JFormDesigner 3.1.1 Manual

Version: 3.1.1

Contents

JFormDesigner

Introduction

JFormDesigner is an innovative GUI designer for Java Swing user interfaces. Its outstanding support for JGoodies FormLayout, Clearthought's TableLayout and GridBagLayout makes it easy to create professional looking forms.

JFormDesigner is available in three editions: as stand-alone application and as IDE plug-ins for Eclipse and IntelliJ IDEA. This documentation covers all editions. If there are functional differences between the editions, they are marked with: **Stand-alone**, **Eclipse plug-in**, **IntelliJ IDEA plug-in**, **JBuilder plug-in** or **IDE plug-ins**.

Key features

Easy and intuitive to use, powerful and productive	JFormDesigner provides an easy-to-use but powerful user interface. Easily drag and drop components, resize components using the handles, set properties, etc. Powerful features like in-place-editing, keyboard navigation, automatic component ordering (for grid based layout managers), IntelliGap, auto-insert columns/rows, drag and drop of columns/rows, bean morphing, layout manager changing increase your productivity.
IDE plug-ins and stand-alone application	JFormDesigner is available as IDE plug-ins for Eclipse, IntelliJ IDEA and JBuilder and as stand-alone application.
JGoodies FormLayout and TableLayout support	These open-source layout managers allow you to design high quality forms. JGoodies FormLayout support includes column/row specifications (alignment, size, resize behavior), IntelliGap (automatically handles gap columns/rows) and column/row grouping (makes widths/heights equal). Also other parts of the JGoodies Forms framework are supported (DLU borders, component factory). TableLayout is fully supported (column/row size, gaps, alignment).
Advanced GridBagLayout support	The advanced GridBagLayout support allows the specification of horizontal and vertical gaps (as in TableLayout). JFormDesigner automatically computes the `GridBagConstraints.insets` for all components. This makes designing a form with consistent gaps using GridBagLayout much easier. No longer wrestling with `GridBagConstraints.insets` ;-)
Column and row headers	The column and row headers (for grid based layout managers) show the structure of the layout (including column/row indices, alignment, growing, grouping) and allow you to insert or delete columns/rows and change column/row properties. It's also possible to drag and drop columns/rows (incl. contained components and gaps). This allows you to swap columns or move rows in seconds.

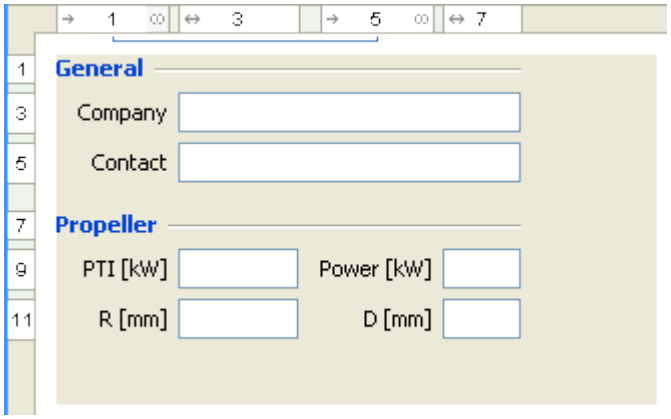
Localization support	Localizing forms using properties files has never been easier. Specify a resource bundle name and a prefix for keys when creating a new form and then forget about it. JFormDesigner automatically puts all strings into the specified resource bundle (auto-externalizing). It also updates resource keys when renaming components, copies resource strings when copying components and removes resource strings when deleting components. You can also externalize and internalize strings, edit resource bundle strings, add locales, switch locale used in Design view, in-place-edit text of current locale.
Java code generator or runtime library	Either let JFormDesigner generate Java source code for your forms (the default) or use the open-source (BSD license) runtime library to load JFormDesigner XML files at runtime. Your choice. Turn off the code generator in the Preferences, if you don't need it.
Generation of nested classes	The Java code generator is able to generate and update nested classes. You can specify a class name for each component in your form. This allows you to organize your source code in an object-oriented way.

What's New in JFormDesigner 3.1

JFormDesigner 3.1 introduces several new features and enhancements. This topic describes some of the significant or more interesting changes. Please have a look at the changelog for a complete list of changes.

Animated transitions

Animated transitions on layout changes in Design view are not just cool, they also help you to better see what happens on changes.



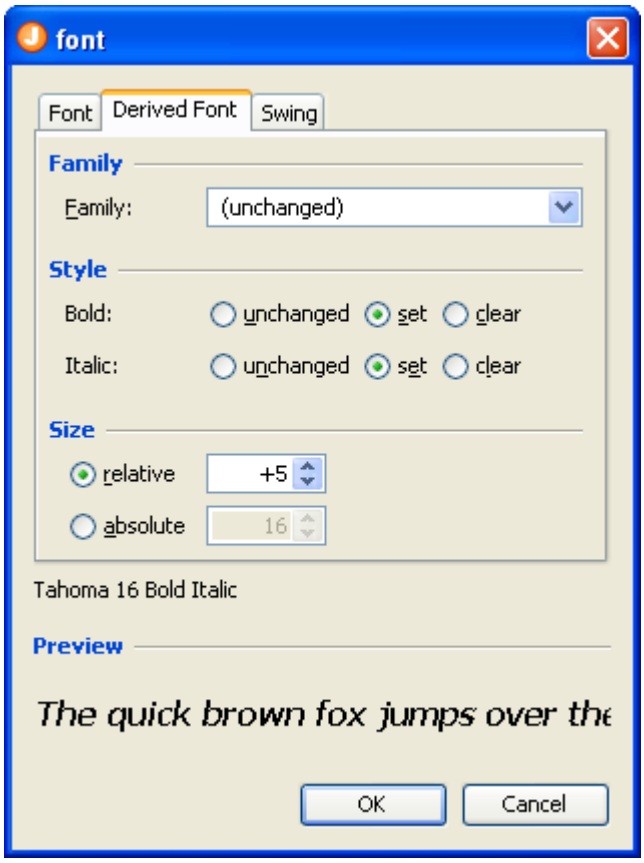
(animated image)

Relative font specification

Derived fonts are computed based on the default font of the component (from the current look and feel). They are recommended if you just need a bold/italic or a larger/smaller font (e.g. for titles) because they are platform independent. If your application runs on several look and feels (e.g. several operating systems), derived fonts ensure that the font family stays consistent.

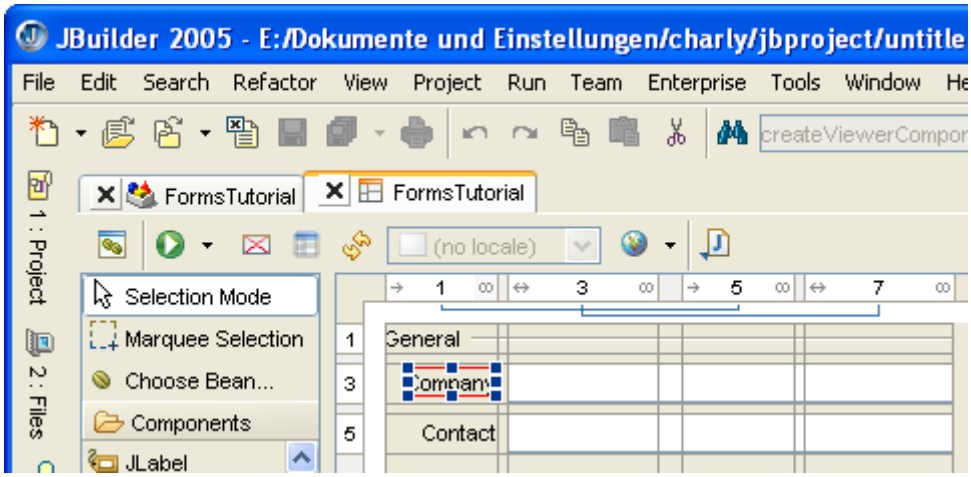
In the Properties view, you can quickly change the style (bold and italic) and the size of the font.

In the custom editor you can choose one of the tabs to specify either absolute fonts, derived fonts or fonts defined in the look and feel.



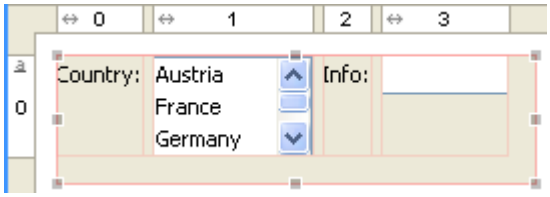
JBuilder plug-in

The JBuilder plug-in fully integrates JFormDesigner into JBuilder 2005 and 2006. For JBuilder 2007 use the Eclipse plug-in.



