InfoNode®
Docking Windows
Developer's Guide

Revision: 1.3
IDW Version: 1.4.0
Last Modified: 2005-12-04

Copyright © 2005 NNL Technology AB
All rights reserved.

InfoNode® is a registered trademark of NNL Technology AB in Sweden.

Java is a registered trademark of Sun Microsystems, Inc. in the U.S. and other countries.

Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

The authors assumes no resposibility for errors or omissions, or for damages resulting from the use of the information contained herein.

Contents

1 Introduction 1
1.1 About This Document 1
1.2 Intended Audience 1
1.3 About InfoNode Docking Windows 1
1.4 Examples and Demo Applications 2
1.5 Abbreviations 2
1.6 References 2
2 Basic Concepts 3
2.1 Window Tree 3
2.2 UI Items 3
2.2.1 Root Window 4
2.2.2 View 4
2.2.3 View Title Bar Buttons 4
2.2.4 Window Buttons on Tab 4
2.2.5 Split Window 5
2.2.6 Tab Window 5
2.2.7 Tab Window Buttons 5
2.2.8 Window Bar 5
2.2.9 Minimized Window Tab 5
2.2.10 Floating Window 5
3 Window Classes 6
3.1 DockingWindow 6
3.2 View 6
3.3 RootWindow 7
3.4 SplitWindow 7
3.5 AbstractTabWindow 7
3.6 TabWindow 8
3.7 WindowBar 8
3.8 FloatingWindow 8
4 Creating a Simple Application 10
4.1 Threading 10
4.2 Creating a Root Window 10
4.3 Creating a Window Layout 11
4.4 Enabling a Window Bar 12
5 The Window Tree 14
5.1 Traversing a Window Tree 14
5.2 Moving a Window 14
5.3 Removing a Window 16
6 Window Layouts 17
6.1 Creating a Window Layout Programatically 17
6.2 Using Your Application as Window Layout Designer 17
6.3 Serializing a Window Layout 19
6.3.1 Writing a Window Layout 19

6.3.2 Reading a Window Layout 19
7 View Serialization 21
7.1 ViewSerializer 21
7.2 Static Views 21
7.2.1 StringViewMap 21
7.2.2 ViewMap 21
7.3 Dynamic Views 22
7.4 Mixing Static and Dynamic Views 22
8 Managing Focus 24
9 Window Operations 25
9.1 Closing a Window 25
9.2 Minimizing a Window 25
9.3 Maximizing a Window 26
9.4 Restoring a Window 26
9.5 Undocking a Window 26
9.6 Docking a Window 27
9.7 Enabling and Disabling User Operations 27
10 Window Listeners 28
11 Popup Menus 30
12 Window Titles 31
13 Drag and Drop 32
13.1 Trigger an External Drag 32
13.2 Drop Types 32
13.2.1 Interior Drop 33
13.2.2 Split Drop 33
13.2.3 Child Drop 33
13.2.4 Insert Tab Drop 33
13.3 Enable or Disable Recursive Tabs 33
13.4 Enable or Disable Drag and Drop 34
13.5 Advanced Filtering of Drag and Drop Operations 34
14 Mouse Button Listeners 36
15 Window Actions 37
16 Heavyweight Components in IDW 38
16.1 Important Restrictions 38
16.2 Enabling Heavyweight Support 38
17 Applets and Web Start Applications 40
17.1 Security Permission 40
17.1.1 Hover 40
17.1.2 Floating Windows and Heavyweight Components 40
17.1.3 Requesting and Granting Permission 40
17.2 Floating Windows in an Applet 41
18 IDW Properties Classes 42
18.1 Overview 42
18.2 RootWindowProperties 43
18.3 DockingWindowProperties 43
18.4 DockingWindowDropFilterProperties 44

18.5 TabWindowProperties 44
18.6 WindowBarProperties 44
18.7 SplitWindowProperties 44
18.8 FloatingWindowProperties 45
18.9 WindowTabProperties 45
18.10 WindowTabStateProperties 45
18.11 WindowTabButtonProperties 46
18.11.1 Creating a ButtonFactory 46
18.12 ViewProperties 47
18.13 ViewTitleBarProperties 47
18.14 ViewTitleBarStateProperties 47
18.15 TabbedPanel::TabbedPanelProperties 48
18.16 TabbedPanel::TitledTabProperties 48
18.17 TabbedPanel::TitledTabStateProperties 48
19 Themes 49
19.1 Creating a Theme 49
19.2 Using a Theme 49
19.3 Enabling Title Bar Style 50
19.4 Look and Feel Docking Theme 51
20 Properties 53
20.1 Property 53
20.2 PropertyGroup 53
20.3 Typed Properties 53
20.4 PropertyValueHandler 53
21 Property Maps 54
21.1 Property Map Classes 54
21.1.1 PropertyMap 54
21.1.2 PropertyMapGroup 55
21.1.3 PropertyMapValueHandler 55
21.1.4 PropertyMapProperty 55
21.1.5 PropertyMapFactory 55
21.1.6 PropertyMapContainer 55
21.2 Advanced Features 55
21.2.1 Super Maps 55
21.2.2 Property Map Composition 56
21.2.3 Property Value References 57
21.2.4 Listeners 58
21.2.5 Weak Listeners 58
21.2.6 Batch Processing 59
21.2.7 Serialization 59
21.3 ComponentProperties 59

1 Introduction

1.1 About This Document

This document is a developer's guide to IDW, InfoNode Docking Windows. It covers the most important concepts found in IDW. For more detailed information about the classes, methods and properties referred to in this document, please see the Javadoc and complete property listing at http://www.infonode.net/index.html?idwdoc.

1.2 Intended Audience

The intented audience of this document is Java developers who have some experience with AWT/Swing programming. You should know the basics of `Borders`, `Colors`, `Components` etc.

1.3 About InfoNode Docking Windows

InfoNode Docking Windows is a Java Swing framework for docking windows. It is released under two licenses, GPL and a commercial license. The GPL version can be used for free by any open source project released under the GPL. The commercial license is for projects that can't use the GPL. See http://www.infonode.net for more information about IDW and the licensing.

Docking windows are internal application windows which the user can re-arrange by dragging them. Unlike common windows found in operation system GUIs, the docking windows are “docked” to each other and can't be moved around freely (with some exceptions). Docking windows are commonly used in IDEs like Eclipse, Netbeans and the Microsoft developer tools. However, their usage is not in any way limited or connected to IDEs, many GUI applications can benefit from using a docking windows framework that allows the user to customize the layout of the application to his/her needs and preferences. Using IDW is similar to creating an application layout using `JSplitPanes`, `JTabbedPanes` and layout managers, but the GUI will become much more flexible and customizable.

Some of the features of IDW are:

- Unlimited depth of nested split and tab windows. For example, you can put two windows in a split pane that is located in a tab of a tab pane. There is practically no limitation on the window layout.
- Tabs can be placed on any side of the tab window. The text and icon of the tabs can be rotated in any direction.
- Windows can be minimized to any edge of an application. They can be dragged to and from the edges.
- Windows can be undocked to floating windows that can be moved anywhere on the desktop.

- Support for heavyweight components. This causes some restrictions, see chapter *16 Heavyweight Components in IDW*.
- A tab window can be maximized to cover the entire window space (except the window bars). It can later be restored to its original location.
- A window or window tree can be undockded to a floating window.
- Easily save and load the window layout using streams.
- Theme support.
- Flexible properties system which allows you to customize the look and behaviour of IDW to suit your application.

1.4 Examples and Demo Applications

The source code for an example application using IDW is included in the IDW distribution. Web Start demos of the example application and a more advanced application can be found at http://www.infonode.net/index.html?idwdemo.

1.5 Abbreviations

- IDW – InfoNode Docking Windows
- ITP – InfoNode Tabbed Panel

1.6 References

[1] *"InfoNode Tabbed Panel Developer's Guide"*, NNL Technology AB

2 Basic Concepts

This chapter describes the basic concepts of IDW.

2.1 Window Tree

The IDW framework consists of a number of different window types. Each window can have a *window parent* and is then a *child window* of that window. A window can have one or more child windows depending on its type. Windows connected through this parent-child relationship forms a *window tree*. There can be an unlimited number of window trees. Window trees with a *root window* at the root can be displayed on the screen.

2.2 UI Items

Figure 2.1 shows a typical IDW application with a root window containing a number of different windows. The highlighted parts of the figure are described in the following sections.

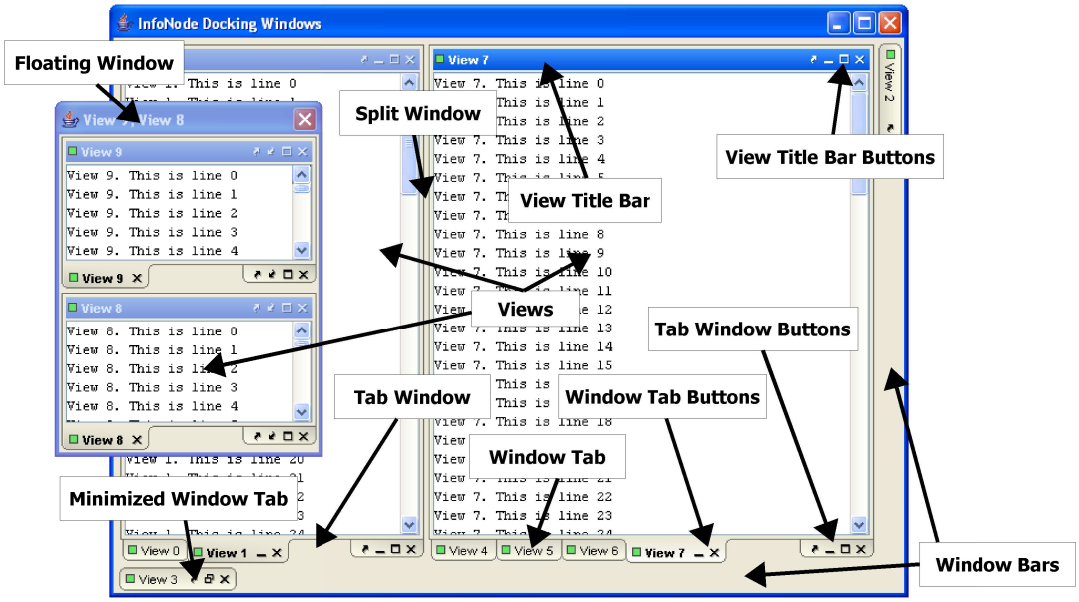

Figure 2.1

A window layout showing the different UI items in a root window

2.2.1 Root Window

Root window is the top most window. It is a container for other windows and can contain up to four window bars, up, down, left and right. The area inside the window bars is called window area. A window can also be maximized in a root window's window area.

2.2.2 View

Every application that uses IDW needs to define a number of *views*. A view is a window containing an application specific lightweight component, for example a `JPanel` or a `JTable`. This is the component you need to define for your application. The view can contain a heavyweight component, see chapter *16 Heavyweight Components in IDW*.

The view can also have a title bar that can be shown on either side of the view's component. Each view has a text title and an optional icon. The title and icon is displayed in a tab in a tab window and/or in the view's title bar.

2.2.3 View Title Bar Buttons

The view title bar's buttons can be used for minimizing, maximizing, restoring, undocking, docking and closing a view. You can customize which buttons are shown when the view is focused or not focused.

2.2.4 Window Buttons on Tab

The windows' buttons can be used for minimizing, maximizing, restoring, undocking, docking and closing a window. The buttons are shown on the tab connected to the

window they operate. You can customize which buttons are shown for focused tabs, selected tabs and normal (unselected) tabs.

2.2.5 Split Window

This is a window that is similar to a `JSplitPane`. It contains two other windows, one on each side of the split divider. The split divider can be either horizontal or vertical and can be dragged to resize the windows.

2.2.6 Tab Window

A tab window is similar to a `JTabbedPane` in that it contains a number of tabs where one tab is selected at a time. Each tab shows the title and the icon of the window connected to that tab. When a tab is selected the window connected to the selected tab is shown inside the tab window.

2.2.7 Tab Window Buttons

The tab window buttons can be used for minimizing, maximizing, restoring, undocking, docking and closing a tab window (including the windows inside it). You can customize which buttons that should be visible in a tab window.

2.2.8 Window Bar

The window bars are located on the edges of a root window. They are similar to tab windows except that it is possible that no tab is selected in the window bar. The title and icon of the tabs in a window bar are rotated along the orientation of the window bar, so window bars on the left and right sides paints their tabs vertically.

2.2.9 Minimized Window Tab

When a window is minimized a tab connected to it is placed on a window bar. When the tab is selected a content panel is displayed containing the window connected to the selected tab. The size of the content panel can be adjusted by the user by dragging the edge of it with the mouse. The same content panel is used for all the windows located on a window bar. The content panel is shown until focus is moved to another window or the selected tab is deselected, which can be done by clicking on it.

2.2.10 Floating Window

A floating window is a window that floats on top of a root window. Windows inside a floating window are *undocked*. A window can be maximized inside a floating window, the same way as a windows can be maximized in a root window. Windows can be dragged from a root window to a floating window, from a floating window to a floating window and from a floating window to a root window.

