2014-04-17T21:45:00Z -
2014-04-22T21:30:00Z,
value (chart value / percentage complete): 0.0,
activity "null from 2014-04-18T03:00:00Z until 2014-04-23T02:45:00Z,
user object = null",
row = "Default",
layer = "Training"

```

Please notice the three different event types **DRAG_STARTED**, **DRAG_ONGOING**, and **DRAG_FINISHED**. The first one gets fired when the user initiates a drag, the second while the drag is still in progress, and the third one when the drag has finished. This pattern can be observed in JavaFX itself and it was implemented throughout **FlexGanttFX** as well. Make sure to take a look at the various event types defined in the **ActivityEvent** class to find out how much information you can receive when the user performs editing operations.

3. Controls

FlexGanttFX ships with several custom JavaFX 8 controls:

- [GanttChart](#)
- [MultiGanttChartContainer](#)
- [DualGanttChartContainer](#)
- [GraphicsBase](#)
 - [ListViewGraphics](#)
 - [VBoxGraphics](#)
 - [SplitPaneGraphics](#)
 - [SingleRowGraphics](#)
- [Timeline](#)
- [Dateline](#)
- [Eventline](#)

3.1 GanttChart

- [Introduction](#)
- [Structure](#)
- [Master / Detail Panes](#)
- [Standalone vs. Multi- / DualGanttChart](#)

Introduction

A generic JavaFX control to visualize any kind of scheduling data along a timeline. The model data needed by the control consists of rows with activities, links between activities, and layers to group activities together.

Structure

The control consists of several children controls:

- **TreeTableView**: shown on the left-hand side to display a hierarchical structure of rows
- **GraphicsBase**: shown on the right-hand side to display a graphical representation of the model data
- **Timeline**: shown above the graphics view. The timeline itself consists of two child controls.
- **Dateline**: displays days, weeks, months, years, etc...
- **Eventline**: displays various time markers

The screenshot belows shows the initial appearance of an empty Gantt chart control.



Master / Detail Panes

The Gantt chart uses two **MasterDetailPane** instances from **ControlsFX** for the high-level layout. The primary master detail pane displays the tree table as its detail node and the **secondary** master detail pane initially displays a property sheet as its detail node. The property sheet is used at development time and can be replaced with any node by calling **setDetail(Node)**. The property sheet displays a lot of properties that are used by the controls, the renderers, the system layers to fine-tune the appearance of the control. Many of them can be changed at runtime.

Standalone vs. Multi- / DualGanttChart

A Gantt chart can be used standalone or inside a **MultiGanttChartContainer** or **DualGanttChartContainer**. When used in one of these containers the position of the Gantt chart becomes important. The control can be the first chart, the last chart, the only chart, or a chart somewhere in the middle. A "first" or "only" chart always displays a timeline. A "middle" or "last" displays a special header (see **setGraphicsHeader()**). The containers are also the reason why the control distinguishes between a timeline (**getTimeline()**) and a master timeline (**getMasterTimeline()**). The master timeline is the one shown by the "first" chart, while the regular timeline is the one that belongs directly to this instance.

3.1.1 Model

The **Gantt chart** control itself doesn't really have any requirements for a model. It is simply providing convenience methods for the underlying controls (tree table view, **graphics view**). The following table lists the relevant methods:

Method	Description
<pre>void setRoot(R row); R getRoot();</pre>	Sets / gets the root node for the underlying tree table view control.
<pre>ObservableList<Layer> getLayers();</pre>	The list of layers that will be displayed by the graphics view.
<pre>ObservableList<ActivityLink<?>> getLinks();</pre>	The list of links that will be displayed by the graphics view.

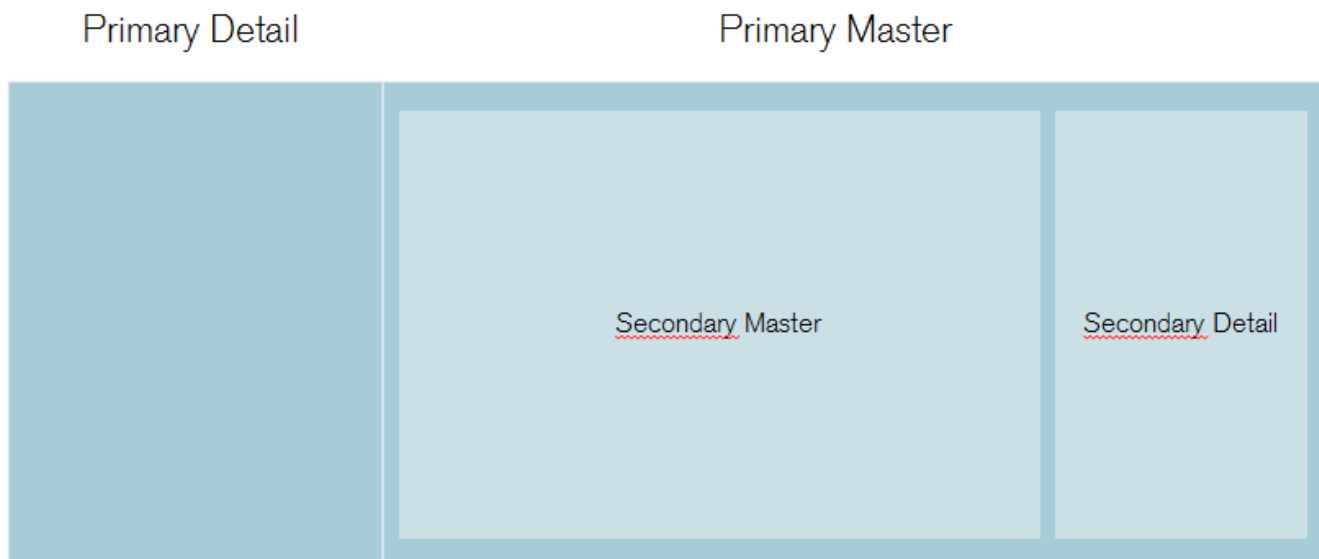
<code>ObservableList<Calendar<?>> getCalendars();</code>	The list of calendars that will be displayed by the graphics view.
--	--

3.1.2 Detail Node

The Gantt chart control is using two nested [master / detail panes](#) (from [ControlsFX](#)). The first one contains the tree table view as its detail node and the second master / detail pane as its master node. The second master / detail pane shows the [graphics view](#) as the master node and a [property sheet](#) as the detail node. The following table lists the related methods:

Method	Description
<code>setDetail(Node);</code> <code>getDetail(Node);</code>	Sets / gets the node that is being shown as the detail node of the secondary master detail pane.
<code>getPrimaryMasterDetailPane();</code>	Returns the primary master / detail pane. This pane shows the tree table view as its detail and the secondary master / detail pane as its master.
<code>getSecondaryMasterDetailPane();</code>	Returns the secondary master / detail pane. This pane shows the graphics view as its master and an optional node as its detail.

The following image illustrates the concept of two nested master / detail panes.



3.1.3 Display Mode

- [Introduction](#)
- [Standard Layout](#)
- [Table Layout](#)
- [Graphics Layout](#)

Introduction

The **displayMode** property of the [Gantt chart](#) control is used to toggle between three different layouts:

- a [standard](#) layout with the tree table view shown on the left-hand side and the graphics area on the right-hand side
- a [table-only](#) layout where the table will fill the entire width of the Gantt chart control
- a [graphics-only](#) layout where the graphics view will fill the entire width of the Gantt chart control

The display mode can be changed by calling the **setDisplayMode()** method.

Standard Layout

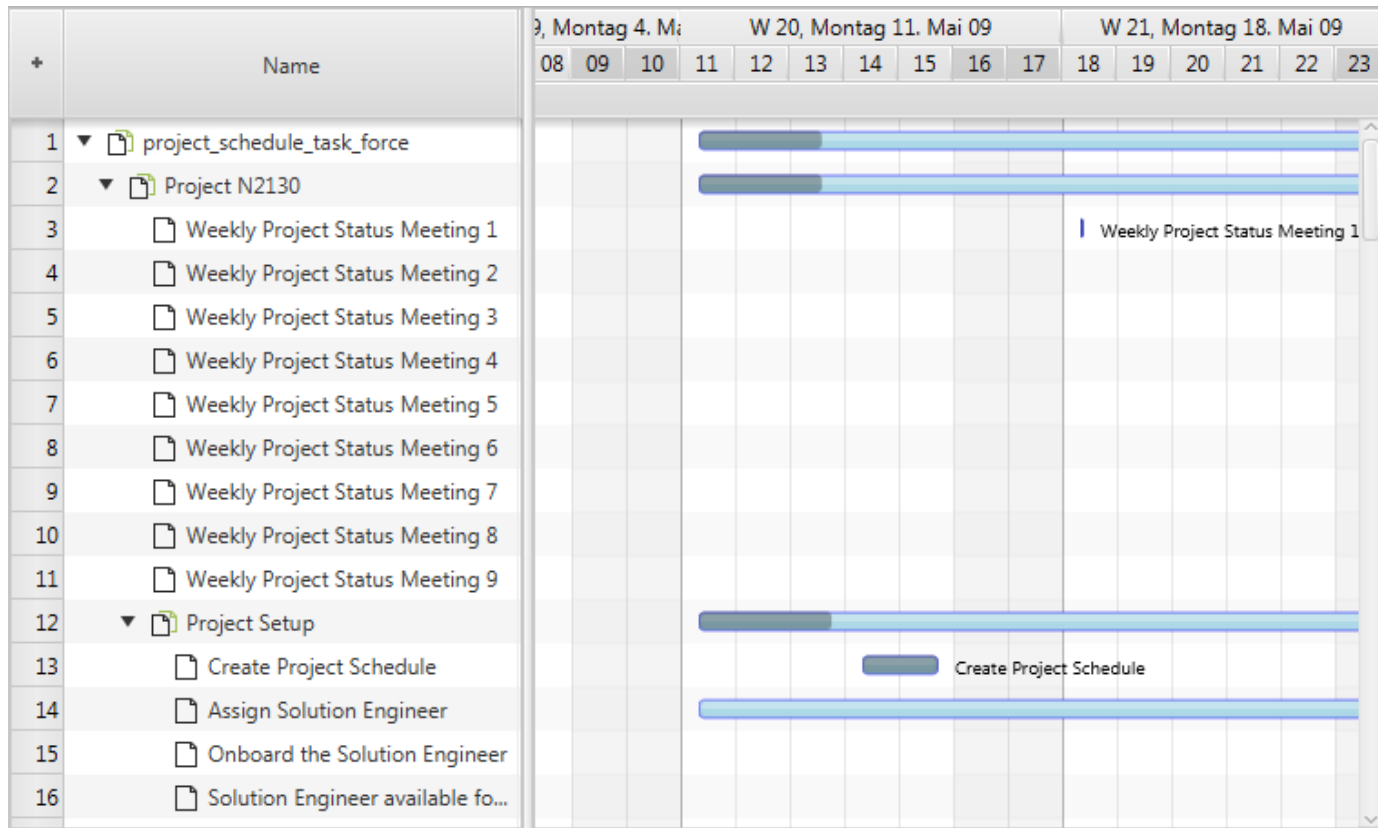
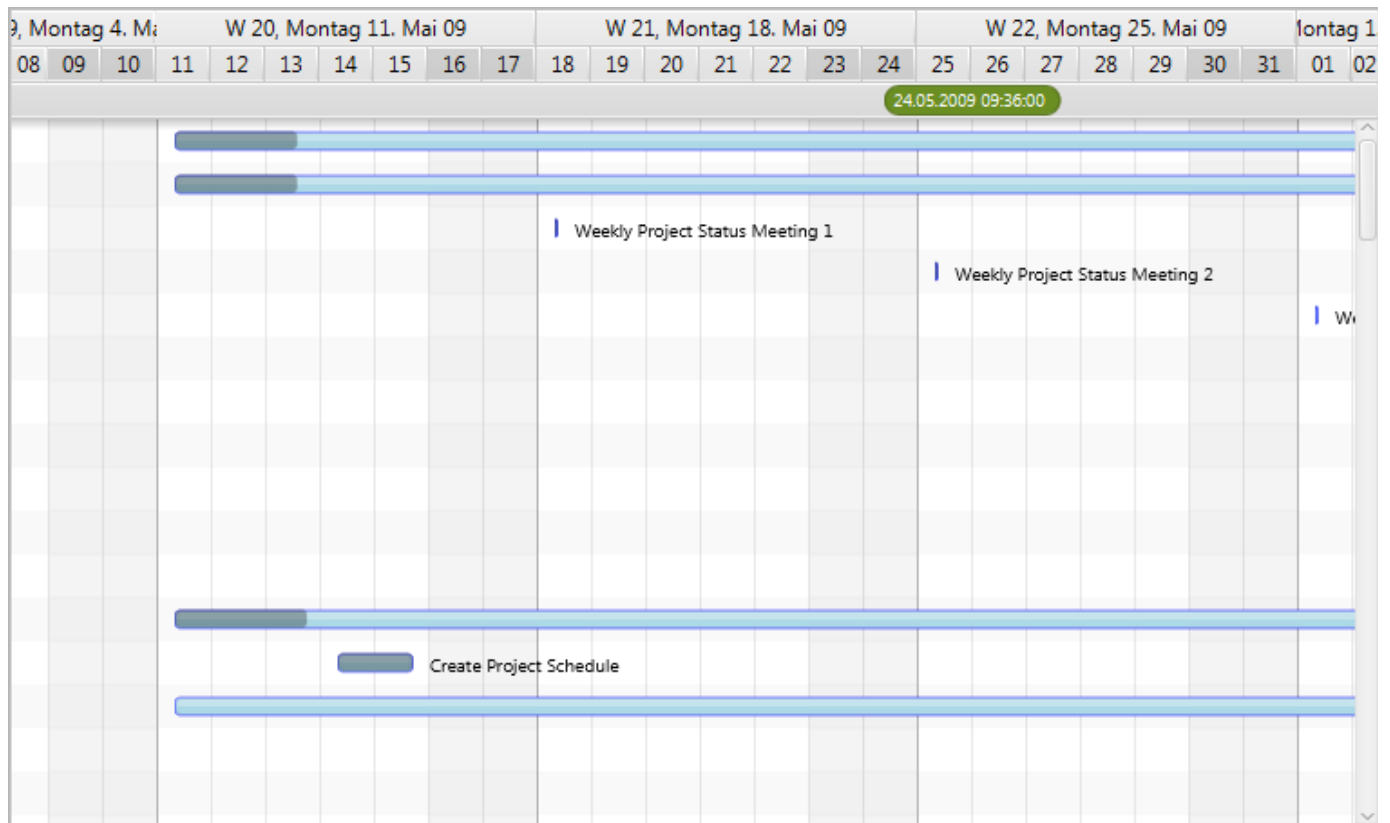


Table Layout

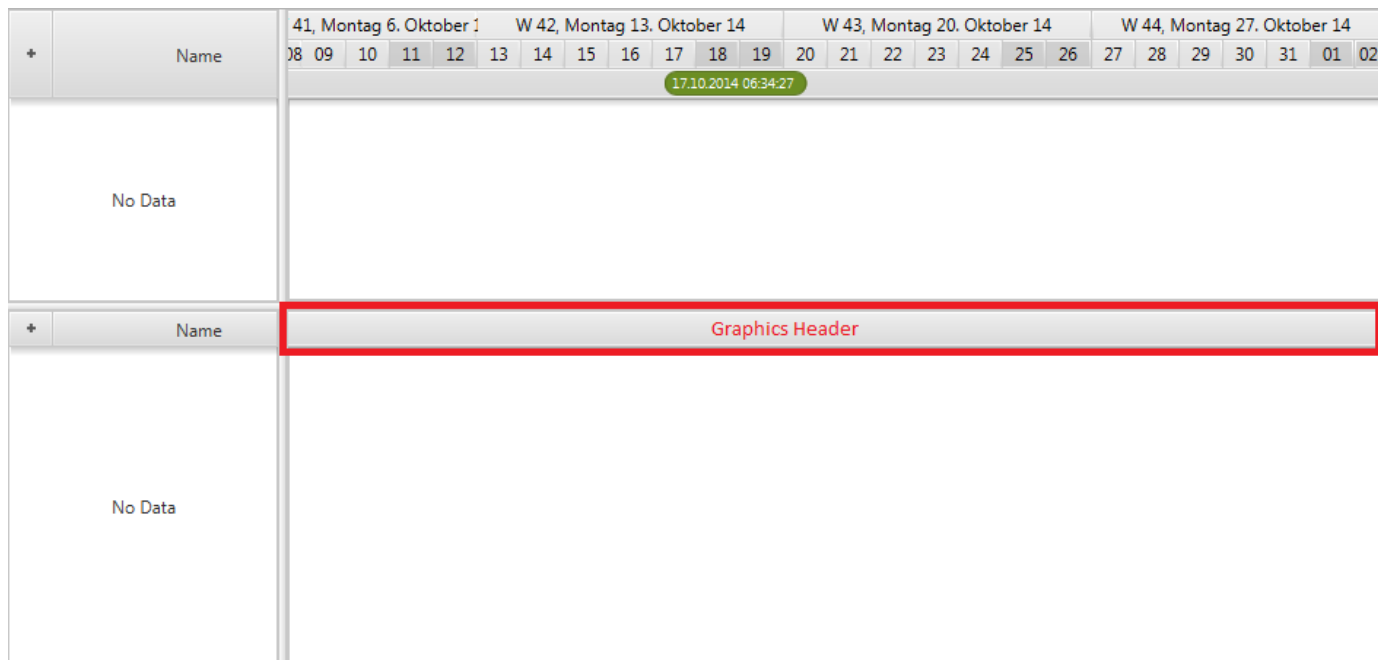
+	Name	%	Complete	Start	Finish	
1	▼ project_schedule_task_force	2 %	<div></div>	11.05.09 08:00	31.08.09 17:00	
2	▼ Project N2130	2 %	<div></div>	11.05.09 08:00	31.08.09 17:00	
3	Weekly Project Status Meeting 1	100 %	<div></div>	18.05.09 08:00	18.05.09 09:00	
4	Weekly Project Status Meeting 2	100 %	<div></div>	25.05.09 08:00	25.05.09 09:00	
5	Weekly Project Status Meeting 3	0 %	<div></div>	01.06.09 08:00	01.06.09 09:00	
6	Weekly Project Status Meeting 4	0 %	<div></div>	08.06.09 08:00	08.06.09 09:00	
7	Weekly Project Status Meeting 5	0 %	<div></div>	15.06.09 08:00	15.06.09 09:00	
8	Weekly Project Status Meeting 6	0 %	<div></div>	22.06.09 08:00	22.06.09 09:00	
9	Weekly Project Status Meeting 7	0 %	<div></div>	29.06.09 08:00	29.06.09 09:00	
10	Weekly Project Status Meeting 8	0 %	<div></div>	06.07.09 08:00	06.07.09 09:00	
11	Weekly Project Status Meeting 9	0 %	<div></div>	13.07.09 08:00	13.07.09 09:00	
12	▼ Project Setup	8 %	<div></div>	11.05.09 08:00	10.06.09 17:00	
13	Create Project Schedule	100 %	<div></div>	14.05.09 08:00	15.05.09 17:00	
14	Assign Solution Engineer	0 %	<div></div>	11.05.09 08:00	05.06.09 17:00	
15	Onboard the Solution Engineer	0 %	<div></div>	08.06.09 08:00	09.06.09 17:00	
16	Solution Engineer available fo...	0 %	<div></div>	09.06.09 17:00	09.06.09 17:00	

Graphics Layout



3.1.4 Graphics Header

The graphics header node is a replacement for the timeline when the [Gantt chart](#) control is being used in a multi Gantt chart context, for example when used in a [DualGanttChartContainer](#) or a [MultiGanttChartContainer](#).



This node can be set by calling **GanttChart.setGraphicsHeader(Node)**. The node passed to this method can be anything. The only important thing to be aware of is that the preferred height of this node has to be set to the same value as the preferred height of the tree table header.

3.1.5 Row Header

- [Introduction](#)
- [Row Header Type](#)
- [Row Header Factory](#)

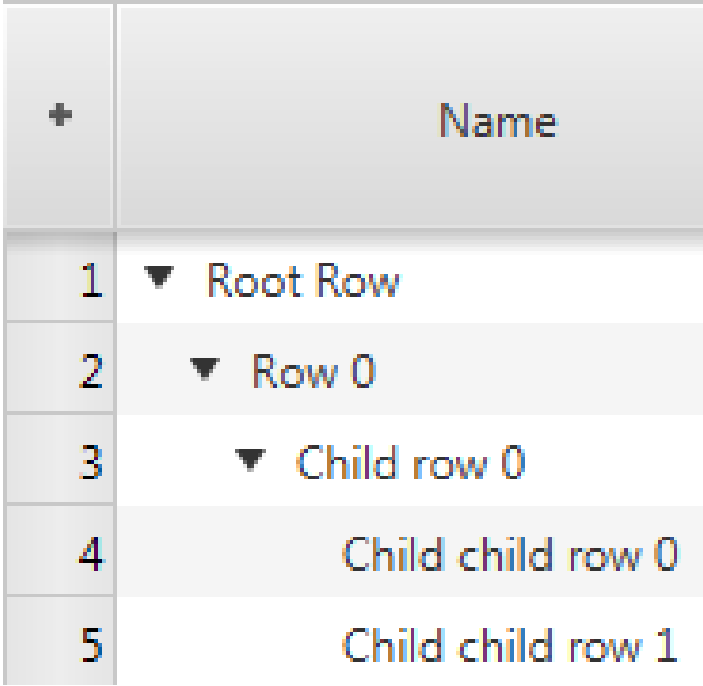
Introduction

The first column in the tree table view is called "row header". This column is provided by the framework and can not be removed. By default it is used to display row numbers but can be reused for other purposes. The RowHeader class is a subclass of TreeTableColumn with some special logic to it. It supports row resizing and might call back on a factory to produce its graphic.

Row Header Type

The enumerator **RowHeaderType** defines the different ways the row header can be used.

Value	Description												
GRAPIC_NODE	Makes the row header cells display a custom node for each row. <div><table><tr><td>+</td><td>Name</td></tr><tr><td>⦿</td><td>▼ Root Row</td></tr><tr><td>⚠</td><td>▼ Row 0</td></tr><tr><td></td><td>▼ Child row 0</td></tr><tr><td>⦿</td><td>Child child row 0</td></tr><tr><td></td><td>Child child row 1</td></tr></table></div>	+	Name	⦿	▼ Root Row	⚠	▼ Row 0		▼ Child row 0	⦿	Child child row 0		Child child row 1
+	Name												
⦿	▼ Root Row												
⚠	▼ Row 0												
	▼ Child row 0												
⦿	Child child row 0												
	Child child row 1												
LEVEL_NUMBER	Makes the row header cells display the level number of the current row (1, 1.1, 1.2, 2, 2.1, 2.2, 2.3, ...). <div><table><tr><td>+</td><td>Name</td></tr><tr><td></td><td>▼ Root Row</td></tr><tr><td>1</td><td>▼ Row 0</td></tr><tr><td>1.1</td><td>▼ Child row 0</td></tr><tr><td>1.1.1</td><td>Child child row 0</td></tr><tr><td>1.1.2</td><td>Child child row 1</td></tr></table></div>	+	Name		▼ Root Row	1	▼ Row 0	1.1	▼ Child row 0	1.1.1	Child child row 0	1.1.2	Child child row 1
+	Name												
	▼ Root Row												
1	▼ Row 0												
1.1	▼ Child row 0												
1.1.1	Child child row 0												
1.1.2	Child child row 1												

ROW_NUMBER	Makes the row header cells display the number of the current row (1, 2, 3,).	
		

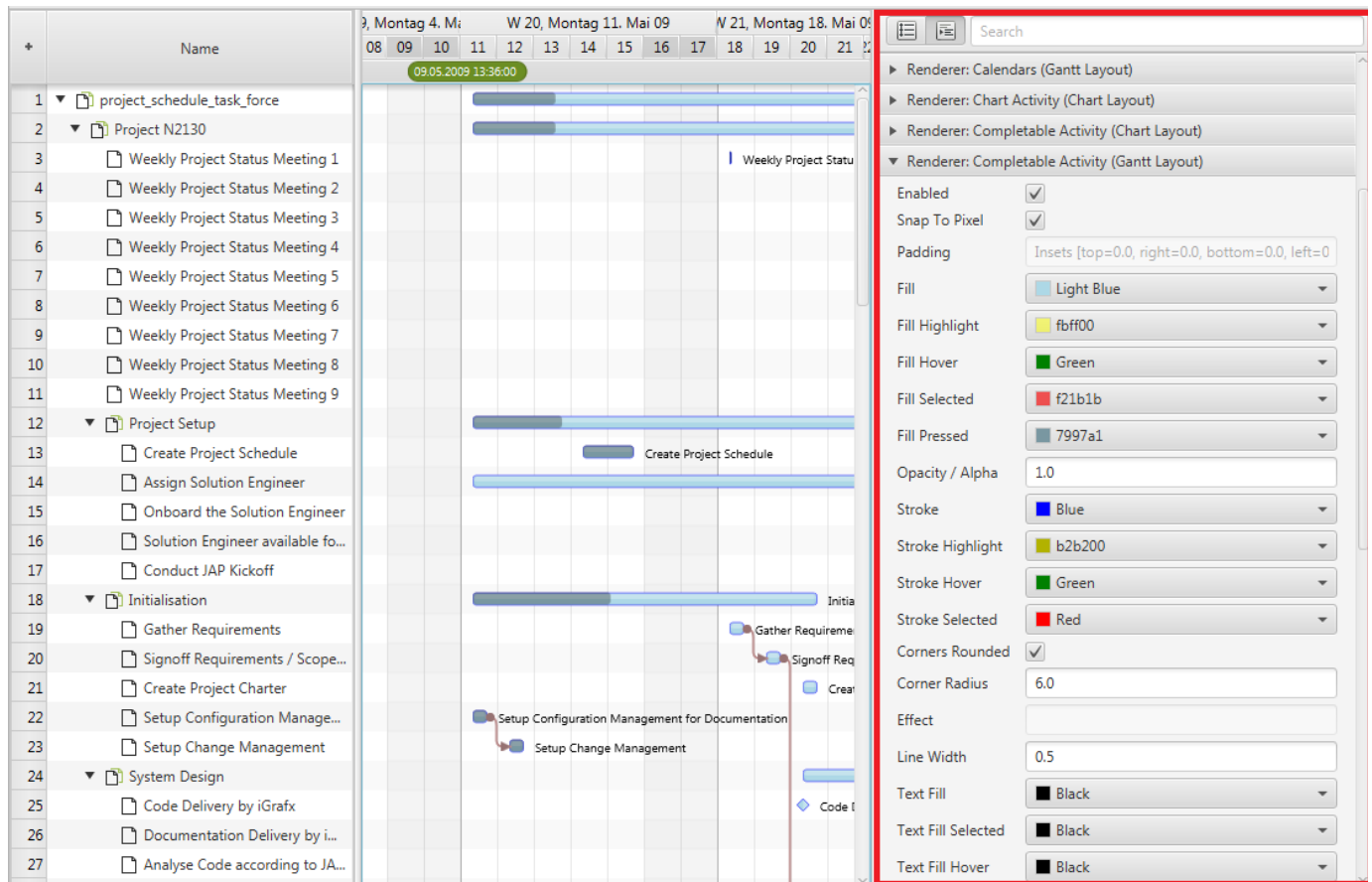
Row Header Factory

If the graphic node header type is chosen then the row header will call back on the row header node factory that is supplied by the GanttChart class. The following example shows how to register a possible implementation of such a factory.

Row Header Node Factory
<pre>ganttChart.setRowHeaderNodeFactory(row -> { public Node call(R row) { Button delete = new Button("Delete"); delete.setOnAction(evt -> deleteRow(row)); return delete; } });</pre>

3.1.6 Property Sheet

The default control used for the Gantt chart detail node property is the [property sheet](#) from the [ControlsFX](#) open source project. It is used to display the properties of the [Gantt chart](#) itself and its subcontrols ([timeline](#), [dateline](#), [eventline](#), [graphics](#)). It also shows the properties of all [renderers](#) registered with the controls. The screenshot below shows the property sheet as it presents itself when the detail node of the primary master detail pane becomes visible. It can be made visible by calling **GanttChart.setShowDetail(true)**.



When writing your own renderers you can override the **getPropertySheetItems()** method and add your own items to the list of items returned by the superclass.

3.1.7 Other Features

- [Introduction](#)
- [Fixed cell size](#)
- [Master Timeline](#)
- [Tree Table Scrollbar](#)
- [Timeline Scrollbar](#)
- [Position](#)
- [Factory Methods](#)

Introduction

This page describes several of the smaller and normally less important features of the Gantt chart control.

Fixed cell size

The tree table view and the list view of JavaFX both support a property called **fixedCellSize**. It can be used to improve the performance of both controls. This is done by setting it to a value other than -1. A value like that informs the controls that each cell will have the same height, which allows for faster algorithms to be used when updating the controls. The **GanttChart** class also defines this property in order to ensure that the tree table view and the list view used by it use the same cell size. If set the Gantt chart will not use the **height** property of the **rows** and will also not allow the user to resize the rows.

Master Timeline

The **GanttChart** control defines a property called **masterTimeLine**. This property is used when the Gantt chart is being used in a multi Gantt

chart context (e.g. [DualGanttChartContainer](#) or [MultiGanttChartContainer](#)). In these situations it is the timeline of the top Gantt chart that is the basis for rendering weekends, grid lines, etc. Every Gantt chart still has its own [timeline](#) subcontrol but they will all know which one is the master.

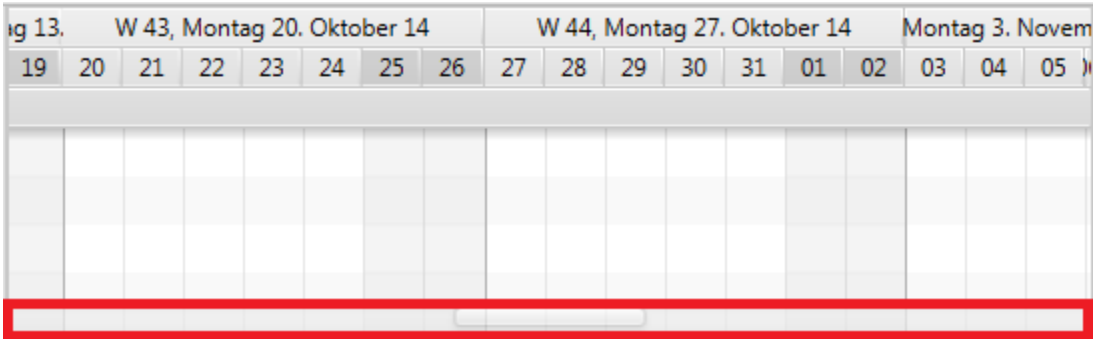
This is framework functionality that applications should normally not interfere with.

Tree Table Scrollbar

Another subcontrol found in the **GanttChart** control is the tree table scrollbar. In *FlexGanttFX* the Gantt chart manages its own horizontal scrollbar for the tree table view. This is done so that the scrollbar can be placed inside a [HiddenSidesPane](#) control from the [ControlsFX](#) project.

Timeline Scrollbar

The graphics view also uses a [HiddenSidesPane](#) for its horizontal scrolling control. However, this control is not a regular scrollbar but a specialization of the [PlusMinusSlider](#) control from ControlsFX. The [TimelineScrollBar](#) allows the user to scroll to the left and right at different speeds, depending on how far the thumb is away from the center location.

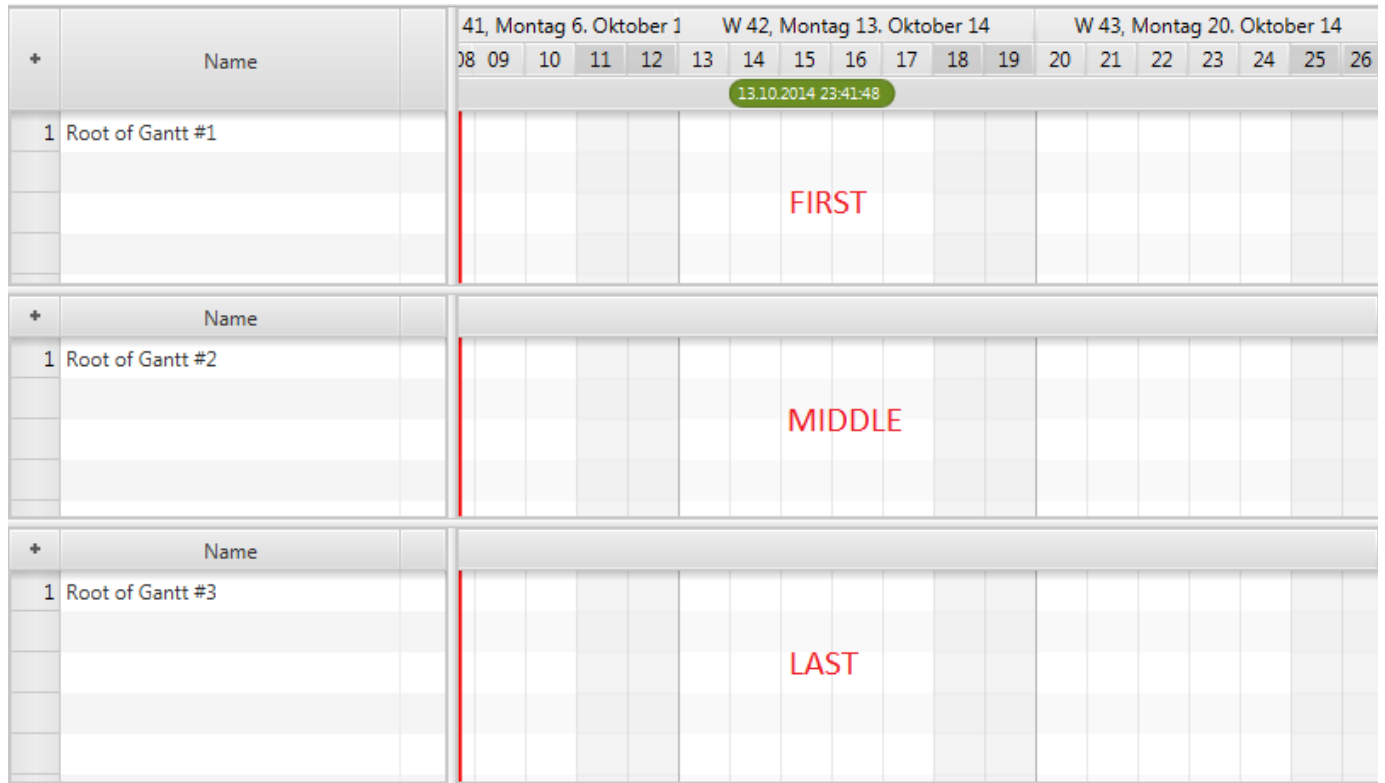


Position

The position property of the **GanttChart** class is used to inform the Gantt chart where it is located within a multi Gantt chart context (e.g. [DualGanttChartContainer](#) or [MultiGanttChartContainer](#)). Possible values are:

Value	Description
ONLY	The Gantt chart is the only one. This is the default value and will not change if not used in a multi Gantt chart context.
FIRST	The Gantt chart is shown at the top of the container.
MIDDLE	The Gantt chart is not the first and not the last one. It is also not the only one.
LAST	The Gantt chart is shown at the bottom of the container.

The screenshot below shows three charts in a [MultiGanttChartContainer](#) and their position values.



This is framework functionality that applications should normally not interfere with.

Factory Methods

There are several protected factory methods used for creating the subcontrols. These methods can be overridden to create subclasses of these controls.

Method	Description
TreeTableView createTreeTable();	Creates the tree table view shown on the left-hand side of the Gantt chart. A typical use case for replacing this table is when you already have a tree table view specialization with advanced filtering or interaction options. You might want to use the same tree table view that your application is already using in other places.
Timeline createTimeline();	Creates the timeline.
GraphicsBase createGraphics();	Creates the graphics view. A use case for replacing the standard one might be that your application adds a couple of nodes to the graphics view. Maybe some kind of overlap on top of the graphics (e.g. a radar).
RowHeader createRowHeader();	Creates the row header column for the tree table view.

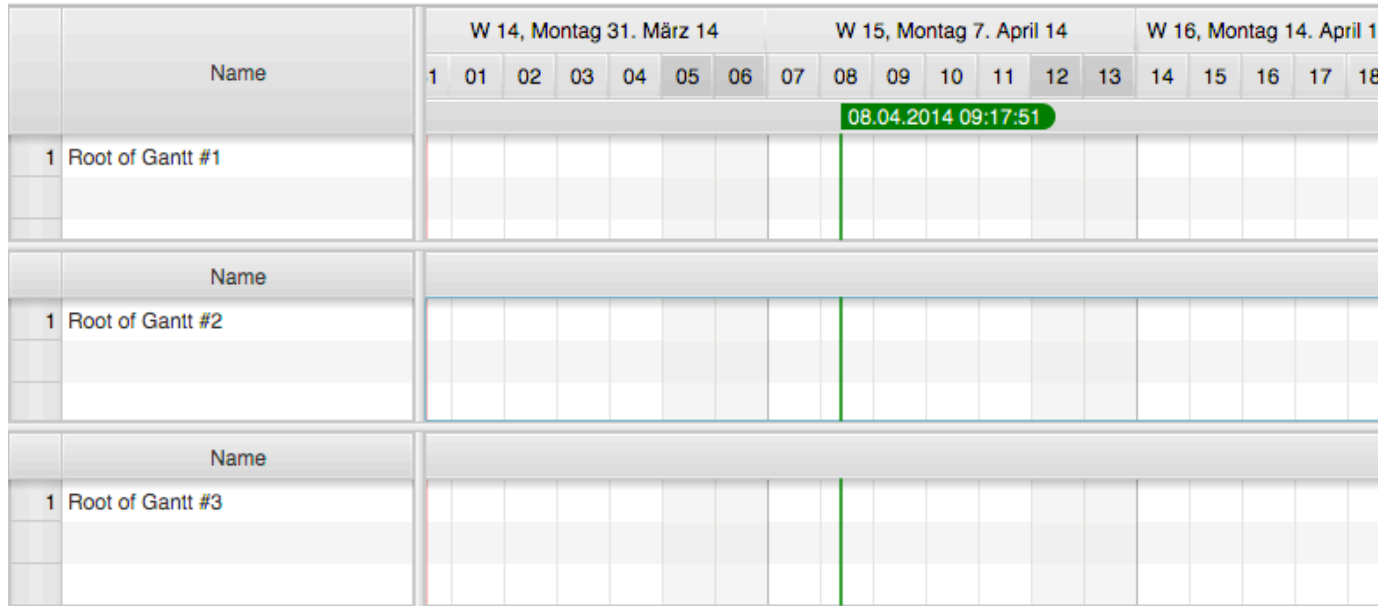
3.2 MultiGanttChartContainer

- [Introduction](#)

Introduction

A container capable of displaying multiple instances of [GanttChart](#) and keeping their layouts (same table width, same timeline) and their scrolling

and zooming behaviour in synch. The screenshot below shows the initial appearance of an empty multi Gantt chart container.

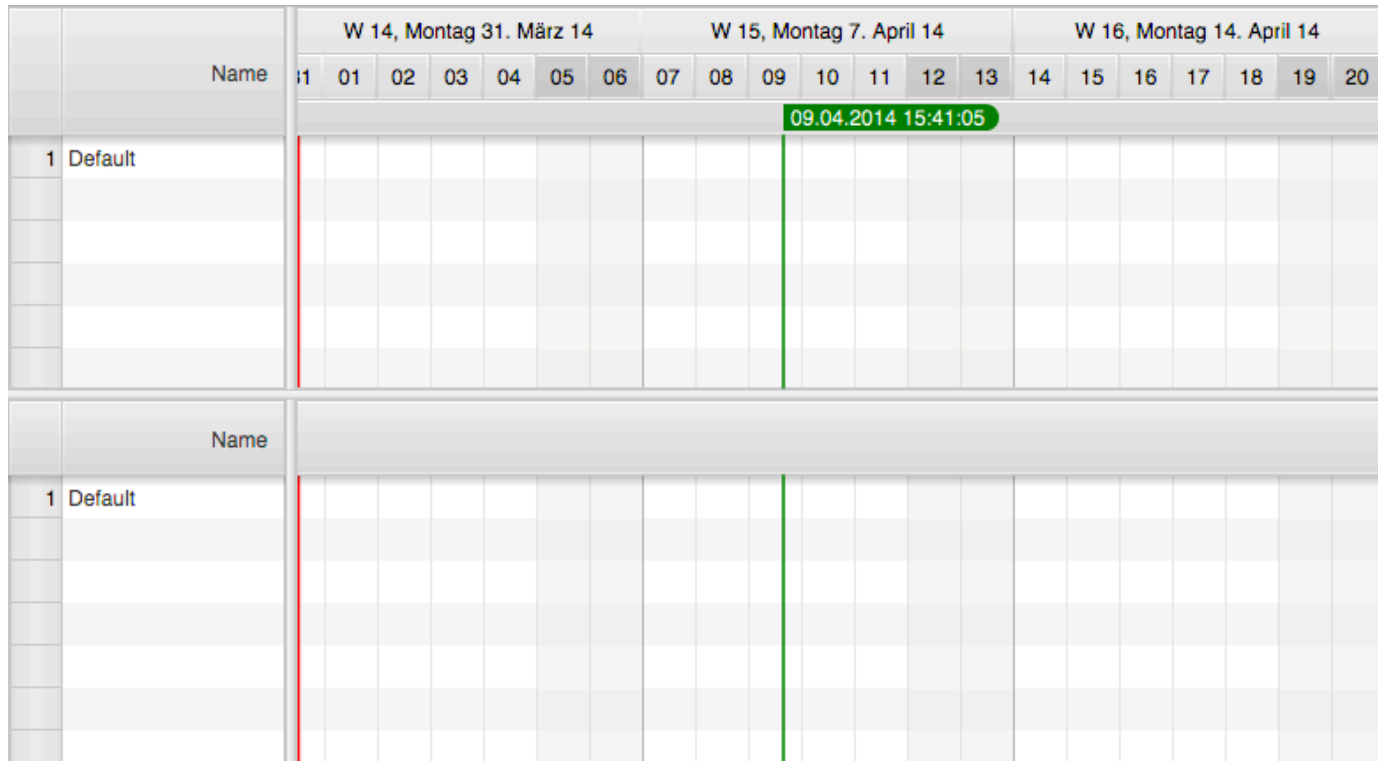


3.3 DualGanttChartContainer

- Introduction

Introduction

A specialization of [MultiGanttChartContainer](#) capable of displaying exactly two instances of [GanttChart](#) and keeping their layouts (same table width, same timeline) and their scrolling and zooming behavior in synch. The container distinguishes between a primary and a secondary Gantt chart, where the secondary Gantt chart is located in the detail node section of a [MasterDetailPane](#). It can be hidden or shown on demand. Each one of the two Gantt charts can have its own header and footer. Initially only the primary header and the secondary footer are used. The header for displaying an instance of [GanttChartToolBar](#) and the footer for displaying an instance of [GanttChartStatusBar](#). The screenshot below shows the initial appearance of an empty Dual Gantt chart control.



3.4 QuadGanttChartContainer

- Introduction

Introduction

A specialization of `MultiGanttChartContainer` capable of displaying exactly four instances of `GanttChart` and keeping their layouts (same table width, same timeline) and their scrolling and zooming behavior in synch. The container distinguishes between the Gantt chart locations `UPPER_LEFT`, `UPPER_RIGHT`, `LOWER_LEFT`, `LOWER_RIGHT`. The timelines of the `UPPER_LEFT` and `LOWER_LEFT` Gantt charts are scrolling in sync and the timelines of the `UPPER_RIGHT` and `LOWER_RIGHT` are scrolling in sync.

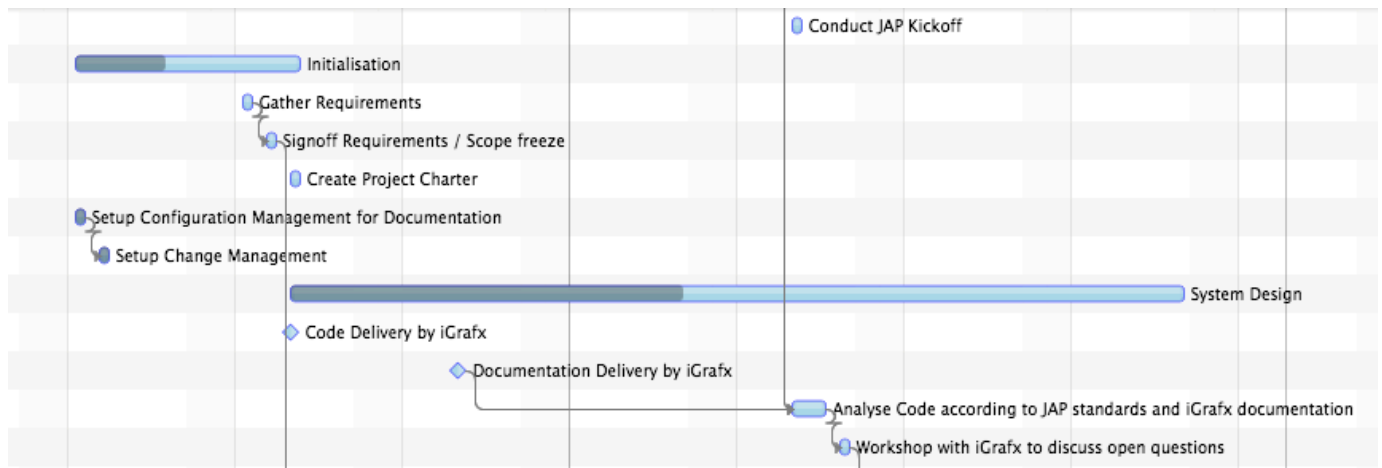
[illegible]

3.5 GraphicsBase

- Introduction
- Rendering
- System Layers
- Editing
- Events
- Hitpoint Detection
- Context Menu

Introduction

The graphics view control is responsible for the rendering of activities and system layers, editing of activities, event notifications, hit detection, system layer management, and for context menu support.



Rendering

The graphics control uses the **Canvas** node and the direct drawing API of it (as opposed to the deferred rendering done via the Scenegraph). This is due to the large data volumes often displayed by Gantt charts. Directly rendering an activity into a bitmap is much faster than updating the scene graph, reapplying CSS styling, laying out nodes. The graphics control uses a pluggable renderer architecture where renderer instances can be mapped to activity types, very similar to the way Swing was doing it. The following code is an example of how to register a custom renderer for a given "Flight" activity and layout type. Please note that the graphics view is capable of displaying activities in three different layouts, hence the layout type must also be passed to the method.

Renderer Registration

