



MiG Calendar™ Getting Started Guide

Release 2.0

MiG InfoCom AB
S:t Olofsgatan 28a
753 32 Uppsala
Sweden

www.miginfo.com

COPYRIGHT © MiG InfoCom AB.
All rights reserved.

Java is a trademark registered ® to Sun Microsystems.
<http://java.sun.com>

Table of Contents

Getting Started Guide.....	5
Preface.....	5
Resources and Developer Support.....	5
Contacting Support via Email.....	5
Contacting support via online forums.....	5
MiG Calendar Product Site.....	5
Bug Reports.....	5
Setting up MiG Calendar for usage.....	7
Classpath.....	7
Add License Code.....	7
Setting up a Swing Application.....	7
Examples.....	8
What Approach to Use?.....	8
JavaBeans.....	8
Manual Approach.....	8
Themed Approach.....	9
The Themed Approach.....	9
The ThemeDateAreaContainer.....	10
Visual Date Range.....	10
Theme Properties.....	11
Themes and Updating.....	11
Headers.....	12
Activities.....	13
Customizing how Activities Look and Feel.....	14
Customizing Activities Size and Position.....	15
Grid Structure.....	16
Customizing Grid Lines.....	16
Segments for Nonlinear Time.....	18
Month View.....	19
Categories.....	20

The Manual Approach.....	20
Filter Rows and Categories.....	24
Customizing how Activities Look and Feel.....	26
Provide your own AShape.....	26
Decorators.....	29
The Basics.....	29
For DefaultDateArea and AbstractDateHeaders.....	30
Examples of Decorators Included.....	30
Getting and Setting Properties.....	31
Deploying applications with the component.....	31
Continued Reading.....	32

Getting Started Guide

Preface

This document aims at providing enough information to get started using the MiG Calendar component in your application. The [MiG Calendar Tutorial](#) contains information on how the component is structured including a overview of its different parts.

The [MiG Calendar API JavaDoc](#) will provide details and should be used as a reference. It can be found at the web site indicated below and should also normally be installed adjacent to this document.

Many IDE:s (Integrated Development Environment) of today have good support for inline help using JavaDocs. The standard HTML JavaDocs for the MiG Calendar component is installed by default and can also be obtained from the site as described below. We highly recommend using this feature as it increases productively when creating applications with this component.

Although all developers independent of prior experience can benefit from reading this document, general knowledge of the standard Java API and OOP (Object Oriented Programming) will help understand some of the details and why they are implemented in a certain way.

Resources and Developer Support

MiG InfoCom AB provides support through email and the online forums. Information and updated tutorials will be made available on the MiG Calendar product site

Contacting Support directly via Email

support@miginfocom.com

Submit a support ticket

<http://www.migcalendar.com/support.php>

MiG Calendar Product Site

www.migcalendar.com

Bug Reports

Please submit a support ticket.

Setting up MiG Calendar for usage

Classpath

In order to use the calendar component your application needs to find it. How to do this depends entirely on your environment but normally you add it to your classpath, possibly in the project settings in your IDE. The file to make available is `migcalendar.jar` or `migcalendarbean.jar`.

Note! From v5.5 You should use `migcalendarbean.jar` for *development* and `migcalendar.jar` for *deployment*. They differ only by the fact that the latter has more information in the file, such as all the property editors needed for visual development. `migcalendar.jar` contains the whole component but all the unnecessary information is removed to provide the smallest possible file. `migcalendar.jar` is about half the size of `migcalendarbean.jar`.

Add License Code

Since the product you are using is commercial it will need to be fed a license code before it can be used. This license code should be entered before any usage if the component is commenced or the component will enter evaluation mode.

This is normally done in your application's `main(...)` method or some other place that is run very early in the startup sequence. This is done like this

```
com.miginfocom.util.LicenseValidator.setLicenseKey(File/String/InputStream);
```

Setting up a Swing Application

In order to show the MiG Calendar component you will have to create a normal Swing application. How to do that is outside the scope of this tutorial but it can be done in just a few lines. See the Swing trail in the Java Tutorial at Sun Microsystems site. It is currently located at:

```
http://java.sun.com/docs/books/tutorial/uiswing/index.html
```

The examples below will assume that you have a basic Swing

application up and working so that you have a workplace (a `JPanel` for instance) to add the component to. You can also look at the end of this guide which contains a complete working example.

If everything was done correctly as described above you should now be able to use the component.

There was a demo application, including source code, installed with the component. You can use that as reference material and modify it as you see fit.

Examples

There are examples showing how to use the JavaBeans. They are installed in the installation folder under `examples`. E.g.

```
C:\Program Files\MiG  
InfoCom\MiGCalendar6\examples
```

What Approach to Use?

This is one of the first questions you have to ask yourself. There are three main approaches but from v6.0 the preferred one is the JavaBean approach.

JavaBeans

This is from v6.0 of MiG Calendar the **preferred** approach. It means using the JavaBeans in `com.miginfocom.beans` package (there are currently 15 beans). There is a separate guide how to use these beans installed with this document called "Visual JavaBeans". It contains everything you need to know.

There is also a Flash Movie on www.migcalendar.com that shows shows how the beans work in an IDE.

It should also be noted that even though you don't use an IDE for "visual" programming the beans is probably still the best choice since there is less to learn to get started and they are as powerful as either approached below.

Note that you can use the Manual approach outlined below in almost any case you want even more power than the beans provide.

Manual Approach

If you want to start from scratch and have complete control you will probably want to go the non-themed, or manual, way. This will be the exception though, since the themed version is very flexible yet simpler to use.

You will have to add headers, decorators, activity layouts,

grid properties and a lot of other stuff manually. It is not as daunting as it may sound but a quite thorough read through of the `DefaultDateArea` and `DateAreaContainer` javadocs is a definitely recommended.

All properties that can be set and changed for the `DefaultDateArea` is outside the context of this getting started guide, for more information see the [API JavaDoc](#).

Themed Approach

Note! From v6.0 of the component this approach isn't recommended! Please use the *JavaBeans* approach. The section about themes below is still worth the reading as the base component is still the same and concepts translated to the other approaches.

