

# DockingFrames

Insights of version BETA 1.0.0

by Beni

## Table of Contents

1 Basics.....	2
1.1 The tree.....	2
1.1.1 Leaf: Dockable.....	2
1.1.2 Node: DockStation.....	2
1.1.3 Root: DockController.....	3
1.2 Making life simple: DockFrontend.....	4
2 Load & Save.....	4
2.1 Local layout: DockableProperty.....	4
2.2 Global layout: DockSituation.....	5
2.3 DockFrontend.....	5
3 Drag & Drop.....	5
3.1 The default behavior.....	5
3.2 Invoking the mechanism.....	6
3.3 Merging two Dockables.....	6
3.4 Constraints for combinations.....	6
4 Decorations.....	6
4.1 Title.....	6
4.1.1 Instantiating a DockTitle.....	6
4.2 Actions.....	9
4.2.1 Types of actions and their use.....	9
4.2.2 How to apply a DockAction.....	10
4.2.2.1 Dockable.....	10
4.2.2.2 ActionGuard.....	11
4.2.2.3 ActionOffer.....	11
4.2.2.4 parent DockStations.....	11

# 1 Basics

This chapter will present concepts that are needed to use DF as client.

## 1.1 The tree

DF is roughly organized as a tree. Clients have to create the elements of the tree (root, nodes, leaves), and users will be able to change the layout of the tree by simple drag&drop operations.

Some of the elements of the tree are part of the graphical user interface, others are not.

### 1.1.1 Leaf: Dockable

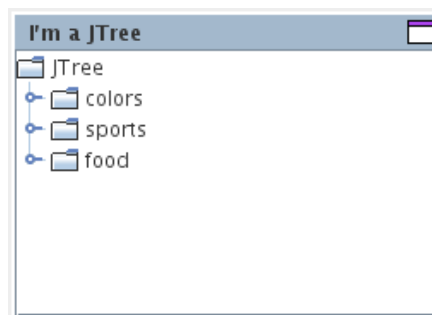
A Dockable is a graphical component like a JPanel, and a set of properties like an icon or title-text.

Clients will normally instantiate a DefaultDockable, and use its “content pane” to add their content to the graphical user interface.

A small example might look like this:

```
1: DefaultDockable dockable = new DefaultDockable();
2: dockable.setTitleText( "I'm a JTree" );
3: dockable.getContentPane().setLayout( new GridLayout( 1, 1 ) );
4: dockable.getContentPane().add( new JScrollPane( new JTree() ) );
```

The result would then be a Dockable of this form:



### 1.1.2 Node: DockStation

A Dockable alone is as usefull as a JButton without a Window to show it. Every Dockable needs a DockStation as parent. The DockStation knows where and how to show its children. A DockStation can be a Dockable by itself, so a tree of DockStations and Dockables can be built.

This codesnippet shows how some specific implementations of Dockable and DockStation can be combined.

```
1: SplitDockStation split = new SplitDockStation();
2: StackDockStation stack = new StackDockStation();
3:
4: stack.setTitleText( "Stack" );
5: stack.drop( new DefaultDockable( "One" ) );
6: stack.drop( new DefaultDockable( "Two" ) );
7:
8: split.drop( stack, SplitDockProperty.WEST );
9: split.drop( new DefaultDockable( "Three" ), SplitDockProperty.EAST );
```

What happens here?

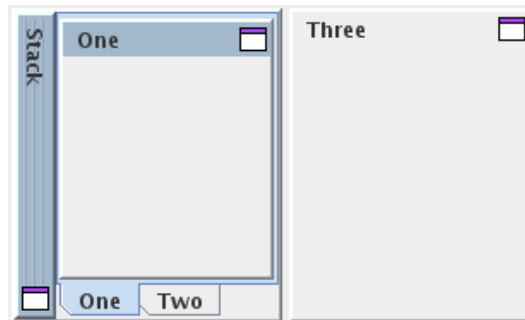
In line 1 and 2, two DockStations are created.

A StackDockStation is also a Dockable, so in line 4 we set the title, and in line 9 we set the parent of the station.