```
GanttChart ganttChart = new GanttChart();
GraphicsBase<?> graphics = ganttChart.getGraphics();
graphics.setActivityRenderer(
    Flight.class, // the type of activities that will be rendered
    GanttLayout.class, // the type of layout where the renderer will be used
    new FlightRenderer(graphics)); // the actual renderer instance
```

System Layers

Activities are not the only thing that need to be displayed. There are also the current time ("now"), grid lines, inner lines, agenda / chart lines, and so on. All of these things are rendered by so-called [system layers](#). The graphics control manages two lists of these layers. One list for background layers and one list for foreground layers.

Background layers are drawn "behind" activities, foreground layers are drawn "in front of" activities. Each one of these lists are already pre-populated but can be changed by the application. For more information on the available system layers, please refer to their [individual documentation](#).

System layers can be turned on and off directly via the API of the graphics control. There is a boolean property for each layer. The value of these properties can be set by calling the methods that follow the pattern **setShowXYZLayer**. System layers that are controlled like this will appear and disappear with a fade in / fade out animation, while calling **SystemLayer.setVisible(boolean)** directly will be without any animation.

Editing

Two different callbacks are used to control the editing behaviour of activities. The first maps a mouse event / mouse location to an `GraphicsBase.EditMode` and can be registered by calling `setEditModeCallback(Class, Class, Callback)`. The second callback is used to determine whether a given editing mode / operation can be applied to an activity at all. This callback is registered by calling `setActivityEditingCallback(Class, Callback)`. Most applications will only need to work with the second callback and keep the defaults for the edit mode locations (for example: right edge used to change end time, left edge used to change start time).

Events

Events of type `ActivityEvent` are sent whenever the user performs a change inside the graphics view. Applications that want to receive these events can either call any one of the `setOnActivityXYZEvent()` methods or by adding an event handler directly via `addEventHandler(ActivityEvent.ACTIVITY_XYZ, ...)`. Events are fired while the change is being performed and once it has been completed. For this the `ActivityEvent` class lists event types with the two different endings `CHANGING` and `CHANGED`.

Hitpoint Detection

The graphics view provides support for finding out information about a given position. Activities can be found by calling `getActivityBoundsAt(double, double)` or `getActivityRefAt(double, double)`. The time at an x-coordinate can be looked up by calling `getTimeAt(double)`. The opposite direction is also available: a location can be found for a given time by calling `getLocation(Instant)`.

Context Menu

Context menus can be set on any control in JavaFX but due to the complexity of the graphics view it does make sense to provide additional built-in support. By calling `setContextMenuCallback(Callback)` a context menu specific callback can be registered with the graphics control. This callback will be invoked when the user triggers the context menu. A callback parameter object (see `GraphicsBase.ContextMenuParameter`) will be passed to the callback already populated with the most important values that might be relevant for building a context menu.

3.4.1 System Layers

- [Introduction](#)
- [Available Layers](#)

Introduction

System layers are used in the background and foreground of each row. A background layer gets drawn before the activities are drawn while a foreground layer gets drawn after the activities are drawn. Each layer is specialized on drawing one type of information: current time, selected time intervals, grid lines, and so on. The [graphics view](#) manages the layers in two lists and provides convenience methods to easily look them up.

Method	Description
<code>getBackgroundSystemLayers()</code>	Returns the complete list of system layers used in the background of activities.
<code>getForegroundSystemLayers()</code>	Returns the complete list of system layers used in the foreground of activities.
<code>getBackgroundSystemLayer(Class)</code>	Returns the system background layer instance of the given type.
<code>getForegroundSystemLayer(Class)</code>	Returns the system foreground layer instance of the given type.
<code>getSystemLayer(Class)</code>	Returns the system layer instance of the given type, no matter if it is a foreground or background layer.

- Layers can be added to or removed from the graphics view by adding them to or removing them from the foreground or background list.
- Once you have looked up a layer you can set its properties to customize its appearance. The most common properties are used for line colors and widths.

System Layer Example