3 Window Classes

IDW consists of a number of window classes which are located in the `net.infonode.docking` package. Each window class stores property values in a properties object, see chapter *18 IDW Properties Classes* for more information. *Figure 3.1* shows the IDW window classes and their relationships. The classes are described in the following sections.

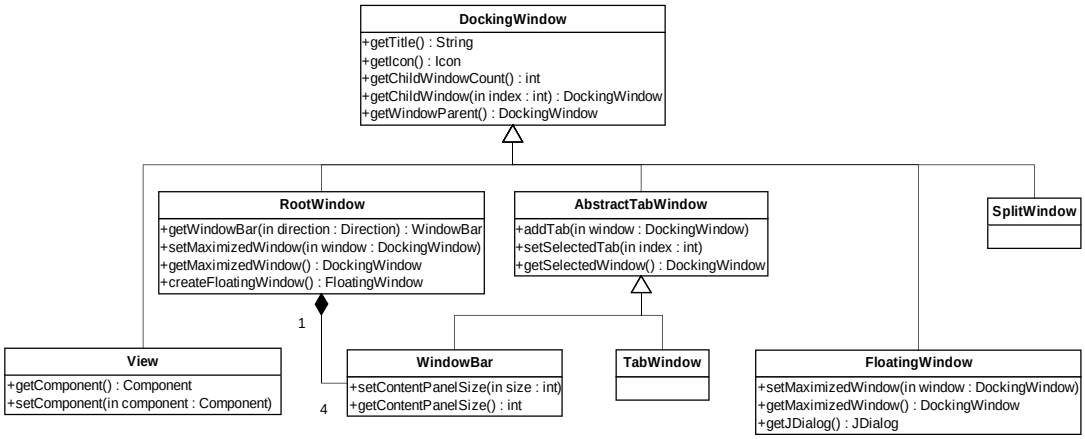

Figure 3.1

The relationships between the window classes

3.1 DockingWindow

`DockingWindow` is the base class for all window classes and contains methods common to all window types in IDW. `DockingWindow` contains methods for getting the window title, `getTitle()`, and the window icon, `getIcon()`. A `DockingWindow` can have one window parent, see `getWindowParent()`, and several child windows, see `getChildWindowCount()` and `getChildWindow(int index)`.

A `DockingWindow` stores its property values in a `DockingWindowProperties` object which can be retrieved using `DockingWindow.getWindowProperties()`. It uses the `DOCKING_WINDOW_PROPERTIES` object from the `RootWindowProperties` of the `RootWindow` it is located in as super object to its properties object.

3.2 View

A `View` contains a component. A `View` stores its property values, for example title and icon, in a `ViewProperties` object which can be retrieved using `View.getViewProperties()`. It uses the `VIEW_PROPERTIES` object from the `RootWindowProperties` of the `RootWindow` it is located in as super object to its properties object.

A `View` can also show a title bar on either side of the component (one side at a time). The title bar's property values are stored in `ViewTitleBarProperties` that is retrieved using `View.getViewProperties().getViewTitleBarProperties()`. The title bar can have two states, one when the View had focus and one when the View is not focused. The `FOCUSED_PROPERTIES` in `ViewTitleBarStateProperties` are added as super object to the `NORMAL_PROPERTIES` (also in `ViewTitleBarProperties`) when the `View` has focus and is then removed when the `View` loses focus.

3.3 RootWindow

A `RootWindow` has no window parent. It has four `WindowBar`s as child windows, possibly one window in its center and `FloatingWindow`s if any have been created. The center window is stretched to fill the entire window area of the `RootWindow`.

A `RootWindow` can also have a maximized window. The maximized window replaces the center window on the screen, but it is not a child window of the `RootWindow`, it keeps the location in the window tree it had before it was maximized. The maximized window can be set and removed using the `RootWindow.setMaximizedWindow()` method.

A `RootWindow` stores its property values in a `RootWindowProperties` object which can be retrieved using `RootWindow.getRootWindowProperties()`.

3.4 SplitWindow

A `SplitWindow` can have up to two child windows, the left/upper window and the right/lower window. One of these window slots can be empty as a result of a window removal, but this is only temporary since a `SplitWindow` with an empty slot is replaced in the window tree with its remaining child window.

A `SplitWindow` has a divider location which is a float value that can be adjusted between 0 – 1. It determines the size ratio of the child windows. When the `SplitWindow` is resized the divider location value remains unchanged, and thus the size ratio of the child windows also remains unchanged.

A `SplitWindow` stores its property values in a `SplitWindowProperties` object which can be retrieved using `SplitWindow.getSplitWindowProperties()`. It uses the `SPLIT_WINDOW_PROPERTIES` object from the `RootWindowProperties` of the `RootWindow` it is located in as super object to its properties object.

3.5 AbstractTabWindow

`AbstractTabWindow` is the base class for the `TabWindow` and `WindowBar` classes. It contains a `net.infonode.tabbedpanel.TabbedPanel` (see [1] for more information) which is used for displaying its tabs and the window connected to the selected tab. You can add tabs, see `addTab()`, and get the window connected to the selected tab, see `getSelectedWindow()`.

3.6 TabWindow

A `TabWindow` is an `AbstractTabWindow` that always has one tab selected and the window connected to the selected tab is visible.

A `TabWindow` stores its property values in a `TabWindowProperties` object which can be retrieved using `TabWindow.getTabWindowProperties()`. It uses the `TAB_WINDOW_PROPERTIES` object from the `RootWindowProperties` of the `RootWindow` it is located in as super object to its properties object.

3.7 WindowBar

A `WindowBar` shows its tabs on the edges of a `RootWindow`. A `WindowBar` can have zero or one tab selected. Connected to each `WindowBar` is a content panel which displays the window connected to the selected tab on top of the contents of the `RootWindow`. If there is no tab selected the content panel is not displayed. The size of the content panel can be read with `getContentPanelSize()` and set with `setContentPanelSize()`. The size can also be adjusted by the user by dragging the edge of the content panel with the mouse. The size is the width for vertical `WindowBar`s and the height for horizontal `WindowBar`s. The content panel is always stretched in the other direction.

A `WindowBar` stores its property values in a `WindowBarProperties` object which can be retrieved using `WindowBar.getWindowBarProperties()`. It uses the following steps to construct its properties object, `rootWindowProperties` is the `RootWindowProperties` of the `RootWindow` the `WindowBar` is located in:

1. The `rootWindowProperties.getTabWindowProperties()` object is added as super object to its `WindowBarProperties.getTabWindowProperties()` object.
2. The `rootWindowProperties.getWindowBarProperties()` object is added as super object to its `WindowBarProperties` object.
3. A `WindowBarProperties` object with property values depending on the `WindowBar` location, for example tab direction and tabbed panel orientation, is added as super object to its properties object.

3.8 FloatingWindow

A `FloatingWindow` is a window that can contain other windows and is floating on top of the `RootWindow`. The `FloatingWindow` is put inside a `JDialog` that can be moved around, or placed on a separate screen from the root window. The `JDialog` can be retrieved using `FloatingWindow.getJDialog()` and it is possible to set for example a `JMenuBar` in the `JDialog`. A `FloatingWindow` is a direct child window of a `RootWindow`.

A `FloatingWindow` stores its property values in a `FloatingWindowProperties` object which can be retrieved using `FloatingWindow.getFloatingWindowProperties()`. It uses the `FLOATING_WINDOW_PROPERTIES` object from the `RootWindowProperties` of the

`RootWindow` it is located in as super object to its properties object.

When a window is undocked a new `FloatingWindow` is created and added as child window to the `RootWindow` of the undocked window. The undocked window is then added as child window to the `FloatingWindow`. When the window is docked, it's restored to the location where it was undocked. When there are no more child windows in the `FloatingWindow` it is automatically closed and removed from the `RootWindow`. It is possible to disable the auto close in the `FloatingWindowProperties` for the `FloatingWindow` so that the `FloatingWindow` is still visible (but empty) and windows can be dropped into it.

4 Creating a Simple Application

This chapter takes you through the basics of creating an application that uses IDW. For a more complete example application, see the example included in the IDW distribution.

4.1 Threading

IDW is not thread safe and all constructor and method calls, including `RootWindow` creation, should be made in the AWT event dispatching thread. Use `SwingUtilities.invokeLater()` and `SwingUtilities.invokeAndWait()` when calling IDW from other threads.

4.2 Creating a Root Window

To use the docking windows library you must create a `RootWindow` instance. The root window is the toplevel container for all docking windows. The easiest way to create a root window is to call the static `DockingUtil.createRootWindow()` method. Before you use it you must create a couple of views and put them in a `ViewMap`.

Create a root window with 5 views each containing a JLabel:

```
View[] views = new View[5];
ViewMap viewMap = new ViewMap();

for (int i = 0; i < views.length; i++) {
  views[i] = new View("View " + i, null, new JLabel("This is view " + i + "!"));
  viewMap.addView(i, views[i]);
}

RootWindow rootWindow = DockingUtil.createRootWindow(viewMap, true);
```

The example uses a `ViewMap` where each view is assigned a unique integer ID which is used for identifying the view when reading and writing window layouts, see chapter *7 View Serialization* and chapter *6 Window Layouts* for more information about this. The `DockingUtil.createRootWindow()` method creates a `RootWindow` with a `TabWindow` containing all the views found in the `ViewMap`. It also adds a `WindowPopupMenuFactory` to the `RootWindow`, see chapter *11 Popup Menus* for more information.

After adding the `RootWindow` to a `JFrame` and displaying that, you application should look similar to that in *Figure 4.1*.

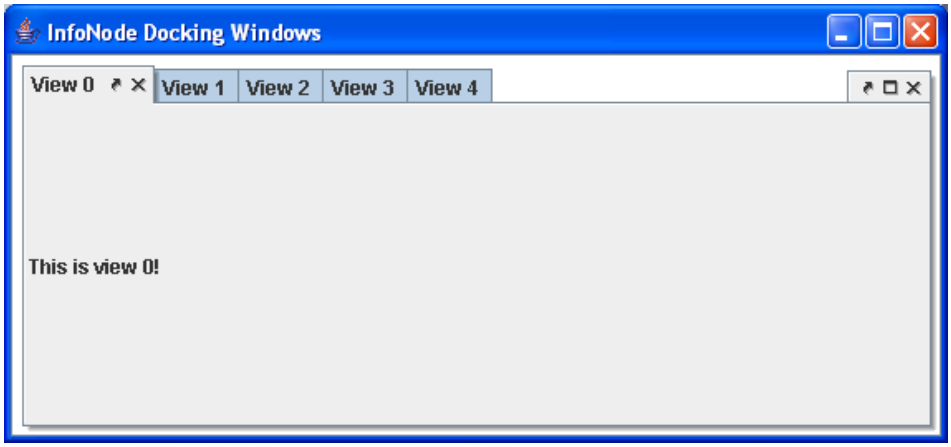

Figure 4.1

A simple IDW application

4.3 Creating a Window Layout

The widow layout in *Figure 4.1* might not be the most user friendly setup for your application, so you might want to change this window setup using the `RootWindow.setWindow()` method.

Create a more complex window layout using nested split windows and tab windows:

```
rootWindow.setWindow(
  new SplitWindow(true,
                  0.4f,
                  new SplitWindow(false,
                                  views[0],
                                  new SplitWindow(false, views[1], views[2])),
                  new TabWindow(new DockingWindow[]{views[3], views[4]})));
```

Your frame should now look similar to *Figure 4.2*.

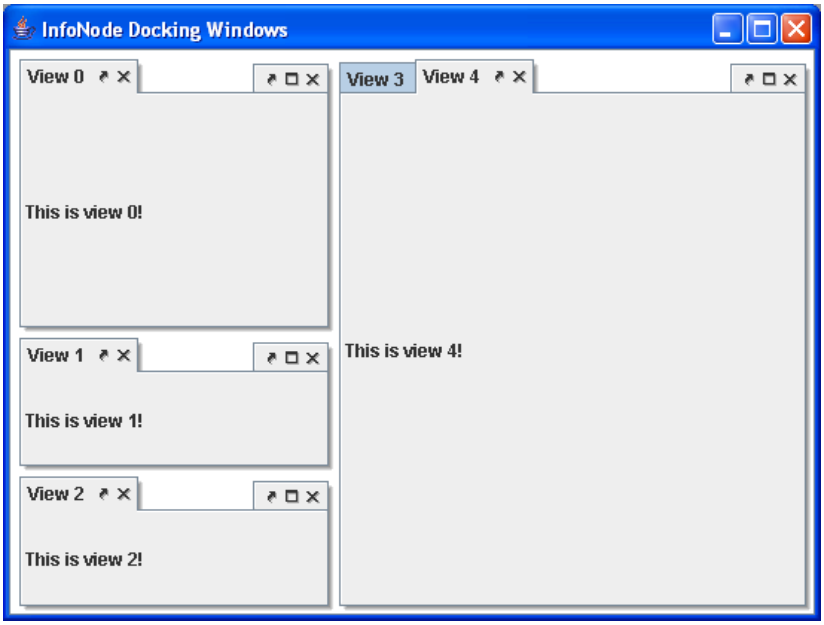

Figure 4.2

Applying a custom window layout

Note that you don't have to explicitly remove windows that you add to other windows, this is handled automatically. There is no limitation on the depth of the window tree, so you can add split and tab windows inside other split and tab windows.