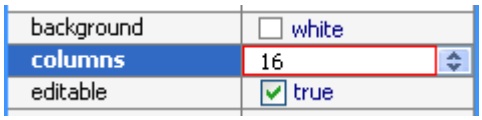
Baseline alignment (Java 6)

Baseline alignment, which was introduced to GridBagLayout and FlowLayout in Java 6, is now supported. Components are vertically aligned along the baseline of the prevailing row.



Spinner editor for numbers

For numbers, a spinner editor in the Properties view makes it now easier to increase or decrease the value using the arrow buttons or **Up** and **Down** keys.



Quicker component aligning

In grid-based layout managers (e.g. FormLayout), you can now quicker change the alignment of selected components because the alignment commands are now directly in the context menu and no longer in submenus.

Native libraries support

You can now use JavaBeans that use native libraries (e.g. JOGL). Add the JAR files and the native libraries to the new Native Library Paths preferences page.

Switch locale at runtime in your application

The Java code generator can now put the code to initialize localized texts into a method initComponentsI18n(). You can invoke this method from your code to switch the locale of a form at runtime. Enable this option either globally in the Localization (Java Code Generator) preferences or per form.

Improved Preferences import and export

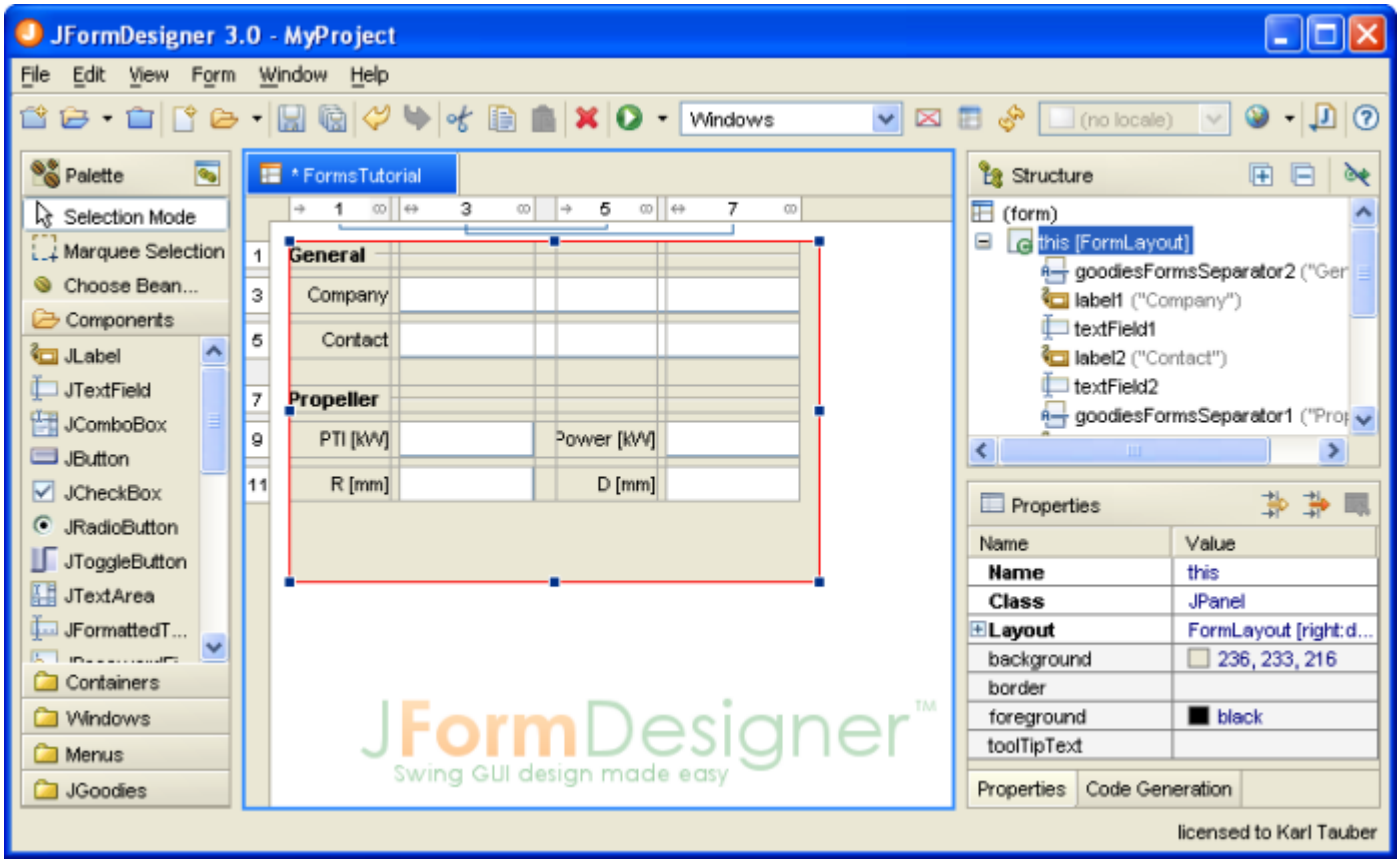
Introduced the possibility to export only parts of the preferences. Also "Import" and "Export" buttons have been added to the IDE plug-ins preferences/settings dialogs for importing and exporting of JFormDesigner .prefs files, which are compatible with all editions of JFormDesigner (IDE plug-ins and stand-alone). Now you can exchange JFormDesigner preferences across all editions.

Runtime library open sourced

The JFormDesigner runtime library jfd-loader.jar, which can be used to load .jfd files at runtime, is now open source licensed under the BSD license.

User Interface

This is the main window of JFormDesigner stand-alone edition:



The main window consists of the following areas:

- Main Menu: Located at the top of the window.
- Toolbar: Located below the main menu.
- Palette: Located at the left side of the window.
- Design View: Located at the center of the window.
- Structure View: Located at the upper right of the window.
- Properties View: Located at the lower right of the window.
- Error Log View: Located below the Design view. This view is not visible in the above screenshot.

Menus

You can invoke most commands from the main menu (at the top of the main frame) and the various context (right-click) menus.

Main Menu

The main menu is displayed at the top of the JFormDesigner main window of the **stand-alone** edition.

File menu	
New Project	Creates a new project.
Open Project	Opens an existing project.
Reopen Project	Displays a submenu of previously opened projects. Select a project to open it.
Project Properties	Displays the project properties.
Close Project	Closes the active project.
New Form	Creates a new form.
Open Form	Opens an existing form.
Reopen Form	Displays a submenu of previously opened forms. Select a form to open it.
Close	Closes the active form.
Close All	Closes all open forms.
Save	Saves the active form and generates the Java source code for the form (if Java Code Generation is switched on in the Preferences).
Save As	Saves the active form under another file name or location and generates the Java source code for the form (if Java Code Generation is switched on in the Preferences).
Save All	Saves all open forms and generates the Java source code for the forms (if Java Code Generation is switched on in the Preferences).
Import	Imports NetBeans or IntelliJ IDEA .form files and creates new JFormDesigner forms. Use **File > Save** to save the new form in the same folder as the .form file. This also updates the .java file.
Exit	Exits JFormDesigner. Mac OS X: this item is in the JFormDesigner application menu.

Edit menu	
Undo	Reverses your most recent editing action.
Redo	Re-applies the editing action that has most recently been reversed by the Undo action.
Cut	Cuts the selected components to the clipboard.
Copy	Copies the selected components to the clipboard.
Paste	Pastes the components in the clipboard to the selected container of the active form.
Rename	Renames the selected component.
Delete	Deletes the selected components.

View menu	
Show Diagonals	Shows diagonals.
Squint Test	Simulates evaluating a graphic layout by squinting your eyes. This tests legibility and whether the overall layout is a strong, clear layout. You can change the squint intensity in the Preferences.
Refresh	Refresh the Design view of the active form. Reloads all classes used by the form and recreates the form preview shown in the Design view. Use this command, if you changed the code of a component used in the form to reload the component classes.

Form menu	
Test Form	Tests the active form. Creates live instances of the form in a new window. You can close that window by pressing the **Esc** key when the window has the focus. If your form contains more than one top-level component, use the drop-down menu in the toolbar to test another component.
Localize	Edit localization settings, resource bundle strings, create new locales or delete locales.
New Locale	Creates a new locale.
Delete Locale	Deletes an existing locale.
Externalize Strings	Moves strings to a resource bundle for localization. Use this command to start localizing existing forms.
Internalize Strings	Moves strings from a resource bundle into the form and remove the strings from the resource bundle.
Generate Java Code	Generates the Java code for the active form. Normally it's not necessary to use this command because when you save a form, the Java code will be also generated.