The advantage of the themed version is that a lot of configuration is taken care of with the default settings for the `CalendarTheme`. The theme can also be changed, preferably before the component is created, in one place and that will migrate to the correct part automatically. For instance you can set the background for a `Header` without knowing anything about how to get a hold of, or even create and inject, the header. The only thing you need is the correct key, and the keys are thoroughly documented, which makes finding it easy.

The downside is that the simplicity to use it hampers the control somewhat. Since the theme classes (e.g. `ThemeDateAreaContainer` and `ThemeDateArea`) creates all headers, grid lines specifications, grid segments etc, you will have a harder time tweaking the creation of them than you might with the manual versions. Normally this is not a problem, the theme has so many different properties that you will probably not even use them all, but there are situations where you might need more control.

It should be noted that it is still possible to change and tweak the themed components, especially if auto update for the `Theme` is turned off, but *what* should be changed and *when* is a bit less clear.

The themed versions of the classes are actually subclasses of the manual ones. They read from the `Theme` and set and reset properties on its base class when creating it, and possibly for every change depending on if the theme is set to *auto update* changes or not.

The Themed Approach

The ThemeDateAreaContainer

To create a component and to show it is really easy. Just create a `ThemeDateAreaContainer` and it will create and use the default `CalendarTheme` settings to create everything. The default settings are somewhat minimalistic though, so it won't be very pretty. The "myContext" theme context is just a key so we know how to get a handle to the created theme later. The constructor will create a `Theme` of the type `CalendarTheme` and register it in the `Themes` singleton class for us, using the provided context as the key.

```
ThemeDateAreaContainer container = new ThemeDateAreaContainer("myContext");
```



Illustration 1 Default Theme settings

Visual Date Range

If you want to specify between which dates the date area should be shown you change the visual range:

```
long startMillis = new GregorianCalendar(...).getTimeInMillis();  
long endMillis = new GregorianCalendar(...).getTimeInMillis();  
  
DateRange visibleRange = new DateRange(startMillis, endMillis, true, null, null);  
container.getDateArea().setVisibleDateRange(visibleRange);
```

Theme Properties

If you want to change the properties of the theme, here is how you do that. It will set a green background. Not pretty, but we know if it works..

```
Theme theme = Themes.getTheme("myContext");  
theme.putValue(CalendarTheme.KEY_GENERIC_BACKGROUND, Color.GREEN);
```

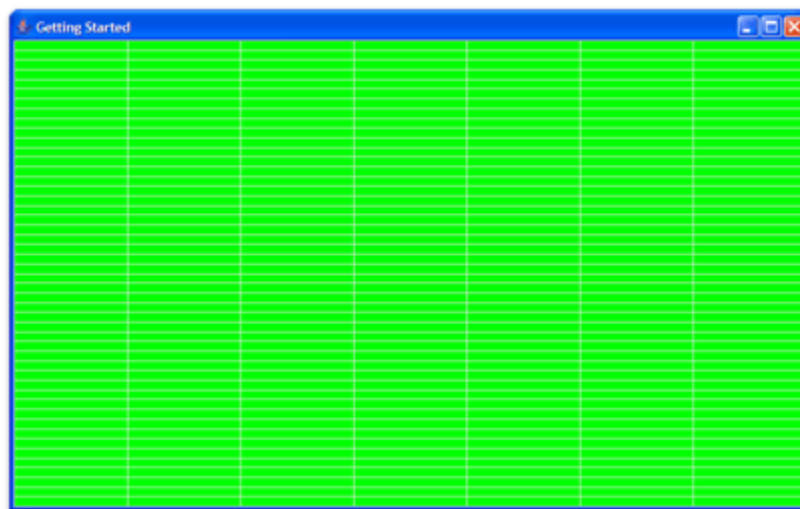


Illustration 2 Horrible green background

Themes and Updating

How come the date area is updated even though we only changed the theme? The default setting for the theme, which is interpreted by the `ThemeDateAreaContainer`, is that changes should be migrated to the container and date area. This is convenient since you don't have to explicitly tell those classes to update themselves. If you have many changes in a batch, you may want turn it of and handle the updating yourself for greater speed.

Depending on what you changed, here are some methods to use for explicitly force an update. They reload/recreate different amounts of information. Read the javadocs for further information.

```
ThemeDateAreaContainer.recreateAll();
```

```
ThemeDateArea.reloadStartupStructureFromTheme();
ThemeDateArea.reloadFromTheme();
ThemeDateArea.recreateAll();
```

Headers

There are no headers in the default theme, so let's add one. Since we normally want to set the headers before the container is created, so it doesn't have to be recreated directly, we change the code a bit to create the theme beforehand.

```
Theme theme = new CalendarTheme("myContext");

CellDecorationRow headerRow = new CellDecorationRow(
    DateRange.RANGE_TYPE_DAY,
    new DateFormatList("E' 'dd'/'M"),
    new AtFixed(20f),
    new Font("SansSerif", Font.PLAIN, 12)
);

GridLineRepetition gridLines = new GridLineRepetition(1, new Color(220, 220, 220));

String mainKey = CalendarTheme.KEY_HEADER_;
theme.addToList(mainKey + "North/CellDecorationRows#", headerRow);
theme.addToList(mainKey + "North/GridLines/PrimaryDim#", gridLines);

container = new ThemeDateAreaContainer("myContext");
```



Illustration 3 Header at top

The code above are using the simplest of the constructors to

create the header row. A lot more can be specified including mouse over paints, row count, size and label positioning within the header cell. The keys used are explained in the javadocs for `CalendarTheme` and since they are list keys (ends with a #) the theme's `addToList(..)` method should be used.