4.4 Enabling a Window Bar

By default all four `WindowBar`s of a `RootWindow` are disabled, which means the windows inside the `RootWindow` can't be minimized.

Enable the bottom (down) window bar and add view 3 to it:

```
rootWindow.getWindowBar(Direction.DOWN).setEnabled(true);
rootWindow.getWindowBar(Direction.DOWN).addTab(views[3]);
```

The frame should now look similar to that in *Figure 4.3*.

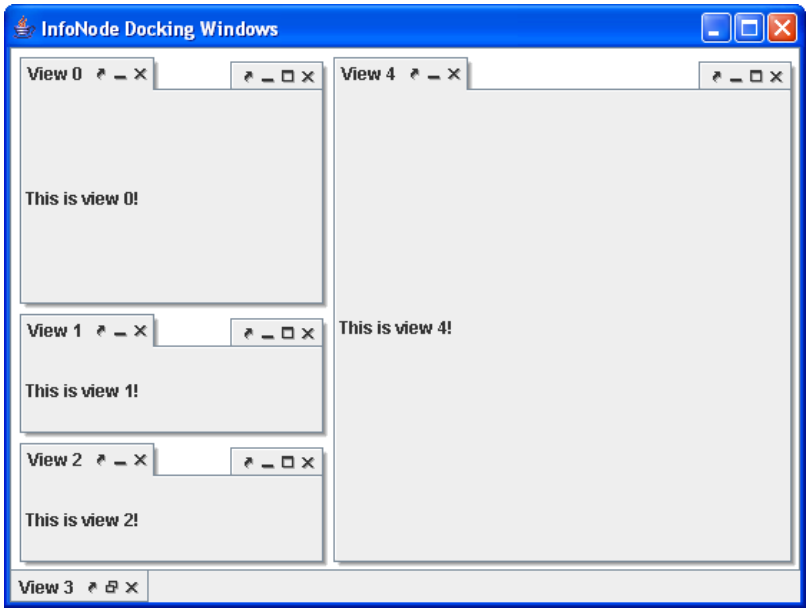

Figure 4.3

Enabling the bottom WindowBar

Note that the window and tab window minimize buttons now are visible because the windows can be minimized to the enabled `WindowBar`.

5 The Window Tree

Figure 5.1 shows the window tree after running the code in section *4.2 Creating a Root Window*.

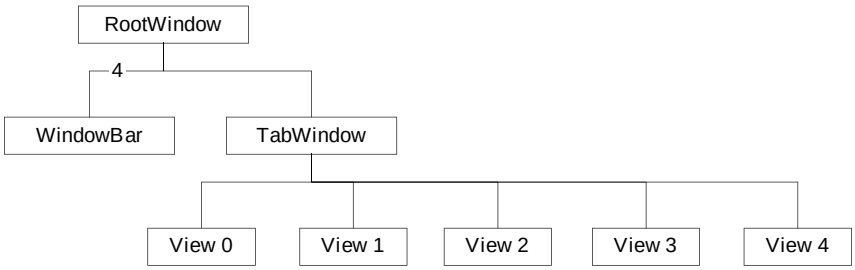

Figure 5.1

The window tree at application start

5.1 Traversing a Window Tree

A window tree can be traversed using the following methods in the `DockingWindow` class:

`int getChildWindowCount()`	Returns the number of child windows a window has.
`DockingWindow getChildWindow(int index)`	Returns the child window with the given index.
`DockingWindow getWindowParent()`	Returns the parent window of a window, or null if there is none.

5.2 Moving a Window

The user can move a window by dragging its tab or the view title bar (if visible) with the mouse. The library will automatically create and remove split and tab windows when needed. *Figure 5.2* shows the user dragging view 3 and splitting it with view 0.

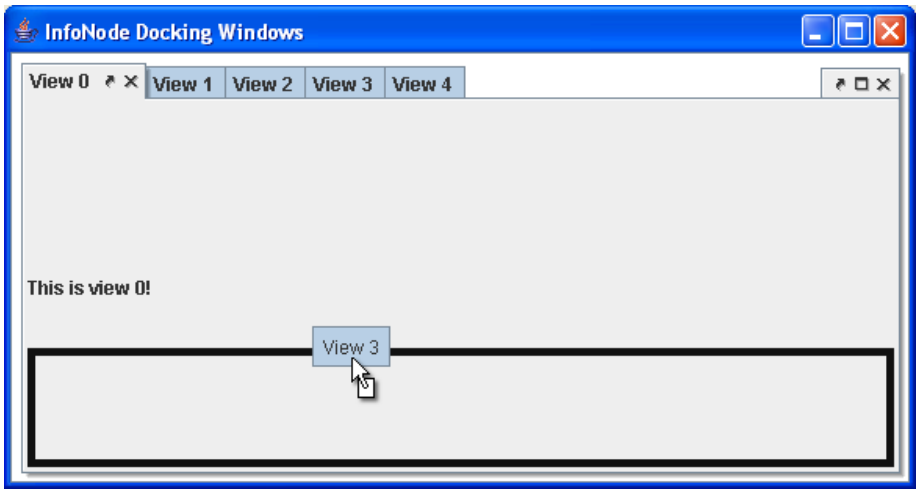

Figure 5.2

Dragging View 3

Figure 5.3 shows the result after dropping the view.

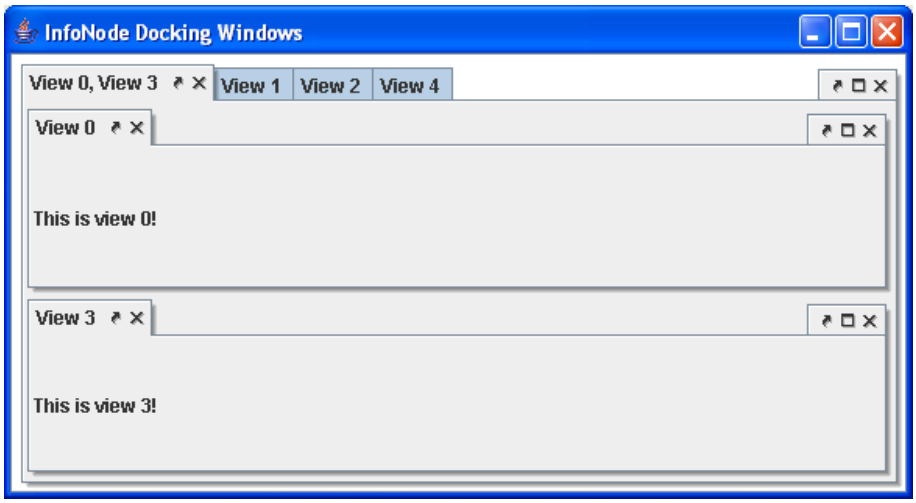

Figure 5.3

View 3 has been dropped

A `SplitWindow` with view 0 and view 3 was created and replaced view 0 in the first tab of the `TabWindow`. Notice that the `SplitWindow` is created inside the `TabWindow`, this is called a nested window. Also notice that two new `TabWindows` were created inside the `SplitWindow` to contain the views. They were created to display the icon and title of the views. *Figure 5.4* shows the window tree for *Figure 5.3*, after view 3 has been dropped.

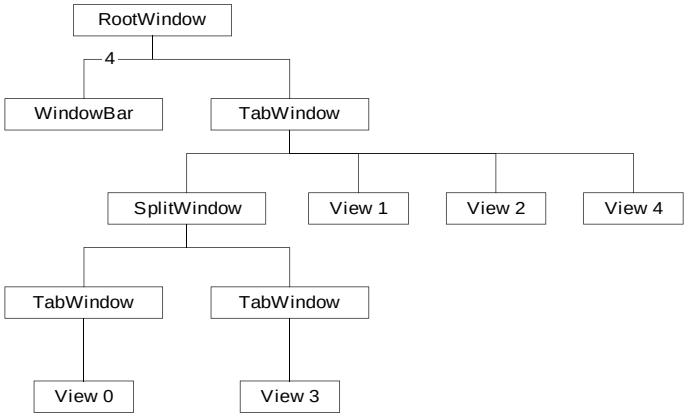

Figure 5.4

The window tree after dropping view 3

5.3 Removing a Window

Continuing from the example in the previous section, view 3 is now closed. *Figure 5.5* shows the window tree after view 3 has been closed.

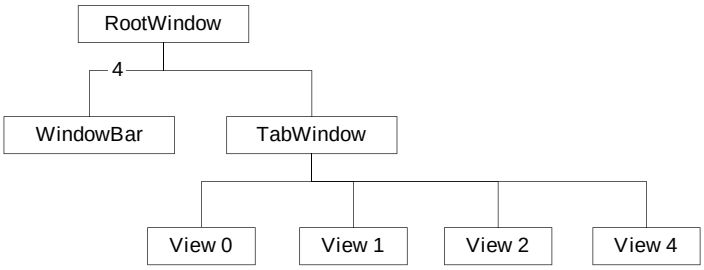

Figure 5.5

The window tree after closing view 3

As seen in the figure, removing view 3 caused some other windows to be implicitly removed. This is done to avoid empty spaces on the screen and reduce the number of windows shown to the user. The following happened after view 3 was closed:

1. The `TabWindow` containing view 3 was removed because it became empty.
2. The `SplitWindow` was replaced by the `TabWindow` containing view 0 because that became the only child window of the `SplitWindow`.
3. The `TabWindow` containing view 0 was removed because it only contained one child window and the window parent is another `TabWindow`.

6 Window Layouts

IDW window layouts can be created programatically or by the user rearranging windows. Window layouts and window property values can be serialized to Java streams so that they can be restored later on. The views can also be serialized, see chapter *7 View Serialization*.

6.1 Creating a Window Layout Programatically

A window layout is basically a window tree. The window layout can be set by either creating a `RootWindow` with an initial window layout or by calling `RootWindow.setWindow()` with a layout at any time.

Setting a window layout using nested split windows and tab windows:

```
View[] views = new View[5];
ViewMap viewMap = new ViewMap();

for (int i = 0; i < views.length; i++) {
  views[i] = new View("View " + i, null, new JLabel("This is view " + i + "!"));
  viewMap.addView(i, views[i]);
}

RootWindow rootWindow = DockingUtil.createRootWindow(viewMap, true);

// Creating a window tree as layout
DockingWindow myLayout =
  new SplitWindow(true,
                  0.4f,
                  new SplitWindow(false,
                                  views[0],
                                  new SplitWindow(false, views[1], views[2])),
                  new TabWindow(new DockingWindow[]{views[3], views[4]}));

// Set the layout
rootWindow.setWindow(myLayout);
```

6.2 Using Your Application as Window Layout Designer

It's might be quite hard to create and advanced layout programatically, for example your application's default layout when it is first started. You'll have to figure out the window tree by hand. Therefore it's better and easier if you can use your own application as layout designer i.e. rearrange your windows and when you're satisfied, retrieve the layout.

This can be done in two ways, either you use serialization, see section *6.3 Serializing a Window Layout* , to save/load your layout or you can use `Util.DeveloperUtil` that has methods that will help you create layouts programatically. It should be noted that serialization is the preferred way to for example load/save user layouts during run-time. The methods in `DeveloperUtil` are only meant to be used during the development of your application to for example create an initial default layout that is not dependent on serialization.

Let's say you have created your application. Now you want to create a nice looking default layout programatically. You can then add a line of code that will show an

additional frame containing Java pseudo-like code for the current layout in your `RootWindow`. It will show the layout for all windows connected to your `RootWindow`, i.e. the top-level window inside the `RootWindow`, windows on `WindowBar`s and windows in `FloatingWindow`s. The frame contains a button that when clicked will retrieve the current layout for your `RootWindow`.

Creating a frame showing the layout for myMainRootWindow:

```
// Frame showing layout for myMainRootWindow.
// Don't forget to remove this line when release compiling
DeveloperUtil.createWindowLayoutFrame("My Main RootWindow", myMainRootWindow)
  .setVisible(true);
```

In *Figure 6.1* you see a quite complex window layout.