In line 5 and 6, we add some children to the StackDockStation.

In line 8 and 9, we add some children to the `SplitDockStation`. The constants `WEST` and `EAST` are a hint for the `SplitDockStation`, where to put the new children.

The result of this code will produce this layout:



There are different `DockStations`:

- `StackDockStation`: organizes its children in a stack, like a `JTabbedPane`
- `SplitDockStation`: all children are put alongside with each other. A gap between the children allows the user to change the sizes of the children.
- `FlapDockStation`: shows a set of buttons, one for each child. The user can press one of the buttons, and a window will open showing the selected child.
- `ScreenDockStation`: creates a new window for each child. These windows float above the main application window.

### 1.1.3 Root: `DockController`

The `DockController` is the center of DF. He is responsible for “global” actions. The `DockController` seldomly does something by himself, but he knows where to find an object that can do its work. Every `DockController` has its own realm. There can be many `DockControllers` in one application, however they can't interact with each other. Normal applications will need only one `DockController`.

There is one method that needs to be mentioned: “`setSingleParentRemove`”. Clients can activate a mechanism that will cut out unnecessary nodes from the tree. This mechanism is per default offline, because in some applications removing nodes without asking would lead to a very strange behaviour.

Now follows a fully functional application showing how the whole tree is built:

```
1: public static void main( String[] args ){
2:     DockController controller = new DockController();
3:     controller.setSingleParentRemove( true );
4:
5:     SplitDockStation station = new SplitDockStation();
6:     controller.add( station );
7:
8:     station.drop( new DefaultDockable( "One" ) );
9:     station.drop( new DefaultDockable( "Two" ), SplitDockProperty.NORTH );
10:    station.drop( new DefaultDockable( "Three" ), SplitDockProperty.EAST );
11:
12:    JFrame frame = new JFrame();
13:    frame.add( station.getComponent() );
14:
15:    frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
16:    frame.setBounds( 20, 20, 400, 400 );
17:    frame.setVisible( true );
18: }
```

What happens here?

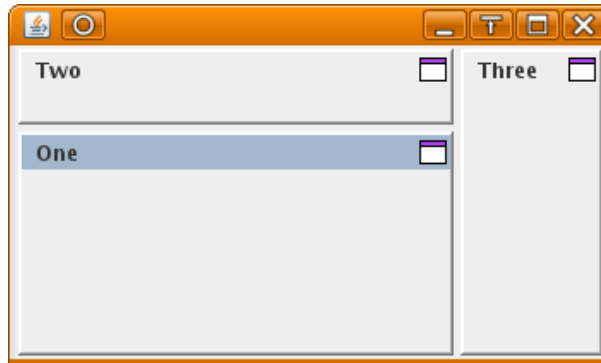
In line 1 and 2, the DockController is created and its properties set.

In line 5 and 6, a DockStation is instantiated, and added to the DockController. It's imperative to register the root-DockStations to the DockController, otherwise the tree could not be completed.

In lines 8 to 10, some Dockables are added to the DockStation.

In line 13 is the graphical representation of the DockStation added to a JFrame.

The resulting application will look like this:



## 1.2 Making life simple: DockFrontend

Instead of using a DockController, clients can also use a DockFrontend. The DockFrontend has the ability to do some of the common tasks which almost every application would like to do.

Clients can register the root-DockStations through the “addRoot”-method. They have to provide a unique name for every DockStation.

Also some Dockables can be registered through the “add”-method. Registered Dockables will then have a little button in their title, with which the user can close (or “hide”) the Dockable.

Another nice ability of DockFrontend is, to read or write the whole layout of the application from or into a DataInput/OutputStream (see the methods “read”, “write”, “save” and “load”).

## 2 Load & Save

DF has tools to save and load the layout (the structure of the tree, the location and size of the Dockables, etc.).

### 2.1 Local layout: DockableProperty

Local layout means the location of exactly one element of the tree. Local information is very independent from the rest of the tree, however local information is never complet. So sometimes there is not enough information to recreate a layout later on.

Local information can be usefull to store the location of a Dockable before it is removed from the tree, and than later to add it again to the tree. Local information can't be used to store the location of all Dockables, then close the application and later restart the application.

