MiG InfoCom AB

MiG Themes™ Developer's Guide

Release 1.0

Mikael Grev & Ellen Bergdahl

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

www.miginfocom.com

Document Revision History

Version	*Date*	*Comments*
0.9	December 2004	Preview Release
1.0	January 2005	Initial Release

Table of Contents

MiG Themes Developer's Guide

Preface

This document aims at providing information about how to develop applications that uses the MiG Themes component.

The **MiG Themes Technical Specification** 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. Currently the UML JavaDoc is used a the technical specification. It will be converted to a PDF at a later time.

Many IDE:s (Integrated Development Environment) of today have good support for inline help using *javadocs*. The standard HTML javadocs for the MiG Themes component are 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 Themes product site

Contacting Support via Email

`support@miginfocom.com`

Contacting support via online forums

`www.miginfocom.com/forum/`

MiG Theme's Product Site (soon)

`www.miginfocom.com/migthemes/`

Bug Reports

Via Email or forums as indicated above.

How Themes work

NOTE! As of v6.0 of the MiG Calendar Component it is recommended to use the JavaBean approach.

Overview

A `Theme` in this component is a collection of properties that may be hierarchical and the values have to conform to certain capabilities. The properties can be linked to each other to make several properties have a single edit point. A special kind of property may contain a list of values, all with the same capabilities. The list will maintain the insertion order.

The key is a string that looks much like a path to a file. E.g. `"Look/Background/color"`. The keys are normally defined as static strings, but they can also be dynamically created.

The value of a property is of type `Object`, but it is type checked at runtime through one of more `PropertyCapability` objects assigned to the property or hierarchy of properties.

`Theme` objects are typically stored in the `Themes` object which has `static` accessor methods. You provide a `context` string which identifies the theme to get. There can be different `Theme`s of the same type and/or of different types stored in the `Themes` object.

You typically subclass `Theme` to define your own theme. This is a simple process and you basically only supply the information needed to define the keys, the rest is handled by the abstract `Theme` class.

All `Theme`s are editable with the GUI Theme Editor application. The Theme Editor can also be seamlessly incorporated into you own applications since it is contained in a normal `JPanel`. If you have a custom value object an `XxxPropertyEditor` can very easily be written to enable visual editing of that value. All boiler plate code is implemented in `AbstractPropertyEditor` (which extends `PropertyEditorSupport`, the JavaBeans standard editor type), you only have to provide the components to show and edit the value.

Theme Context

The theme context is a `String` that identifies a particular `Theme` instance. It is used as a token, or key, to get the theme from the `Themes.getInstance(context)` static method. This is so that the theme reference don't have to be passed around all the time and that the actual implementation can be changed at any time.

Capabilities

A property in the `Theme` can have any number of `PropertyCapability` objects assigned to it. These capabilities outlines which type of object that property can hold, including if it can be `null`. The capability specifies the class type of the value and if it is a `Comparable` also a valid range.

Every store of properties, including loading of them, checks that the capabilities for a property isn't broken. This ensures that the `Theme` is always legal in terms of the types of value for the properties.

The legal `PropertyEditor`s to use for visually editing a property value in a `Theme` will be selected on the property capabilities that are registered for that key. If there are many different property capabilities, which is quite normal, all of them will be selectable to edit the value. This is a lot more flexible than the normal JavaBeans `PropertySupport` classes, since they only can be coupled to one specific class type.

Keys

The keys are really just simple strings and they need not be any special at all to work. However the Theme Editor and the `Theme` class will recognize a hierarchy among the keys if formatted in a special way. Organizing the keys into hierarchies will enable them to

- be shows visually in a tree structure in the Theme Editor. This makes browsing them much simpler.
- have capabilities set on a whole sub tree. Saves resources and simplifies changes.
- create a sub tree from one or more array of keys, making it possible to set up large structurally repeating trees with ease.
- give context to keys. The key "`Header/North/color`" have better and more definable structure than "`northHeaderColor`" would have.

Also see under *Making Your Own Themes* below for further information about the key hierarchy and folder keys.

List Keys

List keys are a special type of key that allow for an ordered list of key, which still comply with the given capabilities. If the key wouldn't be treated in a special way the capabilities would only be able to specify that it should be of type `java.util.List`, now the list elements will be validated instead.

List keys ends with a hash (#) to differentiate them from normal keys. Special methods exists in the base class `Theme` to handle them. Example key: "`Grid/rowHeight#`".

Linked Keys

A key can be linked to another key, or rather the value of a key is linked to the value of another key. This is to create one edit point for when you don't want to have unique objects for every key. You could for instance like all keys that denotes the foreground colors for week days to a "`WeekDay/defaultWeekDayColor`" key. That way you can change them all by changing that key, you don't have to change all seven. You can also just set for instance the Sunday key to `Color.RED` and that will only affect Sunday.

Load and Save

The Themes class has load and save methods to/from `File`s, `URL`s and `Stream`s. It is using the standard JavaBeans `XMLEncoder` and `XMLDecoder` for the transformation to XML. This ensures you can load the themes from any application that is run on JDK 1.4 or later without extra classes. The only thing you need is the very small .jar from this component and your particular `Theme` class. This also ensures that there is no vendor lock-in.