To make the header it a bit snazzier let's use a `ShapeGradientPaint` object provided with this component. Exchange the `headerRow` object above with:

```
ShapeGradientPaint headerBackground = new ShapeGradientPaint(
    new Color(235, 235, 235),
    new Color(255, 255, 255),
    90, 0.7f, 0.6f, false
);

CellDecorationRow headerRow = new CellDecorationRow(
    DateRange.RANGE_TYPE_DAY,
    new DateFormatList("E' 'dd'/'M"),
    new AtFixed(20f),
    AbsRect.FILL,
    headerBackground,
    Color.DARK_GRAY,
    new DefaultRepetition(),
    new Font("SansSerif", Font.PLAIN, 12)
);
```



Illustration 4 Header with snazzy look

Activities

OK, now we have a date area that shows a grid of a week with a header that shows the week days and dates. Next we'll show how to add some activities (E.g. an Event or 'Todo') to the date area. First we should create an `Activity` and add it to the `ActivityDepository`.

```
ImmutableDateRange actRange = new ImmutableDateRange(
    System.currentTimeMillis(), DateRange.RANGE_TYPE_HOUR, 4, null, null
);

Activity activity = new DefaultActivity(actRange, new Integer(1234));
activity.setSummary("Hello, World!");

ActivityDepository.getInstance().addBrokedActivity(activity, null);
```

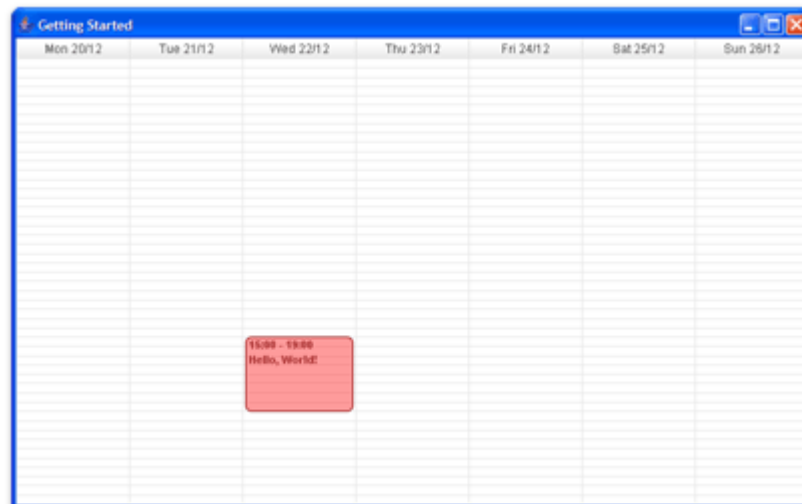


Illustration 5 Added an activity

Important!

Note that activity support is turned **off** by default; This means that you will have to turn it on in order to see the activities. You do that with:

```
container.getDateArea().setActivitiesSupported(true);
```

Customizing how Activities Look and Feel

How the activities look like can't be changed with the theme.

They are drawn with a `Decorator` like everything else but only the layer index in which the activities are drawn can be changed with the `CalendarTheme`. The size and position **can** be changed with the theme since those are properties of the `ActivityLayout` system and not how it looks.

See *Customizing how activities look and feel* under the manual section below for an explanation on how to change the look of an `ActivityView`.

Customizing Activities Size and Position

There are currently three different `ActivityLayouts` delivered with the component; `FlexGridLayout`, `HideLayout` and `TimeBoundsLayout`. They can all be customized to great extent using the theme.

Under the `CalendarTheme.KEY_LAYOUTS_AUTO_INSTALL` list key you can add your own layouts if needed. With the *Theme Editor* you can configure and reorder the built in layouts. See the [Theme Editor Tutorial](#) for more information on how to do this.

To set a layout with a little more space around the activity than default add these rows just before you create the container:

```
TimeBoundsLayout layout = new TimeBoundsLayout(  
    new AtFixed(15), new AtStart(15), new AtEnd(-15), 15);  
  
theme.removeAllFromList(CalendarTheme.KEY_LAYOUTS_AUTO_INSTALL);  
theme.addToList(CalendarTheme.KEY_LAYOUTS_AUTO_INSTALL, layout);
```

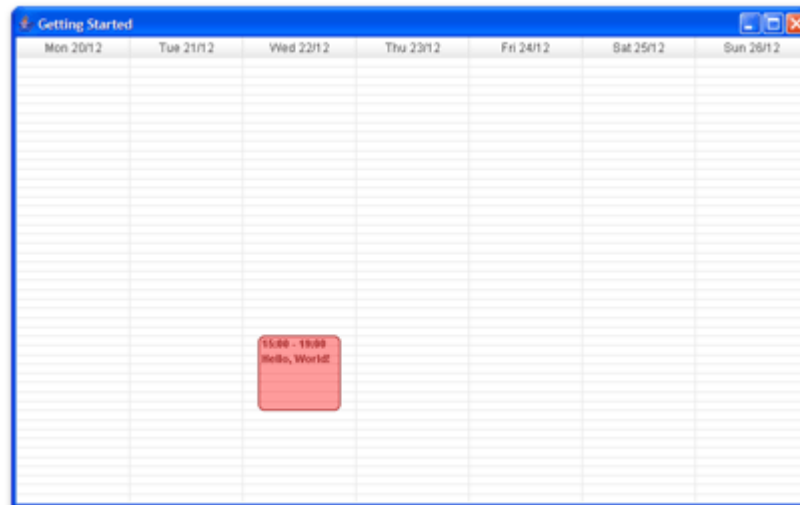


Illustration 6 Reduced layout bounds

We removed all other layouts first so we know that this is the one that will be used for all `ActivityViews`.

Tip!

You can have multiple layouts active at the same time and the `ActivityLayoutBroker` set in the `DateArea` will chose which `ActivityLayout` that will layout which `ActivityView`. You can even install you own `ActivityLayoutBroker` under the key `CalendarTheme.KEY_LAYOUTS_BROKER`. The default value of `null` will use the `DefaultDateArea` as the broker.

More on activity layouts in the [MiG Calendar Tutorial](#).

Grid Structure

The `DateArea` uses a `DateGrid` to map points in time to screen coordinates and also to manage the underlying table-like structure. The grid is recreated by the `DateArea` when the structure changes. Increasing the visible range or row size specifications would for instance recreate the grid.

How much time one row/column spans and in which direction time primarily flows are two of the key properties of the grid. Grid line and grid row sizes and paints are also maintained by the `DateArea` and is changeable from the `CalendarTheme`.

There is a *startup* part in the `CalendarTheme`; it contains information on how the grid should be structured when the `DateArea` is first created. It can later be changed by calling methods on the `DateArea`.

Customizing Grid Lines

Let's say that we want odd grid lines to be a bit brighter since they only show 30 min boundaries.