DF can provide local information through a DockableProperty. The following example shows how the DockableProperty can be read, and applied:

```
1: DockStation root = DockUtilities.getRoot( dockable );
2: DockableProperty location = DockUtilities.getPropertyChain( root, dockable );
3:
```

```
4: dockable.getDockParent().drag( dockable );
5:
6: root.drop( dockable, location );
```

In line 1 we get the root of the subtree containing a Dockable “dockable”.

In line 2 the location of “dockable” is read.

Then in line 4 the Dockable is removed from the tree.

And in line 6 it is reinserted into the tree hopefully at the same location.

Note that the method “drop” returns a boolean indicating whether the operation was a success or not.

DockableProperties can be stored using a PropertyTransformer.

## **2.2 Global layout: DockSituation**

Global layout means the complete layout of a whole tree.

The class DockSituation can be used to store the global layout. It has a method named “write” which requests a map containing all root-DockStations and a stream to write into. The DockSituation will use a DockFactory to store the elements of the tree. Every element has a property “factoryID” which will be used to identify the appropriate factory.

The method “read” is the exact reverse of “write”. The method does creates a new instance of every element in the tree.

Of course there are some drawbacks. For one, every Dockable with different content needs its own factory. And creating new instances of elements is counterintuitive for elements that should only exist once.

So there is another class named “PredefinedDockSituation”. This class has a method called “put” which requests an element of the tree and optionally a unique id for the element. Later on, these elements will be used instead of instantiating new elements. Note that clients do not have to register factories for the elements which were “put” to the PredefinedDockSituation.

In the case that not all elements should be stored, a DockSituationIgnore can be applied to a DockSituation (using the method “setIgnore”). The DockSituationIgnore will filter out those elements which are not to be stored.

## **2.3 DockFrontend**

Developers using a DockFrontend might have a look at the “read”, “write”, “load” and “save”-methods.

# **3 Drag & Drop**

## **3.1 The default behavior**

The drag & drop operation normally is handled by the DockRelocator (which is created by the DockController).

The DefaultDockRelocator registers Mouse- and MouseMotionListeners to several Components, most importantly to the titles of the Dockables.

When the user initiates a mouse-drag operation, the DefaultDockRelocator first shows a window containing the title of the Dockable. Then he uses the knowledge of the DockController to find a DockStation which might become the parent of the dragged Dockable. The methods “prepareMove” or “prepareDrop” of DockStation are used for that.

When the user releases the mouse, the methods “drop” or “move” of the next parent is called. Then the operation is finished.

### **3.2 Invoking the mechanism**

There is a possibility to remotely steer a drag & drop operation. Clients can use the methods “createRemote” or “createDirectRemote” of DockRelocator to get an object which can control the DockRelocator.

The RemoteRelocator (result of “createRemote”) is intended for a connection with a Mouse-/MouseMotionListener.

On the other hand, the DirectRemoteRelocator is intended for a usage that has no connection to the (real) mouse.

### **3.3 Merging two Dockables**

When the user lays two Dockables above each other, they are merged. Most DockStations will use a Combiner to create a new Dockable consisting of the old ones. If not told otherwise, the DockStation will ask the DockTheme (using “getTheme” of DockController) of its DockController to provide a Combiner (using “getCombiner” of DockTheme).

Clients might exchange the Combiner of the DockStation itself, or of the DockTheme, in order to create new effects when merging occurs.

### **3.4 Constraints for combinations**

Not every relationship is a good one. Sometimes a Dockable just don't should be a child of some DockStation.

There are several mechanism to prevent unhealthy relationships:

- Every Dockable has a methods called “accept”.
- Every DockStation has a method “accept”.
- And finally, the DockController can carry a whole set of DockAcceptances (see “addAcceptance” in DockController). Every DockAcceptance has methods called “accept”.

Whenever a drag&drop operation is about to end, all these “accept”-methods must agree to the new relationship.

## **4 Decorations**

DF is part of a graphical user interface, so there have to be some thoughts about eye-candy.