If you are storing objects in the theme that haven't got a XML persistence delegate (most interesting Sun JDK classes do) you can simply add a delegate for your class to `IOUtil`.

`PersistenceDelegate`s can be written to accommodate for any type of object. Sun Microsystems hosts good tutorials on the subject. They are typically one-liners for normal types of objects, or a few lines for more complex ones. If you follow the common `get`/`set` pattern for properties in your classes, and provide a `public` empty constructor, they are even automatically XML persistable, you don't have to do *anything* to make it work.

The Theme Editor

The Theme Editor is a GUI application to visually edit your themes. It can edit any theme that extends the class `Theme` and complies with the simple rules on how to make a theme.

You can load, save and edit themes. There are also a few settings available from the menus, but they should be self explanatory.

Custom Editor Panels

<Information pending for MiG Theme Editor component release>

Extending Theme Editor

<Information pending for MiG Theme Editor component release>

Making Your Own Themes

Making your own theme that is editable in the Theme Editor and/or be usable programmatically from your application is easy. Just follow these simple steps:

- Create a class that extends `Theme` or a subclass of `Theme`.
- Define a set of key that should be used as `public static final String` objects. This is not strictly necessary, but it will probably simplify things and minimize typing errors.

Normally in the constructor:

- For every key in the theme create one or many `PropertyCapabiliy` objects and set them on the key (or a parent folder key, more on that later) with any of `Theme`'s `setCapabilities(..)` methods.
- For every key in the theme set a default value with any of `Theme`'s `setDefaultValue(..)` methods. Note that also a default value of `null` should explicitly be set since it differs from *no value*.
- Call `Theme.transferDefaultsToTheme()` to actually set the theme to it's default values.

Your done.

Suggested Key Pattern

It is possible to create a String constant for every key in the theme, but if keys somewhat repetitive (which they usually are to some degree) there are a couple of convenience methods in the `Theme` class that you can use.

Lets say you have four headers, `North`, `South`, `East` and `West`, which all have three `Color` properties: `Outer`, `Middle` and `Inner`. You could make model this with 12 explicitly created keys:

```
public static final String HEADER_NORTH_OUTER_COLOR “Header/North/outerColor”;
public static final String HEADER_NORTH_MIDDLE_COLOR = “Header/North/middleColor”,
public static final String HEADER_NORTH_INNER_COLOR = “Header/North/innerColor”,
```

```
public static final String HEADER_SOUTH_OUTER_COLOR = "Header/South/outerColor"

and so on for eight more rows...
```

This wouldn't be very simple to index in a loop, since there in no notion of which keys denote NORTH or which are OUTER. Here is another way to make them more indexable:

```
public static final String[] DIRECTIONS_ = {"North/", "South/", "East/", "West/"};
public static final String[] PLACE_KEYS = {"outerColor", "middleColor", "innerColor"};
public static final String HEADER_ = "Header/"
```

Now the keys are more loop friendly, your application can use a nested for loop to get the colors and set them on the correct header object.

The `Theme` methods understands this as well, both for setting capabilities and default values. For setting the capabilities for all keys in you custom `Theme` you can use:

```
String[][] keyArr = new String[][] {DIRECTIONS_, PLACE_KEYS}
setCapabilities(HEADER_, keyArr, <the capability to set>);
```

That would have to be written out in 12 very similar lines if all keys where defined as unique fields.

If you have a whole sub tree that only consists of keys with the exact same capabilities you need only to set one capability on the parent folder key. In this scenario it would mean setting it only on the "`Header/`" key would work. Actually "`Header/`" is not a key, it is a *folder key*. Folder keys can't be used on their own, or store property values, but they can sometimes be used to denote a sub tree of properties.

```
setCapabilities(HEADER_, <the capability to set>);
```

Note that if you do this you can **not** have a key with at different capability named for instance "`Header/size`", since that is in the same sub tree as "`Header/`". You **can** have it if you explicitly set the capabilities for that key though.

The algorithm to find the capabilities for a certain key is to check the key itself and then recursively try with the closest parent folder key, then its parent and so on.

In the same way it is easy to set default values in a structured way with the command:

```
String[][] keyArr = new String[][] {DIRECTIONS_, PLACE_KEYS}
setDefaultValue(HEADER_, keyArr, <the default value(s) to set>);
```

With default values you can not set it only on the folder key, **every** key has to have its own default value.

Immutable Values

The values in the theme should always be treated as *Immutable*. You should never get a value from the theme and change that value for whatever reason. You should always make a copy of it first.

If the value in the theme should be changed a new value should be created and set in the theme. The reasons for this are many but mainly:

- To be able to track changes in the theme. If the values in the there are changed while n the theme the theme will not know they've changed and can not notify it's listeners.
- To be able to use the same value for many keys. This is to save resources, mainly memory. It also reduces startup time since not as many objects has to be created up front.
- Immutable values are thread safe.

If possible only object types that really are immutable, and thus can't be changed, should be used. This will make subtle bugs less likely. But even if that is not possible, great care should be taken to never change an object that are in the theme.

Shared Capabilities

The `PropertyCapability` is considered immutable and can be shared for more than one key. If the theme contains many keys with the same capabilities the actual same objects should be used to save resources. Capabilities for common object types, such as `Insets`, `Boolean`, and `Color`, have `public static` references in the `Theme` class that can, and should, be used.