Grid lines are repetition specifications evaluated in the order they exist in the property list stored under the key:

```
String primGLKey = CalendarTheme.KEY_GRID_GRIDLINES_ + "PrimaryDim#";
```

Default there is a `GridLineRepetition` that says: *'start at grid line 0 and accept every grid line, setting color to gray and width to 1'*.

We can leave that repetition be and insert a new repetition that goes something like: *'start at index 1 and accept every second grid line, setting color to light gray and width to 1'*. Since this repetition is before the default one it is evaluated first. It will only specify a 'hit' on odd grid lines though, making them light gray. All other grid lines will be caught by the second, default, rule making all other grid lines darker gray.

You can easily insert repetitions that just changes a single grid line, for instance the first or last or around lunch or whatever. You can even reference them from the last grid line making them very flexible to work with even if you don't know beforehand how many you have. You could even say that the middle 15% of the grid lines should be five pixels wide and painted yellow..

Here are some code that make odd grid lines lighter a lighter gray:

```
String glKey = CalendarTheme.KEY_GRID_GRIDLINES_ + "PrimaryDim#";
GridLineRepetition glRep = new GridLineRepetition(1, 2, 1, new Color(245, 245, 245));
theme.addToList(glKey, 0, glRep); // add at index 0 to make first key in the list
```

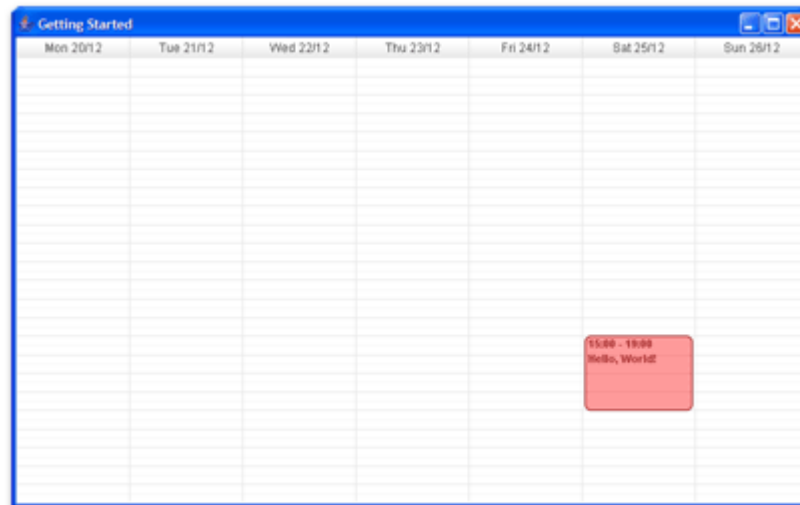


Illustration 7 Even/odd grid lines

Segments for Nonlinear Time

Normally time between, say, 08.00 and 18.00 (06.00 pm) is the most interesting when it comes to planning, especially for work. The most common solution to this is to show this time and let the user scroll to times before and after this more interesting time period. This has a number of problems, the most obvious is that you have to scroll both up and down to see if there is anything planned early morning or late night.

MiG Calendar has a unique and elegant solution to this problem. It supports nonlinear time, per row or for a group of rows, in both primary and/or secondary dimension. So what does it mean? It means that you can, for instance, make the rows denoting 00.00 – 08.00 take up less space per minute than say 08.00-18.00 and then 18.00 – 24.00 can take up less space per minute again.

You do this by dividing rows (wither in the primarily or secondary dimension) into `GridSegments` and giving those segments sizes, either absolute or relative. The `GridSegments` can have min/preferred/max sizes set on them ensuring a very flexible layout independent of the container size. You can for instance specify that a `GridSegment` should: *'prefer a size of 10% of available bounds, but be no less than 2 pixels and no bigger than 10 pixels'*.

Here is a some code that makes grid rows 0 to 15 (00.00 – 08.00) occupy 10% of available bounds but no less than 1 pixel and no more than 4 pixels.

```

AtFixed min = new AtFixed(1);
AtFraction preferred = new AtFraction(0.1f); // Preferred can be absolute or relative
AtFixed max = new AtFixed(4);

GridSegment segment = new GridSegment(16, min, preferred, max);

theme.addToList(CalendarTheme.KEY_GRID_SEGMENTS_ + "PrimaryDim#", 0, segment);

```

As the `GridSegments` are also in a list, and there is a default one at index 0 we add this new one before and as such defines the first segment to be 16 rows with the provided size constraints.

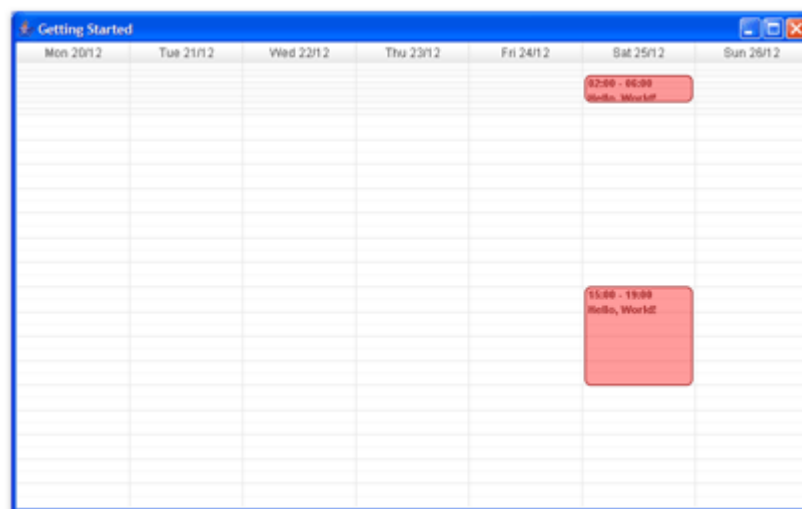


Illustration 8 Non-linear time

Note that even though the activities have equal lengths time wise the one in the compressed time is shorter.

Reducing the number of grid lines, or rather setting their size to zero, in the compressed time as well as other improvements are left as an exercise to the reader. Here is a screen shot on how it can look if you make some more adjustments. The theme shown here are one of the example themes delivered with the component, so they are free, and encouraged, to dissect and experiment on.