### **4.1 Title**

A DockTitle is a component linked to exactly one Dockable, showing some properties of its Dockable.

The default-behaviour would be to show the icon, the text and the DockActions of the linked Dockable, but various implementations can show whatever they like.

#### **4.1.1 Instantiating a DockTitle**

Every Dockable has a method “getDockTitle”. A client should use that method to create a new DockTitle when necessary. Since there are different versions of DockTitles for different purposes,

the method “getDockTitle” needs a DockTitleVersion.

A DockTitleVersion is a unique identifier telling which kind of DockTitle a method should produce. DockTitleVersions should be created by the DockTitleManager (see “getTitleManager” of DockController), using one of the “register”-methods.

The “register”-methods of DockTitleManager need a string which is a unique id, and a DockTitleFactory. A DockTitleFactory simply creates DockTitles. There are different “register”-methods, each representing another priority for the factory. If two factories are registered under the same key, then the one with the higher priority will be used. The highest priority is “client”, the lowest “default”, and “theme” is in the middle.

By the way, there is a special DockTitleFactory called “ControllerTitleFactory”. This factory forwards every call to the default-factory provided by the DockTheme (see “getTheme” of DockController and “getTitleFactory” of DockTheme). This factory should be used, whenever the very basic DockTitle is needed.

This rather complex way of creating a DockTitle allows developers to create either individual DockTitles for each Dockable (by overriding “getDockTitle” of Dockable) or to make global changes (by registering new DockTitleFactories at the DockTitleManager).

The one object that asked for the DockTitle needs to call the “bind”-method of Dockable in order to connect the DockTitle with its Dockable. The method “unbind” should be called when a DockTitle is no longer used. This gives the DockTitle the opportunity to remove listeners.

Let's write down the steps necessary to use a DockTitle again:

- Get a DockTitleVersion from the DockTitleManager as early as possible. Use the appropriate “register”-method, normally it would be “registerDefault”.
- Call “getDockTitle” of Dockable to create a new DockTitle. Use the DockTitleVersion of the last step.
- Use the method “bind” of Dockable to connect the DockTitle to its Dockable.
- Use the method “unbind” of Dockable to remove the connection of the last step.

**Note:** never ever invoke “bind” or “unbind” of DockTitle directly, these methods are intended to be used by the DockController only.

Let's have a look at this example:

```
1: public class OpenViewList extends DefaultDockable{
2:     private JPanel panel = new JPanel();
3:     private Map<Dockable, DockTitle> titles =
4:         new HashMap<Dockable, DockTitle>();
5:     private DockTitleVersion version;
6:
7:     public OpenViewList( DockController controller ){
8:         setTitleText( "Open views" );
9:         version = controller.getDockTitleManager().registerDefault(
10:             "open view", new ControllerTitleFactory() );
11:
12:         getContentPane().setLayout( new GridLayout( 1, 1 ) );
13:         getContentPane().add( new JScrollPane( panel ) );
14:
15:         controller.getRegister().addDockRegisterListener( new DockAdapter(){
16:             @Override
17:             public void dockableRegistered(
18:                 DockController controller, Dockable dockable ){
19:
20:                 DockTitle title = dockable.getDockTitle( version );
21:                 if( title != null ){
22:                     title.setOrientation(
```

```

23:         Orientation.FREE_HORIZONTAL );
24: dockable.bind( title );
25: titles.put( dockable, title );
26: panel.add( title.getComponent() );
27: panel.setLayout( new GridLayout(
28:     panel.getComponentCount(), 1, 2, 2 ) );
29: panel.revalidate();
30:     }
31: }
32:
33: @Override
34: public void dockableUnregistered(
35:     DockController controller, Dockable dockable ){
36:
37:     DockTitle title = titles.remove( dockable );
38:     if( title != null ){
39:         dockable.unbind( title );
40:         panel.remove( title.getComponent() );
41:         panel.setLayout( new GridLayout(
42:             panel.getComponentCount(), 1, 2, 2 ) );
43:         panel.revalidate();
44:     }
45: }
46: });
47: }
48: }