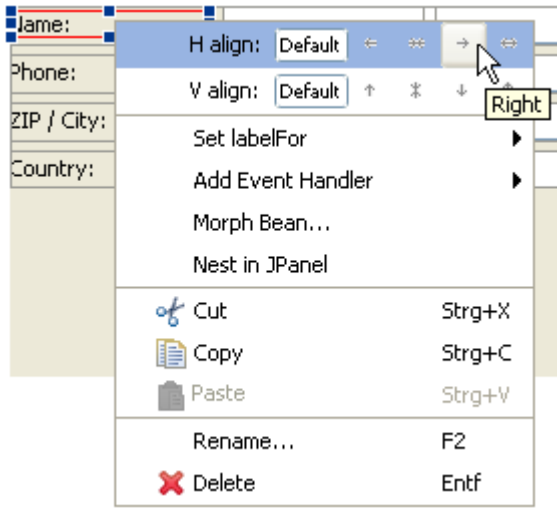
Window menu	
Activate Designer	Activates the Design view.
Activate Structure	Activates the Structure view.
Activate Properties	Activates the Properties view.
Activate Error Log	Activates the Error Log view. By default, the Error Log view is not visible. It automatically appears if an error occurs.
Next Form	Activates the next form.
Previous Form	Activates the previous form.
Preferences	Opens the Preferences dialog. Mac OS X: this item is in the JFormDesigner application menu.

Help menu	
Help Contents	Displays help topics.
What's New	Displays what's new in the current release.
Tip of the Day	Displays a list of interesting productivity features.
Register	Activates your license.
License	Displays information about your license.
About	Displays information about JFormDesigner and the system properties.

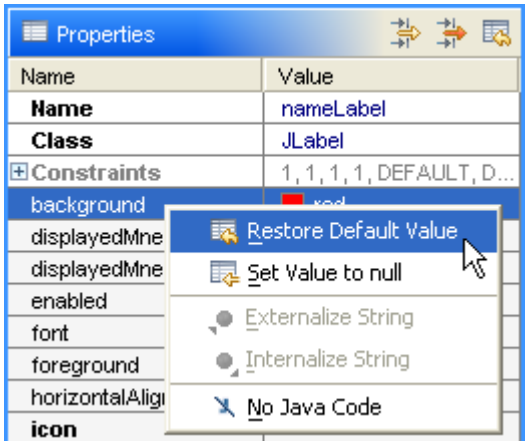
Context menus

Context menus appear when you're right-click on a particular component or control.

Design view context menu:



Properties view context menu:

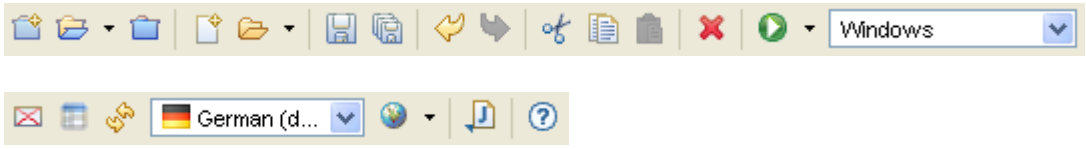


Toolbars

Toolbars provides shortcuts to often used commands.

Main Toolbar

This is the toolbar of JFormDesigner **stand-alone** edition. Many of commands are also used in the toolbars of the **IDE plug-ins**.

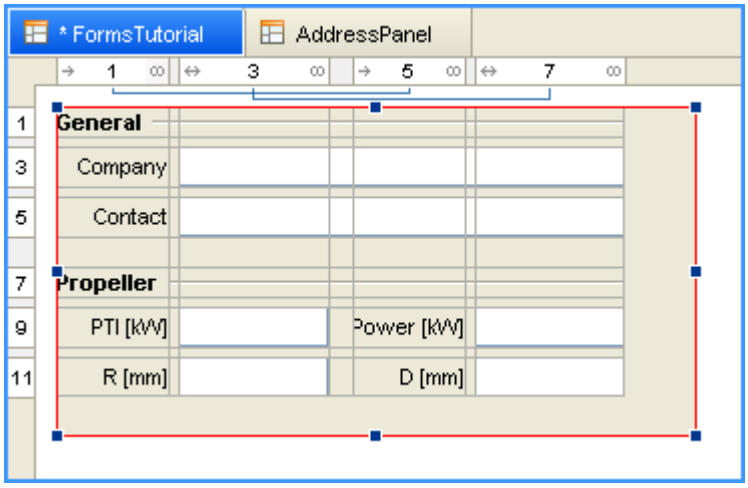


New Project	Creates a new project.
Open Project	Opens an existing project.
Project Properties	Displays the project properties.
New Form	Creates a new form.
Open Form	Opens an existing form.
Save	Saves the active form and generates the Java source code for the form (if Java Code Generation is switched on in the Preferences).
Save All	Saves all open forms and generates the Java source code for the forms (if Java Code Generation is switched on in the Preferences).
Undo	Reverses your most recent editing action.
Redo	Re-applies the editing action that has most recently been reversed by the Undo action.
Cut	Cuts the selected components to the clipboard.
Copy	Copies the selected components to the clipboard.
Paste	Pastes the components in the clipboard to the selected container of the active form.
Delete	Deletes the selected components.
Test Form	Tests the active form. Creates live instances of the form in a new window. You can close that window by pressing the **Esc** key when the window has the focus. If your form contains more than one top-level component, use the drop-down menu to test another component.
Windows	Allows you to change the look and feel of the components in the Design view. You can add other look and feels in the Preferences.
Show Diagonals	Shows diagonals.
Squint Test	Simulates evaluating a graphic layout by squinting your eyes. This tests legibility and whether the overall layout is a strong, clear layout. You can change the squint intensity in the Preferences.
Refresh	Refresh the Design view of the active form. Reloads all classes used by the form and recreates the form preview shown in the Design view. Use this command, if you changed the code of a component used in the form to reload the component classes.
German (Austria)	Allows you to change the locale of the form in the Design view. "(no locale)" is show if the form is not localized. Use **Form > Externalize Strings** to start localizing a form.
Localize	Edit localization settings, resource bundle strings, create new locales or delete locales.

Generate Java Code	Generates the Java code for the active form. Normally it's not necessary to use this command because when you save a form, the Java code will be also generated.
Help Contents	Displays help topics.

Design View

This view is the central part of JFormDesigner. It displays the opened forms and lets you edit forms.



Stand-alone: At top of the view, tabs are displayed for each open form. Click on a tab to activate a form. To close a form, click the symbol that appears on the right side of a tab if the mouse is over it. An asterisk (*) in front of the form name indicates that the form has been changed.

IDE plug-ins: The Design view is integrated into the IDEs, which have its own tabs.

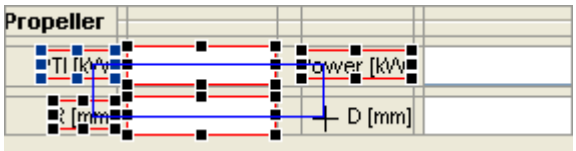
On the left side of the view and below the tabs, you can see the column and row headers. These are important controls for grid based layout managers. Use them to insert, delete or move columns/rows and change column/row properties.

In the center is the design area. It displays the form, grids and handles. You can drag and drop components, resize, rename, delete components or in-place-edit labels.

Selecting components

To select a single component, click on it. To select multiple components, hold down the **Ctrl** (Mac OS X: **Command**) or **Shift** key and click on the components. To select the parent of a selected component, hold down the **Alt** key (Mac OS X: **Option** key) and click on the selected component.

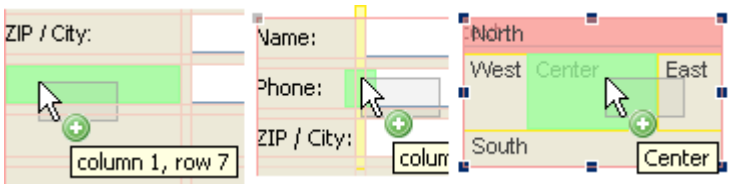
To select components in a rectangular area, select **Marquee Selection** in the Palette and click-and-drag a rectangular selection area in the Design view. Or click-and-drag on the free area in the Design view. All components that lie partially within the selection rectangle are selected.



The selection in the Design view and in the Structure view is synchronized both ways.

Drag feedback

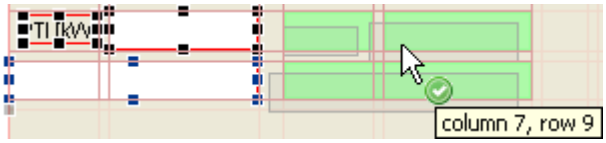
JFormDesigner provides four types of drag feedback.



The gray figure shows the outline of the dragged components. It always follows the mouse location. The green figure indicates the drop location, the yellow figure indicates a new column/row and red figures indicate occupied areas.

Move or copy components

To move components simply drag them to the new location. You will get reasonable visual feedback during the drag operation.



To copy components, proceed just as moving components, but hold down the **Ctrl** key (Mac OS X: **Option** key) before dropping the components.

You can cancel all drag operations using the **Esc** key.

Resize components

Use the selection handles to resize components. Click on a handle and drag it.

The green feedback figure indicates the new size of the component. The tool tip provides additional information about location, size and differences.