Month View

If we wanted to show five weeks in a month like view, we would do something like the following. Note that you will at least have to remove the segment code from above or the grid will be very small.

```

final int dayType = DateRangeI.RANGE_TYPE_DAY;
final int weekType = DateRangeI.RANGE_TYPE_WEEK;
final int primDir = SwingConstants.HORIZONTAL;

theme.putValue(CalendarTheme.KEY_STARTUP_PRIMARY_DIM_CELL_TYPE, dayType);
theme.putValue(CalendarTheme.KEY_STARTUP_PRIMARY_DIM_DIRECTION, primDir);
theme.putValue(CalendarTheme.KEY_STARTUP_PRIMARY_DIM_TYPE_COUNT, 1);
theme.putValue(CalendarTheme.KEY_STARTUP_SECONDARY_DIM_WRAP_BOUNDARY, weekType);
theme.putValue(CalendarTheme.KEY_STARTUP_VISUAL_RANGE_TYPE, weekType);
theme.putValue(CalendarTheme.KEY_STARTUP_VISUAL_RANGE_COUNT, 5);

```

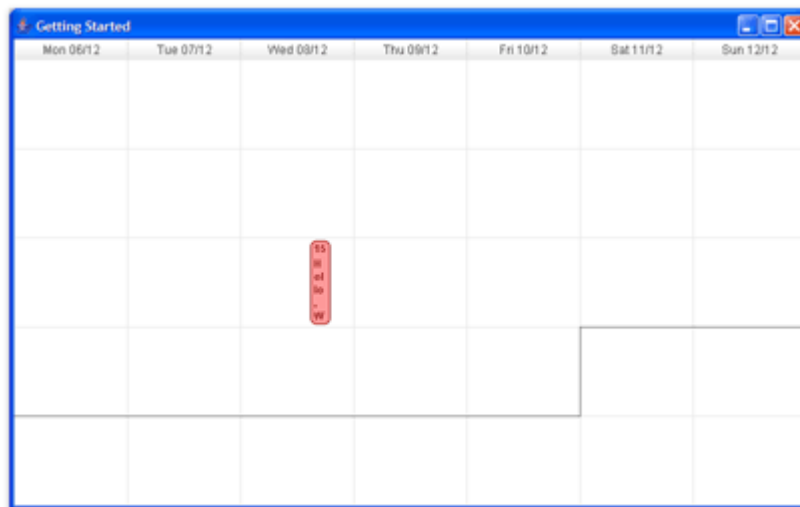


Illustration 9 Structure changed to show five weeks in month view

You should now have enough information to go and test some of the other features of the MiG Calendar. All keys are documented in the [Technical Documentation](#) and the javadocs for `CalendarTheme`.

Categories

The Themed approach doesn't support Categories. Category rows and a category header must be added though the manual approach below. It is still possible to use the themes approach and the in code add the category header.

The Manual Approach

Even though almost all types of applications can be created using the themed approach described above, you sometimes need more control and less automation. Doing it manually is not hard, but it will be more verbose since you have to create

and add all supporting objects, such as `Decorators`, `ActivityLayouts` and date/time rounders for activities. There are also a whole slew of properties to change if needed, something that can be cumbersome doing manually.

Everything that exists in the `CalendarTheme` can also be set manually. Some objects that should be created and added have their properties spread out over multiple properties. You will also have to know where that object should be injected, for instance `Headers` should be added to the `DateAreaContainer` but `ActivityLayouts` should be added to `DefaultDateArea`.

You will have to read the [MiG Calendar Tutorial](#) to get to know what objects to use and where to put them.

To create a non-themed version of the component here's how:

```
DefaultDateArea dateArea = new DefaultDateArea();
DateAreaContainer container = new DateAreaContainer(dateArea);
```

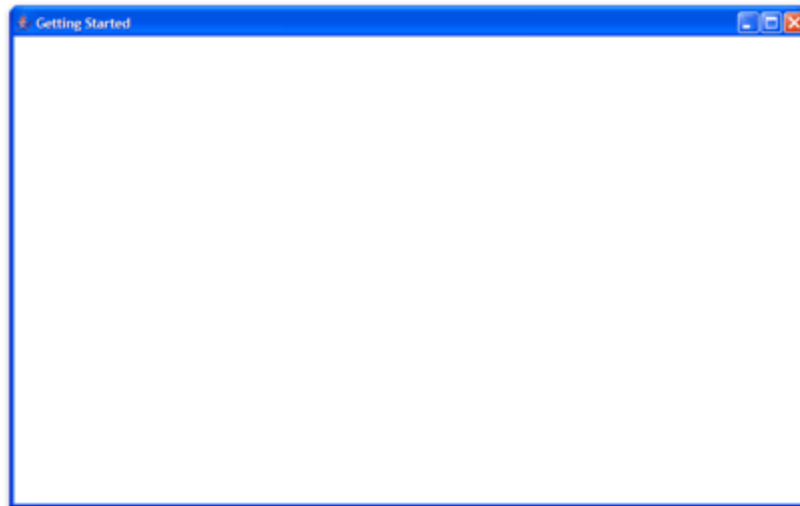


Illustration 10 Empty DateAreaContainer

We only see a white background. That is because we haven't installed (added) any `Decorators` yet (or rather `GridDecorators`, which extends the interface `Decorator` with the ability to get the `GridContainer`).

Lets add grid lines first:

```
GridLineDecorator glDecorator = new GridLineDecorator(dateArea, 60);
dateArea.addDecorator(glDecorator);
```

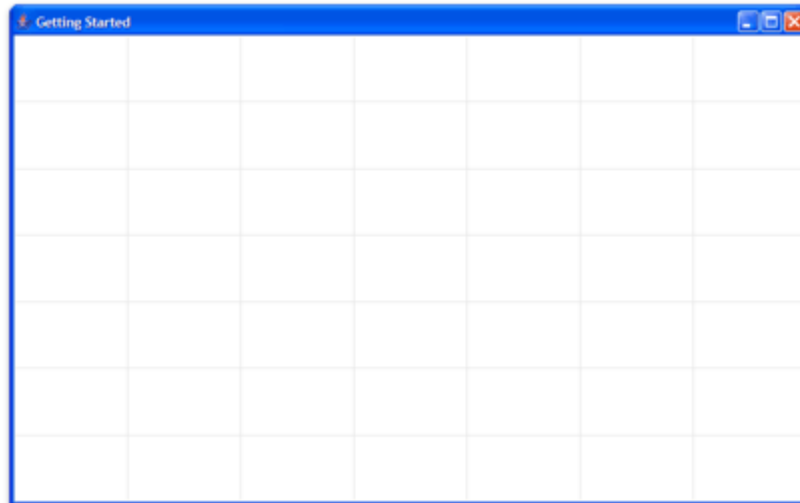


Illustration 11 With GridLineDecorator

If you run it now the grid lines will show. 60 is the relative layer in which this decorator will be painted. Lower layer will be painted first and thus be overwritten by higher layers.