```

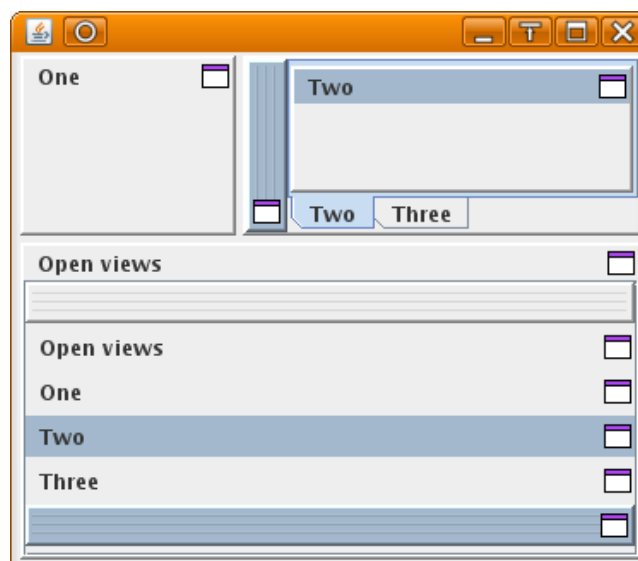
What's happening here? This is a Dockable showing a DockTitle for each Dockable that was added to the tree after it was instantiated.

On line 9, the DockTitleVersion is created.

In lines 20 to 29 a DockTitle is added, in line 20 the DockTitle is created, in line 24 the DockTitle is connected to its Dockable.

In lines 35 to 43 a DockTitle is removed, in line 39 the connection between the DockTitle and its Dockable is broken.

The screenshot shows how an application using three normal Dockables and the OpenViewList might look like. The first, empty DockTitle belongs to the SplitDockStation which is the root of the subtree containing visible elements.





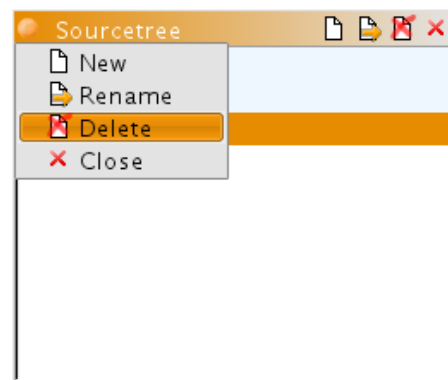
## 4.2 Actions

A DockAction is something that has a graphical representation, and reacts somehow when the user clicks onto this representation.

Every Dockable has a set of DockActions, some of them can be set by the developer, others are added by DF.

DockActions are normally shown on a DockTitle, but other locations are possible, like a drop-down-menu.

Have a look at the image. There are four DockActions associated with the Dockable “Sourcetree”. The DockActions are shown as buttons in the top right corner, and as menu-items in the JMenu at the top left corner.



There are different types of DockActions, and there are different views:

A ActionType describes how a DockAction works, a ViewTarget where the DockAction will be shown. An ActionType might be something like “BUTTON”, describing a DockAction which always does the same action when triggered. A ViewTarget might be something like “MENU” to tell that a DockAction is shown as menu-item in a menu.

ActionTypes and ViewTargets create a table where each cell tells how to create the graphical representation of a specific type of DockAction to be displayed on a specific target.

This table is known as the ActionViewConverter (see “getActionViewConverter” of DockController). Clients might introduce new ActionTypes or ViewTargets using this converter.

### 4.2.1 Types of actions and their use

The interface “DockAction” is rather simple and only contains a method to create a view of the DockAction for a specific ViewTarget.

Clients surely can implement this interface, but then they have to create their own views. It's better to use one of the existing sub-interfaces, since there are already views for them. All of these DockActions have icons, text and tooltips:

- **ButtonDockAction:** represents a button. The user clicks onto a button, and the action always executes the same piece of code.
- **MenuDockAction:** A very simple menu, the content of the menu is a DockActionSource, containing other DockActions. Normally the user can click onto a button, and a drop down menu opens where he can choose which other DockAction to execute.
- **DropDownAction:** A more sophisticated menu. The behavior is almost the same as MenuDockAction, but the content (icon, text, ...) of the last invoked DockAction is shown as content of the DropDownAction.