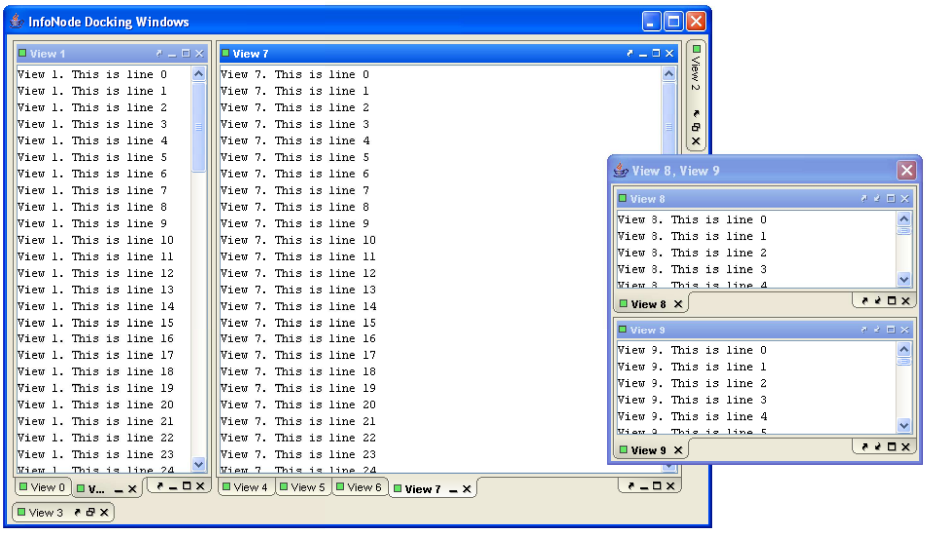

Figure 6.1

A quite complex window layout

The outputted Java pseudo-code shown in *Figure 6.2* can be copied almost directly into your code. You will have to substitute the `RootWindow` and `View` references to the references to your real `RootWindow` and `View`s.

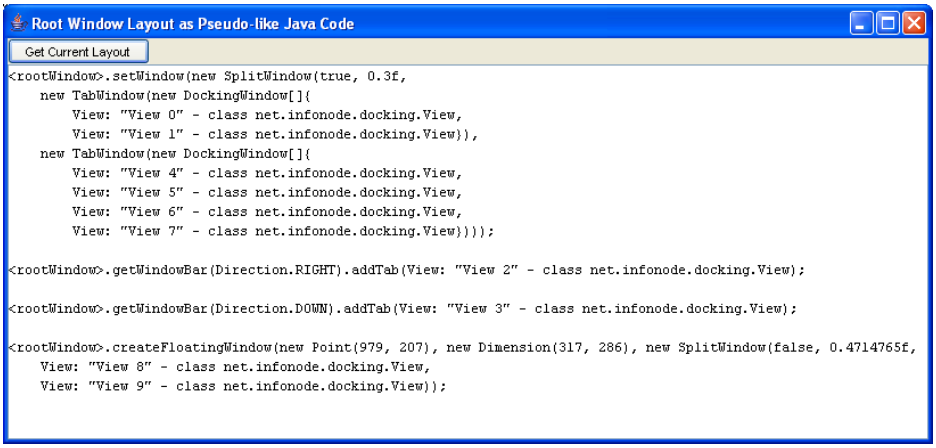

Figure 6.2

Java pseudo-like code for the window layout in Figure 6.1

6.3 Serializing a Window Layout

It is possible to read/write window layouts and window property values. This makes it possible to restore a layout and property values later on, for example when the application is restarted. The views can also be serialized, see chapter *7 View Serialization*.

6.3.1 Writing a Window Layout

The window layout can be written to a stream using the `RootWindow.write()` methods. The property values in the window properties objects can optionally also be written on the stream. Note that only property values modified in the properties object of the window is written, not property values in super objects from themes for example. `View`s inside the root window will be written using the `ViewSerializer` set in the root window.

Write the window layout and the window property values to a byte array:

```
ByteArrayOutputStream bos = new ByteArrayOutputStream();
ObjectOutputStream out = new ObjectOutputStream(bos);
rootWindow.write(out);
out.close();
```

6.3.2 Reading a Window Layout

A previously written window layout can be read from a stream using the `RootWindow.read()` methods. Property values in the window property objects can optionally be read if they are available in the stream. Window layouts written with previous versions of IDW can always be read by later versions. However, window layouts written in newer versions of IDW cannot always be read by older versions.

`View`s inside the root window will be read using the `ViewSerializer` set in the root window.

Read the window layout and the window property values from a byte array:

```
```

```
// Read the window state from a byte array
rootWindow.read(new ObjectInputStream(
  new ByteArrayInputStream(bos.toByteArray())));
```

7 View Serialization

This chapter describes how to handle the serialization of views in different scenarios.

7.1 ViewSerializer

When saving and loading window layouts the `RootWindow` uses a `ViewSerializer` for handling the serialization of views. You don't have to specify a `ViewSerializer` for a `RootWindow`, but then you can't read and write window layouts.

When writing a window layout to a stream the `ViewSerializer` writes an identifier or some part of a view to the stream in the `ViewSerializer.write()` method so that it later can recreate that view in the `ViewSerializer.read()` method when the window layout is read. If a `ViewSerializer` can't recreate a view when reading from the stream it can return `null` from `read()` which causes the view to be removed from the window tree.

7.2 Static Views

Static views are views that are created during application start and that are always available to the user. The easiest way to handle static views is using a view map. The view map classes implement the `ViewSerializer` interface and thus handles view serialization for you. The view map classes also implements the `ViewFactoryManager` interface which is used in `WindowMenuUtil` for displaying the available views on a popup menu.

Views can be added to and removed from a view map without breaking compatibility with previously saved window layouts. If a view that has been removed from a view map is found in a serialized window layout it will be removed from the window tree when the layout is read. Views that has been added after the window layout was written will not be included in the window tree.

There are two view map classes in IDW, `StringViewMap` and `ViewMap`.

7.2.1 StringViewMap

The `StringViewMap` identifies each view using a `String`. This can be the view title or any other `String`. It's important that a `String` which identifies a view is unique and isn't modified or it will be impossible to load old window layouts. So if you're using the view title as view identification string, you should be certain that you never change the title of the view.

7.2.2 ViewMap

The `ViewMap` class uses integer values for identification of views. The same rule as for `StringViewMap`s applies here, you shouldn't change the integer value used to identify a view if you want to be able to load old window layouts.

Example of how an application would typically use a view map:

```
private static final int FILE_BROWSER_VIEW_ID = 0;
private static final int CLASS_BROWSER_VIEW_ID = 1;
private static final int METHOD_BROWSER_VIEW_ID = 2;

private RootWindow createRootWindow() {
  View fileBrowserView = new View("File Browser", null, new FileBrowser());
  View classBrowserView = new View("Class Browser", null, new ClassBrowser());
  View methodBrowserView = new View("Method Browser", null, new MethodBrowser());

  ViewMap viewMap = new ViewMap();
  viewMap.addView(FILE_BROWSER_VIEW_ID, fileBrowserView);
  viewMap.addView(CLASS_BROWSER_VIEW_ID, classBrowserView);
  viewMap.addView(METHOD_BROWSER_VIEW_ID, methodBrowserView);

  return new RootWindow(viewMap);
}
```

7.3 Dynamic Views

If you have an application where views are created and added dynamically, fo example as a result of user actions, and you want to be able to serialize the window layout you should create a custom `ViewSerializer`. For example, if you display contents of a file in an editor view you might want to identify that view with the file name and when the window layout is loaded you read the file name from the stream and create and return a new editor view for that file.

Example of creating a view serializer for an editor:

```
ViewSerializer viewSerializer = new ViewSerializer() {
  public void writeView(View view, ObjectOutputStream out) throws IOException {
    // Write the editor file name to the stream
    out.writeUTF(((Editor) view.getComponent()).getFileName());
  }

  public View readView(ObjectInputStream in) throws IOException {
    // Read the file name from the stream
    String fileName = in.readUTF();

    try {
      // Create a new editor view
      return new View(fileName, null, new Editor(fileName));
    }
    catch (IOException e) {
      // The file couldn't be loaded, skip this view
      return null;
    }
  }
};

RootWindow editorRootWindow = new RootWindow(viewSerializer);
```

Alternatively, you can of course use normal Java serialization of the component inside a `View`. Just use the object streams to write and read your object.

7.4 Mixing Static and Dynamic Views

A common way to mix static and dynamic views are by separating them into different `RootWindow`s. For example, if you create an IDE you can create a `RootWindow` that contains a number of static views like file browser, class browser, method browser and an editor view. This outer `RootWindow` can use a `ViewMap` to handle its views.

Inside the editor view you create another `RootWindow` that contains dynamically created editor views. This `RootWindow` can for example use the file name to identify the views, as described in the previous section.

If you mix static and dynamic views in the same `RootWindow`, you can use the `MixedViewHandler` class to simplify the serialization of views. The static views are then serialized using a normal view map and the dynamic views are serialized using a custom ViewSerializer.

8 Managing Focus

The `View` in a `RootWindow` that contains the AWT focus owner can be retrieved using `RootWindow.getFocusedView()`. If no `View` contains the focus owner, `null` is returned. Each tab connected to a window that contains the focus owner uses the special property values found in `TabWindowProperties.getFocusedProperties()` and `TabWindowProperties.getFocusedButtonProperties()`. See chapter *18 IDW Properties Classes* for more information.

Each window remembers the child window that last contained the focus owner, see `DockingWindow.getLastFocusedChildWindow()`, and it is possible to restore focus to that window using the `DockingWindow.restoreFocus()` method. This method also makes sure that the focused window is displayed on the screen by selecting its tab in a `TabWindow` or `WindowBar`. The method is called recursively on the last focused child windows until a `View` is found, where the focus owner is set to the last component that had focus inside the view.

9 Window Operations

Window operations can be performed either by the user via buttons/dragging in the UI or programatically via method calls. All window operations are notified through the `DockingWindowListener`s that are registered on the windows, see chapter *10 Window Listeners*.

9.1 Closing a Window

A window can be closed by calling `DockingWindow.close()` or `DockingWindow.closeWithAbort()`. Closing a window will remove it from the window tree its located in.

Both close methods will call the `DockingWindowListener.windowClosed()` method of all listeners of the window and the window's ancestors. `DockingWindow.closeWithAbort()` will also first call the `DockingWindowListener.windowClosing()` method of the listeners and abort the close operation if any listener throws an `OperationAbortedException` exception.

9.2 Minimizing a Window

A window is minimized when it has a `WindowBar` as ancestor. To minimize a window use the `DockingWindow.minimize()` or `DockingWindow.minimizeWithAbort()`methods, or add it to a `WindowBar` or another window that is minimized. `DockingWindow.minimize()` and `DockingWindow.minimizeWithAbort()` will add the window to the last `WindowBar` it was located on last time it was minimized. If the window has never been minimized it will added to a suitable `WindowBar`. You can set and get the preferred minimize direction with the methods `DockingWindow.setPreferredMinimizeDirection()` and `DockingWindow.getPreferredMinimizeDirection()`. To minimize a window to a specific `WindowBar` use the `DockingWindow.minimize(Direction direction)` or `DockingWindow.minimizeWithAbort(Direction direction)` method.

All minimize methods will call the `DockingWindowListener.windowMinimized()` method of all listeners of the window and the window's previous (before the minimize) ancestors. `DockingWindow.minimizeWithAbort()` will also first call the `DockingWindowListener.windowMinimizing()` method of the listeners and abort the minimize operation if any listener throws an `OperationAbortedException` exception.

The window location is stored before it is minimized and the window can be restored to this location by calling `DockingWindow.restore()`.

9.3 Maximizing a Window

A window can be maximized by calling `DockingWindow.maximize()`, `DockingWindow.maximizeWithAbort()`, `RootWindow.setMaximizedWindow()` or `FloatingWindow.setMaximizedWindow()`if the window is inside a `FloatingWindow`. A maximized window will be displayed at the top in the center of the `RootWindow` or `FloatingWindow`.

The window tree is not modified when a window is maximized, it will still have the same window parent. You can check if a window is maximized using the `DockingWindow.isMaximized()` method. `RootWindow.getMaximizedWindow()` or `FloatingWindow.getMaximizedWindow()` will return the maximized window, or `null` if there is no maximized window.

All maximize methods will call the `DockingWindowListener.windowMaximized()` method of all listeners of the window and the window's ancestors. `DockingWindow.maximizeWithAbort()` will also first call the `DockingWindowListener.windowMaximizing()` method of the listeners and abort the maximize operation if any listener throws an `OperationAbortedException` exception.

A maximized window can be un-maximized by calling the `DockingWindow.restore()` method, or by calling `RootWindow.setMaximizedWindow()` or `FloatingWindow.setMaximizedWindow()`with a null argument.

9.4 Restoring a Window

The `DockingWindow.restore()` and `DockingWindow.restoreWithAbort()` methods will restore a window to the location it had before it was closed, minimized or maximized. When restoring a view, a copy of an old view parent window might be created and placed as window parent to the view. Restoring a split or tab window will not restore the window to its original location inside the `RootWindow`, but rather restore all the views located inside the window. So, don't assume that certain instance of a tab or split window will be placed inside a root window after it has been restored.

All restore methods will call the `DockingWindowListener.windowRestored()` method of all listeners of the window and the window's previous (before the restore) ancestors. `DockingWindow.restoreWithAbort()` will also first call the `DockingWindowListener.windowRestoring()` method of the listeners and abort the restore operation if any listener throws an `OperationAbortedException` exception.