Notice that the default visible count and type is different from the one provided by the default `CalendarTheme`. Here the default is to show six weeks in a month-like view.

We didn't actually tell the decorator how the grid lines should look like, `GridLineDecorator` will get that information from the `DateGrid` object for every repaint.

Another important decorator is the `ActivityViewDecorator`. It is a special one since it's actually an inner class of `DefaultDateArea`. It is so simple because it has no meaning outside of that class and it needs access to its members. It is not added by default (none area) so we have to add it. The syntax for creating an inner class may seem a bit strange. Here is how you add it:

```
AtFixed forcedSize = new AtFixed(16);
TimeBoundsLayout layout = new TimeBoundsLayout(
    new AtFixed(2), new AtStart(2), new AtEnd(-2), 2,
```

```

        forcedSize, forcedSize, forcedSize
    );
dateArea.addActivityLayout(layout);
dateArea.addDecorator(dateArea.new ActivityViewDecorator(70));

```

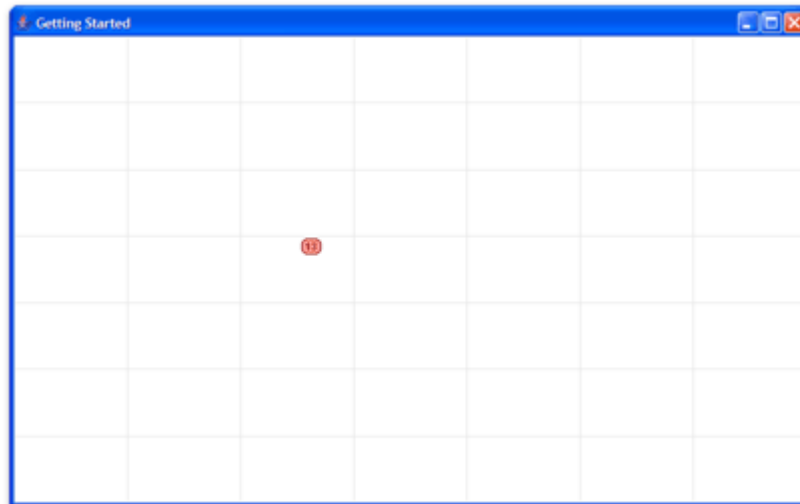


Illustration 12 With an ActivityDecorator

As you see we also had to add an `ActivityLayout`. Otherwise the `ActivityView` would not get laid out in the grid, and thus be invisible. We also set the *min*, *preferred* and *max* size to 16 pixels. The size is referring to the secondary dimension (vertical here) since the start and size in the primary dimension is depending on the date range of the `Activity`.

What if we want the activity to horizontally (the primary dimension) be rounded to a full day, so that it would be more like a list in every day cell? We just *visually* round it, or rather set a `DateRangeRounder` in the `TimeBoundsLayout`.

```

BoundaryRounder dayRounder = new BoundaryRounder(DateRangeI.RANGE_TYPE_DAY);
layout.setVisualDateRangeRounder(dayRounder);

```



Illustration 13 Visually rounded to day

Filter Rows and Categories

Every column/row in the grid can be divided into sub rows. These sub rows can be divided further into sub rows so it is basically hierarchical. These sub rows are usually filtered in some way to only show a certain kind of activities so they appear separate from each other. This is for instance how the TV-schedule is done in the demo application.

You can write your own filter code to filter on whatever, but normally Categories are used as filters. Since categories are also hierarchical it is quite easy to map a category tree, or part there of, to sub rows.

The simplest way to do this is to set `CategoryFilters` on the `DefaultDateArea`. The creator class that creates the `DateGrid` will pick this up and create sub rows automatically. You can also exchange the date grid creator itself to create the rows as you want but that is an advanced approach.

Below is a code snippet that shows how to set up three sub rows rows in every main row in the grid.

```
// (id, name, parentID)
Category mark = new Category(new Integer(0), "Mark", null);
Category susan = new Category(new Integer(1), "Susan", null);
Category michael = new Category(new Integer(2), "Michael", null);

// (category, includeSubCategories, acceptUncategorized)
CategoryFilter[] peopleFilter = new CategoryFilter[] {
```

```

    new CategoryFilter(mark, false, false),
    new CategoryFilter(susan, false, false),
    new CategoryFilter(michael, false, false)
};

defaultDateArea.setRowFilters(peopleFilter);
dateAreaContainer.revalidate();

```

The date area does not by default show a category header to show the names of the filtered rows from above, one have to add a `SubRowGridHeader` for that. This header has much in common with the date headers, only it shows a sub row property (such as the name) rather than the date. They are very flexible in that a category header can contain multiple rows with cells that can be merged in very flexible ways. The simplest way to experiment with this is to use the `North/WestCategoryHeader` JavaBeans with a visual IDE such as `JFormDesigner` or `netBeans`.