```
GraphicsBase<?> graphics = ganttChart.getGraphics();
NowLineLayer nowLayer = graphics.getBackgroundSystemLayer(NowLineLayer.class);
nowLayer.setStroke(Color.ORANGE);
nowLayer.setLineWidth(3); // thick line
```

System Layers vs. Model Layers

Please note that system layers are not related in any way to [model layers](#). A system layer is basically a renderer for some graphical feedback while a model layer is used for grouping activities.

Available Layers

The following table lists all system layers that are shipping with **FlexGanttFX**. The last two columns (FG, BG) are used to indicate whether the layer is used as a foreground or as a background layer.

Layer	Description	FG	BG
AgendaLinesLayer	Draws the horizontal grid lines for a row if the row or any of its inner lines are using the agenda layout .		✓
CalendarLayer	Draws the entries returned by the calendars attached to a row or attached to the entire graphics view. The calendar layer uses pluggable renderers that are mapped to the entry types. Applications can register their own renderers by calling CalendarLayer.setCalendarActivityRenderer() .		✓
ChartLinesLayer	Draws the horizontal grid lines for a row if the row or any of its inner lines are using the chart layout .		✓
GridLinesLayer	Draws the vertical grid lines based on the scale resolutions currently present in the dateline . The layer can be configured to display 0 to 3 grid line levels. If the dateline is, for example, showing days and weeks then a level of 2 would cause the layer to draw grid lines for days and weeks, while a grid line level of 1 would only render grid lines for days.		✓
HoverTimeIntervalLayer	Draws the hover time interval specified by the dateline . If the mouse cursor hovers over a week in the dateline then the layer will fill the time interval defined by this week with a highlighting color.		✓
InnerLinesLayer	Draws separator lines between inner lines. <div>By default the line width property of this layer is set to 0 and the lines will not be drawn at all. To change this simply set a line width greater than 0.</div>		✓
LayoutLayer	Draws the layout padding areas. Each layout may have some padding added to its top and bottom. This layer fills the padding area with a solid color.		✓
NowLineLayer	Draws a vertical line at the location of the current time / now time. The current time is defined in the timeline model .		✓
RowLayer	Draws the background of each row. The layer can be configured with pluggable renderers that are mapped to the type of the row. Applications can register their own renderers by calling RowLayer.setRowRenderer() . For more information please read 3.4.7 Row Rendering .		✓
ScaleLayer	Draws a scale for an entire row or for each line within the row. Scales vary depending on the layout used for the row / line. The scale for the chart layout displays the minimum and maximum values while the scale for the agenda layout displays a time scale (8am, 9am, 10am,). The labels and dashes in the scale layer have to align perfectly with the lines drawn by the agenda lines layer and the chart lines layer.	✓	
SelectedTimeIntervalsLayer	Draws the time intervals that were selected by the user (or the application) in the dateline .		✓
ZoomIntervalLayer	Draws the zoom interval as defined by the timeline. The zoom interval gets created by the user via the help of the timeline lasso.		✓

3.4.2 Drag & Drop

- [Introduction](#)
- [Events](#)
- [Drag And Drop Info Property](#)
- [Feedback Types](#)
- [Drag Image Provider](#)

Introduction

The platform (Windows, Mac) provided drag and drop (DnD) facilities are used in **FlexGanttFX** only to move an activity from one [row](#) and to another. All other editing operations are handled with standard mouse events (pressed, dragged). The new row might actually be a row in another [Gantt chart](#). The default way to initiate a DnD is to move the mouse cursor into the center of an [activity](#) while pressing the **SHIFT** key. This will change the cursor to the DnD cursor if this kind of editing operation is supported by the targeted activity (see also "[3.4.4 Activity Editing](#)"). The DnD will terminate once the user lets go of the mouse button.

Events

Just like all the other editing operations DnD will also trigger several events during its execution. The following table lists them:

Event Type	Description

DRAG_STARTED DRAG_ONGOING DRAG_FINISHED	These event types are fired if the editing operation is EditMode.DRAGGING .
VERTICAL_DRAG_STARTED VERTICAL_DRAG_ONGOING VERTICAL_DRAG_FINISHED	These event types are fired if the editing operation is EditMode.DRAGGING_VERTICAL .

The edit mode DRAGGING_HORIZONTAL does not use platform DnD. Hence the event types HORIZONTAL_DRAG_STARTED / ONGOING / FINISHED are not listed above.

Drag And Drop Info Property

A special property called **dragAndDropInfo** is available on the [graphics view](#) to monitor the DnD operation. This is in addition to the standard event types mentioned above. The info stored in this property provides the application with the most important information required about the dragged activity.

Field	Description
row	The row over which the mouse cursor / the dragged activity is currently hovering.
activityBounds	The bounds of the dragged activity (contains an activity reference and the actual activity).
dragEvent	The last drag event (drag ongoing or drag dropped).
dropInterval	The time interval where the activity would be or was actually dropped.
offset	The offset where the mouse grabbed the activity (needed for visual feedback of the drag).

Feedback Types

FlexGanttFX provides different ways of visualizing the DnD feedback. The enumerator **DragAndDropFeedback** lists the following values which can be set by calling the **setDragAndDropFeedback()** method on **GraphicsBase**.

Value	Description
NATIVE	A snapshot image of the activity will be taken and placed below the mouse cursor. The image will be set at the moment the drag gesture gets recognized. Optionally a drag image provider can be used. <div> The size of the image might be different than the size of the activity (platform-specific). </div>
RENDERED	The dragged activity will be constantly rendered on a separate canvas on top of the graphics area. The activity is guaranteed to keep its original size.
RENDERED_GRID_SNAPPED	The dragged activity will be constantly rendered on a separate canvas on top of the graphics area. The activity is guaranteed to keep its original size. The currently active grid will be used to make the dragged activity snap to the grid locations.

Drag Image Provider

If the DnD feedback type has been set to **NATIVE** then it is possible to pass a custom image for the drag operation. This can be accomplished by setting a drag image provider on **GraphicsBase** by calling **setDragImageProvider()**. This method accepts a callback lambda expression. The input for the callback will be an [ActivityRef](#) and the output will be an image.

Drag Image Provider

```
GraphicsBase<?> graphics = ganttChart.getGraphics();
graphics.setDragImageProvider(ref -> createImage(ref));
```

The default image is a snapshot of the activity at the moment when the drag started.

3.4.3 Event Handling

- [Introduction](#)
- [Activity Events](#)
- [Activity Events Hierarchy](#)
- [Activity Event Properties](#)
- [Lasso Events](#)
- [Lasso Event Hierarchy](#)
- [Lasso Info](#)
- [Links / Further Reading](#)

Introduction

The [graphics view](#) fires standard JavaFX events in order to let applications react to change. The concepts used for event handler support in **Flex GanttFX** are the same as the ones found in the standard JavaFX controls.

Activity Events

Activity events are fired whenever the user deletes or edits an activity. To receive an activity event simply register an event handler with the [graphics view](#) via one of the convenience methods.

Single Activity Event Handler

```
GraphicsBase<?> graphics = ganttChart.getGraphics();
graphics.setOnActivityChangeFinished(evt ->
    System.out.println("An activity has changed"));
```

If you need to register more than one handler for a specific event type then use this approach:

Multiple Activity Event Handlers

```
GraphicsBase<?> graphics = ganttChart.getGraphics();
graphics.addEventHandler(ActivityEvent.ACTIVITY_CHANGE_FINISHED,
    evt -> System.out.println("Listener 1"));
graphics.addEventHandler(ActivityEvent.ACTIVITY_CHANGE_FINISHED,
    evt -> System.out.println("Listener 2"));
```

The following table lists all supported activity event types and the convenience setter methods of the [graphics view](#). These methods are used to quickly register an event handler for the given event type.

Event Types	Methods	Description
ACTIVITY_DELETED	<code>setOnActivityDeleted()</code>	Fired whenever the user deletes an activity via the backspace key.

ACTIVITY_CHANGE	setOnActivityChanged()	The parent event type of all activity changes. Can be used to to receive a notification for any kind of activity change.
ACTIVITY_CHANGE_STARTED ACTIVITY_CHANGE_ONGOING ACTIVITY_CHANGE_FINISHED	setOnActivityChangeStarted() setOnActivityChangeOngoing() setOnActivityChangeFinished()	Fired whenever an activity change has started, is ongoing, or has finished.
CHART_HIGH_VALUE_CHANGE_STARTED CHART_HIGH_VALUE_CHANGE_ONGOING CHART_HIGH_VALUE_CHANGE_FINISHED	setOnChartHighValueChangeStarted(); setOnChartHighValueChangeOngoing(); setOnChartHighValueChangeFinished();	Fired whenever the user has started editing, is in the process of editing, or has finished editing the "high" value of a high / low chart activity.
CHART_LOW_VALUE_CHANGE_STARTED CHART_LOW_VALUE_CHANGE_ONGOING CHART_LOW_VALUE_CHANGE_FINISHED	setOnChartLowValueChangeStarted(); setOnChartLowValueChangeOngoing(); setOnChartLowValueChangeFinished();	Fired whenever the user has started editing, is in the process of editing, or has finished editing the "low" value of a high / low chart activity.
CHART_VALUE_CHANGE_STARTED CHART_VALUE_CHANGE_ONGOING CHART_VALUE_CHANGE_FINISHED	setOnChartValueChangeStarted(); setOnChartValueChangeOngoing(); setOnChartValueChangeFinished();	Fired whenever the user has started editing, is in the process of editing, or has finished editing a chart value of a chart activity.
DRAG_STARTED DRAG_ONGOING DRAG_FINISHED	setOnActivityDragStarted(); setOnActivityDragOngoing(); setOnActivityDragFinished();	Fired whenever the user has started dragging, is in the process of dragging, or has finished dragging an activity via platform-provided drag & drop. This event type is used when the user can freely move the activity around, vertically and horizontally.
END_TIME_CHANGE_STARTED END_TIME_CHANGE_ONGOING END_TIME_CHANGE_FINISHED	setOnActivityEndTimeChangeStarted(); setOnActivityEndTimeChangeOngoing(); setOnActivityEndTimeChangeFinished();	Fired whenever the user has started changing, is in the process of changing, or has finished changing the end time of an activity.
HORIZONTAL_DRAG_STARTED HORIZONTAL_DRAG_ONGOING HORIZONTAL_DRAG_FINISHED	setOnActivityHorizontalDragStarted(); setOnActivityHorizontalDragOngoing(); setOnActivityHorizontalDragFinished();	Fired whenever the user has started changing, is in the process of changing, or has finished changing the time interval (start <u>and</u> end time) of an activity. Changing this time interval makes the activity move horizontally, either to the right (future) or the left (past).
PERCENTAGE_CHANGE_STARTED PERCENTAGE_CHANGE_ONGOING PERCENTAGE_CHANGE_FINISHED	setOnActivityPercentageChangeStarted(); setOnActivityPercentageChangeOngoing(); setOnActivityPercentageChangeFinished();	Fired whenever the user has started changing, is in the process of changing, or has finished changing the "percentage complete" value of an activity.
START_TIME_CHANGE_STARTED START_TIME_CHANGE_ONGOING START_TIME_CHANGE_FINISHED	setOnActivityStartTimeChangeStarted(); setOnActivityStartTimeChangeOngoing(); setOnActivityStartTimeChangeFinished();	Fired whenever the user has started changing, is in the process of changing, or has finished changing the start time of an activity.

VERTICAL_DRAG_STARTED	setOnActivityVerticalDragStarted();	Fired whenever the user has started dragging, is in the process of dragging, or has finished dragging an activity via platform-provided drag & drop. This event type is used when the user can only drag the activity vertically (reassign an activity to a different row).
VERTICAL_DRAG_ONGOING	setOnActivityVerticalDragOngoing();	
VERTICAL_DRAG_FINISHED	setOnActivityVerticalDragFinished();	

Activity Events Hierarchy

The event types defined in the **ActivityEvent** class are defining an event hierarchy. All events are input events (InputEvent.ANY) and they change the activity. Some of them get fired when the user starts the change, some while the change is ongoing, and some when the change is finished.

- InputEvent.ANY
 - ACTIVITY_CHANGE
 - ACTIVITY_DELETED
 - ACTIVITY_CHANGE_STARTED // All event types that signal "start"
 - CHART_VALUE_CHANGE_STARTED
 - CHART_HIGH_VALUE_CHANGE_STARTED
 - CHART_LOW_VALUE_CHANGE_STARTED
 - DRAG_STARTED
 - END_TIME_CHANGE_STARTED
 - HORIZONTAL_DRAG_STARTED
 - PERCENTAGE_CHANGE_STARTED
 - START_TIME_CHANGE_STARTED
 - VERTICAL_DRAG_STARTED
 - ACTIVITY_CHANGE_ONGOING // All event types that signal "ongoing"
 - CHART_VALUE_CHANGE_ONGOING
 - CHART_HIGH_VALUE_CHANGE_ONGOING
 - CHART_LOW_VALUE_CHANGE_ONGOING
 - DRAG_ONGOING
 - END_TIME_CHANGE_ONGOING
 - HORIZONTAL_DRAG_ONGOING
 - PERCENTAGE_CHANGE_ONGOING
 - START_TIME_CHANGE_ONGOING
 - VERTICAL_DRAG_ONGOING
 - ACTIVITY_CHANGE_FINISHED // All event types that signal "finished"
 - CHART_VALUE_CHANGE_FINISHED
 - CHART_HIGH_VALUE_CHANGE_FINISHED
 - CHART_LOW_VALUE_CHANGE_FINISHED
 - DRAG_FINISHED
 - END_TIME_CHANGE_FINISHED
 - HORIZONTAL_DRAG_FINISHED
 - PERCENTAGE_CHANGE_FINISHED
 - START_TIME_CHANGE_FINISHED

- VERTICAL_DRAG_FINISHED

Activity Event Properties

Applications are obviously interested in the attributes of an activity. Not only the new values of these attributes (for example the new start time) but also the old values (start time before the change). The new values are already available on the activity as they are being set while the user performs the change. The old values are stored on the event object. The following table lists the methods on **ActivityEvent** to retrieve these values.

Method	Description	Event Types
<code>getOldTime()</code>	Returns the old start <u>or</u> end time of the activity.	END_TIME_CHANGE_ START_TIME_CHANGE_
<code>getOldTimeInterval()</code>	Returns the old start <u>and</u> end time of the activity.	DRAG_ HORIZONTAL_DRAG_ VERTICAL_DRAG_
<code>getOldRow()</code>	Returns the old row where the activity was located before.	DRAG_ VERTICAL_DRAG_
<code>getOldValue()</code>	Returns the old value of "percentage complete" or "chart value".	CHART_VALUE_CHANGE_ CHART_HIGH_VALUE_ CHART_LOW_VALUE_ PERCENTAGE_CHANGE_

Lasso Events

The user can use a lasso to select activities. Events are fired when this happens. To receive a lasso event simply register an event handler with the graphics view via one of the convenience methods.

Singe Lasso Event Handler

```
GraphicsBase<?> graphics = ganttChart.getGraphics();
graphics.setOnLassoFinished(evt ->
    System.out.println("The lasso was used"));
```

If you need to register more than one handler for a specific event type then use this approach:

Multiple Lasso Event Handlers