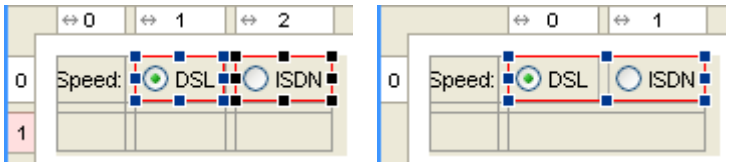
Whether a component is resizable or not depends on the used layout manager.

Morph components

The "Morph Bean" command allows you to change the class of existing components without loosing properties, events or layout information. Right-click on a component in the Design or Structure view and select **Morph Bean** from the popup menu.

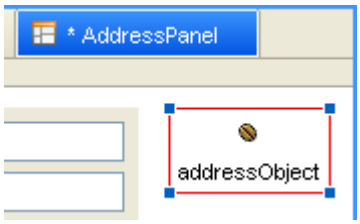
Nest in Container

The "Nest in Container" command allows you to nest selected components in a new container (usually a JPanel). Right-click on a component in the Design or Structure view and select **Nest in JPanel** from the popup menu. The new container gets the same layout manager as the old container and is placed at the same location where the selected components were located. For grid-based layout managers, the new container gets columns and rows and the layout constraints of the selected components are preserved.



Non-visual beans

To add a non-visual bean to a form, select it in the Palette (or use **Choose Bean**) and drop it into the free area of the Design view. Non-visual beans are shown in the Design view using proxy components.



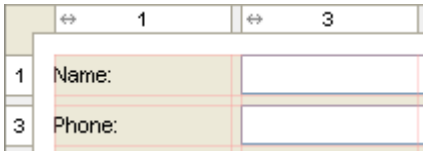
Red beans

If a bean could not instantiated (class not found, exception in constructor, etc), a **red bean** will be shown in the designer view as placeholder.

To fix such problems, add required jars to the classpath in the Project dialog and then select **View > Refresh** from the menu (or press **F5**) to refresh the designer view.

Headers

The column and row headers (for grid based layout managers) show the structure of the layout. This includes column/row indices, alignment, growing and grouping.



Use them to insert, delete or move columns/rows and change column/row properties. Right-clicking on a column/row displays a popup menu. Double-clicking shows a dialog that allows you to edit the column/row properties.

If a column width or row height is zero, which is the case if a column/row is empty, then JFormDesigner uses a minimum column width and row height. Columns/rows having a minimum size are marked with a light-red background in the column/row header.

Selecting columns/rows

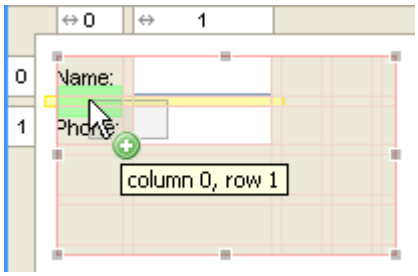
You can select more than one column/row. Hold down the **Ctrl** key (Mac OS X: **Command** key) and click on another column/row to add it to the selection. Hold down the **Shift** key to select the columns/rows between the last selected and the clicked column/row.

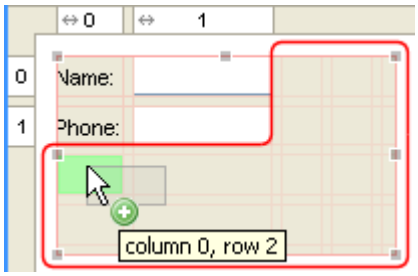
Insert column/row

Right-click on the column/row where you want to insert a new one and select **Insert Column** / **Insert Row** from the popup menu. The new column/row will be inserted before the right-clicked column/row. To add a column/row after the last one, right-click on the area behind the last column/row.

If the layout manager is FormLayout, an additional gap column/row will be added. Hold down the **Shift** key before selecting the command from the popup menu to avoid this.

Besides using the popup menu, you can insert new columns/row when dropping components on column/row gaps or outside of the existing grid. In the first figure, a new row will be inserted between existing rows. In the second figure, a virtual grid is shown below/right to the existing grid and a new row will be added.





Delete columns/rows

Right-click on the column/row that you want delete and select **Delete Column** / **Delete Row** from the popup menu.

If the layout manager is FormLayout, an existing gap column/row beside the removed column/row will also be removed. Hold down the **Shift** key before selecting the command from the popup menu to avoid this.

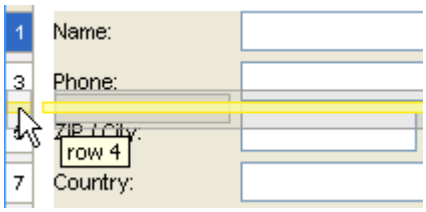
Split columns/rows

Right-click on the column/row that you want split and select **Split Column** / **Split Row** from the popup menu.

If the layout manager is FormLayout, an additional gap column/row will be added. Hold down the **Shift** key before selecting the command from the popup menu to avoid this.

Move columns/rows

The headers allow you to drag and drop columns/rows (incl. contained components and gaps). This allows you to swap columns or move rows in seconds. Click on a column or row and drag it to the new location. JFormDesigner updates the column/row specification and the locations of the moved components.



If the layout manager is FormLayout, then existing gap columns/rows are also moved. Hold down the **Shift** key before dropping a column/row to avoid this.

Resize columns/rows

To change the (minimum) size of a column/row, click near the right edge of a column/row and drag it.

FormLayout supports minimum and constant column/row sizes. Hold down the **Ctrl** key to change the minimum size. TableLayout supports only constant sizes and GridBagLayout supports only minimum sizes.

Header symbols

Following symbols are used in the headers:

Column Header	
←	Left aligns components in the column.
→	Right aligns components in the column.
↔	Center components in the column.
⇔	Fill (expand) components into the column.
⊢	Grow column width.
∞	Group column with other columns. All columns in a group will get the same width.

Row Header	
↑	Top aligns components in the row.
↓	Bottom aligns components in the row.
↕	Center components in the row.
⇕	Fill (expand) components into the row.
a	Baseline aligns components in the row.
a	Aligns components above baseline in the row.
a	Aligns components below baseline in the row.
⊤	Grow row height.
∞	Group row with other rows. All rows in a group will get the same height.

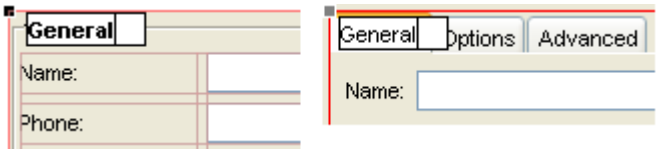
In-place-editing

In-place-editing allows you to edit the text of labels and other components directly in the Design view. Simply select a component and start typing. JFormDesigner automatically displays a text field that allows you to edit the text.

You can also use the **Space** key or double-click on a component to start in-place-editing. Confirm your changes using the **Enter** key, or cancel editing using the **Esc** key.

In-place-editing is available for all components, which support one or the properties `textWithMnemonic`, `text` and `title`.

In-place-editing is also supported for the title of `TitledBorder` and the tab titles of JTabbedPane.



`TitledBorder`: double-click on the title of the `TitledBorder`; or select the component with the `TitledBorder` and start in-place-editing as usual.

`JTabbedPane`: double-click on the tab title; or single-click on the tab, whose title you want to edit and start in-place-editing as usual.

Keyboard Navigation

Keyboard navigation allows you to change the selection in the designer view using the keyboard. This allows you for example to edit a bunch of labels using in-place-editing without having to use the mouse. You can use the following keys:

Key	Description
Up	move the selection up
Down	move the selection down
Left	move the selection left
Right	move the selection right
Home	select the first component
End	select the last component

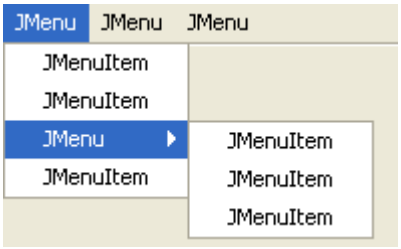
Note: Keyboard navigation is currently limited to one container. You cannot move the selection to another container using the keyboard.

Menu Designer

The menu designer makes it easy to create and modify menu bars and popup menus. It supports in-place-editing menu texts and drag-and-drop menu items.

Menu bar structure

The following figure shows the structure of a menu bar. The horizontal bar on top of the image is a `JMenuBar` that contains `JMenu` components. The `JMenu` contains `JMenuItem`, `JCheckBoxMenuItem`, `JRadioButtonMenuItem` or Menu Separator components. To create a sub-menu, put a `JMenu` into a `JMenu`.