9.5 Undocking a Window

A window can be undocked (placed in a `FloatingWindow`) by calling `DockingWindow.undock()` or `DockingWindow.undockWithAbort()`. If the window is already in a `FloatingWindow` it will be undocked again and placed in a new `FloatingWindow`.

Both undock methods will call the `DockingWindowListener.windowUndocked()` method of all listeners of the window and the window's previous (before the undock) ancestors. `DockingWindow.undockWithAbort()` will also first call the `DockingWindowListener.windowUndocking()` method of the listeners and abort the undock operation if any listener throws an `OperationAbortedException` exception.

Dragging and dropping a window outside the `RootWindow` or the `FloatingWindow` will undock the window. This behaviour can be disabled by setting the property `UNDOCK_ON_DROP` in the `DockingWindowProperties` to false.

9.6 Docking a Window

A window can be docked by calling `DockingWindow.dock()` or `DockingWindow.dockWithAbort()`. Note that it is not the tab window, split window etc that are docked. The views inside those windows are docked instead. A `View` is always docked back to the `RootWindow`. to the location it had prior to it was undocked.

Both dock methods will call the `DockingWindowListener.windowDocked()` method of all listeners of the window and the window's previous (before the dock) ancestors. `DockingWindow.dockWithAbort()` will also first call the `DockingWindowListener.windowDocking()` method of the listeners and abort the dock operation if any listener throws an `OperationAbortedException` exception.

9.7 Enabling and Disabling User Operations

The user can close, minimize, maximize, undock, dock and restore windows using buttons, popup menues etc. Each of these operations can be enabled/disabled using the properties found in `DockingWindowProperties`. Note that disabling an operation only affects the UI and does not disable any methods related to these operations in the `DockingWindow` class.

You can call the `isClosable()`, `isMinimizable()`, `isMaximizable()` and `isRestorable()` methods in `DockingWindow` to check if an operation is enabled. These methods might take into account additional information besides the related property value, for example `isMinimizable()` will return false if there are no enabled `WindowBar`s.

10 Window Listeners

Docking window listeners can be implemented using the `DockingWindowListener` interface. You can add and remove listeners to a docking window using the `DockingWindow.addListener()` and `DockingWindow.removeListener()` methods. A listener added to a window will received events for that window and all descendants of that window. `DockingWindowListener` contains the following methods:

- `windowAdded()` Called when a window has been added as child window to a window.
- `windowRemoved()` Called when a child window has been removed from a window.
- `windowShown()` Called when a window has been shown, ie its tab has been selected in an `AbstractTabWindow`.
- `windowHidden()` Called when a window has been hidden, ie its tab has been deselected in an `AbstractTabWindow`.
- `viewFocusChanged()` Called when the view that contains the focus owner has changed.
- `windowClosing()` Called before a window is closed. If an `OperationAbortedException` is thrown the close operation is aborted.
- `windowClosed()` Called after a window has been closed.
- `windowMaximizing()` Called before a window is maximized. If an `OperationAbortedException` is thrown the maximize operation is aborted.
- `windowMaximized()` Called when a window has been maximized in either a `RootWindow` or a `FloatingWindow`.
- `windowMinimizing()` Called before a window is maximized. If an `OperationAbortedException` is thrown the minimize operation is aborted.
- `windowMinimized()` Called when a window has been minimized i.e. moved to a `WindowBar`.
- `windowRestoring()` Called before a window is maximized. If an `OperationAbortedException` is thrown the restore operation is aborted.
- `windowRestored()` Called when a window has been restored from maximize, minimize or closed.
- `windowUndocking()` Called before a window has been undocked. If an `OperationAbortedException` is thrown the close operation is aborted.
- `windowUndocked()` Called after a window has been undocked.

- `windowDocking()` Called before a window has been docked. If an `OperationAbortedException` is thrown the close operation is aborted. This is called once for the window that "wants" to dock. The actual dock is performed for each `View` found in the subtree starting at window.

- `windowDocked()` Called after a `View` has been docked to the `RootWindow`.

11 Popup Menus

You can enable a popup menu for a window that is shown on a mouse popup trigger on the window component or its tab in a `AbstractTabWindow`. The popup menu for a docking window is created by a `WindowPopupMenuFactory` set with the `DockingWindow.setPopupMenuFactory()` method. By default there is no popup menu factory, so no popup menu is shown.

A popup menu factory of a window is recursively inherited to all child windows that has no popup menu factory set. The most common scenario is that you want the same popup menu factory for all window in a `RootWindow`. Setting the popup menu factory on the `RootWindow` will accomplish this.

Create a popup menu with a close item for the views in a root window:

```
rootWindow.setPopupMenuFactory(new WindowPopupMenuFactory() {
  public JPopupMenu createPopupMenu(final DockingWindow window) {
    JPopupMenu menu = new JPopupMenu();

    // Check that the window is a View
    if (window instanceof View) {
      menu.add("Close").addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          window.close();
        }
      });
    }

    return menu;
  }
});
```

12 Window Titles

You can customize the title text returned by the `DockingWindow.getTitle()` method by setting the `TITLE_PROVIDER` property in the `DockingWindowProperties`. The property value is an implementation of the `net.infonode.docking.title.DockingWindowTitleProvider` interface. The interface contains one method that returns a string title for a window.

Two title providers are included in the distribution:

- `SimpleDockingWindowTitleProvider` which just concatenates all the titles of the views found in the window. This is the default title provider which is used when no title provider has been set in the `DockingWindowProperties`.
- `LengthLimitedDockingWindowTitleProvider` which returns a length limited window title. It primarily uses the titles of views that are displayed on screen and fills up the rest of the title with views that are not displayed on screen.

When creating a custom title provider it's common to traverse the window tree to find views and then call `getViewProperties().getTitle()` on the views to get the view title from the view properties.

13 Drag and Drop

This chapter describes how to trigger an external drag and how to customize drag and drop operations.

13.1 Trigger an External Drag

Drag and drop of docking windows is normally handled internally by IDW but it's possible to trigger a dragging of a window from the outside of IDW. The drag is triggered by calling the `DockingWindow.startDrag()` method on a window. This method takes the `RootWindow` in which the window can be dropped as parameter and returns a `DockingWindowDragger` object which is used to control the drag and drop operation.

The easiest way handle external drag and drop is to create a `DockingWindowDragSource` for the component which you want to be able to trigger drag from and implement a `DockingWindowDraggerProvider` for the source. Typically the `DockingWindowDraggerProvider` implementation would create a new or use an existing view and return the `DockingWindowDragger` returned by `startDrag()` on the view. The `DockingWindowDragSource` class will handle mouse events and drag abort for you.

Starting an outside drag of a View from a JLabel:

```
final View myView = new View("My View", null, new JTextArea("My View"));
JLabel label = new JLabel("Drag View");
new DockingWindowDragSource(label, new DockingWindowDraggerProvider() {
  public DockingWindowDragger getDragger(MouseEvent mouseEvent) {
    return myView.startDrag(rootWindow);
  }
});
```

13.2 Drop Types

There are several types of drops that can occur when a docking window is dropped after a drag operation. These drop types depends on where the window was dropped. The table below shows what drop types each docking window supports. The types are described in the following chapters. These drop types can be filtered to customize the dropping behaviour, see *13.5 Advanced Filtering of Drag and Drop Operations*.

All drops starts at the root of the window tree and are then propagated down the window tree to the window into which the dragged window was dropped. Docking windows in the tree are asked if they would accept a drop of the window beeing dragged. If a drop is not accepted the drop cannot be completed i.e. if the window is dropped anyway, the drop will be aborted.

The drop type order is as follows:

- The window is asked if it supports split drop.
- If not, then it's asked if it supports child drop.

- If not, then it's asked if it supports interior drop or insert tab drop.
- If not, the drop cannot be completed.

	Interior Drop	Split Drop	Child Drop	Insert Tab Drop
RootWindow	x		x	
SplitWindow	x	x	x	
TabWindow		x	x	x
WindowBar			x	x
View		x		
FloatingWindow	x		x	

13.2.1 Interior Drop

This drop type occur when a window is dropped inside a `RootWindow` or a `FloatingWindow` that don't have any top-level window, i.e. they appear empty. It also occurs when a window is dropped onto the split divider in a `Splitwindow`.

13.2.2 Split Drop

This drop type occur when a window is dropped in another docking window and that window will be split i.e. replaced by a `SplitWindow` containing the dropped window and the window that was split.

13.2.3 Child Drop

This drop type occur before a child window of a docking window is asked if it accepts a drop.

13.2.4 Insert Tab Drop

This drop type occur when a window is dropped into the tab area of a `TabWindow` or onto `WindowBar`. It will result in the insertion of a tab containing the dropped window.

13.3 Enable or Disable Recursive Tabs

There's a convenience property called `RECURSIVE_TABS_ENABLED` in the `RootWindowProperties` that enables or disables recursive tabs i.e. dragging and dropping a window so that a nested `TabWindow` inside another `TabWindow` occurs.

Disable recursive tabs:

```
rootWindow.getRootWindowProperties().setRecursiveTabsEnabled(false);
```

13.4 Enable or Disable Drag and Drop

It's possible to enable or disable drag and drop in the `DockingWindowProperties` for a window. By doing this in the `DockingWindowsProperties` in the `RootWindowProperties` all dragging and dropping can be enabled or disabled. It's also possible to disable the dragging of the split divider in a `SplitWindow` in the `SplitWindowProperties`.

Freeze the layout by disabling drag:

```
// Disable drag ooperations
rootWindow.getRootWindowProperties().getDockingWindowProperties()
  .setDragEnabled(false);

// Disable split divider dragging
rootWindow.getRootWindowProperties().getSplitWindowProperties()
  .setDividerLocationDragEnabled(false);
```

13.5 Advanced Filtering of Drag and Drop Operations

`DropFilter`s makes it possible to control which drops that are allowed in a drag and drop operation. The package `net.infonode.docking.drop` contains several classes that make it possible to filter possible window drops on a per window basis.

A `DropFilter` is an interface that takes a `DropInfo` as argument and either return `true` if the drop can be accepted or `false` if the drop should be rejected. `DropInfo` is the base class for information about a drop. There are specialized classes for each type of drop, `SplitDropInfo`, `InteriorDropInfo`, `ChildDropInfo` and `InsertTabDropInfo`. The specialized drop infos contain information about the drop such as the window beeing dragged, the window that is currently asked if it accepts drop of the dragged window and other data that could be of importance when deciding if a drop could be accepted.

The package also contains two predefined singleton `DropFilter`s called `AcceptAllDropFilter` and `RejectAllDropFilter`.

The `DockingWindowProperties` contains a properties object called `DropFilterProperties`. This makes it possible to assign a `DropFilter` for each drop type a window supports. By default, drops of all windows are allowed. If a `DropFilter` is assigned to a drop type that is not supported by the window, for example *insert tab drop* in a `SplitWindow`, the filter is ignored.

Now let's filter and reject all drops of `TabWindow`s onto the split divider in any of the `SplitWindow`s in a `RootWindow`. Dropping on the split divider in a `SplitWindow` is an *interior drop* type.

Reject all interior drops of Tab Windows into any Split Window in a Root Window:

```
rootWindow.getRootWindowProperties().getDockingWindowProperties()
  .getDropFilterProperties().setInteriorDropFilter(
    new DropFilter() {
      public boolean acceptDrop(DropInfo dropInfo) {
        InteriorDropInfo interiorDropInfo = (InteriorDropInfo)dropInfo;

        // If the drop window is a split window and the drag window is a
        // tab window, no drops are allowed
        if (interiorDropInfo.getDropWindow() instanceof SplitWindow &&
            interiorDropInfo.getWindow() instanceof TabWindow)
```

```
            return false;

        return true;
      }
    });
```

14 Mouse Button Listeners

A mouse button listener implements the `net.infonode.gui.MouseButtonListener` interface and receives a `MouseEvent` when a mouse button is pressed, released or clicked on a docking window tab. Mouse button listeners can be added and removed from a docking window using the `DockingWindow.addTabMouseButtonListener()` and `DockingWindow.removeTabMouseButtonListener()` methods. The listeners will be called in the reverse order they were added, so the last added listener will be called first. When all mouse button listeners for a window has been called, the listeners of the parent window will be called and so on.

The `RootWindow` adds a default mouse button listener that handles restore and maximize on double click etc. The listener code is not run if the mouse event has been consumed using `InputEvent.consume()`. So, if you don't want the default mouse listener to run in some cases, just add a mouse button listener to the `RootWindow` and in the listener implementation call `InputEvent.consume()` on the mouse event.

15 Window Actions

`SimpleAction` is a class located in `net.infonode.gui.action`. It models an immutable action that can be performed. It has a name and an optional icon, and can be in either enabled or disabled state. A `SimpleAction` can be converted to a normal Swing `Action` using the `toSwingAction()` method.

An instance of `DockingWindowAction` can create a `SimpleAction` for a docking window. A `DockingWindowAction` have a name and an optional icon. The `DockingWindowAction` class is located in the `net.infonode.docking.action` package along with a number of common docking windows actions, for example `CloseWithAbortWindowAction` and `MinimizeWindowAction`.