```
GraphicsBase<?> graphics = ganttChart.getGraphics();
graphics.addEventHandler(LassoEvent.SELECTION_FINISHED,
    evt -> System.out.println("Listener 1"));
graphics.addEventHandler(LassoEvent.SELECTION_FINISHED,
    evt -> System.out.println("Listener 2"));
```

The following table lists the event types and the convenience setter methods of the graphics view.

Event Type	Method	Description
------------	--------	-------------

ALL	setOnLassoSelection()	Any lasso operation (start, ongoing, finished).
SELECTION_STARTED	setOnLassoSelectionStarted()	The user has pressed the mouse button and started a drag. The lasso has become visible.
SELECTION_ONGOING	setOnLassoSelectionOngoing()	The user is changing the size of the lasso.
SELECTION_FINISHED	setOnLassoSelectionFinished()	The user has finished the lasso selection. The lasso is no longer visible.

Lasso Event Hierarchy

The event types defined in the **LassoEvent** class are defining an event hierarchy. All events are input events (InputEvent.ANY).

- InputEvent.ANY
 - LassoEvent.ALL
 - LassoEvent.SELECTION_STARTED
 - LassoEvent.SELECTION_ONGOING
 - LassoEvent.SELECTION_FINISHED

Lasso Info

The lasso automatically performs selections of activities but sometimes we might want to know more about the exact nature of this selection or we want to use the lasso for another use case (e.g. for creating new activities). For this reason instances of **LassoEvent** also provide an object of type **LassoInfo**, which carries many attributes that the application can use to react accordingly. The lasso information can be retrieved by calling **LassoEvent.getInfo()**. The following table lists the attributes of **LassoInfo**.

Method	Description
List<ActivityRef<?>> getActivities();	Returns all activities that were selected by the lasso.
Instant getStartTime(); Instant getEndTime();	Returns the start and end time of the lasso according to the location of the <u>left</u> and <u>right</u> edge of the lasso.
LocalTime getLocalStartTime(); LocalTime getLocalEndTime();	Returns the <u>local</u> start and end time. These values are only provided if the <u>upper</u> or <u>lower</u> edge of the lasso is located in an area that uses the AgendaLayout .
List<Row<?, ?, ?>> getRows();	Returns the rows that were touched by the lasso.

Links / Further Reading

- [Oracle JavaFX documentation](#)
- [Event handling examples](#)

3.4.4 Activity Editing

- [Introduction](#)
- [Edit Mode Callback](#)
 - [Edit Mode Callback Parameter](#)
 - [Edit Mode Callback Example](#)
- [Editing Callback](#)

- [Editing Callback Parameter](#)
- [Editing Callback Example](#)

Introduction

Two different callbacks on the [graphics view](#) are used to control the editing behaviour of activities. The first maps a mouse event / mouse location to an editing mode. The second callback is used to determine whether a given editing mode / operation can be applied to an activity at all. Most applications will only need to work with the second callback and keep the defaults for the edit mode locations (for example: right edge used to change end time, left edge used to change start time). The enum **GraphicsBase.EditMode** lists all available editing operations that can be performed on an activity.

Mode	Description
AGENDA_ASSIGNING	Assign an activity in AgendaLayout to another row.
AGENDA_DRAGGING	Drag an activity in AgendaLayout up and down or sideways within the same row.
AGENDA_END_TIME_CHANGE	Change the end time of an activity in AgendaLayout .
AGENDA_START_TIME_CHANGE	Change the start time of an activity in AgendaLayout .
CHART_VALUE_CHANGE	Change the value of a ChartActivity .
CHART_VALUE_HIGH_CHANGE	Change the "high" value of a HighLowActivity .
CHART_VALUE_LOW_CHANGE	Change the "low" value of a HighLowActivity .
DRAGGING	Perform a drag and drop in all directions on an activity.
DRAGGING_HORIZONTAL	Move an activity horizontally within its own row (change start and end time).
DRAGGING_VERTICAL	Perform a drag and drop on an activity in vertical direction only.
END_TIME_CHANGE	Change the end time of an activity.
NONE	Do nothing.
PERCENTAGE_COMPLETE_CHANGE	Change the "percentage complete" value of a CompletableActivity .
START_TIME_CHANGE	Change the start time of an activity.

Edit Mode Callback

The edit mode callback is used to determine the edit mode at the given mouse location. Instances of this callback can be registered via the **GraphicsBase.setEditModeCallback()** method which maps the callback to a combination of activity type and [layout](#) type.

Edit Mode Callback Registration

```
public final void setEditModeCallback(
    Class<? extends MutableActivity> activityType,
    Class<? extends Layout> layoutType,
    Callback<EditModeCallbackParameter, EditMode> callback);
```

Edit Mode Callback Parameter

The parameter object passed to the edit mode callback is of type **EditModeCallbackParameter** and contains the following information:

Field	Description
activityBounds	The bounds of the activity over which the mouse cursor is hovering. The x and y coordinates are relative to the coordinate space of the row where the activity is displayed.
mouseEvent	The mouse event that triggered the lookup of the edit mode (normally a MOUSE_OVER).

Edit Mode Callback Example

The following is a simple example of an editing mode callback.

Edit Mode Callback Example

```
public class MyEditModeCallback implements
    Callback<EditModeCallbackParameter, EditMode> {

    public EditMode call(EditModeCallbackParameter param) {
        MouseEvent event = param.getMouseEvent();
        ActivityBounds bounds = param.getActivityBounds();

        /*
         * If the mouse cursor is touching the left edge of the activity
         * then begin a change of the start time of the activity.
         */
        if (event.getX() - bounds.getMinX() < 5) {
            return EditMode.CHANGE_START_TIME;
        }

        return EditMode.NONE;
    }
}
```

This callback can now be registered like this:

Edit Mode Callback Registration

```
GraphicsBase<?> graphics = ganttChart.getGraphics();
graphics.setEditModeCallback(
    ActivityBase.class,
    GanttLayout.class,
    new MyEditModeCallback());
```

We could have used a lambda expression for the entire callback instance but decided against it in favor of verbosity.

Editing Callback

The editing callback is used to determine if a specific edit mode is currently usable for a given activity. Instances of this callback can be registered via the **GraphicsBase.setActivityEditingCallback()** method which maps the callback to an activity type.

Edit Mode Callback Registration

```
public final void setActivityEditingCallback(
    Class<? extends MutableActivity> activityType,
    Callback<EditingCallbackParameter, Boolean> callback);
```

Editing Callback Parameter

The parameter object passed to the editing callback is of type **EditingCallbackParameter** and contains the following information:

Field	Description
activityRef	The reference to the activity for which to perform the check.
editMode	The edit mode that needs a check.

Editing Callback Example

The following is a simple example of an editing mode callback.

Edit Mode Callback Example

```
public class MyEditingCallback implements
    Callback<EditingCallbackParameter, Boolean> {

    public Boolean call(EditingCallbackParameter param) {
        ActivityRef ref = param.getActivityRef();
        Activity activity = ref.getActivity();

        /*
         * Only allow editing for activities that that have not
         * started, yet.
         */
        if (activity.getStartTime().isAfter(Instant.now())) {

            /*
             * Only allow changes to the start and end time
             * of the activity.
             */
            switch (param.getEditMode()) {
                case CHANGE_START_TIME:
                case CHANGE_END_TIME:
                    return true;
                default:
                    return false;
            }
        }

        return false;
    }
}
```

This callback can now be registered like this:

Editing Callback Registration

```
GraphicsBase<?> graphics = ganttChart.getGraphics();
graphics.setActivityEditingCallback(
    ActivityBase.class,
    new MyEditingCallback());
```

3.4.5 Row Editing

- [Introduction](#)
- [Row Editor Factory](#)
- [Row Controls Factory](#)
 - [Example 1](#)
 - [Example 2](#)

Introduction

The [graphics view](#) not only supports editing [activities](#) but also [rows](#). If a row gets edited the entire row will be flipped around and additional controls will become visible on the "back" of the row. If the back of the row requires more space (height) than the front of the row then the height will be automatically adjusted. The following table lists the methods that are related to row editing:

Method	Description
<code>void startRowEditing(R row);</code>	Initiates the row editing sequence on the given row. The back of the row will become visible and expose controls to change row settings.
<code>void stopRowEditing();</code> <code>void stopRowEditing(R row);</code>	Stops the row editing of all rows or just the given row. The front of the row will become visible again.
<code>ObjectProperty<RowEditingMode></code> <code>rowEditingModeProperty();</code> <code>void</code> <code>setRowEditingMode(RowEditingMode);</code> <code>RowEditingMode getRowEditingMode();</code>	Stores, sets, and retrieves the row edit mode. The enum GraphicsBase.RowEditingMode is used to determine whether the user will be able to edit rows at all, one row at a time, or multiple rows at the same time.
<code>ObservableList<R> getRowsEditing();</code>	An observable list of all rows that are currently being edited (their back is shown).
<code>BooleanProperty animateRowEditor();</code> <code>void setAnimateRowEditor(boolean);</code> <code>boolean isAnimateRowEditor();</code>	Stores, sets, and retrieves a flag that is used to signal whether the exposure of the row back will be immediate or animated.

Row Editor Factory

The row editor factory is used to create the controls for a given row at the moment when the user requests that the row will be edited. The factory is a callback method that gets called with a **GraphicsBase.RowEditorParameter** object. This parameter object stores some fields that can be useful for creating the editor controls and also a method for stopping the row editing.

Method	Description
<code>GraphicsBase</code> <code>getGraphics();</code>	Returns a reference to the graphics view where the editing will occur.
<code>R getRow();</code>	Returns the row for which the row editor will be created.
<code>void</code> <code>stopEditing();</code>	A convenience method for the row editor controls that can be used to signal that the user is done editing the row. This method will usually get invoked by some kind of close button in the editor UI:

A row editor factory might look like this:

Row Editor Example

```
public class MyRowEditorFactory implements
    Callback<RowEditorParameter<R>, Node> {

    public Node call(RowEditorParameter<R> param) {
        VBox box = new VBox();

        /*
         * Bind the text property of the textfield to the name
         * property of the row. This allows us to change the name
         * of the row.
         */
        TextField nameField = new TextField();
        Bindings.bindBidirectional(param.getRow().nameProperty(),
            nameField.textProperty());

        /*
         * A close button to invoke the stopEditing() method
         * on the parameter object.
         */
        Button closeButton = new Button("Close");
        closeButton.setOnAction(evt -> param.stopEditing());
        box.getChildren().addAll(nameField, closeButton);

        /*
         * Return the vbox node.
         */
        return box;
    }
}
```

Row editors can be registered like this:

Row Editor Registration

```
GraphicsBase<?> graphics = ganttChart.getGraphics();
graphics.setRowEditorFactory(new MyRowEditorFactory());
```

Row Controls Factory

To trigger row editing the user interface needs to provide some kind of controls. This can be done in many ways, for example by the help of a context menu on a row. Another way is to use the built-in support for so-called "row controls". These controls appear / disappear every time the mouse cursor enters / exists a row. They are created by a callback implementation. This callback receives a parameter object of type **GraphicsBase.RowControlsParameter**. The following table lists the fields of this type.

Field	Description
graphics	The graphics view for which the callback gets invoked.
row	The row for which controls will be created.

Example 1

A possible implementation of this callback can look like this:

Row Controls Factory

```
public class MyRowControlsFactory extends StackPane
    implements Callback<RowControlsParameter, Node> {

    private Button button;

    public MyRowControlsFactory() {

        /*
         * Important: let mouse events pass through.
         */
        setMouseTransparent(true);
        button = new Button("Press Me");
        getChildren().add(button);
    }

    /*
     * Reuse the button. Simply exchange the action that will
     * happen when the user presses on it.
     */
    public Node call(RowControlsParameter param) {
        button.setOnAction(evt ->
            System.out.println("Pressed on row " +
                param.getRow().getName()));
        return this;
    }
}
```

Please take notice that this factory is a Node object and returns itself every time the call() method gets invoked. Only the action of the button gets replaced with each invocation. This makes perfect sense as row controls are always only shown for one row at a time (as opposed to row editors where several of them can be in use at the same time).

The callback can be registered like this:

Row Controls Factory Registration

```
GraphicsBase<?> graphics = ganttChart.getGraphics();
graphics.setRowControlsFactory(new MyRowControlsFactory());
```

Example 2

The following is the code of the RowControls class in the FlexGanttFX "Extras" project. It adds a simple "Edit" button to the row. When clicked it will show the row editor controls on the back on the row.

RowControls.java

```
/**
 * Copyright (C) 2014 Dirk Lemmermann Software & Consulting (dlsc.com)
 *
 * This file is part of FlexGanttFX.
 */
package com.flexganttfx.extras;

import javafx.geometry.Pos;
import javafx.scene.Node;
import javafx.scene.control.Button;
import javafx.scene.layout.HBox;
import javafx.util.Callback;
import com.flexganttfx.model.Row;
import com.flexganttfx.view.graphics.GraphicsBase.RowControlsParameter;

public class RowControls<R extends Row<?, ?, ?>> extends HBox implements
    Callback<RowControlsParameter<R>, Node> {

    private Button editButton;

    public RowControls() {
        setPickOnBounds(false);
        setMinSize(0, 0);
        setAlignment(Pos.TOP_RIGHT);
        setFillHeight(true);
        editButton = new Button("EDIT");
        editButton.getStyleClass().add("row-controls-button");
        getChildren().add(editButton);
    }

    @Override
    public Node call(RowControlsParameter<R> param) {
        editButton.setOnAction(evt -> param.getGraphics().startRowEditing(
            param.getRow()));
        return this;
    }
}
```

The matching CSS for the button is defined like this:

RowControls Button CSS

```
/*
 * Row controls button are shown when the mouse hovers over a row that can
 * be
 * edited (flipped around).
 */
.row-controls-button {
    -fx-padding: 5 9 7 7;
    -fx-background-insets: 0 4 2 2;
    -fx-background-color: rgba(0,0,0,.5);
    -fx-background-radius: 0;
    -fx-text-fill: white;
    -fx-font-size: 8;
    -fx-font-weight: bold;
}

.row-controls-button:hover,
.row-controls-button:focus {
    -fx-padding: 5 9 7 7;
    -fx-background-insets: 0 4 2 2;
    -fx-background-color: rgba(0,0,0,.6);
    -fx-background-radius: 0;
    -fx-text-fill: white;
    -fx-font-size: 8;
    -fx-font-weight: bold;
}

.row-controls-button:pressed,
.row-controls-button:selected {
    -fx-background-color: rgba(0,0,0,.7);
    -fx-background-radius: 0;
}
```

3.4.6 Activity Rendering

- [Introduction](#)
- [Drawing](#)
- [Default Renderers](#)
- [Activity Bounds](#)
- [Properties](#)

Introduction

The graphics view uses the [canvas API](#) of JavaFX. This is due to the complex nature of a Gantt chart and due to the large data volumes often observed inside of them. Directly rendering large quantities of activities into a bitmap is much faster than constantly updating the scene graph and reapplying CSS styling. FlexGanttFX implements a pluggable renderer architecture where renderer instances can be mapped to activity types, very similar to the way Swing was doing it.

The following code is an example of how to register a custom renderer for a given "Flight" activity type. Please note that the graphics view is capable of displaying activities in different layouts, hence the layout type must also be passed to the method.

Renderer Registration

```
GraphicsBase<?> graphics = ganttChart.getGraphics();
graphics.setActivityRenderer(
    Flight.class,
    GanttLayout.class,
    new FlightRenderer(graphics));
```

We usually also pass the graphics view to the renderer at construction time. This is needed as renderers will invoke a redraw on the graphics when any of its properties changes. This is very different to the Swing approach. This also implies that renderer instances should only be used for a single graphics view, the one that was passed to their constructor.

The following methods on **GraphicsBase** are used for working with renderers:

Method	Description
<code>void setActivityRenderer(...);</code>	Registers a new renderer for the given activity and layout type.
<code>ActivityRenderer getActivityRenderer(...);</code>	Returns a renderer for the given activity and layout type.

Drawing

Activity renderers have a single entry point for drawing, a method called **draw()**. This method is final and can not be overridden. Once invoked it will call various protected methods to perform the actual drawing. The call hierarchy looks like this:

- public final **draw()** calls ...
 - protected ActivityBounds **drawActivity()**
 - protected void **drawBackground()**
 - protected void **drawBorder()**

Subclasses are free to override any of the three protected methods to customize the activity appearance.

All drawXXX() methods have the same arguments:

Arguments

```
ActivityRef<A> activityRef, // the activity to draw
Position position,        // agenda layout only (first, middle, last, only)
GraphicsContext gc,       // the graphics context into which to draw
double x,                 // the location of the start time of the activity
double y,                 // the y coordinate (0 when drawn on row or line location)
double w,                 // end time location minus start time location
double h,                 // row or line height
boolean selected,         // is activity currently selected?
boolean hover,            // is mouse cursor currently hovering over it?
boolean highlighted,      // is activity currently blinking?
boolean pressed)          // is user currently pressing on it?
```






Default Renderers

The following table lists the various activity renderers that are provided by default.

Renderer Class	Description

ActivityRenderer	The most basic renderer for activities. Draws a filled rectangle at the location of the activity. All default renderers are subclasses of this type.
ActivityBarRenderer	Draws a bar instead of filling the entire area. The height of the bar can be specified. <div>Also supports text in several locations inside and outside the bar.</div>
ChartActivityRenderer	Draws a ChartActivity vertically depending on its chart value.
CompletableActivityRenderer	Subclass of the bar renderer. Draws a CompletableActivity as a bar with a section of its background filled with another color. The size of the section depends on the percentage complete value of the activity.

These default renderers are attached to this page and can be downloaded here:

File	Modified 
 ChartActivityRenderer.java Base renderer for chart activities.	Oct 07, 2014 by Dirk Lemmermann [Administrator]
 ActivityRenderer.java The base class for all activity renderers.	Oct 07, 2014 by Dirk Lemmermann [Administrator]
 ActivityBarRenderer.java The base class for all activity renderers that want to display a thin bar.	Oct 07, 2014 by Dirk Lemmermann [Administrator]
 CompletableActivityRenderer.java Base renderer for completable activities.	Oct 07, 2014 by Dirk Lemmermann [Administrator]

 [Download All](#)

Activity Bounds

Every activity renderer is responsible for returning an instance of **ActivityBounds** after drawing the activity. These bounds are an essential piece for the framework and many operations will only work properly if these bounds are valid. They are being used for editing activities, for hitpoint detection, for laying out links, for context menus, and so on. The following table lists the attributes of the **ActivityBounds** class.

Attribute	Description
activity	The activity for which these are the bounds.
activityRef	An activity referene pointing to the activity.
layer	The layer on which the activity was drawn.
layout	The layout that was used when the activity was drawn.
lineIndex	The index of the line on which the activity is located (-1 if activity is on the row, not a line).
position	The position of the bounds when the activity was drawn in agenda layout (first, middle, layout). This is needed because the same activity might be rendered in several pieces across several days.
row	The row where the activity was drawn.

Please ignore the attributes **overlapColumn**, **overlapCount**, and the list **overlapBounds**. These are all used internally for [agenda layout](#) related operations.

Properties

All renderers define several properties that can be used to customize their apperance. Many of these properties are depenent on the "pseudo state" of the activity: hover, pressed, selected, highlighted. To make it easier to lookup the right color at the right time several convenience

methods are available:

Renderer	Method	Description
Renderer	<pre>protected Paint getFill(boolean selected, boolean hover, boolean highlighted, boolean pressed);</pre>	Returns the color to use for the activity background depending on pseudo states passed.
ActivityRenderer	<pre>protected Paint getStroke(boolean selected, boolean hover, boolean highlighted, boolean pressed);</pre>	Returns the color to use for the activity border depending on pseudo states passed.
ActivityBarRenderer	<pre>protected Paint getTextFill(boolean selected, boolean hover, boolean highlighted, boolean pressed);</pre>	Returns the color to use for text depending on pseudo states passed.

3.4.7 Row Rendering

- [Introduction](#)
- [Row Renderer](#)

Introduction

The system layer [RowLayer](#) supports pluggable renderers in order to customize the background of each row depending on the row type. In the tutorial we have seen that we can have Aircraft rows and Crew rows. For clarity these two rows could have different background colors. This is something that could be done with a row renderer.

Row Renderer

All row renderers have to subclass **RowRenderer**. This class defines a final public method called **draw()** that gets called by the framework. It then calls the protected method **drawRow()** which subclasses can override. A possible implementation might look like this:

Custom Row Renderer

```
public class AircraftRowRenderer extends RowRenderer<Aircraft> {

    public AircraftRowRenderer(GraphicsBase<?> graphics) {
        super(graphics, "Aircraft Row Renderer");
    }

    protected void drawRow(Aircraft row,
                           GraphicsContext gc,
                           double w,
                           double h,
                           boolean selected,
                           boolean hover,
                           boolean highlighted,
                           boolean pressed) {
        gc.setFill(Color.ORANGE);
        gc.fillRect(0, 0, w, h);
    }
}
```

This renderer can now be registered with the RowLayer like this:

Row Renderer Registration

```
GraphicsBase<?> graphics = ganttChart.getGraphics();
graphics.getSystemLayer(RowLayer.class).setRowRenderer(
    Aircraft.class, new AircraftRowRenderer());
```

The RowRenderer base implementation is attached to this page and can be downloaded here:

File	Modified ▲
  RowRenderer.java Row renderer base implementation.	Oct 07, 2014 by Dirk Lemmermann [Administrator]

3.4.8 Context Menu

- [Introduction](#)
- [Code Example](#)

Introduction

There are two ways to register a context menu with the graphics view. The standard way by calling **GraphicsBase.setContextMenu(ContextMenu)** or by registering a context menu callback by calling **GraphicsBase.setContextMenuCallback()**. The advantage of the second option is that a parameter object of type **ContextMenuParameter** will be passed to the callback method. This parameter object contains the most relevant parameters that most context menus will require in order to let the user perform some kind of action on the graphics view.

Please note that a context menu callback will have precedence over a standard context menu.

Code Example

The following snippet shows an example of a context menu callback implementation. Here we simply add a menu item for each activity that was found at the mouse location where the context menu was requested by the user.

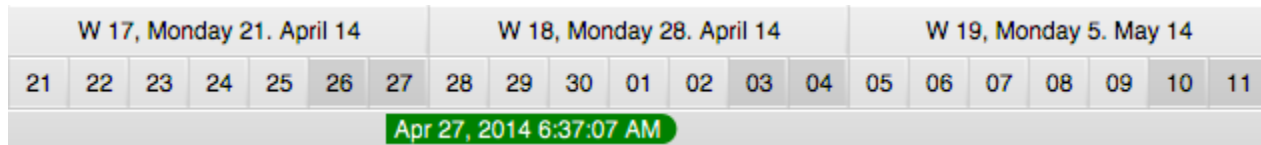
Context Menu Callback