Here is some code to set up an example category header. It is taken from the demo application and can be viewed in a context there.

```

DateArea dateArea = mainCalendarContainer.getDateArea();

int size = 15;
SubRowGridHeader header = new SubRowGridHeader(dateArea, 1, new Color(220, 220, 220),
                                                size, 1, SwingConstants.TOP);
header.setBackgroundPaint(new ShapeGradientPaint(new Color(230, 230, 230),
                                                  new Color(250, 250, 250), 90, 1, 0.5f, false));

DefaultSubRowLevel row = new DefaultSubRowLevel("$gridRowName$",
                                                new AtFixed(size),
                                                AbsRect.FILL,
                                                null,
                                                Color.DARK_GRAY,
                                                null,
                                                999,
                                                new Font("sansserif", Font.PLAIN, 11),
                                                null,
                                                AtFraction.CENTER,
                                                AtFraction.CENTER,
                                                TextAShape.TYPE_SINGE_LINE,
                                                DefaultSubRowLevel.APPLY_TO_ALL
);
row.setTextAntiAlias(GfxUtil.AA_HINT_ON);

header.addDecorator(new SubRowHeaderDecorator(header, row, 100, true));
header.addDecorator(new GridLineDecorator(header, 110));

mainCalendarContainer.setHeader(header, DateAreaContainer.NORTH, 1);

```

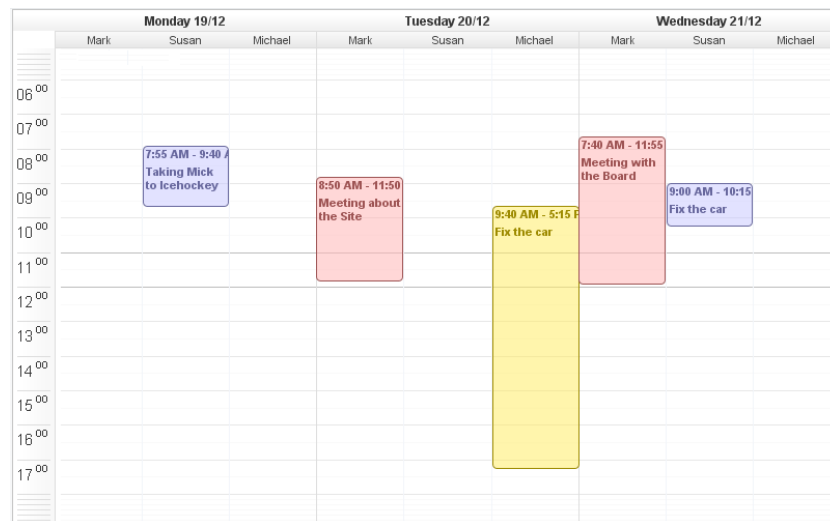


Illustration 14: Date area with category filter rows and a Category Header

Customizing how Activities Look and Feel

The default installed `Decorator` that paints the `ActivityViews` uses the in the `DateArea` specified `ActivityViewRenderer` (which is an interface). That renderer can easily be exchanged for something completely different, but it is a `ShapeRenderer` by default.

You basically have two choices if you want to customize how the activities are painted:

1. Provide a custom `AShape` that looks like you want. This is the preferred choice for almost any use case. `AShapes` have build in support for mouse and key event handling and `DefaultDateArea` will support that. How to get the `ShapeRenderer` to use your new `AShape` will be shown later.
2. Write your own `ActivityViewRenderer`. This is very simple to do, but it can be hard to enable user interaction with the rendered views since it has to be done manually.

Provide your own `AShape`

Creating an `AShape` that works and looks good is easy. Creating one that look equally good in **all** sizes and aspect ratios is a bit harder since it usually includes setting conditional visibility on some parts and size/position

constrains on other parts. For a full primer on AShapes set the [AShape Tutorial](#) and the [MiG Calendar Technical Specification](#) documents.

```
Rectangle shape = new Rectangle(0, 0, 1, 1);  
//Ellipse2D shape = new Ellipse2D.Float(0, 0, 1, 1); // Alternative.  
  
FillAShape box = new FillAShape("fill", shape, AbsRect.FILL, Color.BLUE, null);  
RootAShape root = new RootAShape(box);
```

The `root` object is the shape to use. AShapes are stored in a hierarchy with a `RootAShape` at the top. The root shape can be viewed as the handle for the whole shape and must exist at the very top of the hierarchy tree.

Notice that the `FillShape` itself doesn't impose what geometry it should fill, that is provided in the constructor and can be any Java2D Shape, including `GeneralPath` and our own version with better, constrainable, coordinates types; `PolygonShape`.

As always, to see exactly what the arguments mean and what the objects can do see the [MiG Calendar Technical Specification](#).

Now we need to tell our component how to use our newly created AShape. As it turns out, this can be done in more than one way.

1. Exchange the `Decorator` in `DefaultDateArea` for your own. The most flexible solution, but much of the boiler plate work that is handled automatically by `DefaultDateArea` has to be remade by you.
2. Exchange `ActivityViewRenderer` in the `DefaultDateArea` with a proprietary one that uses the new shape when painting. Also very flexible but much easier to implement since you will be provided with the `ActivityViews` to paint and the total bounds. There is nothing 'AShape' about this approach though, you can use pure Java2D if you like.
3. Get the `DefaultAShapeProvider` from the the `ActivityViewRenderer`, which we know is a `AShapeRenderer`, and instruct it to use a some other `RootAShape` for some `paintContext`. `DefaultAShapeProvider` uses the `paintContext` of the `Activity` to look up a `RootAShape` to return. There are both a map for the actual renderer and a fall back

global map. Look at the JavaDoc for `DefaultAShapeProvider` for how to do this. Then just set the `paintContext` on the activities to the same context as you mapped in in the renderer and everything will be handled automatically. You can even use the static method `DefaultAShapeProvider.setShapeGlobally(..)` to set it, which is the simplest solution. The `null` context in the global map will give you the global fall back shape and the only installed by default. That one can be exchanged to make the mapping for the whole application.

There are quite a few different ways to inject your own `RootAShape` that should be used for painting those activities. Which way you choose depends on how late in the decision process you want to inject it. Earlier gives you more choice on the actual implementation and later gives you more help from the framework, and you have to use a `RootAShape`.

Here is the outline of the process of painting the `ActivityViews`, as short as possible:

Explanation The `GridDecorator` that is installed in the `DefaultDateArea` to paint the activities is a `ActivityViewDecorator`. It is using a `AShapeRenderer` to do the actual painting. That renderer gets the `RootAShape` to use from its `DefaultAShapeProvider`. That factory always has a default `RootAShape` to return (from the global map with `paintContext null`), and you can even change that one. On the actual `DefaultAShapeProvider` you can change the mapping between a `paintContext` and a `RootAShape`, you can also change it globally with a static method.