- **SelectableDockAction:** A DockAction which has two states, “selected” and “not selected”. Clicking on the graphical representation might change this state. There are often two implementations of this DockAction, either as “CHECK” (state changes always when clicking) or as “RADIO” (state changes to “selected” when clicking, but has to be deselected by another action).

There are two sets of classes implementing the above interfaces.

- **SimpleButtonAction, SimpleMenuAction, SimpleDropDownAction, SimpleSelectableAction.Check, SimpleSelectableAction.Radio:**  
These classes provide a single property for icon, text, etc. Several Dockable can share one of these DockAction, and the user will always experience the same behavior. SimpleDockActions are enough for most applications.
- **GroupedButtonAction, GroupedSelectableAction.Check, GroupedSelectableAction.Radio:**  
These classes provide a map for each property. Every Dockable is then mapped to a key, which is used to read the informations from the maps. Different Dockables may share a key, and the key for a Dockable can be changed at any time.  
In the end, it's like having several SimpleDockActions, and exchanging them whenever the key is exchanged.

## 4.2.2 How to apply a DockAction

The question remains, how a DockAction can be injected into the framework.

Whenever some view wants to show the DockActions for a Dockable, the view calls the method “listActionOffers” of DockController. This method collects the DockActions from various sources:

- The Dockable itself
- All the parent DockStations of the Dockable
- The ActionGuards
- The ActionOffers

### 4.2.2.1 Dockable

Since DockActions are bound to a Dockable, its natural that every Dockable contains a list (a DockActionSource) of DockActions. Clients should set their own list of DockAction when a new Dockable is created. Replacing the list later on will not have any effect, however changing the content of the list will be noted immediately.

This codesnipet shows how a DockAction can be assigned to a DefaultDockable:

```
1: DefaultDockable dockable = new DefaultDockable( "One" );
2:
3: DefaultDockActionSource source = new DefaultDockActionSource();
4: dockable.setActionOffers( source );
5:
6: SimpleButtonAction action = new SimpleButtonAction();
7: source.add( action );
8: action.setIcon( ... );
9: action.setText( ... );
```

In line 1, a new Dockable is created.

In lines 3 and 4, the list of DockActions is created and assigned.

In lines 6 to 9, a new DockAction is created, added to the list of DockActions, and some properties

of the DockAction are changed.

#### **4.2.2.2 ActionGuard**

If a more “global approach” is necessary, an ActionGuard can be used. An ActionGuard is added to the DockController (see “addActionGuard” of DockController), and asked for a list of DockActions whenever someone wants to show the DockActions of a Dockable.

This is an example how an ActionGuard might look like.

```
1: public class Guard implements ActionGuard{
2:     public boolean react( Dockable dockable ) {
3:         return "One".equals( dockable.getTitleText() );
4:     }
5:
6:     public DockActionSource getSource( Dockable dockable ) {
7:         DefaultDockActionSource source = new DefaultDockActionSource();
8:         source.setHint( new LocationHint(
9:             LocationHint.ACTION_GUARD, LocationHint.MIDDLE ) );
10:        source.add( ... );
11:        return source;
12:    }
13: }
```

The method “react” in line 2 to 4 tells whether a Dockable should receive DockActions from this guard or not. In this case, only Dockables with a title “One” should get new DockActions.

The method “getSource” in lines 6 to 12 then creates a list of DockActions for those Dockables which were selected by “react”.

The LocationHint created in lines 8 and 9 tells the views which show more than just one DockActionSource, where the list should be placed relatively to the other lists.

#### **4.2.2.3 ActionOffer**

Yet another way is using an ActionOffer. An ActionOffer is responsible to collect the DockActions for a Dockable from various sources. Of course, an ActionOffer could insert its own DockActions while doing that. ActionOffers are added to the DockController through “addActionOffer”, and only one ActionOffer can serve a Dockable.

#### **4.2.2.4 parent DockStations**

Last but not least, every DockStation has methods “getDirectActionOffers” and “getIndirectActionOffers”. These methods return DockActions either for the direct or for all children of the DockStation.