```
GraphicsBase<?> graphics = ganttChart.getGraphics();
graphics.setContextMenuCallback(param -> {
    ContextMenu menu = new ContextMenu();
    for (ActivityRef<?> ref : param.getActivities()) {
        Activity activity = ref.getActivity();
        MenuItem item = new MenuItem("Move " + activity.getName());
        item.setOnAction(evt -> moveActivity(activity));
        menu.getItems().add(item);
    }
    return menu;
});
```

3.6 Timeline

- [Navigation](#)
- [Zooming](#)
- [Scrolling](#)
- [TimeTracker](#)
- [Visible Time Interval](#)

The timeline control is a container for the [Dateline](#) and the [Eventline](#). It is displayed above the [Graphics](#) control and provides several methods for scrolling and zooming, both of which can be done with or without animation. The timeline also keeps track of the current time (see [TimeTracker](#)).



Navigation

The timeline is used to navigate through time. It provides methods to jump to the current time or a given time. It can be requested to show a specific time unit ("show days"), or a time range.

Method	Description
<code>showNow()</code> <code>showNow(boolean center)</code>	Changes the start time of the timeline model in such a way that the current time (as already stored in the TimelineModel) will be displayed either on the left edge of the dateline or right in the middle.
<code>showTime(Instant time)</code> <code>showTime(Instant time, boolean center)</code>	Changes the start time of the timeline model in such a way that the given time will be displayed either on the left edge of the dateline or right in the middle.

<code>showRange(Instant start, Instant end)</code> <code>showRange(Instant start, Duration duration)</code> <code>showRange(TimeInterval range)</code>	Changes the start time and the "millis per pixel" value of the timeline model in such a way that the given time range will become fully visible in the dateline .
<code>showTemporalUnit(TemporalUnit unit, double width)</code>	Changes the start time and the "millis per pixel" value of the timeline model in such a way that the given time unit will be used in the dateline . Each cell in the dateline will be as wide as the given width.

It should be noted that the timeline in cooperation with the dateline can only make a best-effort attempt at fulfilling these requests as they depend on the availability of dateline resolutions in the [dateline model](#).

The methods above can be executed with or without animation. This animation can be controlled via the help of two properties: **moveAnimated** and **moveDuration**. The appropriate getter and setter methods for these properties are available on **Timeline**.

Zooming

The timeline is responsible for managing anything related to zooming. The user can press the + / - keys to increase the zoom level by a specific zoom factor or he can select a time interval via a "lasso" by dragging the mouse and holding down the SHIFT key. The result will be a selected time interval which is stored in the read-only property **selectedTimeInterval**. The timeline listens for changes to this property and will automatically try to display the selected time interval across the entire available width, ultimately causing a zoom in operation.

Method	Description
<code>zoomIn()</code> <code>zoomOut()</code>	Makes the timeline modify the timeline model in such a way that the resulting visible time range will be the current time range multiplied by / divided by the current zoom factor.
<code>zoom(double factor, boolean zoomIn, Instant frozenTime)</code>	Performs a zoom operation with the given zoom factor (either zoom in or out). The timeline will try to keep the given "frozen" time at its current location. This kind of behaviour is very useful for a pinch-based zoom, where the UI zooms "into" a specific time.
<code>setZoomLassoEnabled(boolean);</code> <code>boolean isZoomLassoEnabled();</code>	Controls the availability of the zoom lasso.



Just like the moving operations the zoom operations can also be executed in an animated or non-animated way. To control this the two properties **zoomAnimated** and **zoomDuration** are available.

Another property is used to fine-tune the zooming behaviour as some applications prefer to either keep the start time, the end time, or the center time while zooming. For this the application can set the **zoomMode** property. Possible values of this enum are **KEEP_START_TIME**, **KEEP_END_TIME**, or **CENTER**.

Scrolling

The timeline supports scrolling to the left and right in two different speeds.

Method	Description
<code>scrollLeft()</code> <code>scrollLeftFast()</code>	Changes the start time property of the timeline model in such a way that the dateline will end up starting with an earlier time.
<code>scrollRight()</code> <code>scrollRightFast()</code>	Changes the start time property of the timeline model in such a way that the dateline will end up starting with a later time.

These methods can be invoked by the user via the  and  keys. Scrolling will be fast if the user presses  at the same time.

TimeTracker

The timeline control is responsible for tracking time. This means that it updates the property now of the underlying timeline model. The timeline implements methods for starting and stopping time tracking, however the actual update of now will be delegated to a time tracker class.

Method	Description
<code>startTimeTracking()</code> <code>stopTimeTracking()</code>	Starts and stops time tracking. These methods invoke the equivalent methods on the TimeTracker class.
<code>timeTrackerProperty()</code> <code>setTimeTracker(TimeTracker tracker)</code> <code>TimeTracker</code> <code>getTimeTracker()</code>	The time tracker property and its getter and setter methods. The default time tracker (uses the system time) can be replaced with a custom one.

Visible Time Interval

Two read-only properties are keeping track of the earliest and latest times shown by the timeline . They are called **visibleStartTime** and **visibleEndTime** and the methods **getVisibleStartTime()**, **getVisibleEndTime()**, and **getVisibleDuration()** can be used to work with them.

3.5.1 Timeline Model

- [Introduction](#)
- [Start Time & Millis Per Pixel](#)
- [Now Time / Now Location](#)
- [Time & Coordinate Calculations](#)
- [The Horizon](#)
- [Highest & Lowest Temporal Unit](#)

Introduction

The timeline uses a model of type **TimelineModel**. This model provides the most important parameters for the timeline and the dateline in order for them to work properly. The timeline model can be typed for different temporal units. *FlexGanttFX* ships with a **ChronoUnitTimelineModel** and a **SimpleUnitTimelineModel**.

Start Time & Millis Per Pixel

The two most important properties of the **TimelineModel** are the **startTime** and the **millisPerPixel** (MPP) properties. The start time determines the first visible time in the Gantt chart while the current width of the timeline in combination with the MPP value determine the last visible time and hence the visible time range. Increasing the MPP value will cause the timeline to show a larger time range while reducing this value will result in a shorter time range. The methods found in the Timeline class for showing a time, scrolling to a time, zooming into a range are all playing with these two variables to achieve their purpose. The following table lists the methods related to these properties:

Method	Description
<code>ObjectProperty<Instant></code> <code>startTimeProperty();</code> <code>setStartTime(Instant time);</code> <code>Instant getStartTime();</code>	Stores, sets, and retrieves the current <u>start</u> time, the first visible time in the Gantt chart. <div>The earliest possible start time can be restricted via the horizonStartTime property.</div>

<pre> DoubleProperty millisPerPixel(); setMillisPerPixel(double mpp); double getMillisPerPixel(); </pre>	Stores, sets, and retrieves the millis per pixel value (mpp). <div> The default value of mpp is $24 * 60 * 60 * 1000 / 30$. This results in days having the width of 30 pixels. </div>
--	---

Now Time / Now Location

Gantt charts often have a requirement to mark the "current" time. This time can either be the system time (**`java.time.Instant.now()`**) or an arbitrary value controlled by the application. The latter is often the case in software that runs some kind of simulation and the Gantt chart is used to track the simulation time. To support these use cases the timeline model defines a property called **now**.

The value of now is usually updated by a [time tracker](#) that can be controlled via the [timeline](#).

Method	Description
<pre> ObjectProperty<Instant> nowProperty(); void setNow(Instant now); Instant getNow();now </pre>	Stores, sets, and retrieves the current time.
<pre> ReadOnlyDoubleProperty nowLocation(); double getNowLocation(); </pre>	Stores and retrieves the location of the current time. The now location is calculated by the model based on the start time, and the millis per pixel value. <div> This property is a read-only property as the now <u>location</u> is always dependent on the value of the now <u>time</u>. The location can only be changed by changing now itself. </div>

Time & Coordinate Calculations

The primary purpose of the timeline model is to convert time into a location and vice versa. For this the model provides several methods:

Method	Description
<pre>double calculateLocationForTime(Instant);</pre>	Returns the x coordinate for the given time.
<pre>Instant calculateTimeForLocation(double);</pre>	Returns the time for the given x coordinate.

The Horizon

Scheduling applications often work with a horizon, defined by an earliest and latest time. These times might be based on the loaded dataset (min / max calculation of the start and end times of the activities) or the planning horizon (Q1, Q2, Q3, Q4). Setting the values of **horizonStartTime** and **horizonEndTime** ensures that the user will not be able to scroll to a time outside the horizon.

Highest & Lowest Temporal Unit

Not all applications require all available units of a temporal unit. **`java.time.temporal.ChronoUnit`** for example defines units for nanos until millennia. The **highestTemporalUnit** and the **lowestTemporalUnit** property enable the application to restrict the unit range to something more sensible, e.g. hours to months.

3.5.2 Time Tracker

- [Introduction](#)
- [Example](#)

Introduction

A time tracker is used to update the property **now** of the [TimelineModel](#). In most cases the time "now" will be equivalent to the system time but in simulation software this might not be the case. The time tracker is used by the [timeline](#) and can be replaced by calling **Timeline.setTimeTracker(TimeTracker)**. However, a default tracker is already installed and can be started by calling **Timeline.startTimeTracking()**.

Example

The following is the entire code of the default time tracker class.

```
TimeTracker

/**
 * Copyright (C) 2014 - 2016 Dirk Lemmermann Software & Consulting
 (dlsc.com)
 *
 * This file is part of FlexGanttFX.
 */
package com.flexganttfx.view.timeline;

import java.time.Instant;
import java.util.logging.Level;

import com.flexganttfx.core.LoggingDomain;
import com.flexganttfx.model.timeline.TimelineModel;

import javafx.application.Platform;
import javafx.beans.property.ReadOnlyObjectProperty;
import javafx.beans.property.ReadOnlyObjectWrapper;

/**
 * A time tracker can be used to update the property
 * {@link TimelineModel#nowProperty()}. In most cases the time "now" will
 be
 * equivalent to the system time but in simulations this might not be the
 case.
 * The time tracker can be used in combination with the {@link
 TimelineModel} by
 * binding the {@link TimelineModel#nowProperty()} to the
 * {@link TimeTracker#timeProperty()}.
 *
 * @since 1.0
 */
public class TimeTracker extends Thread {

    private boolean running = true;

    private long delay = 1000;

    private boolean stopped;

    /**
     * Constructs a new tracker.
     */
}
```



```

    *
    * @since 1.0
    */
    public TimeTracker() {
        setName("Time Tracker");
        setDaemon(true);
    }

    private final ReadOnlyObjectWrapper<Instant> time = new
ReadOnlyObjectWrapper<> (
        this, "time", Instant.now());

    public final ReadOnlyObjectProperty<Instant> timeProperty() {
        return time.getReadOnlyProperty();
    }

    public final Instant getTime() {
        return time.get();
    }

    /**
     * Returns the delay in milliseconds between updates of
     * {@link TimelineModel#nowProperty()}. The default is 1000 millis.
     *
     * @return the default delay between update calls
     * @since 1.0
     */
    public final long getDelay() {
        return delay;
    }

    /**
     * Sets the delay between updates of {@link
TimelineModel#nowProperty()}.
     * The default is 1000 millis.
     *
     * @param millis
     *         the new delay
     * @throws IllegalArgumentException
     *         if the delay is zero or smaller
     * @since 1.0
     */
    public final void setDelay(long millis) {
        if (millis <= 0) {
            throw new IllegalArgumentException(
                "$NON-NLS-1$" + "delay must be larger than zero but was" + millis);

            this.delay = millis;
        }
    }

    /**

```

```

    * Starts the tracking of the time.
    *
    * @since 1.0
    */
    public final void startTracking() {
        if (stopped) {
            throw new IllegalStateException(
                "Time tracker has already been stopped and can not be
started again.");
        } else {
            running = true;
            start();
        }
    }

    @Override
    public void run() {
        while (running) {
            Platform.runLater(() -> time.set(getNow()));
            try {
                Thread.sleep(delay);
            } catch (InterruptedException e) {
                LoggingDomain.CONFIG.log(Level.WARNING,
                    "problem in update thread", e); //$NON-NLS-1$
            }
        }
    }

    /**
     * Stops the tracking of the time.
     *
     * @since 1.0
     */
    public final void stopTracking() {
        stopped = true;
        running = false;
    }

    /**
     * Override to return the instant that will be set as "now" on the
timeline
     * model. The default implementation uses {@link Instant#now()}.
     *
     * @see TimelineModel#setNow(Instant)
     *
     * @return the "now" instant
     */
    protected Instant getNow() {

```

```

        return Instant.now();
    }
}

```

3.7 Dateline

- [Introduction](#)
- [Scale Resolutions](#)
- [Primary Temporal Unit](#)
- [Timezone](#)
- [Selection Model](#)
- [Hover Time Interval](#)
- [Events](#)
- [Cell Factory](#)

Introduction

The dateline is a control that displays the actual dates (Mo, Tu, We, ...) in cells that are placed on one or more rows. The dateline is timezone aware, keeps track of currently selected time intervals and the current hover time interval. It also fires events whenever the visible time range changes (e.g. after scrolling left or right).

Scale Resolutions

The dateline can display one to five rows. Each row is called a "dateline scale" and each one of these scales displays a "resolution". A resolution is comprised of a temporal unit (e.g. day, week, month), a pattern for formatting, and a quantity. The quantity is needed to specify resolutions like "5 minutes", "15 minutes", and so on. The entire list of resolutions that are currently shown by the dateline can be retrieved by calling **getScaleResolutions()**.

One example for a use of this method is given by the system layer **GridLinesLayer**. It calls this method in order to use the resolutions to calculate the locations of the vertical grid lines. For this the **Resolution** class offers the methods **truncate()** to go to the beginning of a unit (e.g. the beginning of a day) and **increment()** to go to the next unit (e.g. the next day). For more information on **Resolution** please go to the [dateline model](#) documentation.

Primary Temporal Unit

A dateline with three scales could for example display the resolutions "month", "week", and "day". The smallest resolution "day" gets displayed at the bottom of the dateline. The temporal unit **ChronoUnit.DAYS** that is used by this resolution is also called "primary temporal unit". The current value of this unit is stored in the read-only property **primaryTemporalUnit**. The value of this property is used when querying activities from [activity repositories](#). This way the repository can decide how fine-grained the result of its invocation will be or if certain activities will not be shown at all.

One example for a good use of the primary temporal unit is the **WeekendCalendar** class. It implements **Calendar**, which is an extension of **ActivityRepository**. The purpose of the **WeekendCalendar** is to return the weekend days (Saturday, Sunday) for a given time interval. When it gets invoked it will not return anything if the primary temporal unit is too large or too small. It makes no sense to return weekend information if the user is currently looking at minutes or decades.

Timezone

The dateline needs to know for which timezone it is displaying the dates (e.g. EST or GMT). Hence it features the property **zoneIdProperty()**. It is writable and can be set via **setZoneId()**. The value of this property can be made visible in the control by calling **setZoneId(true)**.

Selection Model

The dateline control allows the user to perform single or multiple selections of time intervals by clicking the primary mouse button while pressing the shortcut modifier key (CTRL on Windows / Linux, Option on Mac). Whether single or multiple selection is supported depends on the value of **selectionModeProperty()**.

Only those intervals can be selected that are currently visible in any one of the rows / scales. So if the dateline is currently showing weeks and days then the user can only select an entire weeks or entire days. This list of selected intervals can be retrieved by calling **getSelectedTimeIntervals()**.

Hover Time Interval

When the mouse cursor hovers over the dateline it also implies that it is hovering over a time interval. Depending on the resolution shown in the dateline row / scale at the given mouse location the interval might be an entire week or a single day. Whatever it is, the interval will be stored in the read-only property `hoverTimeIntervalProperty()`.

Events

Applications can listen to scrolling events fired by the dateline when they need to react to any changes in the currently visible time range. This is done by passing an event listener to the `setOnVisibleRangeChanged()` method or by calling `addEventListener(DatelineScrollingEvent.ANY, myListener)`.

Cell Factory

The dateline control is capable of displaying different types of temporal units. **ChronoUnit** (Mon, Tue, Wed, ...) and **SimpleUnit** (1, 2, 3, 4, ...) are supported by default. Each unit type has its own visual representation. To accommodate for this the dateline control delegates the creation of dateline cells to a pluggable factory that was previously mapped to a specific temporal unit type.

Cell Factories

```
setCellFactory(SimpleUnit.class,  
    unit -> new SimpleUnitDatelineCell());  
setCellFactory(ChronoUnit.class,  
    unit -> new ChronoUnitDatelineCell());
```

If a new temporal unit type needs to be displayed then a new factory needs to be registered in the same way.

3.6.1 Dateline Model

- [Introduction](#)
- [Chrono Unit Dateline Model](#)
- [Simple Unit Dateline Model](#)
- [Timezones](#)

Introduction

The dateline model provides the `dateline` control with various pieces of information so that it can layout itself correctly.

- **Resolutions** - a resolution defines which temporal unit to show (e.g. hours) and how to format it. It also contains the information whether it can be shown in a top, bottom, or middle scale. Each model usually defines a long list of such resolutions. The more resolutions are defined the more flexible the dateline control becomes when it comes to zooming in and out.
- **Time Zones** - The dateline control allows the user to switch between different time zones. The model defines which zones are available.
- **Scale Count** - The dateline control is composed of a set of dateline scales (top, bottom, several middle scales). The model can be used to define the currently visible, the minimum and the maximum number of scales that the user can choose to see.
- **Temporal Units** - The dateline control calls back onto the model to lookup the "next" temporal unit after it has either failed or succeeded to create a scale for the current unit.

The dateline model is a typed model. FlexGanttFX ships with two specializations: **ChronoUnitDatelineModel** and **SimpleUnitDatelineModel**.

Chrono Unit Dateline Model

The **ChronoUnitDatelineModel** class is a specialization for the **ChronoUnit** temporal that is part of JDK 8. It requires scale resolutions of type **ChronoUnitResolution**. The following listing is the implementation of this model and illustrates how to define and add resolutions and also how the resolution is used to go to from one temporal unit to the next.

ChronoUnitDatelineModel

```
/**  
 * Copyright (C) 2014 Dirk Lemmermann Software & Consulting (dlsc.com)
```

```

*
* This file is part of FlexGanttFX.
*/
package com.flexganttfx.model.dateline;

import static com.flexganttfx.model.dateline.Resolution.Position.BOTTOM;
import static com.flexganttfx.model.dateline.Resolution.Position.MIDDLE;
import static com.flexganttfx.model.dateline.Resolution.Position.ONLY;
import static com.flexganttfx.model.dateline.Resolution.Position.TOP;
import static java.time.temporal.ChronoUnit.CENTURIES;
import static java.time.temporal.ChronoUnit.DAYS;
import static java.time.temporal.ChronoUnit.DECADES;
import static java.time.temporal.ChronoUnit.HOURS;
import static java.time.temporal.ChronoUnit.MICROS;
import static java.time.temporal.ChronoUnit.MILLENNIA;
import static java.time.temporal.ChronoUnit.MILLIS;
import static java.time.temporal.ChronoUnit.MINUTES;
import static java.time.temporal.ChronoUnit.MONTHS;
import static java.time.temporal.ChronoUnit.SECONDS;
import static java.time.temporal.ChronoUnit.WEEKS;
import static java.time.temporal.ChronoUnit.YEARS;
import java.time.temporal.ChronoUnit;

public final class ChronoUnitDatelineModel extends
DatelineModel<ChronoUnit> {

    public ChronoUnitDatelineModel() {
        addResolution(new ChronoUnitResolution(MILLIS, "EEEE, dd. MMMM YYYY,
HH:mm:ss:SSS", 1, TOP, ONLY));
        addResolution(new ChronoUnitResolution(MILLIS, "EEEE, dd.MM.YY,
HH:mm:ss:SSS", 1, TOP, ONLY));
        addResolution(new ChronoUnitResolution(MILLIS, "E, dd.MM.YY,
HH:mm:ss:SSS", 1, TOP, ONLY));
        addResolution(new ChronoUnitResolution(MILLIS, "dd.MM.YY, HH:mm:ss:SSS",
1, TOP, ONLY));
        addResolution(new ChronoUnitResolution(MILLIS, "dd.MM, HH:mm:ss:SSS", 1,
TOP));
        addResolution(new ChronoUnitResolution(MILLIS, "SSS", 1, BOTTOM));
        addResolution(new ChronoUnitResolution(MILLIS, "SSS", 5, BOTTOM));
        addResolution(new ChronoUnitResolution(MILLIS, "SSS", 10, BOTTOM));
        addResolution(new ChronoUnitResolution(MILLIS, "SSS", 15, BOTTOM));
        addResolution(new ChronoUnitResolution(SECONDS, "EEEE, dd. MMMM YYYY,
HH:mm:ss", 1, TOP, ONLY));
        addResolution(new ChronoUnitResolution(SECONDS, "EEEE, dd.MM.YY,
HH:mm:ss", 1, TOP, ONLY));
        addResolution(new ChronoUnitResolution(SECONDS, "E, dd.MM.YY, HH:mm:ss",
1, TOP, ONLY));
        addResolution(new ChronoUnitResolution(SECONDS, "dd.MM.YY, HH:mm:ss", 1,
TOP, ONLY));
        addResolution(new ChronoUnitResolution(SECONDS, "dd.MM, HH:mm:ss", 1,
TOP));
        addResolution(new ChronoUnitResolution(SECONDS, "HH:mm:ss", 1, MIDDLE));
    }

```

```

    addResolution(new ChronoUnitResolution(SECONDS, "ss", 1, BOTTOM));
    addResolution(new ChronoUnitResolution(SECONDS, "ss", 5, BOTTOM));
    addResolution(new ChronoUnitResolution(SECONDS, "ss", 10, BOTTOM));
    addResolution(new ChronoUnitResolution(SECONDS, "ss", 15, BOTTOM));
    addResolution(new ChronoUnitResolution(MINUTES, "EEEE, dd. MMMM YYYY,
HH:mm", 1, TOP, ONLY));
    addResolution(new ChronoUnitResolution(MINUTES, "EEEE, dd.MM.YY, HH:mm",
1, TOP, ONLY));
    addResolution(new ChronoUnitResolution(MINUTES, "E, dd.MM.YY, HH:mm", 1,
TOP, ONLY));
    addResolution(new ChronoUnitResolution(MINUTES, "dd.MM.YY, HH:mm", 1,
TOP, ONLY));
    addResolution(new ChronoUnitResolution(MINUTES, "dd.MM, HH:mm", 1, TOP));
    addResolution(new ChronoUnitResolution(MINUTES, "HH:mm", 1, MIDDLE));
    addResolution(new ChronoUnitResolution(MINUTES, "mm", 1, BOTTOM));
    addResolution(new ChronoUnitResolution(MINUTES, "mm", 5, BOTTOM));
    addResolution(new ChronoUnitResolution(MINUTES, "mm", 10, BOTTOM));
    addResolution(new ChronoUnitResolution(MINUTES, "mm", 15, BOTTOM));
    addResolution(new ChronoUnitResolution(HOURS, "EEEE, dd. MMMM YYYY,
HH:mm", 1, TOP, ONLY));
    addResolution(new ChronoUnitResolution(HOURS, "EEEE, dd. MMMM YYYY", 1,
TOP));
    addResolution(new ChronoUnitResolution(HOURS, "EEEE, dd.MM.YY, HH:mm", 1,
TOP, BOTTOM, ONLY));
    addResolution(new ChronoUnitResolution(HOURS, "E, dd.MM.YY, HH:mm", 1,
TOP, ONLY));
    addResolution(new ChronoUnitResolution(HOURS, "dd.MM.YY, HH:mm", 1, TOP,
ONLY));
    addResolution(new ChronoUnitResolution(HOURS, "dd.MM, HH:mm", 1, TOP,
ONLY));
    addResolution(new ChronoUnitResolution(HOURS, "H:mm", 1, MIDDLE,
BOTTOM));
    addResolution(new ChronoUnitResolution(HOURS, "H:mm", 3, MIDDLE,
BOTTOM));
    addResolution(new ChronoUnitResolution(HOURS, "H:mm", 6, MIDDLE,
BOTTOM));
    addResolution(new ChronoUnitResolution(DAYS, "EEEE d. MMMM YYYY", 1, TOP,
ONLY));
    addResolution(new ChronoUnitResolution(DAYS, "EEEE d. MMMM YY", 1, TOP,
ONLY));
    addResolution(new ChronoUnitResolution(DAYS, "E, d. MMMM YY", 1, TOP,
ONLY));
    addResolution(new ChronoUnitResolution(DAYS, "E, d. MMMM", 1, TOP,
ONLY));
    addResolution(new ChronoUnitResolution(DAYS, "E, dd.MM.YY", 1, TOP,
ONLY));
    addResolution(new ChronoUnitResolution(DAYS, "EEEE dd", 1, MIDDLE,
BOTTOM));
    addResolution(new ChronoUnitResolution(DAYS, "E dd", 1, MIDDLE, BOTTOM));
    addResolution(new ChronoUnitResolution(DAYS, "dd.MM", 1, MIDDLE,
BOTTOM));
    addResolution(new ChronoUnitResolution(DAYS, "dd", 1, BOTTOM));
    addResolution(new ChronoUnitResolution(WEEKS, "'W' w, EEEE d. MMMM YY",

```