At last, some code. First #2:

```
DefaultDateArea dateArea = (DefaultDateArea) container.getDateArea();
AShapeRenderer r = (AShapeRenderer) dateArea.getActivityViewRenderer();
r.getShapeProvider().setShape(root, null);
container.getDateArea().recreateActivityViews(); // The views caches the shapes..

or

DefaultAShapeProvider.setShapeGlobally(root, null); // Changes for ALL
```

This will add the `root` shape we created a bit up. It will be a blue rectangle filling the exact bounds of the date range it represents. It can not be interacted with since we haven't set

any interactions on it, see the [AShape Tutorial](#) for information on how to do this.

If all we want to do is fill blue rectangles there is a faster way, the #2 in the list above. It might be used to paint a great number of read only activities with extreme speed.

```
// Setting a new renderer as defined below.
container.getDateArea().setActivityViewRenderer(new BlueRectangleRenderer());

// add outside a method declaration in your class
static class BlueRectangleRenderer implements ActivityViewRenderer
{
    private final Insets REPAINT_MARGIN = new Insets(1, 1, 1, 1);

    public void paint(Graphics2D g2, Rectangle bounds, TimeSpanList actViewList)
    {
        Paint oldPaint = g2.getPaint();
        g2.setColor(Color.BLUE);
        Rectangle clip = g2.getClipBounds();

        for (int ix = 0, ixSz = actViewList.size(); ix < ixSz; ix++) {
            ActivityView actView = (ActivityView) actViewList.get(ix);

            Rectangle[] actBnds = actView.getBounds();
            if (actBnds != null) {
                for (int i = 0; i < actBnds.length; i++) {
                    Rectangle actBnd = actBnds[i];
                    if (actBnd != null && clip.intersects(actBnd)) {
                        g2.fill(actBnd); // The actual paint code! Exchange for something better...
                    }
                }
            }
        }
        g2.setPaint(oldPaint);
    }

    public Insets getRepaintMargin()
    {
        return REPAINT_MARGIN;
    }
}
```

You now have the basics for changing the AShapes for you own, and even for implementing you own proprietary paint algorithm.

In the demo source that was installed with the component you can look at AShapeCreator source code. It contains some example AShapes.

Decorators

The Basics

You have already used a couple of decorators above, but

here area a more thorough explanation.

A `Decorator` is a generic interface that encapsulates how to paint something. A `GridDecorator` is a sub interface and used to decorate things relative to a `Grid`. They have an `layerIndex` that denotes how to order them is a repaint cycle. Lower indexed decorators will be painted before higher indexed ones, making higher indexed decorators appear on top.

They can also interact with the user since the container of them should offer all `InputEvents` (such as `MouseEvent`s and `KeyEvent`s) to the decorators. Event notification stops if a decorator consumes the event. If no decorator consumes it it would normally mean that the container if the decorators should process it. The code that processes the `InputEvents` in the decorators should as always be **very** fast to not introduce delays in the GUI.

For `DefaultDateArea` and `AbstractDateHeaders`

Currently decorators are use in these two classes and their sub classes. `DecoratorSupport` is a class that handles them and can be utilized should you choose to use them for other purposes.

Examples of Decorators Included

The `Decorator` hierarchy is one of the more extensive hierarchies in the MiG Calendar component. This is done to minimize code duplication. For a UML diagram of this structure see the [MiG Calendar Technical Specification](#).

`CellLabelDecorator` – Draws labels in the cells of the `Grid`.

`DateSeparatorDecorator` – Draws separator lines between date boundaries in the grid. For instance dividing months in a month view.

`EvenFieldFillDecorator` – To fill different backgrounds for even/odd date ranges. For instance every other month can have a yellowish background.

`GridLineDecorator` – Draws the grid lines in a `Grid`.

`HeaderShapeGridDecorator` – Draws the labels for headers. Very flexible and can merge cells. Also manages mouse over and press interactions.

`ImageDecorator` – A generic decorator to place images somewhere on the `Grid`.

`NoFitShapeDecorator` – Paints a custom aligned `AShape` in cells that had `ActivityViews` that couldn't fit.

OccupiedDecorator – Merges the date ranges for the `ActivityViews` and draws/fills the background, or some of it, for when there is at least one of them.

There are currently about 20 decorators, though some are abstract. It is also very easy to write your own since getting the positions for dates/cells is trivial courtesy of the `DateGrid` object.

Getting and Setting Properties

Generally the MiG Calendar component follows the standard get/set pattern. Since `DateArea`, or probably more often the `DefaultDateArea`, is normally contained by a `DefaultDateContainer` its properties will not be shown in a RAD (Rapid Application Development) IDE. You will have to manually get the `DateArea` with a call like this:

```
DateArea dateArea = container.getDateArea();
```

The `dateArea` object is the one that contains most properties to change. See the [MiG Calendar Technical Specification](#) for a list of the properties. Since the `DateAreaContainer` is almost always used as the container to host a `DateArea` and `Headers` it is like it is the object you will need a reference to to get a hold of other important objects.

The `DateArea` also contains another important object, the `DateGrid`. It contains the information on how to convert between cells, dates/times and pixel positions. It contains a lot of convenience methods for this.

Deploying applications with the component

The only file you need is `migcalendar.jar`. You might have to include whatever themes you have made and possible `AShape` XML files as well, but that is completely depending on your particular setup.

It is important that you read the license agreement that is included with this component as it contains information of what you as a customer can and cannot do.

Continued Reading

This [Getting Started](#) guide has given you the basics for experimenting on your own. First stop should probably be the Theme Editor since that is a utility that can change almost any of the component's properties and the results are shown immediately.

If you have requirements that can not be met by the properties in the theme you must resort to writing code in order to customize it further. Almost all aspects of the component can be exchanged and/or overridden to extend just about everything. This is by design. To be able to do this you will need to have a thorough understanding of how the different parts fits together. [The MiG Calendar Tutorial](#) and the [API JavaDocs](#) is a must read for doing this. Also, the support forums at <http://www.miginfo.com/forum/> can be used to ask questions.

When you have gotten acquainted with the component you are welcome to make feature requests. The MiG Calendar component consists of the *Calendar* part, the *Theme Editor* and the *AShape* framework. Suggestions for any one of them are appreciated and those should also be posted in the forums, linked above, so that others can see and comment on them.