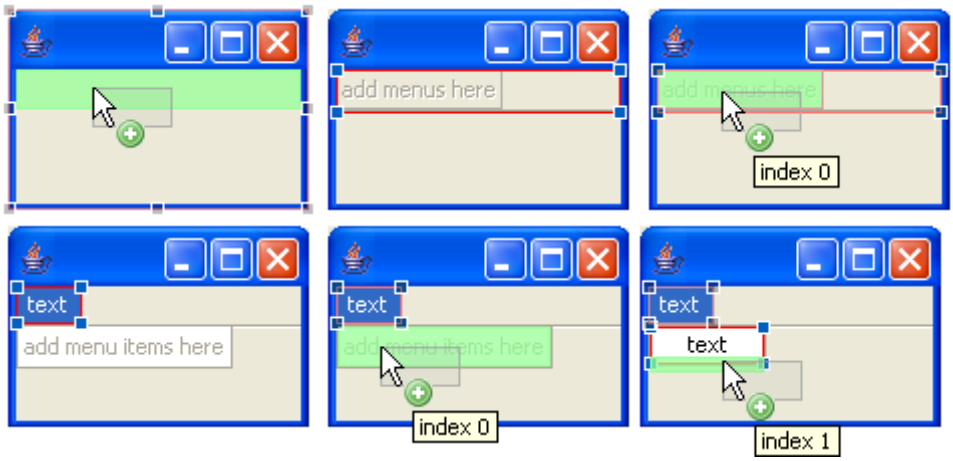
The component palette has a category "Menus" that contains all components necessary to create menus.

Creating menu bars

To create a menu bar:

1. add a `JMenuBar` to a `JFrame`
2. add `JMenus` to the `JMenuBar` and
3. add `JMenuItems` to the `JMenus`

Select the necessary components in the Palette and drop them to the Design view.

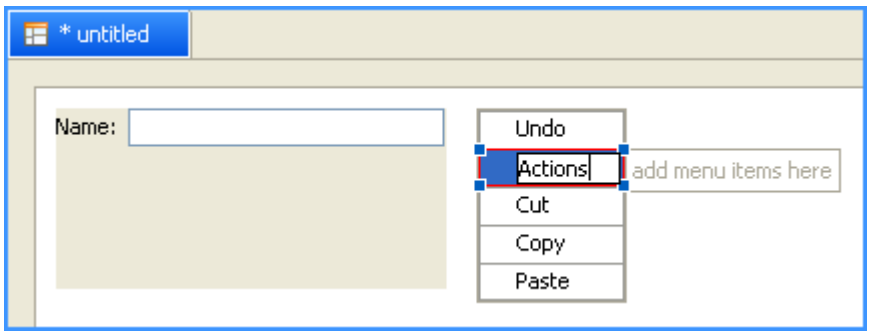


You can freely drag and drop the various menu components to rearrange them.

Creating popup menus

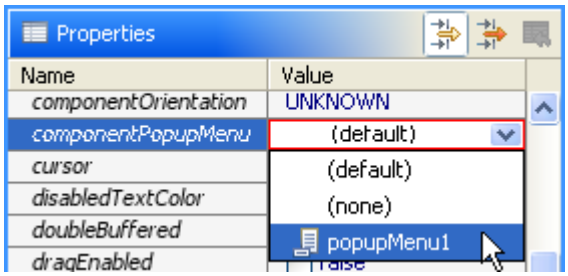
To create a popup menu:

1. add a `JPopupMenu` to the free area in the Design view and
2. add `JMenuItems` to the `JPopupMenu`



Assign popup menus to components

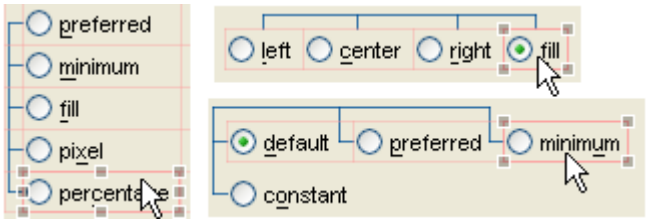
If you use Java 5 or later, you can assign the popup menu to a component in the properties view using the "componentPopupMenu" property. Select the component to which you want attach the popup menu and assign it in the Properties view. Note that you must click on the **Show Advanced Properties** in the toolbar of the Properties view to see the property.



Note that JFormDesigner must run on Java 5 to use the "componentPopupMenu" property. Open the JFormDesigner About dialog and check whether it displays "Java 1.5.x".

Button groups

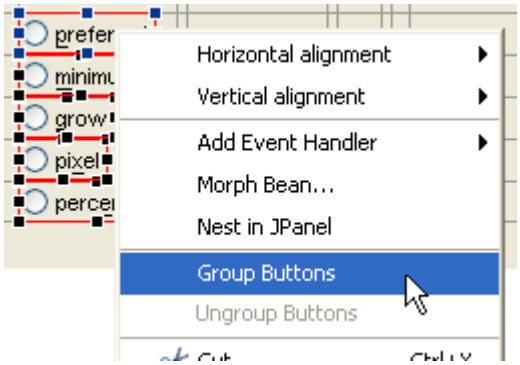
Button groups (`javax.swing.ButtonGroup`) are used in combination with radio buttons to ensure that only one radio button in a group of radio buttons is selected.



To visualize the grouping, JFormDesigner displays lines connecting the grouped buttons.

Group Buttons

To create a new button group, select the buttons you want to group, right-click on a selected button and select **Group Buttons** from the popup menu.



You can extend existing button groups by selecting at least one button of the existing group and the buttons that you want to add to that group, then right-click on a selected button and select **Group Buttons** from the popup menu.

Note that the **Group Buttons** and **Ungroup Buttons** commands are only available in the context menu if the selection contains only components, which are derived from `JToggleButton` (`JRadioButton` and `JCheckBox`).

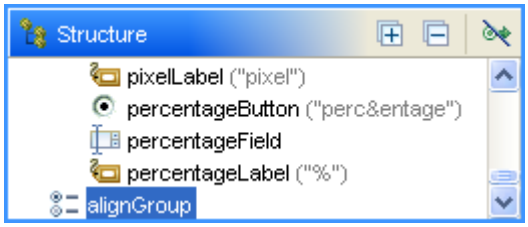
Ungroup Buttons

To remove a button group, select all buttons of the group, right-click on a selected button and select **Ungroup Buttons** from the popup menu.

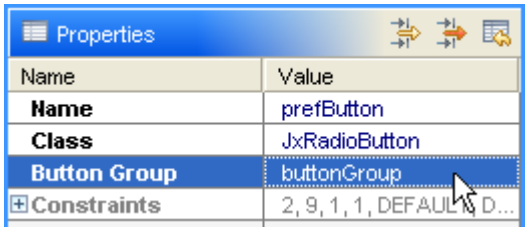
To remove a button from a group, right-click on it and select **Ungroup Buttons** from the popup menu.

ButtonGroup object

Button groups are non-visual beans. They appear at the bottom of the Structure view and in the Design view. JFormDesigner automatically creates and removes those objects. You can rename button group objects.



If a grouped button is selected, you can see the association to the button group in the Properties view.

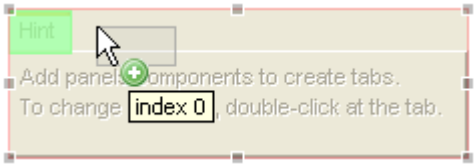


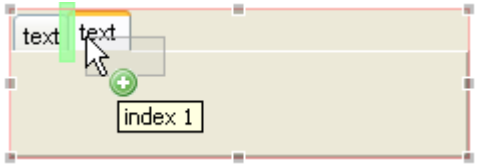
JTabbedPane

JTabbedPane is a container component that lets the user switch between pages by clicking on a tab.

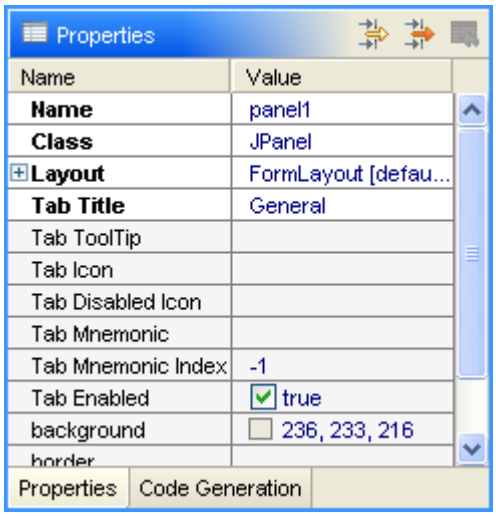
After adding a JTabbedPane to your form, it looks like this one:

To add pages, select an appropriate component (e.g. JPanel) in the palette, move the cursor over the tabs area of the JTabbedPane and click to add it.