```

1, TOP, ONLY));
    addResolution(new ChronoUnitResolution(WEEKS, "'W' w, d. MMMM YY", 1,
TOP, ONLY));
    addResolution(new ChronoUnitResolution(WEEKS, "'W' w, d. MMMM", 1));
    addResolution(new ChronoUnitResolution(WEEKS, "'W' w, E, dd.MM.YY", 1,
TOP, ONLY));
    addResolution(new ChronoUnitResolution(WEEKS, "'W' w, dd.MM.YY", 1, TOP,
ONLY));
    addResolution(new ChronoUnitResolution(WEEKS, "'W' w, dd.MM", 1,
BOTTOM));
    addResolution(new ChronoUnitResolution(WEEKS, "'W' w", 1, MIDDLE,
BOTTOM));
    addResolution(new ChronoUnitResolution(MONTHS, "MMMM YYYY", 1, TOP,
ONLY));
    addResolution(new ChronoUnitResolution(MONTHS, "MMMM", 1, MIDDLE,
BOTTOM));
    addResolution(new ChronoUnitResolution(MONTHS, "MMM", 1, MIDDLE,
BOTTOM));
    addResolution(new ChronoUnitResolution(MONTHS, "M", 1, MIDDLE, BOTTOM));
    addResolution(new ChronoUnitResolution(YEARS, "YYYY", 1));
    addResolution(new ChronoUnitResolution(DECADES, "YYYY", 1));
    addResolution(new ChronoUnitResolution(CENTURIES, "YYYY", 1));
    addResolution(new ChronoUnitResolution(MILLENNIA, "YYYY", 1));
}

```

@Override

```

public final ChronoUnit nextTemporalUnit(ChronoUnit unit) {
    switch (unit) {
        case NANOS:
            return MICROS;
        case MICROS:
            return MILLIS;
        case MILLIS:
            return SECONDS;
        case SECONDS:
            return MINUTES;
        case MINUTES:
            return HOURS;
        case HOURS:
            return DAYS;
        case DAYS:
            return WEEKS;
        case WEEKS:
            return MONTHS;
        case MONTHS:
            return YEARS;
        case YEARS:
            return DECADES;
        case DECADES:
            return CENTURIES;
        case CENTURIES:
            return MILLENNIA;
        default:

```

```
/*  
 * We are ignoring HALF DAYS.  
 */  
return null;
```



```
}  
}  
}
```

Simple Unit Dateline Model

The **SimpleUnitDatelineModel** class is a specialization for the **SimpleUnit** temporal which ships with FlexGanttFX. It requires scale resolutions of type **SimpleUnitResolution**. The implementation of this model class below shows why the unit is called "simple".

SimpleUnitDatelineModel

```
/**  
 * Copyright (C) 2014 Dirk Lemmermann Software & Consulting (dlsc.com)  
 *  
 * This file is part of FlexGanttFX.  
 */  
package com.flexganttfx.model.dateline;  
  
import com.flexganttfx.model.util.SimpleUnit;  
  
public final class SimpleUnitDatelineModel extends  
DatelineModel<SimpleUnit> {  
  
    public SimpleUnitDatelineModel() {  
        for (SimpleUnit unit : SimpleUnit.values()) {  
            addResolution(new SimpleUnitResolution(unit, "", 1));  
        }  
    }  
  
    @Override  
    public SimpleUnit nextTemporalUnit(SimpleUnit unit) {  
        int ordinal = unit.ordinal();  
        if (ordinal < SimpleUnit.values().length - 1) {  
            return SimpleUnit.values()[ordinal + 1];  
        }  
        return null;  
    }  
}
```

Timezones

The dateline model manages a list of zone IDs, which is used by the UI to create menu items for each ID. This way the user can easily toggle between them. The default list is set up in the **DatelineModel** class like this:

DatelineModel

```
/**
 * Constructs a new model and populates the list of available zone IDs.
 */
protected DatelineModel() {
    addZoneId("Europe/Berlin");
    addZoneId("America/New_York");
    addZoneId("Australia/Darwin");
    addZoneId("Australia/Sydney");
    addZoneId("America/Argentina/Buenos_Aires");
    addZoneId("Africa/Cairo");
    addZoneId("America/Anchorage");
    addZoneId("America/Sao_Paulo");
    addZoneId("Asia/Dhaka");
    addZoneId("Africa/Harare");
    addZoneId("America/St_Johns");
    addZoneId("America/Chicago");
    addZoneId("Asia/Shanghai");
    addZoneId("Africa/Addis_Ababa");
    addZoneId("Europe/Paris");
    addZoneId("America/Indiana/Indianapolis");
    addZoneId("Asia/Kolkata");
    addZoneId("Asia/Tokyo");
    addZoneId("Pacific/Apia");
    addZoneId("Asia/Yerevan");
    addZoneId("Pacific/Auckland");
    addZoneId("Asia/Karachi");
    addZoneId("America/Phoenix");
    addZoneId("America/Puerto_Rico");
    addZoneId("America/Los_Angeles");
    addZoneId("Pacific/Guadalcanal");
    addZoneId("Asia/Ho_Chi_Minh");
}
```

Please note that the currently used timezone is managed by the [Dateline control](#), not the model.

3.8 Eventline

- [Introduction](#)
- [Date & Time Formatting](#)
- [Cursor: Location & Time](#)
- [Marked Time Interval](#)

Introduction

The eventline is a control that displays time cursors: time at mouse location, selected time interval. This control is part of the [Timeline](#) and displayed at the bottom of it.

W 17, Monday 21. April 14							W 18, Monday 28. April 14							W 19, Monday 5. May 14						
21	22	23	24	25	26	27	28	29	30	01	02	03	04	05	06	07	08	09	10	11
Apr 27, 2014 6:37:07 AM																				

Date & Time Formatting

Each application has its own requirements regarding the format in which dates and times are displayed. Accordingly the eventline features a date and time formatter that can be replaced by calling **setDateTimeFormatter()**. Formatter instances can be looked up by calling static methods on the **DateTimeFormatter** class, e.g. **DateTimeFormatter.ISO_LOCAL_DATE_TIME**.

Cursor: Location & Time

The eventline keeps track of the mouse cursor location when the mouse hovers over the [timeline](#) or the [graphics](#) control. The location is stored in the read-only **cursorLocationProperty()**. Whenever the location changes the eventline will also update the value of **cursorTimeProperty()**. These two properties make the eventline the perfect provider for cursor information for the entire application.

Marked Time Interval

Whenever the user edits an activity the eventline will display the new time interval occupied by the activity. This interval is stored in the **markedTimeIntervalProperty()**. When its' value changes the eventline will display two additional time cursors, one for the beginning of the time interval and one for its' end.

4. Model

The following table lists the most important model classes for populating a [Gantt chart](#) with data.

Class	Desription
Activity	Activities represent objects that will be displayed below the timeline in the graphics view of the Gantt chart control. Activities can be added to a specific layer on a row.
ActivityRef	An activity reference is used to precisely identify the location of an activity where the location is a combination of row, layer, and the activity itself.
ActivityLink	An activity link can be used to express a dependency between two activities.
ActivityRepository	Activity repositories are used by rows to store and lookup activities.
Row	A (model) row object is used to store the activities found on a (visual) row of the Gantt chart.
Layer	Layers are used to create groups of activities.
LinesManager	A lines manager is used to control the layout of (inner) lines inside a row.
Layout	Each row and each inner line of a row are associated with a layout. The layout influences several aspects during rendering and editing of activities. Additionally several of the system layers used for drawing the row background also utilize the layout information.
Calendar	A calendar is an extension of an activity repository with the addition of a name and a visibility property.

4.1 Activity

- [Introduction](#)
- [Activity Types](#)

Introduction

Activities represent objects that will be displayed below the timeline in the [graphics view](#) of the [Gantt chart](#) control. Activities can be added to a specific layer on a row by calling **Row.addActivity(Layer, Activity)**.

Activity Types

The following table lists all available activity types.

Only mutable activity types can be edited interactively by the user. Any activity type that is not mutable can only be used for read-only purposes.

Activity Interface	Base Implementation	Description & Attributes	Editable
Activity	ActivityBaseBase	<p>The simplest form of an activity.</p> <ul style="list-style-type: none"> • id (String) • name (String) • startTime (Instant) • endTime (Instant) 	
ChartActivity	ChartActivityBaseBase	<p>These activities can be displayed in a chart layout.</p> <ul style="list-style-type: none"> • id (String) • name (String) • startTime (Instant) • endTime (Instant) • chartValue (double) 	
CompletableActivity	CompletableActivityBase	<p>These activities carry a percentage value (completion).</p> <ul style="list-style-type: none"> • id (String) • name (String) • startTime (Instant) • endTime (Instant) • percentageComplete (double) 	
HighLowChartActivity	HighLowChartActivityBase	<p>These activities can be displayed in a chart layout.</p> <ul style="list-style-type: none"> • id (String) • name (String) • startTime (Instant) • endTime (Instant) • high (double) • low (double) 	
MutableActivity	MutableActivityBase	<p>The simplest form of a mutable activity. The user can change the start and end time of these activities.</p> <ul style="list-style-type: none"> • id (String) • name (String) • startTime (Instant) • endTime (Instant) 	✓
MutableChartActivity	MutableChartActivityBase	<p>These activities can be displayed in a chart layout. The user can change the start and end time and the chart value of these activities.</p> <ul style="list-style-type: none"> • id (String) • name (String) • startTime (Instant) • endTime (Instant) • chartValue (double) 	✓
MutableCompletableActivity	MutableCompletableActivityBase	<p>These activities carry a percentage value (completion). The user can change the start and end time and the percentage complete value of these activities.</p> <ul style="list-style-type: none"> • id (String) • name (String) • startTime (Instant) • endTime (Instant) • percentageComplete (double) 	✓
MutableHighLowChartActivity	MutableHighLowChartActivityBase	<p>These activities can be displayed in a chart layout. The user can change the start and end time and the high and low value of these activities.</p> <ul style="list-style-type: none"> • id (String) • name (String) • startTime (Instant) • endTime (Instant) • high (double) • low (double) 	✓

4.1.1 ChartActivity

A chart activity is an add-on interface for [activities](#). It needs to be implemented by activities that want to participate in a [ChartLayout](#). The interface adds a chart value to the activity. The image below shows an example of a chart layout laying out one chart activity per day.



4.1.2 CompletableActivity

A completable activity is an activity that carries a "percentage complete" value between 0 and 100%. Completable activities are drawn with a "[completable activity bar renderer](#)". This renderer fills the background of the activity based on the percentage complete value. The image below shows an example.



4.1.3 HighLowChartActivity

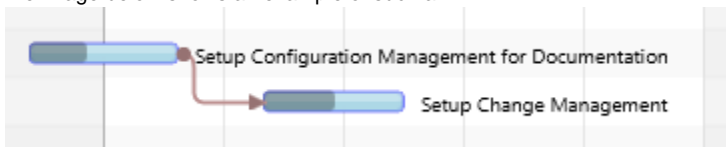
A high low chart activity carries two extra attributes: high and low. These values are used by the [ChartLayout](#) to position them appropriately. One example for a good use case for high low activities are candlestick charts (e.g. stocks open / high / low / close price).

4.2 ActivityRef

An activity reference is used to precisely identify the "location" of an activity. A location is the combination of row, layer, and the activity itself. As the same activity can be located on multiple rows and or multiple layers at the same time it is often necessary to work with an activity reference instead of only the activity.

4.3 ActivityLink

An activity link can be used to model any kind of dependency between two [activities](#). In project planning applications a link would express a predecessor / successor relationship between two tasks, for example "task A must be finished before task B can begin". In other domains a link might simply express that two or more activities need to be scheduled together and that moving one of them requires all others to be moved, too. The image below shows an example of such a link.



A link can be added to the Gantt chart by calling `GanttChart.getLinks().add(myLink);`

4.4 ActivityRepository

- [Introduction](#)
- [Queries](#)
- [Earliest / Latest Time Used](#)
- [Updating Activities](#)
- [Event Handling](#)

Introduction

Activity repositories are used by rows to store and lookup activities. Each row by default owns an [IntervalTreeActivityRepository](#). This default repository can be replaced with a custom one, for example if your application requires a lazy loading strategy.

Queries

The most important functionality of any repository is the ability to query the repository for activities within a given time interval. For this purpose the [ActivityRepository](#) interface defines the [getActivities\(\)](#) method with these parameters:

Parameter	Description
-----------	-------------

Layer layer	Whenever the user scrolls left or right the row will query the repository several times. Once for each layer.
Instant startTime	The start time of the time interval for which the row is querying activities.
Instant endTime	The end time of the time interval for which the row is querying activities.
TemporalUnit unit	The current value of the primary temporal unit currently displayed by the dateline . This is the unit shown at the bottom of the dateline , e.g. days. This parameter can be used to control how fine-grained the result will be. If we know that the user is currently looking at months then it might make sense to aggregate daily activities.
ZoneId zoneId	The timezone shown by the row.

Earliest / Latest Time Used

Each repository implementation needs to be able to answer the question for the earliest and latest times used (earliest start time / latest end time of any activity stored in the repository). This allows the UI to provide controls for easy navigation: "show earliest", "show latest". For this purpose repositories need to implement the **getEarliestTimeUsed()** and **getLatestTimeUsed()** methods.

Updating Activities

Activities need to be removed (`ActivityRef.detachFromRow()`) from their repository before they are being changed and added back (`ActivityRef.attachToRow()`) after they have been changed. This is the only way to ensure that a repository will always have its underlying data structure in synch with the activities. Example: the interval tree data structure only works properly if all its nodes are in their correct location. This can only be guaranteed if the nodes are removed from the tree before they are being changed (otherwise the tree will not find them) and then reinserted with their new value.

Event Handling

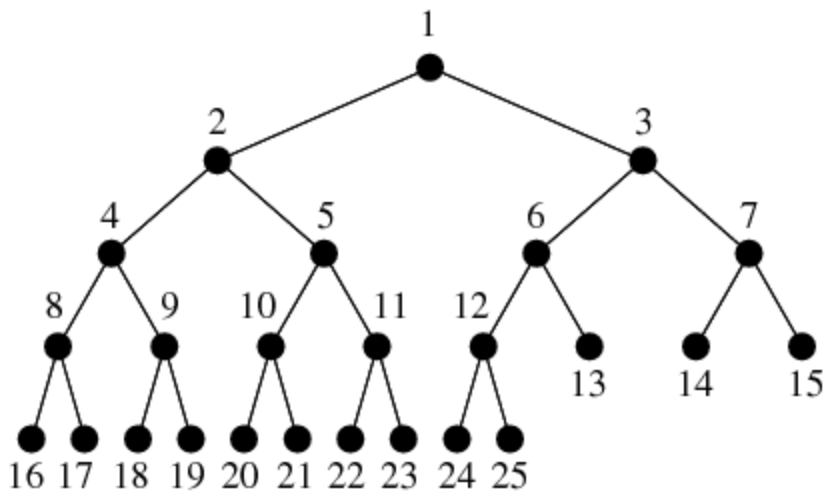
Activity repositories implement listener support so that the UI can update itself if the content of the repository changes. Interested parties can attach handlers by calling **addEventHandler()** or remove handlers by calling **removeEventHandler()**. The event class is called `RepositoryEvent` and it has three event types:

Event Type	Description
ACTIVITY_ADDED	An activity was added to the repository.
ACTIVITY_REMOVED	An activity was removed from the repository.
REPOSITORY_CHANGED	Something has changed the state of the repository.

Each one of these event types will normally trigger a redraw of the row to which the repository belongs.

4.4.1 IntervalTreeActivityRepository

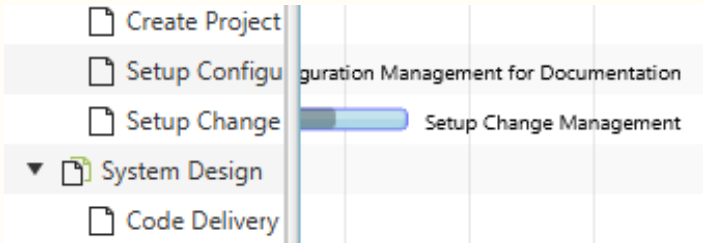
The **IntervalTreeActivityRepository** is an activity repository that is using one or more binary [interval tree](#) data structures for storing activities.



This repository type is the default repository type. It is also the preferred repository for rows with a large number of intervals (hundreds).

4.4.2 ListActivityRepository

The **ListActivityRepository** is an [activity repository](#) using one or more list data structures to store [activities](#). This repository can be configured to return different types of result iterators from its query method. The possible values are defined in **ListActivityRepository.IteratorType**.

Type	Description
BINARY_ITERATOR	Returns an iterator that performs a <u>binary</u> search to find the first activity to draw for a given time interval. It will then iterate over all following activities until it finds an activity that starts after the given time interval.
LINEAR_ITERATOR	Returns an iterator that performs a <u>linear</u> search to find the first activity to draw for a given time interval. It will then iterate over all following activities until it finds an activity that starts after the given time interval.
SIMPLE_ITERATOR	<p>Returns an iterator that does not perform any search at all but will start returning activities immediately, no matter whether they are currently located in the visible time interval of the Gantt chart or not. This iterator is used for rows with only a few activities on them.</p> <div> <p>This iterator is very useful when we want to make sure that the trailing text of an activity will still be shown even if the activity has already scrolled out of the visible area.</p>  </div>

4.5 Row

- [Introduction](#)
- [Type Arguments & Hierarchy](#)
- [Properties](#)

Introduction

A row object is used to store the [activities](#) found on a row of the [Gantt chart](#). These activities are not stored directly on the row but in an [activity repository](#). The default repository is of type [IntervalTreeActivityRepository](#). [Activities](#) can be placed on lines within the row. The row delegates this work to a [LinesManager](#). The default manager is of type [EqualLinesManager](#).

Type Arguments & Hierarchy

Three type arguments are needed to define a row. The first one defines the type of the parent row, the second one defines the type of the children rows, the third one defines the type of the activities shown on the row. The following is an example that defines a "building", that is part of a factory. The building has machines in it. In the building row we are showing shifts.