Actions can be set on the minimize/maximize/close/restore/undock/dock buttons found in window tabs, `View` title bars and the `TabWindow` tab area. The actions are set using the `ACTION` property found in the `WindowTabButtonProperties` class. Setting this property for a button will override the default action performed when clicking the button. The `WindowTabButtonProperties` class also contains a utility method called `setTo()` which in addition to setting the action also sets the button icon and tooltip text properties to the values found in the action.

16 Heavyweight Components in IDW

It is possible to use both lightweight and heavyweight components as the content in a `View`.

16.1 Important Restrictions

IDW will use heavyweight components internally when heavyweight support is enabled so that the z-order is correct when mixing heavyweight and lightweight components. The drag indication, drag label etc will be heavyweight so that they are displayed above any `View`'s heavyweight component.

There are restrictions when using heavyweight components in IDW:

- Must use Java version 1.5 and above otherwise the z-order of the components will not be correct.
- The heavyweight component must support re-parenting i.e. add to/remove from/re-add to containers. This is what happens when you move around windows.
- The mix of lightweight and heavyweight components will cause flickering during the painting of the components. To minimize flickering it is recommended that continuous layout for split window and window bars are disabled.
- The drag rectangle will be a rectangle with the border thickness specified in the `DRAG_RECTANGLE_BORDER_WIDTH` property in the `RootWindowProperties.` The border color will be the color retrieved by `getColor(...)` from the `ComponentPainter` in the `DRAG_RECTANGLE_SHAPED_PANEL_PROPERTIES` in the `RootWindowProperties.` The border will be opaque i.e. it doesn't support transparency.
- The drag label will be opaque and only support rectangular shapes. Any other shape will be rendered inside the rectangular area.
- The window bars' content area will be opaque and only support rectangular shape. Any other shape will be rendered inside the rectangular area.

16.2 Enabling Heavyweight Support

Heavyweight support is enabled by creating a `RootWindow` with heavyweight support. This is done either via the `RootWindow` constructor or via `DockingUtil.createHeavyweightSupportedRootWindow()`. It is not possible to switch between only lightweight support and heavyweight support once the `RootWindow` has been created.

Creating a RootWindow with heavyweight support:

```
RootWindow rootWindow =
  DockingUtil.createHeavyweightSupportedRootWindow(viewMap, true);
```

Mixing heavyweight and lightweight components can cause flickering when for example resizing windows. Therefore it is recommended that continuous layout for `SplitWindow` and `WindowBar` are disabled.

Disabling continuous layout:

```
// Split window
rootWindow.getRootWindowProperties().getSplitWindowProperties()
  .setContinuousLayoutEnabled(false);

// Window bar
rootWindow.getRootWindowProperties().getWindowBarProperties()
  .setContinuousLayoutEnabled(false);
```

All popup menus and tooltips in Swing are per default lightweight. This mean that they might be shown below the heavyweight components. It is recommended that you tell Swing to disable lightweight popupmenus and tooltips.

Disabling lightweight popups and tooltips in Swing:

```
JPopupMenu.setDefaultLightWeightPopupEnabled(false);
ToolTipManager.sharedInstance().setLightWeightPopupEnabled(false);
```

17 Applets and Web Start Applications

This chapter describes some issues regarding using IDW in applets and web start applications.

17.1 Security Permission

IDW does not require any security permissions for most of its features, for example dragging, docking and undocking. This means that you normally don't need to request any permissions from the user in your JNLP file for your web start application or for your applet. There are two exceptions, hover mechanism and floating window, see below.

17.1.1 Hover

The mouse hover mechanism needs the `AWTPermission` "listenToAllAWTEvents". If the permission is not granted, the hover mechanism will be disabled silently i.e. without any warnings or error messages.

The hover mechanism is not a mandatory part of IDW and all the docking/undocking features will work even if the hover mechanism is disabled.

17.1.2 Floating Windows and Heavyweight Components

Dragging windows between floating windows or from the root window to a floating window can be performed without any security permissions.

When you have a root window with **heavyweight** support enabled, see chapter *16 Heavyweight Components in IDW*, and you drag a window very fast into a floating window the floating window might not detect that a window was dragged into it i.e. no split indication and nothing will happen when you drop the window. You can then drag the window outside of the floating window and then slowly in again and it should work.

If you grant the `AWTPermission` "listenToAllAWTEvents" then the detection should work without any problems.

17.1.3 Requesting and Granting Permission

You need to add a request for permission in your JNLP file for your web start application.

Requesting permission in a web start JNLP file:

```
<security>
  <all-permissions/>
</security>
```

It is also possible to permanently grant the the permission for your installed Java runtime environment (JRE) by modifying the java.policy file, see

http://java.sun.com/j2se/1.5.0/docs/guide/security/PolicyFiles.html, that is usually found in the "*jre/lib/security/*" folder in the Java installation folder.

Granting permission in the java.policy file of your jre:

```
grant {
  permission java.awt.AWTPermission "listenToAllAWTEvents";
};
```

17.2 Floating Windows in an Applet

In a normal application using a `Jframe`, a floating window is always shown on top of the root window. This is not possible in an applet because the top level ancestor in an applet is not a window which means that the z-order between the floating windows and the root window cannot be guaranteed. Therefore the floating windows might be shown below the applet.

When you start a drag of a window, all the floating windows will be brought to front i.e. above the applet. This is not guaranteed to work on all platforms. If a floating window is still below the applet the drag detection will not work correctly for the root window. The root window might think that the dragged window is dragged into the floating window that is below the root window. It is then recommended that you move the conflicting floating window to another location on the screen before you start the drag of the window.

18 IDW Properties Classes

This chapter gives an overview of the IDW properties classes. Only some properties are described in this chapter. For a complete listing of the more than **2400** properties that can be changed in IDW see http://www.infonode.net/documentation/idw/properties/rootwindow.html.

18.1 Overview

Figure 18.1 shows the IDW properties classes and their relationships.

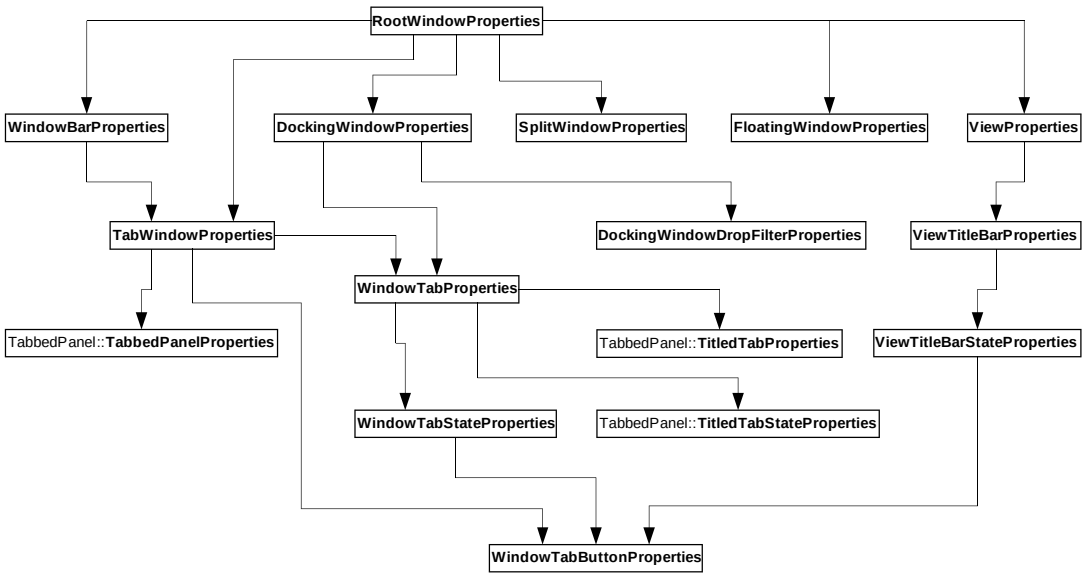

Figure 18.1

IDW properties classes

Each properties class defines a number of properties which are defined as a public static fields. The classes inherit from `PropertyMapContainer` and can be instantiated and act as value containers for its properties. The properties object stores the property values in a `PropertyMap`, see chapter *21 Property Maps* for more information. There is a setter and getter for each property to make it easy to access its value in the `PropertyMap`. Each class also has `addSuperObject()` and `removeSuperObject()` methods which simplifies adding and removing super maps to the internal `PropertyMap`.

The properties classes in *Figure 18.1* are described in the following sections. For each properties class the name, type and description of some important properties in that class are listed.

18.2 RootWindowProperties

`RootWindowProperties` stores property values for a `RootWindow`. When the `RootWindow` of a window is changed the window will add the related property object from the `RootWindowProperties` as super object to its own property object. For example, a `TabWindow` will call `RootWindowProperties.getTabWindowProperties()` and use that object as super object.

Property	Type	Description
`COMPONENT_PROPERTIES`	`ComponentProperties`	The root window component property values.
`WINDOW_AREA_PROPERTIES`	`ComponentProperties`	The window area property values. The window area is the area inside the `WindowBars`.
`DOCKING_WINDOW_PROPERTIES`	`DockingWindowProperties`	Default property values for `DockingWindows` inside this root window.
`TAB_WINDOW_PROPERTIES`	`TabWindowProperties`	Default property values for `TabWindows` inside this `RootWindow`.
`SPLIT_WINDOW_PROPERTIES`	`SplitWindowProperties`	Default property values for `SplitWindows` inside this `RootWindow`.
`FLOATING_WINDOW_PROPERTIES`	`FloatingWindowProperties`	Default property values for `FloatingWindows` inside this `RootWindow`.
`VIEW_PROPERTIES`	`ViewProperties`	Default property values for `Views` inside this `RootWindow`.
`WINDOW_BAR_PROPERTIES`	`WindowBarProperties`	Default property values for `WindowBars` inside this `RootWindow`.
`RECURSIVE_TABS_ENABLED`	`boolean`	If true, makes it possible for the user to create tab windows inside other tab windows when dragging windows. If false, only one level of tab windows is allowed. Changing the value of this property does not alter the window tree. Default value is `true`.

18.3 DockingWindowProperties

`DockingWindowProperties` is used by the `DockingWindow` class and the property values apply to all types of IDW windows. Each window type also has its own `DockingWindowProperties` object where these property values can be specified for a specific window.

Property	Type	Description
TAB_PROPERTIES	WindowTabProperties	Property values for the window tab when the window is located in a TabWindow or a WindowBar.
DROP_FILTER_PROPERTIES	DockingWindowDropFilterProperties	Contains DropFilters for the different kinds of drops supported by the window.

18.4 DockingWindowDropFilterProperties

DockingWindowDropFilerProperties contains property values for setting DropFilters for the different kinds of drop a window supports, see *13 Drag and Drop*.

18.5 TabWindowProperties

TabWindowProperties contains property values for TabWindows and WindowBars.

Property	Type	Description
TABBED_PANEL_PROPERTIES	TabbedPanelProperties	Property values for the tabbed panel in the TabWindow.
TAB_PROPERTIES	WindowTabProperties	Default property values for the window tabs in the TabWindow.
*_BUTTON_PROPERTIES	WindowTabButtonProperties	Property values for the TabWindow buttons.

18.6 WindowBarProperties

WindowBarProperties contains property values for WindowBars.

Property	Type	Description
COMPONENT_PROPERTIES	ComponentProperties	Component property values for the WindowBar.
CONTINUOUS_LAYOUT_ENABLED	boolean	When enabled causes the windows to change size continuously while resizing the WindowBar's content area. Default value is true.

18.7 SplitWindowProperties

SplitWindowProperties contains property values for SplitWindows.

Property	Type	Description
CONTINUOUS_LAYOUT_ENABLED	boolean	When enabled causes the windows to change size continuously while dragging the split window divider. Default value is true.
DIVIDER_SIZE	int	The split pane divider size. Default value is 4.

18.8 FloatingWindowProperties

FloatingWindowProperties contains property values for FloatingWindows.

Property	Type	Description
COMPONENT_PROPERTIES	ComponentProperties	Component property values for the FloatingWindow.
AUTO_CLOSE_ENABLED	boolean	When true the FloatingWindow will automatically close itself when it doesn't contain any child window. Default value is true.

18.9 WindowTabProperties

WindowTabProperties contains property values for window tabs.

Property	Type	Description
TITLED_TAB_PROPERTIES	TitledTabProperties	Property values for the TitledTab used in the window tab.
FOCUSED_PROPERTIES	TitledTabStateProperties	Property values for the TitledTab when the window is focused or a component in the tab's content component has focus. By default the property values from HIGHLIGHTED_PROPERTIES in the TITLED_TAB_PROPERTIES are used.
*_BUTTON_PROPERTIES	WindowTabStateProperties	Property values for the tab buttons in the different tab states.

Change the background color of a window tab to green when the window contains the focus owner:

```
rootWindow.getRootWindowProperties().getTabWindowProperties().getTabProperties().
  getFocusedProperties().getComponentProperties().setBackgroundColor(Color.GREEN);
```

18.10 WindowTabStateProperties

WindowTabStateProperties contains property values for the tab buttons in the different tab states.