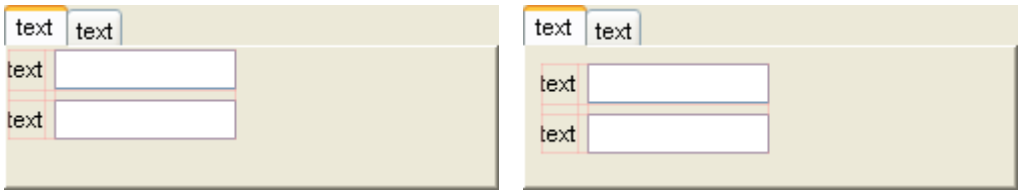


You can see only the components of the active tab. Click on a tab to switch to another page. To change a tab title, double-click on a tab to in-place-edit it. You can edit other tab properties (tool tip text, icon, ...) in the Properties view. Select a page component (e.g. JPanel) to see its tab properties.



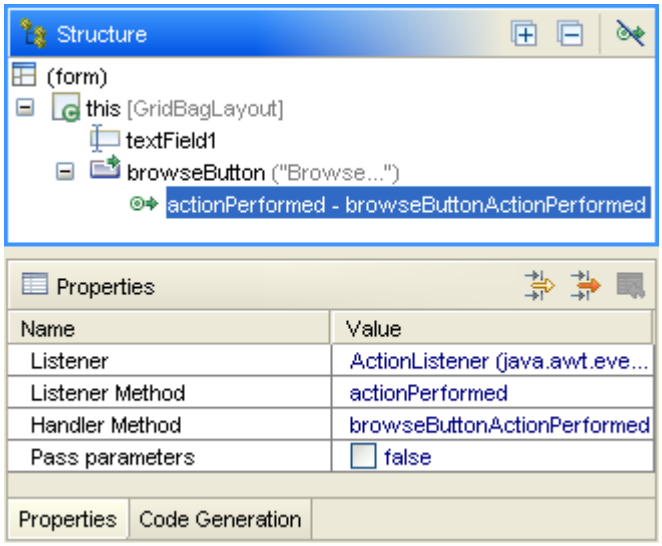
To change the tab order, select a page component (e.g. JPanel) and drag it over the tabs to a new place. You can also drag and drop page components in the Structure view to change its order.

Use an empty border to separate the page contents from the JTabbedPane border. If you are using JGoodies Forms, it's recommended to use `TABBED_DIALOG_BORDER`. Otherwise use an `EmptyBorder`.



Events

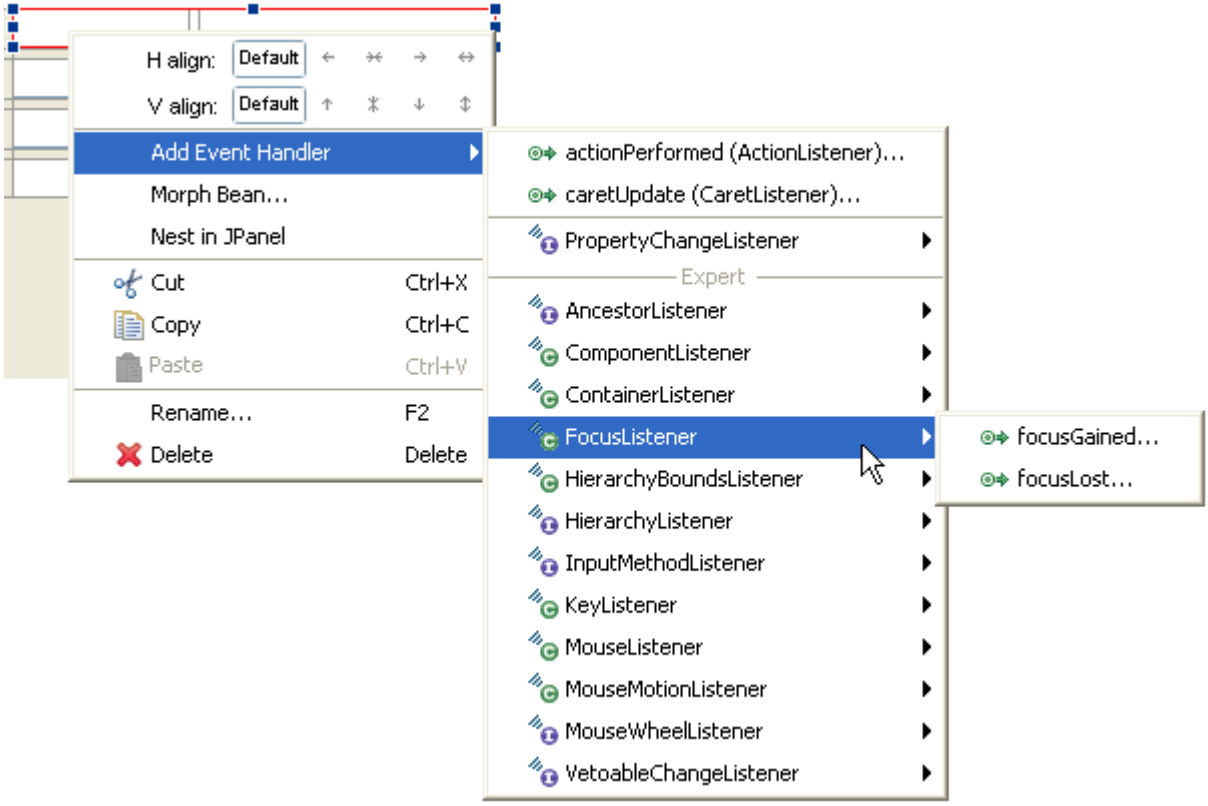
Components can provide events to signal when activity occurs (e.g. button pressed or mouse moved). JFormDesigner shows events in the Structure view and event properties in the Properties view.



IDE plug-ins: Double-click on the event in the Structure view to go to the event handler method in the Java editor of the IDE.

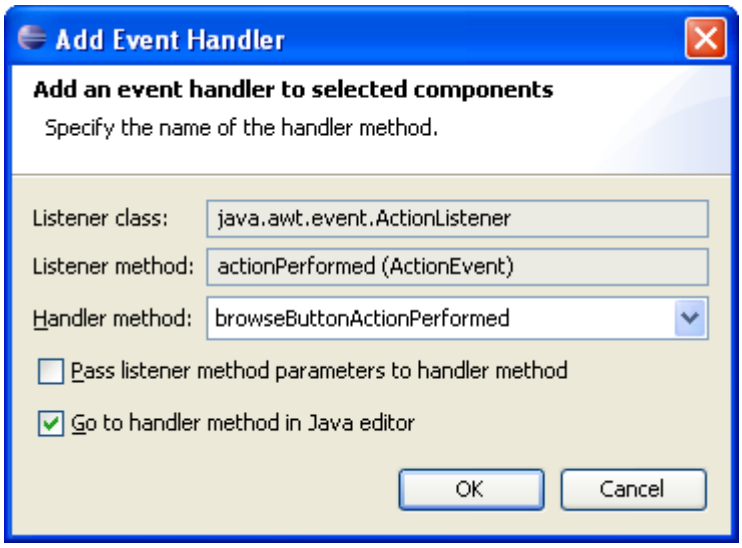
Add Event Handlers

To add an event handler to a component, right-click on the component in the Design or Structure view and select **Add Event Handler** from the popup menu. The events popup menu lists all available event listeners for the selected components and is divided into three sections: preferred, normal and expert event listeners.



The icon in the popup menu indicates that the listener interface will be implemented (e.g. javax.swing.ChangeListener). The icon indicates that the listener adapter class will be used (e.g. java.awt.event.FocusAdapter for java.awt.event.FocusListener). The icons and are used when the listener is already implemented.

After selecting an event listener from the popup menu, you can specify the name of the handler method and whether listener methods should be passed to the handler method in following dialog.



If you add a `PropertyChangeListener`, you can also specify a property name (field is not visible in above screenshot). Then the listener is added using the method `addPropertyChangeListener(String propertyName, PropertyChangeListener listener)`.

The "Go to handler method in Java editor" check box is only available in the **IDE plug-ins**.

Stand-alone: After saving the form, go to your favorite IDE and implement the body of the generated event handler method.

If you use the Runtime Library and the Java code generator is disabled, you must implement the handler method yourself in the target class. See documentation of method `FormCreator.setTarget()` in the JFormDesigner Loader API for details.

Remove Event Handlers

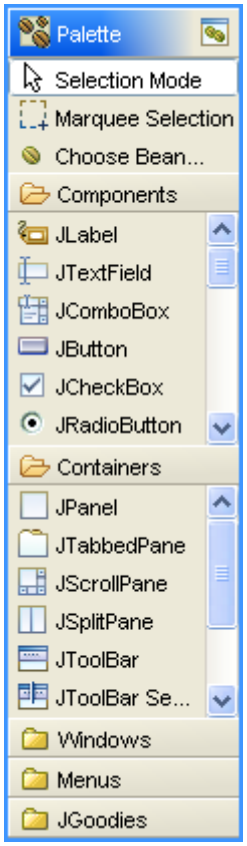
To remove an event handler, select it in the Structure view and press the **Del** key. Or right-click on the event handler and select **Delete** from the popup menu.