```
Building.java  
  
public class Building extends Row<Factory, Machine, Shift> {  
}
```

A model like this would allow us to display a hierarchy in the Gantt chart that might look like this:

- Factory
 - Building 1
 - Building 2
 - Building 3
 - Machine A
 - Machine B
 - Machine B
 - Building 4
 - Machine C
 - Machine D

Properties

Each row has a set of properties.

Property	Description
<code>BooleanProperty expanded</code>	Controls whether the row will show its children rows or not.
<code>DoubleProperty height</code>	The current height of the row.
<code>DoubleProperty minHeight</code>	The minimum height of the row.
<code>DoubleProperty maxHeight</code>	The maximum height of the row.
<code>ObjectProperty<Layout> layout</code>	The layout used by the row. The default is Gantt Layout .
<code>IntegerProperty lineCount</code>	The number of inner lines to show within the row.
<code>ObjectProperty<LinesManager<A>> linesManager</code>	The lines manager used for controlling the lines, their location, their height, the placement of the activities.
<code>StringProperty name</code>	The name of the row, e.g. "Machine 1", "Building 1".
<code>ReadOnlyObjectProperty<P> parent</code>	The parent row. <div>This is a read-only property that is managed internally and updated when a row gets added to the list of children of another row.</div>
<code>ObjectProperty<ActivityRepository> repository</code>	The repository used by the row to store the activities.
<code>BooleanProperty showing</code>	A flag used to signal that the row is currently shown in the UI. This information can be used for optimizing lazy loading strategies.

ObjectProperty<Object> userObject	An optional user object. Used to have a bridge to the business model.
ObjectProperty<ZoneId> zoneId	The timezone represented by the row. Each row can be in its own timezone.

4.6 Layer

Layers are used to group [activities](#) together. Activities on the same layer are drawn at the same time (z-order). A layer has a name, an ID, it can be turned on / off, and their opacity can be changed. These changes have an impact on all activities on that layer. The ID of the layer is used for drag and drop operations of activities between different [Gantt charts](#). Dropped activities will be added to the layer with the same ID. The layer name will be used as the default ID for newly created layers. The ID only needs to be changed if the same layer type will be used with different names in different [Gantt charts](#).

4.7 LinesManager

- [Introduction](#)
- [Line Count](#)
- [Interface](#)
- [Equal Lines Manager](#)
- [Auto Lines Manager](#)

Introduction

A lines manager is used to control the layout of lines inside a [row](#). [Activities](#) located on different lines do not overlap each other, except if the lines themselves overlap each other. Each line can have its own height and a location within the row. Each line can also have its own [Layout](#) . By using lines and [layouts](#) it is possible to display activities that belong to the same row in different ways (see [ChartLayout](#), [AgendaLayout](#), [GanttLayout](#)).

Line Count

The actual number of lines on a row is stored on the **lineCount** property of the row. Simply call `Row.setLineCount(int)` to change its value. If the line count is larger than 0 the row will call back on a its line manager to figure out where each line is located, how high it is, and which activity will be placed on which row. Also the type of layout to use for each line will be retrieved from the manager.

Interface

The following table describes the interface methods.

Method	Description
<code>double getLineHeight(int lineIndex, double rowHeight);</code>	Returns the height of the line with the given index. The height can be computed on-the-fly based on the given available row height.
<code>int getLineIndex(A activity);</code>	Returns the line index for the given activity. This method places activities on different lines.
<code>Layout getLineLayout(int lineIndex);</code>	Returns the layout for the line with the given line index. Each line can have its own layout.
<code>double getLineLocation(int lineIndex, double rowHeight);</code>	Returns the location of the line with the given index. The location can be computed on-the-fly based on the given available row height.

Equal Lines Manager

The **EqualLinesManager** can be used to equally distribute line locations and line heights on a row. Each line will have the same height and the lines will not overlap each other. While this behaviour will be provided by the manager it is still the responsibility of the application to place the activities on different rows and to specify the layout for each line. This is also the reason why the methods **getLineHeight()** and **getLineLocation()** are final while the methods **getLineLayout()** and **getLineIndex()** are not and can be overridden.

Auto Lines Manager

The **AutoLinesManager** can be used to create a dynamic number of lines based on all [activities](#) inside a [repository](#). This lines manager detects

clusters of intersecting activities (start / end time intervals) and ensures that enough lines are available to place the activities in a non-overlapping way. Below you are finding the complete source code of this manager class as a case study. Please note that the manager's **layout()** method needs to be invoked from the outside. A good way to do this is to listen to [ACTIVITY_CHANGE_FINISHED](#) events or even more fine grained [START/END_TIME_CHANGE_FINISHED](#) events.

AutoLinesManager

```
/**
 * Copyright (C) 2014 Dirk Lemmermann Software & Consulting (dlsc.com)
 *
 * This file is part of FlexGanttFX.
 */
package com.flexganttfx.view.util;
import static java.util.Objects.requireNonNull;
import impl.com.flexganttfx.skin.util.Placement;
import impl.com.flexganttfx.skin.util.Resolver;
import java.time.Instant;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import com.flexganttfx.model.Activity;
import com.flexganttfx.model.ActivityRepository;
import com.flexganttfx.model.Layer;
import com.flexganttfx.model.LinesManager;
import com.flexganttfx.model.Row;
import com.flexganttfx.model.layout.EqualLinesManager;
import com.flexganttfx.view.graphics.GraphicsBase;
/**
 * A specialized {@link LinesManager} used for ensuring that activities
 * will not
 * overlap each other. This manager will create as many inner lines as
 * needed
 * and will calculate the placement of all activities on these lines.
 *
 * @param <R>
 *         the type of the row that will be managed
 * @param <A>
 *         the type of the activities that will be managed
 *
 * @since 1.2
 */
public class AutoLinesManager<R extends Row<?, ?, A>, A extends Activity>
    extends EqualLinesManager<R, A> {
    private Map<A, Placement<A>> placements;
    private GraphicsBase<R> graphics;
    /**
     * Constructs a new automatic lines manager. The constructor requires a
     * reference to the graphics view to lookup various parameters that are
     * needed when the manager queries the activity repository of the row
     * (e.g.
     * the currently displayed temporal unit and the list of layers).
     *
     * @param row
     */
}
```

```

    *           the managed row
    * @param graphics
    *           the graphics view where the manager will be used
    *
    * @since 1.2
    */
public AutoLinesManager(R row, GraphicsBase<R> graphics) {
    super(row);
    this.graphics = requireNonNull(graphics);
    layout();
}
/**
 * Returns the graphics view where the manager will be used.
 *
 * @return the graphics view
 * @since 1.2
 */
public final GraphicsBase<R> getGraphics() {
    return graphics;
}
public final void layout() {
    R row = getRow();
    ActivityRepository<A> repository = row.getRepository();
    Instant st = repository.getEarliestTimeUsed();
    Instant et = repository.getLatestTimeUsed();
    if (st == null || et == null) {
        return;
    }
    List<A> allActivities = new ArrayList<>();
    for (Layer layer : graphics.getLayers()) {
        Iterator<A> activities = repository.getActivities(layer, st, et,
            graphics.getTimeline().getDateline()
                .getPrimaryTemporalUnit(), row.getZoneId());
        if (activities != null) {
            activities.forEachRemaining(activity -> allActivities
                .add(activity));
        }
    }
    placements = Resolver.resolve(allActivities);
    if (placements != null && !placements.isEmpty()) {
        Placement<A> p = placements.values().iterator().next();
        row.setLineCount(p.getColumnCount());
    } else {
        row.setLineCount(0);
    }
}
@Override
public int getLineIndex(A activity) {
    if (placements != null) {
        Placement<A> placement = placements.get(activity);
        if (placement != null) {
            return placement.getColumnIndex();
        }
    }
}

```

}

```

    return -1;
}
}

```

4.8 Layout

- [Introduction](#)
- [Layout Types](#)
- [Padding](#)

Introduction

Each row and each inner line of a row are associated with a layout. The layout influences several aspects during rendering and editing of activities. Additionally several of the system layers used to draw the row background also utilize the layout information. The layout can be set by calling `Row.setLayout(Layout)` or when using inner lines by returning them via the [lines manager](#) of the row.

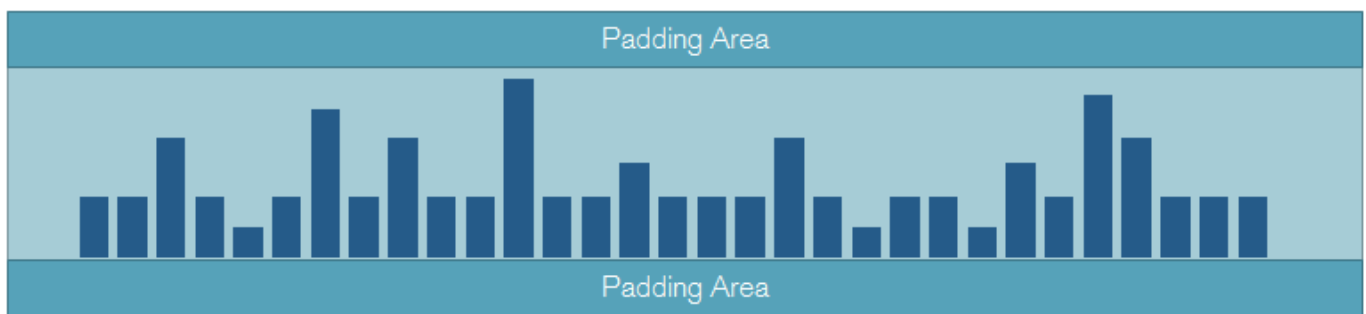
Layout Types

Three layout types are included in FlexGanttFX.

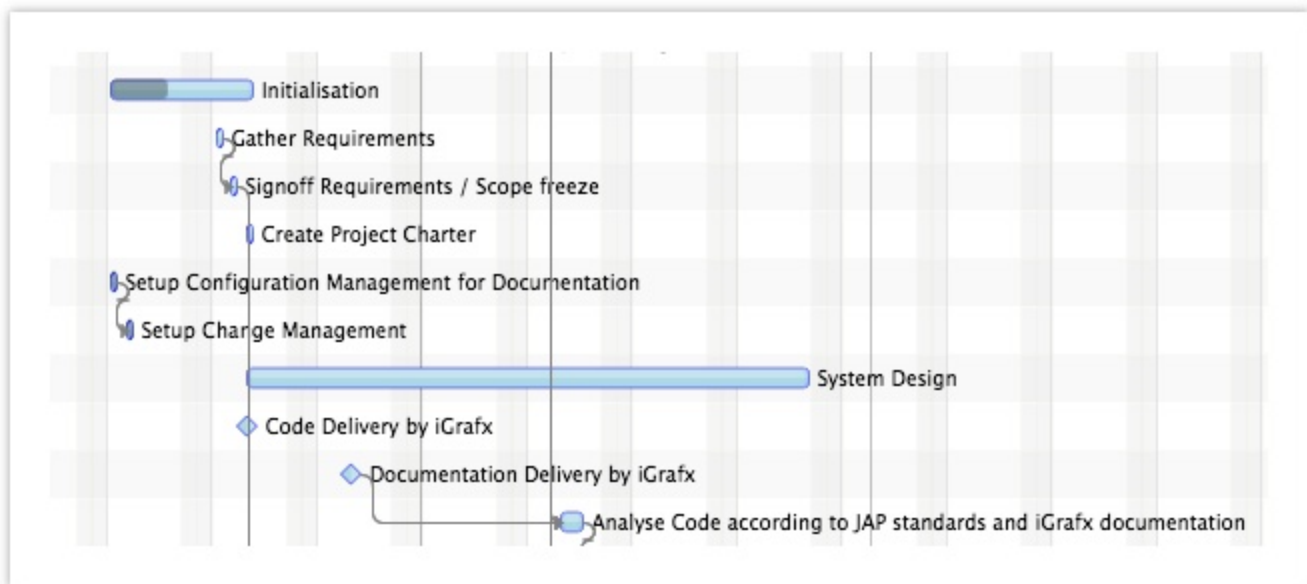
Layout	Description
GanttLayout	Lays out activities horizontally along the timeline. Positions are based on the start and end times of the activities.
AgendaLayout	Lays out activities vertically along a "local time" scale (0 - 24 hours). This makes activities look like calendar entries.
ChartLayout	Lays out activities as chart values. Activities can implement the ChartActivity or the HighLowChartActivity interface.

Padding

All layout types have a **padding** property. It is used to create a visual gap at the top and bottom of each row / line.



4.8.1 Gantt Layout



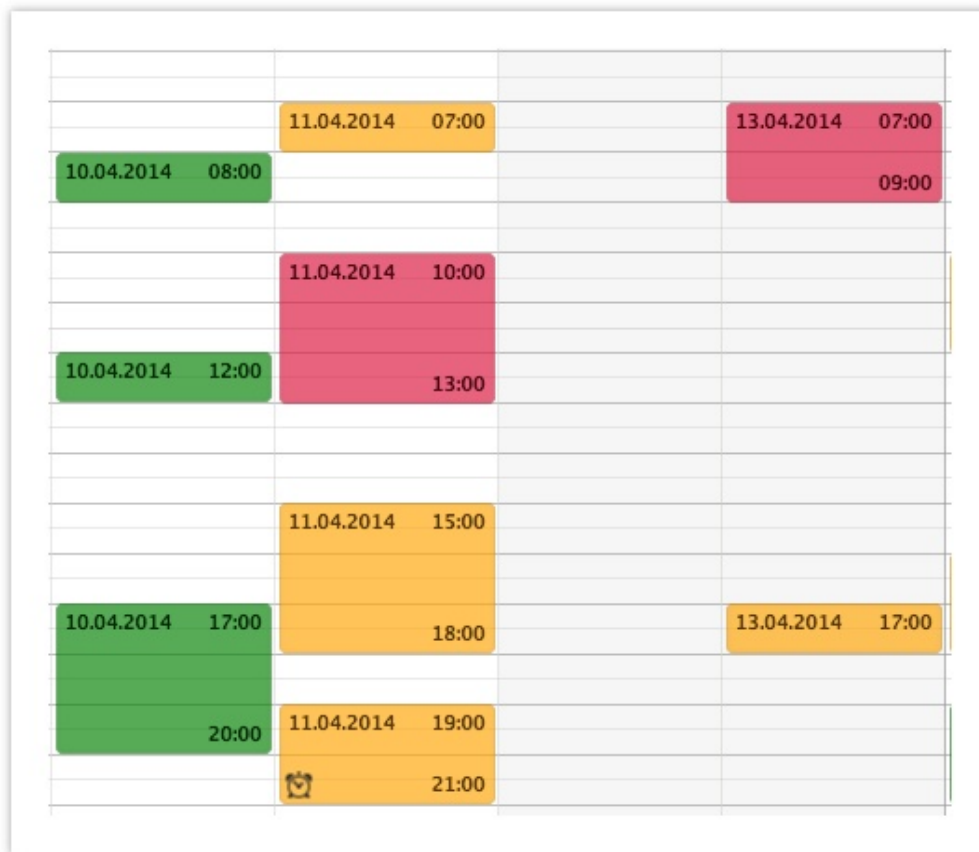
4.8.2 Agenda Layout

- Introduction
- Start and End Time
- Conflict Strategy

Introduction

The agenda layout class is used to lay out activities in a style similar to a regular calendar where a vertical scale will display hours. Activities are used to represent appointments for a given day.

Activities shown in agenda layout might be rendered several times. This is, for example, the case when an activity spans several days.



Start and End Time

The agenda layout class allows you to specify a start and end time. This is used to restrict the time interval that is shown and in which the agenda activities are laid out. In most cases it does not make sense to show the entire 24 hours but only the working hours, e.g. 8am until 6pm. Simply call **AgendaLayout.setStartTime()** or **setEndTime()** to change the time range.

Conflict Strategy

Activities in an agenda layout might intersect with each other. The conflictStrategy property allows you how to handle these situations. The following table lists the possible values.

Strategy	Description
OVERLAPPING	<p>Conflicting agenda entries will be drawn on top of each other but with one of them being indented by a couple of pixels.</p> <p>The indentation amount can be controlled via the overlapOffset property on AgendaLayout.</p>
PARALLEL	Conflicting agenda entries will be displayed in different columns within the same day.

4.8.3 Chart Layout

Introduction

Using the ChartLayout class results in activities being laid out as chart bars. A series of such bars can for example be used to form a capacity profile. Activities of type **ChartActivity** will be placed on a zeroline between the minimum and the maximum value of the layout. The height of the chart activity will be based on the value returned by **ChartActivity.getChartValue()**. Activities of type **HighLowChartActivity** will appear as floating bars. The layout also supports the definition of minor and major chart lines drawn in the row background.



Min & Max Value

The chart layout provides two properties that control the actual layout of the chart activities: **minValue** and **maxValue**. These values have to be managed by the application, not the framework. They can be set by calling **ChartLayout.setMinValue()** or **ChartLayout.setMaxValue()**.

Major & Minor Ticks

A list of major and minor ticks is available on each chart layout instance. Values can be added to these lists in order to render value lines in the background of the row. Example: the min value is equal to 0 the max value is equal to 100. Then it would make sense to define major ticks for the values 50 and 100. Minor ticks might be at 10, 20, 30, 40, 60, 70, 80 and 90.

4.9 Calendar

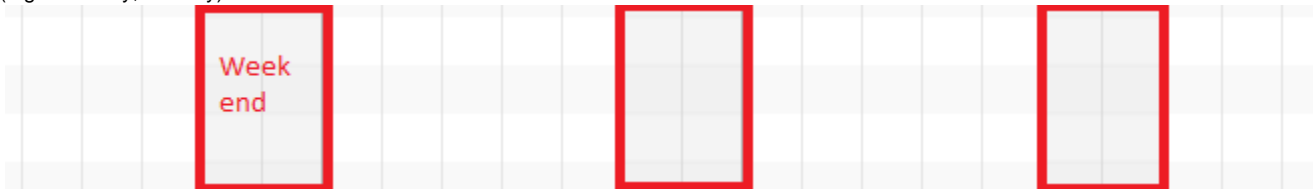
- [Introduction](#)
- [Weekend Calendar](#)

Introduction

A calendar is an extension of an [activity repository](#) with the addition of a **name** and a **visibility** property. Calendars can be added to the whole [Gantt chart](#) or to individual [rows](#) within the Gantt chart. Calendars are used for the background of rows. They can mark things like weekend days or holidays. Calendar information is always shown as read-only. Activities returned by calendars have to be of type **CalendarActivity**. They can not be edited interactively by the user.

Weekend Calendar

There already is a predefined calendar type included in **FlexGanttFX**. It is called **WeekendCalendar** and it is used to mark the weekend days (e.g. Saturday, Sunday).



The following listing shows the entire code of this calendar class. It can be used as a basis for your own calendars.

WeekendCalendar.java

```
/**
 * Copyright (C) 2014 Dirk Lemmermann Software & Consulting (dlsc.com)
 * This file is part of FlexGanttFX.
 */
package com.flexganttfx.model.calendar;
import static java.time.temporal.ChronoUnit.DAYS;
import static java.util.Objects.requireNonNull;
import java.time.DayOfWeek;
import java.time.Instant;
import java.time.ZoneId;
```



```

import java.time.ZonedDateTime;
import java.time.temporal.ChronoUnit;
import java.time.temporal.TemporalUnit;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.EnumSet;
import java.util.Iterator;
import java.util.List;

import javafx.event.Event;

import com.flexganttfx.model.Layer;
import com.flexganttfx.model.repository.RepositoryEvent;
/**
 * A calendar specialized on returning activities that represent weekend
days
 * (default: saturday, sunday). The days that are considered weekend days
can be
 * configured by calling {@link #setWeekendDays(DayOfWeek...)}.
 *
 * @since 1.0
 */
public class WeekendCalendar extends CalendarBase<WeekendCalendarActivity>
{
    private Instant lastStartTime = Instant.MIN;
    private Instant lastEndTime = Instant.MAX;
    private ZoneId lastZoneId;
    private List<WeekendCalendarActivity> entries;
    private EnumSet<DayOfWeek> weekendDays = EnumSet.of(DayOfWeek.SATURDAY,
        DayOfWeek.SUNDAY);
    /**
     * Constructs a new weekend calendar.
     *
     * @since 1.0
     */
    public WeekendCalendar() {
        super("Weekends");
    }
    /**
     * Sets the days of the week that are considered to be a weekend day. By
     * default {@link DayOfWeek#SATURDAY} and {@link DayOfWeek#SUNDAY} are
     * considered weekend days.
     *
     * @param days
     *         the days of the week that are to be considered weekend days
     * @since 1.0
     */
    public void setWeekendDays(DayOfWeek... days) {
        requireNonNull(days);
        weekendDays.clear();
        weekendDays.addAll(Arrays.asList(days));
        Event.fireEvent(this, new RepositoryEvent(this));
    }

```

```

}
/**
 * Returns the days of the week that are to be considered weekend days. By
 * default {@link DayOfWeek#SATURDAY} and {@link DayOfWeek#SUNDAY} are
 * considered weekend days.
 *
 * @return the days of the week used as weekend days
 * @since 1.0
 */
public DayOfWeek[] getWeekendDays() {
    return weekendDays.toArray(new DayOfWeek[weekendDays.size()]);
}
@Override
public Iterator<WeekendCalendarActivity> getActivities(Layer layer,
    Instant startTime, Instant endTime, TemporalUnit temporalUnit,
    ZoneId zoneId) {

    if (!(temporalUnit instanceof ChronoUnit)) {
        /*
         * We only work for ChronoUnit.
         */
        return Collections.emptyListIterator();
    }
    if (startTime.equals(lastStartTime) && endTime.equals(lastEndTime)
        && zoneId.equals(lastZoneId)) {
        /*
         * We already answered this query for the given time interval. Let's
         * return the result from last time.
         */
        if (entries != null) {
            return entries.iterator();
        }
    } else {
        ChronoUnit unit = (ChronoUnit) temporalUnit;
        /*
         * The time interval has changed. Find the weekends within the new
         * interval, but only if the user is currently looking at days or
         * weeks.
         */
        if (isSupportedUnit(unit)) {
            /* Lazily create list structure. */
            if (entries == null) {
                entries = new ArrayList<WeekendCalendarActivity>();
            } else {
                entries.clear();
            }
            ZonedDateTime st = ZonedDateTime.ofInstant(startTime, zoneId);
            ZonedDateTime et = ZonedDateTime.ofInstant(endTime, zoneId);
            findWeekends(st, et);
            lastStartTime = startTime;
            lastEndTime = endTime;
            lastZoneId = zoneId;
            return entries.iterator();
        }
    }
}

```

```

    }
}

return Collections.emptyListIterator();
}
/**
 * Determines if weekends will be shown for the given temporal unit.
 * By default we only show weekends for {@link ChronoUnit#DAYS} and
 * {@link ChronoUnit#WEEKS}. To support more units simply override
 * this method in a subclass.
 *
 * @param unit
 *         the unit to check
 * @return true if weekend information will be shown in the Gantt chart
 * @since 1.0
 */
protected boolean isSupportedUnit(TemporalUnit unit) {
    if (unit instanceof ChronoUnit) {
        ChronoUnit chronoUnit = (ChronoUnit) unit;
        switch (chronoUnit) {
            case DAYS:
            case WEEKS:
                return true;
            default:
                return false;
        }
    }
    return false;
}

private void findWeekends(ZonedDateTime st, ZonedDateTime et) {
    while (st.isBefore(et) || st.equals(et)) {
        if (weekendDays.contains(st.getDayOfWeek())) {
            st = st.truncatedTo(DAYS);
            entries.add(new WeekendCalendarActivity(st.getDayOfWeek()
                .toString(), Instant.from(st), Instant.from(st
                    .plusDays(1)), st.getDayOfWeek()));
        }
        st = st.plusDays(1);
    }
}

```





```
}
```

5. Styling (CSS)

FlexGanttFX ships with several custom controls. Each one of them has its own stylesheet. The following table lists the controls and their associated CSS stylesheets.

Control	Stylesheet
GanttChart	gantt.css
GraphicsBase	graphics.css
Timeline	timeline.css
Dateline	dateline.css
Eventline	eventline.css

For convenience the files have been attached to this page.

File	Modified 
 gantt.css	Oct 07, 2014 by Dirk Lemmermann [Administrator]
 graphics.css	Oct 07, 2014 by Dirk Lemmermann [Administrator]
 dateline.css	Oct 07, 2014 by Dirk Lemmermann [Administrator]
 eventline.css	Oct 07, 2014 by Dirk Lemmermann [Administrator]
 timeline.css	Oct 07, 2014 by Dirk Lemmermann [Administrator]

 [Download All](#)

dateline.css

```
/*
 * dateline.css file of FlexGanttFX
 *
 * Copyright 2014 Dirk Lemmermann Software & Consulting
 */

.dateline {
    -fx-background-color: transparent;
}

.dateline-content {
}

.dateline-cell {
```

```

-fx-padding: 2 6 2 6;
-fx-background-color: transparent;
-fx-border-color:
    derive(-fx-base, 80%)
    linear-gradient(to bottom, derive(-fx-base,80%) 20%,
derive(-fx-base,-10%) 90%)
    derive(-fx-base, 10%)
    linear-gradient(to bottom, derive(-fx-base,80%) 20%,
derive(-fx-base,-10%) 90%),
    /* Outer border: */
    transparent -fx-box-border -fx-box-border transparent;
-fx-border-insets: 0 1 1 0, 0 0 0 0;
-fx-border-width: 0.083333em 0.083333em 0.083333em 0, 0.083333em
0.083333em 0.083333em;
}