18.11 WindowTabButtonProperties

`WindowTabButtonProperties` contains property values for window tab buttons, view title bar buttons and `TabWindow` buttons.

Property	Type	Description
`ICON`	`Icon`	The button icon.
`TOOL_TIP_TEXT`	`String`	The button tool tip text.
`VISIBLE`	`boolean`	True if the button is visible.
`FACTORY`	`net.infonode.gui.button.ButtonFactory`	The button factory. This factory is used to create the button when its first needed. Modifying this property will NOT cause already created buttons to be replaced. The created button will be set to non-focusable and will be assigned the icon from the `ICON` property and the tool tip from the `TOOL_TIP_TEXT` property. An action listener is also added to the button.

Set the close icon for all tab windows and window tabs in a root window:

```
rootWindow.getRootWindowProperties().getTabWindowProperties().
  getCloseButtonProperties().setIcon(icon);

rootWindow.getRootWindowProperties().getTabWindowProperties().getTabProperties().
  getNormalButtonProperties().getCloseButtonProperties().setIcon(icon);
```

18.11.1 Creating a ButtonFactory

You might have noticed that by default all window buttons in IDW are flat without border and has a mouse over highlight. You can replace the default buttons with custom ones by creating your own button factory.

To create button factory implement the `net.infonode.gui.button.ButtonFactory` interface and return an `AbstractButton` object. The window the button should be created for is sent as argument to `ButtonFactory.createButton()`.

Apply your `ButtonFactory` to a `WindowTabButtonProperties` object using the `setButtonFactory()` method. Your `ButtonFactory` will be called when a button is needed. Note that buttons that have been created before you apply your `ButtonFactory` will not be replaced, so it's recommended that you apply the factory just after creating the `RootWindow` and before any windows are added to the `RootWindow`.

After your `ButtonFactory` has been called to create the button, IDW will add a listener to it, set it to non-focusable and set the button icon and tool tip text to the values in the `WindowTabButtonProperties` object.

Create custom close buttons for the window tabs:

```
rootWindow.getRootWindowProperties().getTabWindowProperties().getTabProperties().
  getHighlightedButtonProperties().getCloseButtonProperties().
  setFactory(new ButtonFactory() {
    public AbstractButton createButton(Object object) {
      return object instanceof View ? new JButton("View") : new JButton();
    }
```

```
  });
```

18.12 ViewProperties

`ViewProperties` contains property values for `View`s.

Property	Type	Description
`VIEW_TITLE_BAR_PROPERTIES`	`ViewTitleBarProperties`	Property values for the title bar that can be shown in the `View`.
`ICON`	`Icon`	The view icon.
`TITLE`	`String`	The view title.
`ALWAYS_SHOW_TITLE`	`boolean`	If true the view will always be placed in a `TabWindow` so that its title is shown. Default value is `true`.

18.13 ViewTitleBarProperties

`ViewTitleBarProperties` contains property values for a `View`'s title bar.

Property	Type	Description
`NORMAL_PROPERTIES`	`ViewTitleBarStateProperties`	Property values for the title bar when the `View` is in normal state i.e. not focused.
`FOCUSED_PROPERTIES`	`ViewTitleBarStateProperties`	Property values for the title bar when the `View` has focus.
`VISIBLE`	`boolean`	If true the `View`'s title bar will be visible. Default is `false`.
`HOVER_LISTENER`	`net.infonode.gui.hover.HoverListener`	`HoverListener` that will receive events when the title bar is hovered by the mouse.

18.14 ViewTitleBarStateProperties

`ViewTitleBarStateProperties` contains property values for the title bar title, icon, buttons and for the title bar's look.

Property	Type	Description
ICON	Icon	The title bar icon. Defaults to the View's icon from ViewProperties.
TITLE	String	The title bar title. Defaults to the View's title from ViewProperties.

18.15 TabbedPanel::TabbedPanelProperties

TabbedPanel::TabbedPanelProperties is a properties class found in the ITP library. See [1] for more information.

18.16 TabbedPanel::TitledTabProperties

TabbedPanel::TitledTabProperties is a properties class found in the ITP library. See [1] for more information.

18.17 TabbedPanel::TitledTabStateProperties

TabbedPanel::TitledTabStateProperties is a properties class found in the ITP library. See [1] for more information.

19 Themes

IDW can use themes to customize the look and behaviour. There are several themes included in the `net.infonode.docking.theme` package.

19.1 Creating a Theme

A theme is created by extending `DockingWindowsTheme` found in the package `net.infonode.docking.theme.` A theme has a name and a `RootWindowProperties` object. You can create your own theme by setting properties in the `RootWindowProperties` object.

IDW is per default themeless i.e. no theme is applied. IDW will however still have a default look based on colors, fonts etc for the current look and feel. To make theming simpler so that you do not have to distinguish between themed or not themed a `DefaultDockingTheme` is included. It is basically an empty `RootWindowProperties` object.

19.2 Using a Theme

A theme is applied to a `RootWindow` by adding the `RootWindowProperties` object as super object to the properties object in the `RootWindow`.

Apply /remove the included Shaped Gradient Docking Theme to a root window:

```
DockingWindowsTheme theme = new ShapedGradientDockingTheme();

// Apply theme
rootWindow.getRootWindowProperties().addSuperObject(
  theme.getRootWindowProperties());

// Remove theme
rootWindow.getRootWindowProperties().removeSuperObject(
  theme.getRootWindowProperties());
```

After applying the above theme an application should look something like *Figure 19.1*.

If you want to change to another theme, then you can replace the previous theme with the new theme.

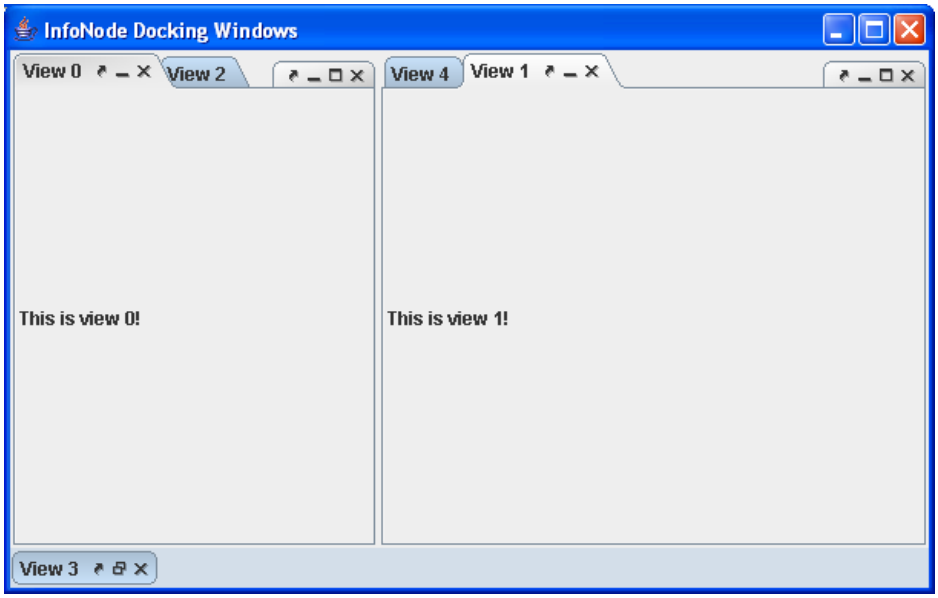

Figure 19.1

Shaped Gradient Docking Theme

Replacing a theme with a new theme:

```
rootWindow.getRootWindowProperties().replaceSuperObject(
  oldTheme.getRootWindowProperties(), newTheme.getRootWindowProperties());
```

19.3 Enabling Title Bar Style

A title bar style is simply a `RootWindowProperties` object that is added as super object to a `RootWindow`'s `RootWindowProperties` object after a `DockingWindowsTheme` has been applied. You could of course set the properties in a `RootWindowsProperties` object yourself and enable view title bar, changing tab area orientation, tab layout etc.

To make it more simple the class `PropertiesUtil` in the package `net.infonode.docking.util` has a method that will create such a `RootWindowProperties` object for you. The object is primarily meant to be used in companion with the included themes.

Enable/disable title bar style to the apllication seen in Figure 19.1:

```
DockingWindowsTheme theme = new ShapedGradientDockingTheme();
rootWindow.getRootWindowProperties().addSuperObject(
  theme.getRootWindowProperties());

RootWindowProperties titleBarStyleProperties =
  PropertiesUtil.createTitleBarStyeRootWindowProperties();

// Enable title bar style
rootWindow.getRootWindowProperties().addSuperObject(
  titleBarStyleProperties);

// Disable title bar style
rootWindow.getRootWindowProperties().removeSuperObject(
  titleBarStyleProperties);
```

The result after enabling title bar style is shown in *Figure 19.2*.

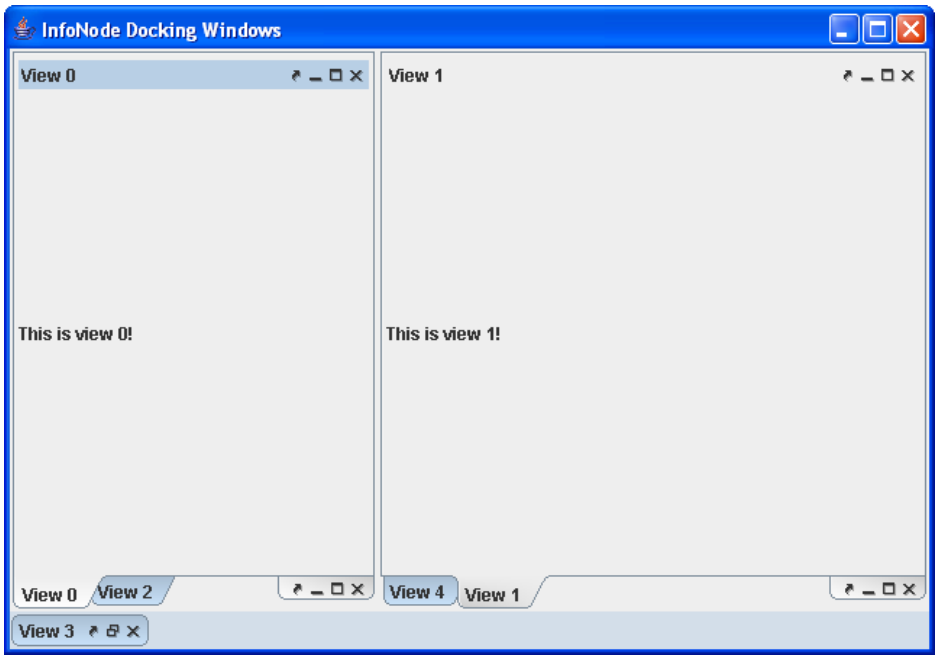

Figure 19.2

Shaped Gradient Docking Theme with title bar style enabled

If you want to change theme and still use title bar style then you only need to replace the old theme's `RootWindowProperties` object with the new theme's `RootWindowProperties` object i.e. there is no need to add/remove the title bar style `RootWindowProperties` object.

19.4 Look and Feel Docking Theme

The Look and Feel Docking Theme is an **experimental** theme that tries to replicate the Swing `JTabbedPane`'s look for the tab windows and the Swing `JInternalFrame`'s title bar look under the active look and feel. This may or may not work depending on the look and feel that is beeing used. It is wise to test if the theme works well together with the look and feel that is going to be used in the application.

The theme tries to replicate the look for the tab window's content area and tabs including hover effects. It will also try to replicate the look for the view title bar based on the `JInternalFrame` title bar look. The buttons such as scroll buttons, title bar buttons etc are not replicated.

The theme uses heavyweight components internally and therefore the theme must be disposed when it is no longer needed.

Disposing the Look and Feel Docking Theme:

```
// First remove look and feel theme properties
rootWindow.getRootWindowProperties()
```

```
  .removeSuperObject(lookAndFeelDockingTheme.getRootWindowProperties());

// Dispose the look and feel docking theme
lookAndFeelDockingTheme.dispose();
```

The theme uses the hover mechanism for tab mouse hover effects, see *17 Applets and Web Start Applications*.

There's no support given for this theme. The theme may change, be removed etc in future versions of InfoNode Docking Windows. It's important that you test that the theme is working with the look and feel you intend to use. If the look and feel for example doesn't render its `JTabbedPane` tabs correctly the same problem will be seen using this theme.

20 Properties

This chapter is included in both the *"InfoNode Docking Windows Developer's Guide"* and *"InfoNode Tabbed Panel Developer's Guide"*.

The InfoNode properties module is located in the `net.infonode.properties` package. The properties module is NOT thread safe, so only one thread at a time should call methods inside the module. IDW and ITP always use the AWT event dispatching thread when calling methods in the properties module. InfoNode properties are similar to Java bean properties, but they are not connected to Java class and method names.

20.1 Property

A `Property` is similar to a Java class field. It has a name, a type and a description. It can belong to a `PropertyGroup`. The value of a `Property` can be any type of object and the value can be stored in any type of object, for example a `PropertyMap` (see chapter *21 Property Maps*). To set the value of a `Property` in an object use the `Property.setValue()` method. To get the value of a Property from an object use the `Property.getValue()` method.