Change Handler Method Name

Select the event handler in the Structure view, press the **F2** key and edit the name in-place in the tree. You can also change the handler method and the "pass parameters" flag in the Properties view.

Palette

The component palette provides quick access to commonly used components (JavaBeans) available for adding to forms.



The components are organized in categories. Click on a category header to expand or collapse a category.

You can add a new component to the form in following ways:

- Select a component in the palette, move the cursor to the Design or Structure view and click where you want to add the component.
- Select **Choose Bean**, enter the class name of the component in the Choose Bean dialog, click OK, move the cursor to the Design or Structure view and click where you want to add the component.

To add multiple instances of a component, press the **Ctrl** key (Mac OS X: **Command** key) while clicking on the Design or Structure view.

The component palette is fully customizable. Right-click on the palette to add, edit, remove or reorder components and categories. Or use the Palette Manager.

Toolbar commands	
Palette Manager	Opens the Palette Manager dialog to customize the palette.

Choose Bean

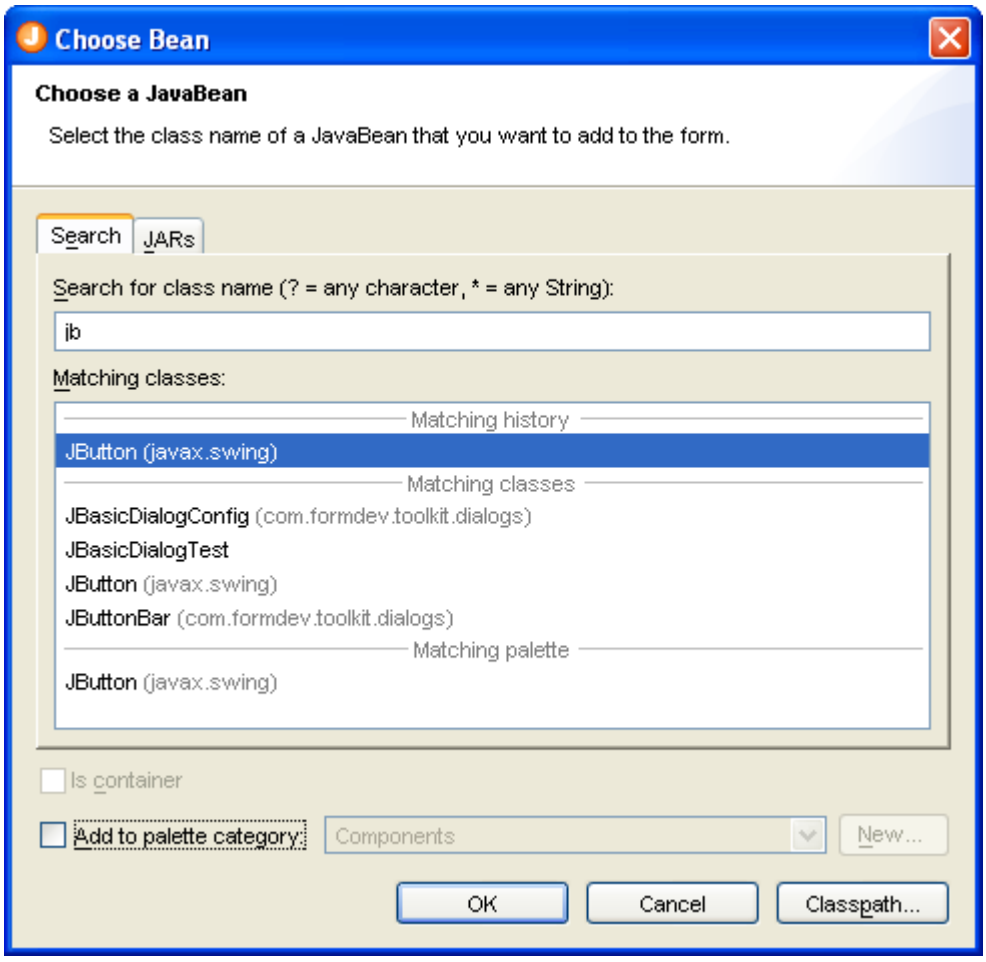
You can use any component that follows the JavaBean specification in JFormDesigner. Select **Choose Bean** in the palette to open the Choose Bean dialog.

Search tab

On this tab you can search for classes. Enter the first few characters of the class you want to choose until it appears in the matching classes list. Then select it in the list and click OK.

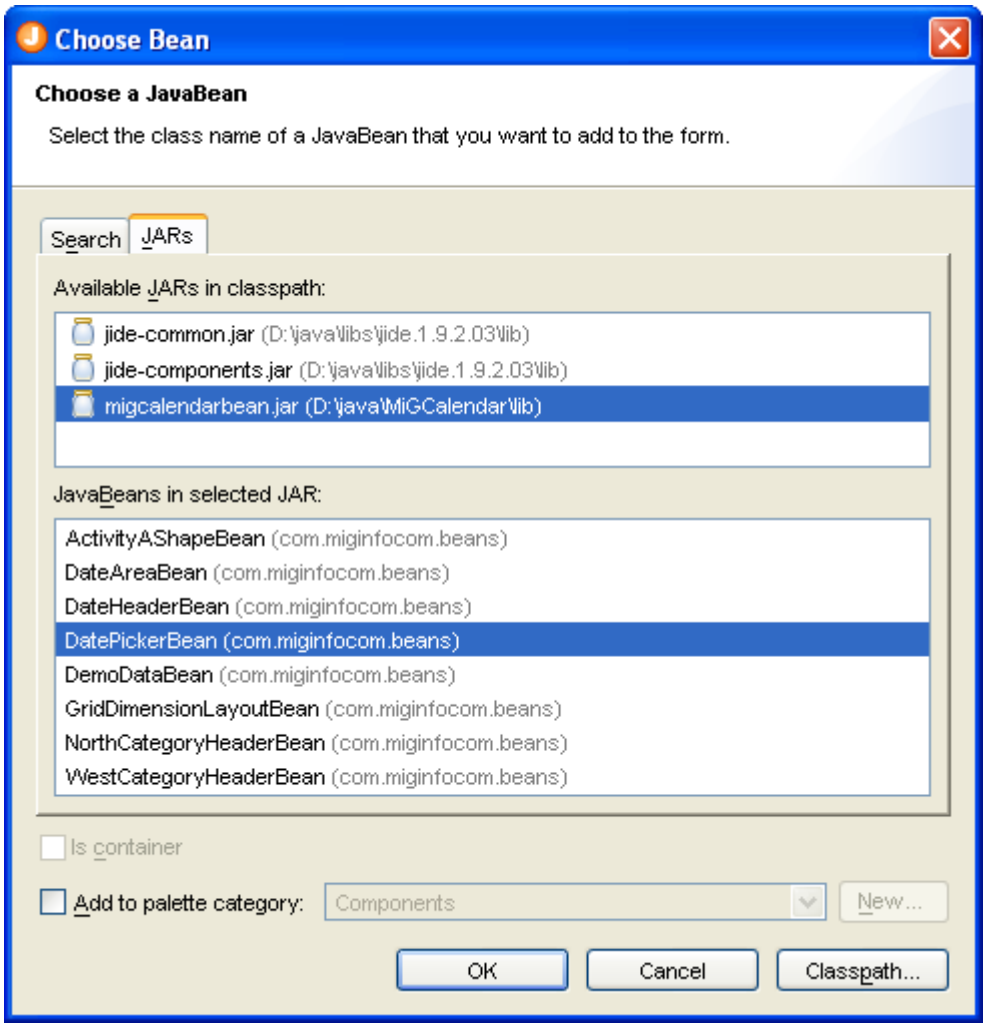
The matching classes list displays all classes that match. It is separated into up to three sections:

- Matching history: classes found in the history of last used classes. If the search field is empty, the complete history is displayed. To delete a class from the history, select it and press the **Delete** key or right-click on it an select **Delete** from the popup menu.
- Matching classes: classes found in the Classpath specified in the current Project.
- Matching palette: classes found in the palette.



JARs tab

On this tab you can select classes that are marked as JavaBean in the JAR's manifest. The provider of the component JAR can mark some classes as JavaBean in the manifest file. Popular 3rd party component libraries like MiG Calendar or JIDE components use this to make it easier to find the few classes, which can be used in GUI builders, in libraries that contain hundreds of classes.



See also http://java.sun.com/j2se/1.5.0/docs/guide/jar/jar.html#Per-Entry%20Attributes

Other options

The **Is Container** check box allows you to specify whether a bean is a container or not.