.dateline-cell:hover {
-fx-background-color: rgba(255.0,255.0,255.0,1.0) ;
-fx-effect: innershadow( three-pass-box , rgba(0.0,0.0,0.0,0.6) , 5.0,
0.0 , 0.0 , 1.0 );
}

.dateline-cell:pressed {
-fx-background-color: rgba(255.0,255.0,255.0,0.8) ;
}

.dateline-cell > .text {
-fx-alignment: center;
}

.dateline-cell > .text:hover {
}

.dateline-cell-simpel > .text {
-fx-alignment: baseline-left;
}

.dateline-cell-selected {
-fx-border-color: transparent transparent transparent -fx-box-border, red
transparent transparent transparent;
-fx-border-width: 1.0, 3.0;
-fx-border-insets: 0, 0.0 0.0 0.0 1.0;
}

.calendar-info {
-fx-background-size: 8;
-fx-background-image: url("calendar.png");
-fx-background-position: right top;
-fx-background-repeat: no-repeat;
}

.dateline-cell-last {

```

```

    -fx-border-width: 0.083333em 0 0.083333em 0.083333em, 0.083333em 0
0.083333em 0;
}

.dateline-cell-first {
    -fx-border-width: 0.083333em 0.083333em 0.083333em 0, 0.083333em 0
0.083333em 0;
}

.scale {
    -fx-background-color: transparent;

    /*
    * Scale height property is currently not supported. Timeline height and
    * tree table header height is now hard-wired and based on font size.
    */

    /* -fx-scale-height: 26.0; */
    -fx-cell-padding: 5;
}

.scale-top,
.scale-middle {
}
.scale-bottom {
}
.scale-only {
}

.centuries {}
.days {}
.decades {}
.half_days {}
.hours {}
.months {}
.micros {}
.millenia {}
.millis {}
.minutes {}
.nanos {}
.seconds {}
.weeks {}
.years {}

.scale-bottom > .saturday, .scale-bottom > .sunday {
    -fx-background-color: rgba(0,0,0,0.1) ;
}

.scale-bottom > .saturday: hover, .scale-bottom > .sunday: hover {
    -fx-background-color: rgba(0,0,0,0.1) ;
}

.january {}

```

```
.february {}
.march {}
.april {}
.may {}
.june {}
.july {}
.august {}
.september {}
.october {}
.november {}
.december {}

.am {
}

.pm {
  -fx-background-color: rgba(255.0,255.0,255.0,0.2);
}

.zone-id-label {
  -fx-background-color:
    linear-gradient(#ffd65b, #e68400),
    linear-gradient(#ffef84, #f2ba44),
    linear-gradient(#ffea6a, #efaa22),
    linear-gradient(#ffe657 0.0%, #f8c202 50.0%, #eea10b 100.0%),
    linear-gradient(from 0.0% 0.0% to 15.0% 50.0%,
rgba(255.0,255.0,255.0,0.9), rgba(255.0,255.0,255.0,0.0));
  -fx-background-radius: 0.0 0.0 0.0 0.0;
  -fx-background-insets: 0.0,1.0,2.0,3.0,0.0;
  -fx-text-fill: #654b00;
  -fx-font-weight: bold;
  -fx-padding: 6.0 10.0 6.0 10.0;
```

```

        -fx-effect: dropshadow(gaussian, rgba(0.0,0.0,0.0,0.5), 3.0, 0.5, 1.0,
1.0) ;
    }

```

eventline.css

```

/*
 * eventline.css file of FlexGanttFX
 *
 * Copyright 2014 Dirk Lemmermann Software & Consulting
 */

/*
 * The eventline uses the same style as the dateline cells. This style
 * is based on the default modena style of the table column headers.
 */
.eventline {
    -fx-background-color: transparent;
    -fx-border-color:
        derive(-fx-base, 80%)
        linear-gradient(to bottom, derive(-fx-base,80%) 20%,
derive(-fx-base,-10%) 90%)
        derive(-fx-base, 10%)
        linear-gradient(to bottom, derive(-fx-base,80%) 20%,
derive(-fx-base,-10%) 90%),
    /* Outer border: */
        transparent -fx-box-border -fx-box-border transparent;
    -fx-border-insets: 0 1 1 0, 0 0 0 0;
    -fx-border-width: 0.083333em 0 0.083333em 0, 0.083333em 0 0.083333em 0;
    -fx-pref-height: 20px;
}

/*
 * The style used for the label that displays the time at the current
 * mouse cursor location.
 */
.time-cursor {
    -fx-font-size: 0.8em;
    -fx-text-fill: white;
    -fx-background-color: olivedrab;
    -fx-background-insets: 1 0 1 0;
    -fx-background-radius: 8;
    -fx-border-color: derive(olivedrab, -20%);
    -fx-border-radius: 8px;
    -fx-padding: 0 8 0 4;
}

/*
 * The style used for the two labels that display the start and end time
 * of the currently edited activity.

```



```
*/
.marked-time {
  -fx-font-size: 0.8em;
  -fx-text-fill: white;
  -fx-background-color: cornflowerblue;
  -fx-background-insets: 1 0 1 0;
  -fx-background-radius: 8;
  -fx-border-color: derive(cornflowerblue, -20%);
  -fx-border-radius: 8;
  -fx-padding: 0 8 0 8;
}

/*
 * Additional style to modify the appearance of the start time only.
 */
.marked-time-start {
}

/*
 * Additional style to modify the appearance of the end time only.
```

```
*/
.marked-time-end {
}
```

gantt.css

```
/*
 * gantt.css file of FlexGanttFX
 *
 * Copyright 2014 Dirk Lemmermann Software & Consulting
 */

/* Define global colors */
.root {
  -tree-table-row-background-even: white;
  -tree-table-row-background-odd: rgb(245,245,245);
}

/*
 * The split pane padding gets removed so that the tree table and the
 * graphics view both completely fill their sides.
 */
.split-pane {
  -fx-padding: 0.0;
}

.scroll-bar {
  -fx-opacity: .75;
}

/*
 * Row header cells are used in the row header column / the first column
 * of the tree table. The row header is used to display row numbers. The
 * user
 * can also resize rows via a mouse drag inside the cell.
 */
.row-header-cell {
  -fx-text-fill: black;
  -fx-background-color: derive(-fx-box-border,30.0%), linear-gradient(to
right, derive(-fx-base,-3.0%), derive(-fx-base,5.0%) 50.0%,
derive(-fx-base,-3.0%));
  -fx-border-color: transparent -fx-box-border -fx-box-border
-fx-box-border ;
}

/*
 * The tree table header has to have the same height as the timeline.
 * The height can depend on the location of the Gantt chart if it is
 * shown in a multi Gantt chart container.
 */
```

```

.gantt-tree-table-view .column-header,
.gantt-tree-table-view-first .column-header {
    -fx-pref-height: 60px;
}

/*
 * The table header is smaller when the Gantt chart is placed in the
 * middle or bottom of a multi Gantt chart context.
 */
.gantt-tree-table-view-middle .column-header,
.gantt-tree-table-view-last .column-header {
    -fx-pref-height: 24px;
}

/*
 * We do not need the vertical scrollbar of the table. We are styling it
 * away by setting its preferred width to zero. But we only do this if the
 * current display mode is "standard" (table and graphics are both
visible).
 */
.tree-table-view-standard-display-mode > .virtual-flow >
.scroll-bar:vertical,
.tree-table-view-standard-display-mode > .virtual-flow >
.scroll-bar:vertical .decrement-arrow ,
.tree-table-view-standard-display-mode > .virtual-flow >
.scroll-bar:vertical .increment-arrow {
    -fx-pref-width: 0.0;
}

/*
 * We do not need the horizontal scrollbar of the table. We are styling it
 * away by setting its preferred width to zero. We are replacing the
scrollbar
 * with our own scrollbar located hidden inside a HiddenSidesPane instance.
The
 * scrollbar only becomes visible if the user moves the mouse cursor close
to
 * the bottom edge of the table.
 */
.gantt-tree-table-view > .virtual-flow > .scroll-bar:horizontal,
.gantt-tree-table-view > .virtual-flow > .scroll-bar:horizontal
.decrement-arrow ,
.gantt-tree-table-view > .virtual-flow > .scroll-bar:horizontal
.increment-arrow {
    -fx-pref-height: 0.0;
}

/*
 * We like to center the column header text and use a normal font weight
 * for it.
 */
.gantt-tree-table-view .column-header .label {
    -fx-alignment: center;
}

```

```

}

.gantt-tree-table-view .column-header, .gantt-tree-table-view .filler {
    -fx-font-weight: normal;
}

/*
 * Alternating row colors inside the table. To make this work we have to
 * also set styles on tree table row cells. Quite nasty if you ask me.
 */
.tree-table-row-cell:even {
    -fx-background-color: -tree-table-row-background-even;
}

.tree-table-row-cell:odd {
    -fx-background-color: -tree-table-row-background-odd;
}

.gantt-tree-table-view > .virtual-flow > .clipped-container > .sheet >
.tree-table-row-cell:selected {
    -fx-background-color: -fx-selection-bar-non-focused;
}

.gantt-tree-table-view:focused > .virtual-flow > .clipped-container >
.sheet > .tree-table-row-cell:selected {
    -fx-background-color: -fx-selection-bar;
}

/*
 * We are adding depth to the table content and graphics content by placing
 * a shadow below the table header and the timeline. This gives the
impression
 * that the content of both really does slide "behind" these header
controls.
 */
.viewport-shadow {
    -fx-pref-height: 6;
    -fx-background-color: linear-gradient(from 0% 0% to 0% 100%,
    rgba(0,0,0,.2), rgba(0,0,0,0));
}

/*
 * The style used by the buttons inside the layers control (layer up, down,
delete).
 */
.layers-navigate-button {
    -fx-background-insets: 0;
    -fx-border-insets: null;
    -fx-background-color: transparent;
    -fx-padding: 0;
}

/*

```

```

* The column headers inside the layers control.
*/
.layers-table-header {
-fx-padding: 0 0 5 0;
-fx-text-fill: gray;
-fx-font-weight: bold;
-fx-alignment: center;
-fx-border-color: transparent transparent lightgray transparent;
}

/*
* The blank area on top of the graphics view that becomes visible
* for Gantt charts in the middle or last position in a multi Gantt
* chart container context.
*/
.graphic-view-header {
-fx-background-color: -fx-body-color;
-fx-border-color:
    derive(-fx-base, 80%)
    linear-gradient(to bottom, derive(-fx-base,80%) 20%,
derive(-fx-base,-10%) 90%)
    derive(-fx-base, 10%)
    linear-gradient(to bottom, derive(-fx-base,80%) 20%,
derive(-fx-base,-10%) 90%),
    /* Outer border: */
    transparent -fx-box-border -fx-box-border transparent;
-fx-border-insets: 0 1 1 0, 0 0 0 0;
-fx-border-width: 0.083333em 0.083333em 0.083333em 0, 0.083333em
0.083333em 0.083333em;
-fx-pref-height: 0px;
}

/*
* The container "around" the timeline and the graphics area.
*/
.timeline-graphics-wrapper {
-fx-background-color: -fx-box-border, -fx-control-inner-background;
-fx-background-insets: 0, 1;
-fx-padding: 1;
}

.timeline-graphics-wrapper:focused {
-fx-background-color: -fx-faint-focus-color, -fx-focus-color,
-fx-control-inner-background;
-fx-background-insets: -2, -0.3, 1;
}

/*
* The time slider is used to scroll the timeline to the left or right.
* It becomes visible when the user moves the mouse cursor close to the
* bottom edge of the graphics view.
*/
.time-slider {

```

```

    -fx-opacity: .75;
    -fx-background-radius: 0.0;
    -fx-border-color: null;
    -fx-border-radius: 0.0;
}

.time-slider:horizontal {
    -fx-background-color: linear-gradient(to bottom, derive(-fx-base,-3%),
derive(-fx-base,5%) 50%, derive(-fx-base,-3%));
    -fx-pref-height: 16.0;
    -fx-max-height: 16.0;
}

.time-slider > * > .slider {
    -fx-show-tick-marks: false;
}

.time-slider > * > .slider > .track {
    -fx-background-color: transparent;
}

.time-slider > * > .slider > .thumb {
    -fx-pref-width: 100;
    -fx-background-color: -fx-outer-border, -fx-inner-border,
-fx-body-color;
    -fx-background-insets: 2.0, 3.0, 4.0;
    -fx-background-radius: 3.0, 2.0, 1.0;
}

.time-slider > * > .slider:focused > .thumb {
}

.time-slider > * > .adjust-plus {
    -fx-pref-width: 0;
    -fx-shape: null;
}

.time-slider > * > .adjust-minus {
    -fx-pref-width: 0;
    -fx-shape: null;
}

/*
 * The styling of the labels showing the zone ID of a row.
 */
.zoneIdLabel {
    -fx-padding: 4 5 4 5;
    -fx-background-color:
        transparent,
        rgba(0,0,0,0.05),
        linear-gradient(#dcca8a, #c7a740),
        linear-gradient(#f9f2d6 0%, #f4e5bc 20%, #e6c75d 80%, #e2c045
100%),

```

```
    linear-gradient(#f6ebbe, #e6c34d);  
-fx-background-insets: 0,2,3,4,5;  
-fx-background-radius: 4;  
-fx-font-family: "Helvetica";  
-fx-font-size: 10px;  
-fx-text-fill: #311c09;  
-fx-effect: innershadow( three-pass-box , rgba(0,0,0,0.1) , 2, 0.0 , 0
```

```
, 1);  
}
```

graphics.css

```
/*  
 * graphics.css file of FlexGanttFX  
 *  
 * Copyright 2014 Dirk Lemmermann Software & Consulting  
 */  
.root {  
    /* The color used for drawing links between activities */  
    -flexganttfx-link-color: rosybrown;  
}  
  
/*  
 * We need to adjust the list view so it stays in synch with the tree  
 * table view. We also have to remove all padding from the list cells  
 * and assign a default row height that is equal to the default row  
 * height defined in Row.java.  
 */  
.list-view {  
    -fx-padding: 0.0;  
}  
  
.list-view:focused {  
    -fx-padding: 0.0;  
}  
  
.list-cell {  
    -fx-padding: 0.0;  
    -fx-pref-height: 24px;  
}  
  
.list-cell-row-pane {  
    -fx-background-color: transparent;  
}  
  
/*  
 * The single row pane, vbox row pane, and the splitpane row pane all have  
 * to set a background color. The list view version doesn't need to as it  
uses  
 * the colors of the rows.  
 */  
.single-row-pane,  
.vbox-row-pane,  
.splitpane-row-pane {  
    -fx-background-color: white;  
}  
.vbox-row-pane {
```



```

    -fx-border-color: transparent transparent, gray, transparent;
    -fx-border-width: .25px;
}

/*
 * The lasso is used to select multiple activities at once.
 */
.activities-lasso {
    -fx-stroke: red;
    -fx-fill: rgba(255.0,0.0,0.0,0.2);
}

/*
 * The cursor lines.
 */
.horizontal-cursor,
.vertical-cursor {
    -fx-stroke: olivedrab;
    -fx-stroke-width: 1.5;
}

.horizontal-cursor-indicator {
    -fx-background-color: green, white;
    -fx-background-insets: 0, 2;
    -fx-background-radius: 5px;
    -fx-padding: 5px;
    -fx-pref-width: 8;
    -fx-pref-height: 8;
}

/*
 * Marked Time Interval
 */
.marked-time-line {
    -fx-stroke-width: 1.5px;
    -fx-stroke: cornflowerblue;
    -fx-stroke-dash-array: 4 3;
}

.marked-start-time-line {
}

.marked-end-time-line {
}

/*
 * Row controls button are shown when the mouse hovers over a row that can
be
 * edited (flipped around).
 */
.row-controls-button {
    -fx-padding: 5 9 7 7;
    -fx-background-insets: 0 4 2 2;
}

```

```
-fx-background-color: rgba(0,0,0,.5);
-fx-background-radius: 0;
-fx-text-fill: white;
-fx-font-size: 8;
-fx-font-weight: bold;
}

.row-controls-button:hover,
.row-controls-button:focus {
    -fx-padding: 5 9 7 7;
    -fx-background-insets: 0 4 2 2;
    -fx-background-color: rgba(0,0,0,.6);
    -fx-background-radius: 0;
    -fx-text-fill: white;
    -fx-font-size: 8;
    -fx-font-weight: bold;
}

.row-controls-button:pressed,
.row-controls-button:selected {
    -fx-padding: 5 9 7 7;
    -fx-background-insets: 0 4 2 2;
    -fx-background-color: rgba(0,0,0,.7);
    -fx-background-radius: 0;
    -fx-text-fill: white;
    -fx-font-size: 8;
    -fx-font-weight: bold;
}

.virtual-grid-popover > .border {
    -fx-padding: 10px;
}

.grid-button,
.grid-button:hover,
.grid-button:selected,
.grid-button:focus,
.grid-button:pressed {
    -fx-font-weight: bold;
    -fx-font-size: 10px;
    -fx-alignment: center-left;
    -fx-label-padding: 2 0 2 0;
    -fx-background-radius: 2;
    -fx-background-insets: 4;
}

.grid-button {
    -fx-background-color: white;
}

.grid-button:hover {
    -fx-background-color: lightgray;
}
```

```

.grid-button:pressed {
  -fx-background-color: gray;
  -fx-text-fill: white;
}

.grid-button:selected {
  -fx-background-color: black;
  -fx-text-fill: white;
}

/*
 * The styles used for the activity links. A link consists of a path and
two
 * regions (one for the start handle, one for the end handle).
 */
.link {
  -fx-stroke-width: 1.5px;
  -fx-stroke: -flexganttfx-link-color;
}

.link-start-handle {
  -fx-border-color: derive(-flexganttfx-link-color, -20%);
  -fx-background-color: derive(-flexganttfx-link-color, -20%);
  -fx-pref-width: 6px;
  -fx-pref-height: 6px;
  -fx-translate-y: -3px;
  -fx-shape: "M 100, 100 m -75, 0 a 75,75 0 1,0 150,0 a 75,75 0 1,0
-150,0";
  -fx-scale-shape: true;
}

.link-end-handle {
  -fx-background-color: derive(-flexganttfx-link-color, -20%);
  -fx-pref-width: 8px;
  -fx-pref-height: 8px;
  -fx-translate-x: -8px;
  -fx-translate-y: -4px;
  -fx-shape: "M 0 0 L 100 50 L 0 100 L 0 0 Z";
  -fx-scale-shape: true;
}

.link-start-handle-rotated {
  -fx-translate-x: -6px;
}

.link-end-handle-rotated {
  -fx-translate-x: 0px;
}

/*
 * The lens for the graphics area. Experimental. Feature planned for 1.2
release earliest.
 */

```

```
.graphics-lens {  
  -fx-effect: innershadow(gaussian, gray, 10, .1, 0, 0);  
  -fx-border-color: gray;  
  -fx-border-insets: 8px;  
}
```

```
.graphics-lens-popover {  
  -fx-border-color: red;  
}
```

timeline.css

```
/*  
 * timeline.css file of FlexGanttFX  
 *  
 * Copyright 2014 Dirk Lemmermann Software & Consulting  
 */  
  
/*  
 * timeline-first/middle/last are styles that are applied depending on the  
 * position of the Gantt chart in a multi Gantt chart context, e.g. the  
 * DualGanttChartContainer or the MultipleGanttChartContainer.  
 */  
.timeline {  
  -fx-background-color: -fx-body-color;  
  /*  
   * The pref height of the timeline has to match the pref height  
   * of the table column headers, so that they  
   */  
  -fx-pref-height: 60px;  
}  
  
.timeline-first {  
}  
.timeline-middle,  
.timeline-last {  
  /*  
   * The pref height of the timeline is smaller if the timeline  
   * is used for the second, third, ... chart in a multi Gantt chart  
   * context.  
   */  
  -fx-pref-height: 24px;  
}  
  
/*  
 * The lasso used for selecting time intervals.  
 */  
.timeline-lasso {  
  /* semi-transparent rectangle, blue by default (modena.css) */  
  -fx-opacity: 60%;  
  -fx-background-color: -fx-accent;  
}
```

6. Logging

- [Introduction](#)
- [Logging Domains](#)
- [Configuration File](#)

Introduction

FlexGanttFX has some built in logging support using the standard `java.util.logging` framework. Several logging domains are defined in the class `com.flexganttfx.core.LoggingDomain`. The following table lists the available domains.

Logging Domains

Domain	Description
CONFIG	Anything related to the configuration of the Gantt chart control. For example: the renderers that are being registered for different activity types.
DND	Displays everything related to a drag and drop operation (native drag and drop / platform provided drag and drop).
EDITING	Reports changes to the start time, end time, percentage complete, chart value, of an activity.
EVENTS	Informs about activities related to events: registered listeners, events that are being sent.
NAVIGATION	Scrolling, zooming.
PERFORMANCE	Informs about performance related aspects.
RENDERING	Anything relate to rendering rows or activities.
REPOSITORY	Lists repository operations.

Configuration File

The following file can be used to configure logging for **FlexGanttFX**.

```
# To use this property file add the following command line argument:
# -Djava.util.logging.config.file=${project_loc}/log.properties
# Specify the handlers to create in the root logger
# (all loggers are children of the root logger)
# The following creates two handlers
# handlers = java.util.logging.ConsoleHandler,
java.util.logging.FileHandler

handlers = java.util.logging.ConsoleHandler

# Set the default logging level for the root logger
.level = INFO

# Set the default logging level for new ConsoleHandler instances
java.util.logging.ConsoleHandler.level = OFF

# Set the default logging level for new FileHandler instances
# java.util.logging.FileHandler.level = ALL

# Set the default formatter for new ConsoleHandler instances
java.util.logging.ConsoleHandler.formatter =
com.flexganttfx.core.LoggingFormatter

# FlexGanttFX logging domains
com.flexganttfx.config.level = OFF
com.flexganttfx.performance.level = OFF
com.flexganttfx.repository.level = OFF
com.flexganttfx.editing.level = OFF
com.flexganttfx.navigation.level = OFF
com.flexganttfx.rendering.level = OFF
com.flexganttfx.dnd.level = OFF
com.flexganttfx.events.level = OFF
```