Some property value containers support removing of a property value with the `Property.removeValue()` method. To check if a property value can be removed use the `Property.valueIsRemovable()`.

A `Property` is type checked, so it can only be assigned values that are compatible with the type of the `Property`.

20.2 PropertyGroup

A `PropertyGroup` is similar to a Java class. It is a collection of `Property`s and has a name, a description and an optional parent group.

20.3 Typed Properties

The `net.infonode.properties.types` package contains a number of `Property` classes which have values that are instances of common Java classes. For example, the `IntegerProperty` class stores values that are instances of the `Integer` class.

20.4 PropertyValueHandler

The typed property classes will take a `PropertyValueHandler` object as constructor parameter. The `PropertyValueHandler` is responsible for storing property values in an object. As mentioned before property values can be stored in any type of object.

21 Property Maps

This chapter is included in both the *"InfoNode Docking Windows Developer's Guide"* and *"InfoNode Tabbed Panel Developer's Guide"*.

The property map module is a part of the InfoNode properties module. A property map is a powerful container for `Property` values.

21.1 Property Map Classes

The property map classes are located in the `net.infonode.properties.propertymap` package. *Figure 21.1* shows the property map classes and their relationships.

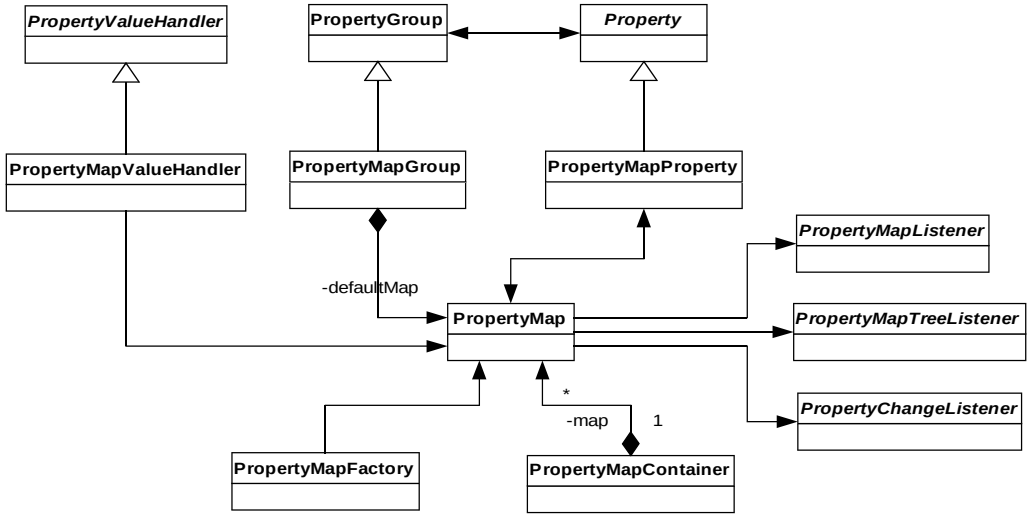

Figure 21.1

Property map classes.

The classes are described in the following sections.

21.1.1 PropertyMap

A `PropertyMap` is a map for `Property` values. `PropertyMap` objects can be created using the methods in `PropertyMapFactory`. You can either create a `PropertyMap` for a `PropertyMapGroup` or a `PropertyMap` that has another `PropertyMap` as super map and the same `PropertyMapGroup` as the super map.

Use the `Property.setValue()` and `Property.getValue()` methods to store and retrieve values in a `PropertyMap`. `PropertyMap` supports removal of property values with the `Property.removeValue()` method.

21.1.2 PropertyMapGroup

A `PropertyMapGroup` is a special `PropertyGroup` which uses `PropertyMap`s as property value storage. Each `PropertyMapGroup` has a `PropertyMap` which contains default values for the `Property`s in the group. These values are returned when a value is not found in a `PropertyMap`. You can access and modify the default values with `PropertyMapGroup.getDefaultMap()`. Note that modifying the default values will NOT trigger listener notifications in other `PropertyMap`s, so the default values should be set before any other `PropertyMap`s are created, for example in a static initializer.

21.1.3 PropertyMapValueHandler

`PropertyMapValueHandler` is used for handling property values stored in a `PropertyMap`.

21.1.4 PropertyMapProperty

A `PropertyMapProperty` is a property that has `PropertyMap`s as values. These properties are automatically assigned a value which cannot be modified.

21.1.5 PropertyMapFactory

Creates `PropertyMap` objects.

21.1.6 PropertyMapContainer

A utility class that is used as base class for properties classes that uses a `PropertyMap` for property value storage.

21.2 Advanced Features

In this section some of the more advanced features of the `PropertyMap` class is described. These features are most easily described with examples using pseudo code. Here are the definitions of the `PropertyMapGroup`s used in the examples:

- `Point` is a `PropertyMapGroup` which contains two `IntegerProperty`s, `X` and `Y`.
- `Line` is a `PropertyMapGroup` which contains two `PropertyMapProperty`s, `Start` and `End`, of type `Point`.

The figures in the examples uses normal arrows with a number to indicate super maps, the number is the search order, and arrows with a diamond to indicate map composition.

21.2.1 Super Maps

A `PropertyMap` can have a number of super maps. If a property value isn't found in a `PropertyMap`, the last added super map is searched for the property value. If it isn't found there, the next super map is searched and so on.

A super map is added using `PropertyMap.addSuperMap()`, and removed using `PropertyMap.removeSuperMap()`. A super map can be replaced using the `PropertyMap.replaceSuperMap()` method.

An example:

1. `P1 = new Point`
2. `P1.X = 1, P1.Y = 2`
3. `P2 = new Point`
4. `P2.Y = 3`
5. `P1` is added as super map to `P2`. Now `P2.X == 1`, the value is found in `P1`, and `P2.Y == 3`, the value is found in `P2`.

21.2.2 Property Map Composition

When a `PropertyMap` is created for a `PropertyMapGroup` that contains `PropertyMapProperty`s, each `PropertyMapProperty` is assigned a new `PropertyMap` as value. Getting the property value will return this `PropertyMap`. The property value is constant and can't be modified.

When a super map, A, is added to a `PropertyMap`, B, which contains `PropertyMap` values, the corresponding `PropertyMap` value in the A is added as super map to each `PropertyMap` value in B. The super map is added after, and thus it's values are overridded by, any super maps explicitly added to a `PropertyMap` value in B. The process is recursively repeated. The reverse operation is performed when a super map is removed.

For example:

1. `L1 = new Line`
2. `P1 = new Point`
3. `P1.X = 1`
4. `P1` is added as super map to `L1.Start`, which means that `L1.Start.X == 1`

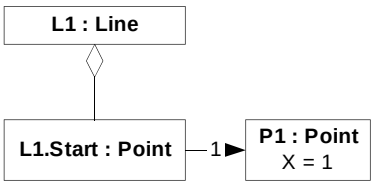

Figure 21.2

The `PropertyMap` hierarchy after step 4

5. `L2 = new Line`

6. `L2.Start.X == 2, L2.Start.Y == 3`
7. `L2` is added as super map to `L1`. This causes `L2.Start` to be added as super map to `L1.Start`, but the values in `P1` overrides the values in `L2.Start`. So now `L1.Start.X == 1` and `L1.Start.Y == 3`.

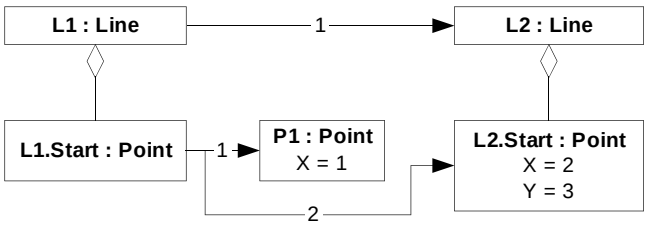

Figure 21.3

The `PropertyMap` hierarchy after step 7

21.2.3 Property Value References

Using `PropertyMap.createRelativeRef()` you can create a property value that is a reference to another property value. The reference is inherited just like normal property values, but its actual value may differ depending in which `PropertyMap` the property value is read. The reference is dereferenced relative to the `PropertyMap` where the property value is read.

Here's an example on how it works:

1. `L1 = new Line`
2. `L1.End.Y = 1`
3. A relative reference is created from `L1.Start.X` to `L1.End.Y`. So, `L1.Start.X == 1`

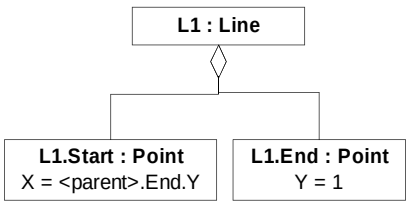

Figure 21.4

The `PropertyMap` hierarchy after step 3

4. `L2 = new Line()`
5. `L1` is added as super map to `L2`. Now `L2.Start.X == 1`, because:

1. No value for `X` is found in `L2.Start`, so `L1.Start` is searched and the reference is found.
2. The reference is dereferenced relative to `L2` which causes a lookup for the value of `L2.End.Y`.
3. No value for `Y` is found in `L2.End`, so the value in `L1.End` is returned.

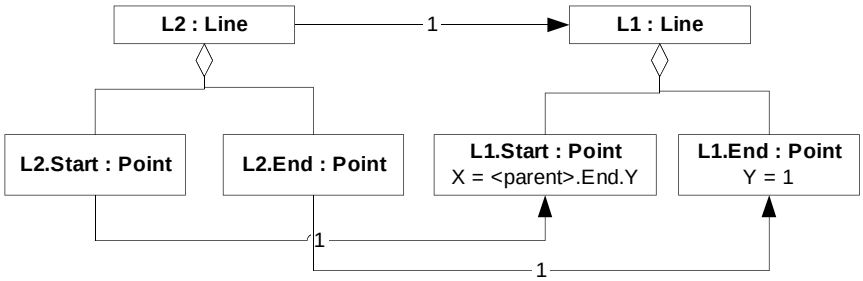

Figure 21.5

The `PropertyMap` hierarchy after step 5

6. `L2.End.Y = 2`. Now `L2.Start.X == 2` because the reference is inherited to `L2` and dereferenced relative to `L2`. Still `L1.Start.X == 1` though.

21.2.4 Listeners

Three types of listeners can be added to a `PropertyMap`:

`PropertyMapListener`	Listens to value changes in the `PropertyMap`, but not changes in child maps. Value changes for multiple properties can be bundled together in a single listener notification.
`PropertyMapTreeListener`	Listens to value changes in the `PropertyMap` and all its child maps recursively. Value changes for multiple properties can be bundled together in a single listener notification.
`PropertyChangeListener`	Listens for a value change for a specific property in a `PropertyMap`.

The listeners are notified when the `Property.getValue()` with the `PropertyMap` would return a different value than before, not only when `Property.setValue()` is called for the `PropertyMap`. This means that when a property value that is referenced is modified listener notifications will be triggered in all maps that reference the value. For example, the listeners of a `PropertyMap` will be notified if a the map contains no value for a property and the value of that property is changed in a super map, or if a new super map where this value is set is added to the `PropertyMap`.

21.2.5 Weak Listeners

A weak listener is a listener that are garbage collected and removed from the `PropertyMap` it listens to when there are no strong or soft references to the listener. Weak listeners are useful for example when you want to listen to a PropertyMap, but you don't want the listener to prevent the map from being garbage collected.

The `PropertyMapWeakListenerManager` class handles weak listeners and contains methods for adding the three listener types as weak listeners to a `PropertyMap`. You must use the remove methods in `PropertyMapWeakListenerManager` when removing a previously added weak listener.

21.2.6 Batch Processing

The `PropertyMapManager.runBatch()` method can be used minimize the number of listener notifications when performing multiple property value changes. All property value modifications done inside `run()` of the `Runnable` passed to `PropertyMapManager.runBatch()` will be bundled together and sent to listeners when the run() method returns. No listener notifications are performed before that. Each listener is guaranteed to be called at most once for each batch operation.

Example setting two property values:

```
PropertyMapManager.runBatch(new Runnable() {
  public void run() {
    X.set(p1, 4);
    Y.set(p1, 5);
  }
});
```

21.2.7 Serialization

A `PropertyMap` can be written to an `ObjectOutputStream` using the `PropertyMap.write()` methods. If the recusive flag is set, all the `PropertyMap`s of all `PropertyMapProperty`s will be written recursively. In the stream the `Property`s are identified by their name. The property values are written using normal serialization. Property values that doesn't implement the `Serializable` interface will not be written to the stream.

A previously written `PropertyMap` is read using the `PropertyMap.read()` method. If the map was written recusively, it is read recusively as well. If a name for a non-existing `Property` is found in the stream the value is skipped. Not all `Property`s must have a value in the stream.

21.3 ComponentProperties

The `net.infonode.properties.gui.util.ComponentProperties` class contains properties common to all `JComponent`s. The property values can be applied to a `JComponent` using the `applyTo()` methods.