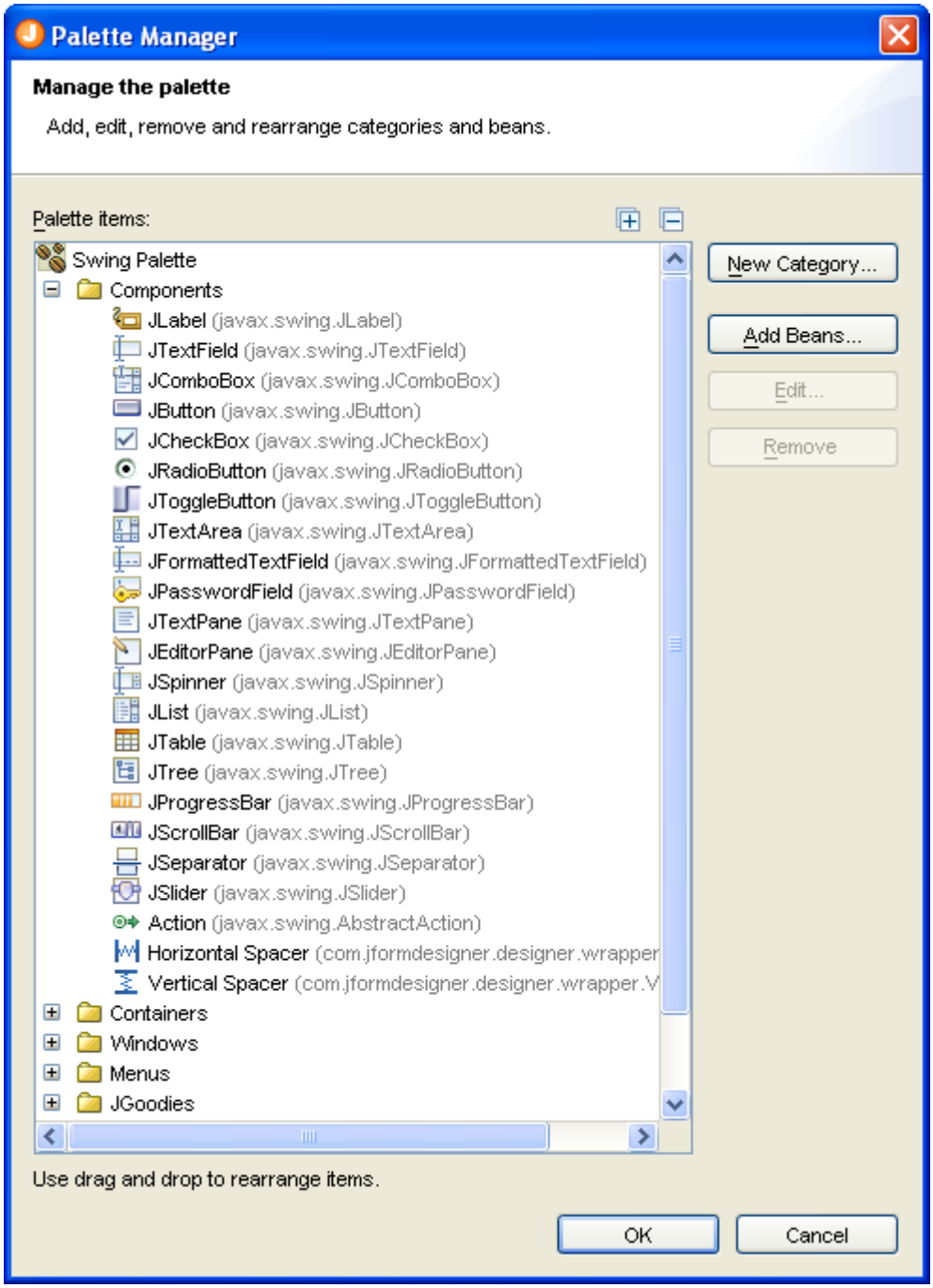
If you select **Add to palette category**, the component will be added to the palette category specified in the following field. Click the **New** button to create a new category for your components if necessary.

Stand-alone: Use the **Classpath** button to specify the location of your component classes. Add your JAR files or class folders.

IDE plug-ins: The classpath specified in the IDE project is used to locate component classes.

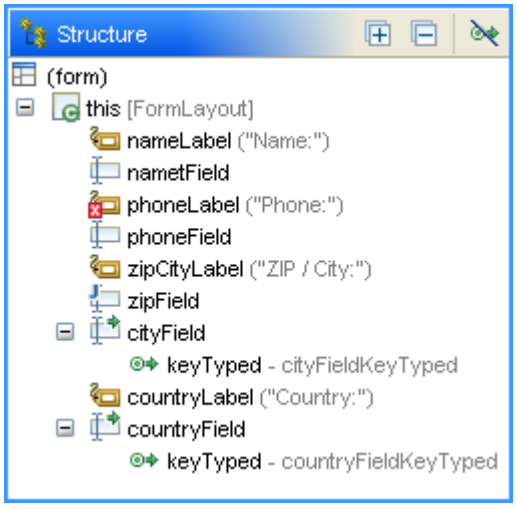
Palette Manager

This dialog allows you to fully customize the component palette. You can add, edit, remove or reorder components and categories.



Structure View

This view displays the hierarchical structure of the components in a form.



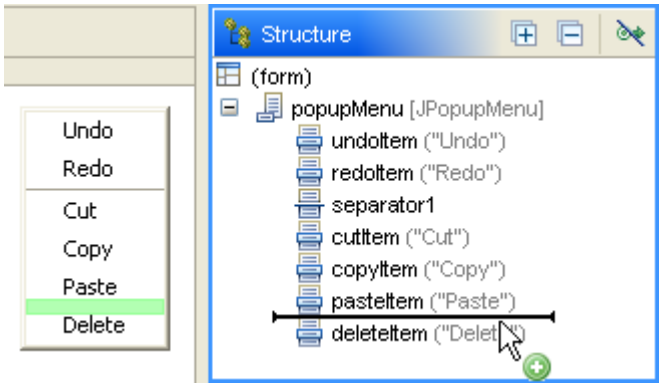
Each component is shown in the tree with an icon, its name and additional information like layout manager class or the text of a label or button. The name must be unique within the form and is used as variable name in the generated Java code.

You can edit the name of the selected component in the tree by pressing the **F2** key. Right-click on a component to invoke commands from the context menu.

The selection in the Structure view and in the Design view is synchronized both ways.

The tree supports multiple selection. Use the **Ctrl** key (Mac OS X: **Command** key) to add individual selections. Use the **Shift** key to add contiguous selections.

The tree supports drag and drop to rearrange components. You can also add new components from the palette to the Structure view. Besides the feedback indicator in the structure tree, JFormDesigner also displays a green feedback figure in the Design view to show the new location.



Various overlay icons are used in the tree to indicate additional information:

Icon	Description
	The component is bound to a Java class. Each component can have its own (nested) class. See Nested Classes for details.
	The component has events assigned to it. The events are shown as child nodes in the tree.
	The component has custom code assigned to it (see **Code Generation** tab in the Properties view). In the above screenshot, the component `zipField` has custom code.
	The variable modifier of the component is set to `public`. See Code Generation tab.
	The variable modifier of the component is set to `default`.
	The variable modifier of the component is set to `protected`.
	The variable modifier of the component is set to `private`.

Icon	Description
	A property (e.g. `JLabel.labelFor`) of the component has a reference to a non-existing component. This can happen if you e.g. remove a referenced `JTextField`. In the above screenshot, the component `phoneLabel` has a invalid reference.

Toolbar commands	
 Expand All	Expand all nodes in the structure tree.
 Collapse All	Collapse all nodes in the structure tree.
 Hide Events	If selected, hides the events from the structure tree.

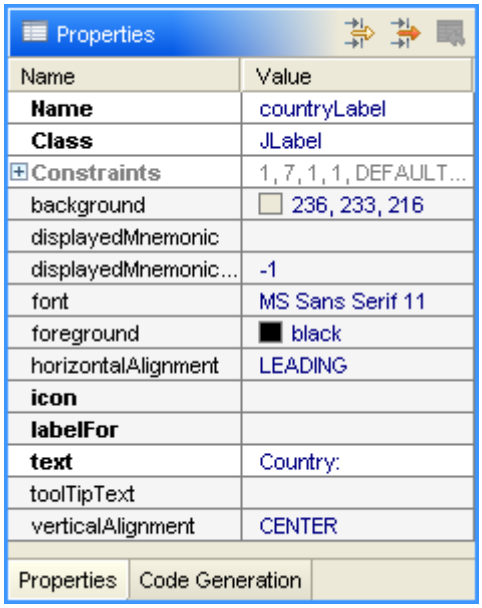
Properties View

The Properties view displays and lets you edit the properties of the selected component(s). Select one or more components in the Design or Structure view to see its properties. If more than one component is selected, only properties that are in all selected components are shown.

The view consists of two tabs (at the bottom of the view).

Properties tab

The **Properties** tab displays the component name, component class, layout properties, client properties and component properties. The list of component properties comes from introspection of the component class (JavaBeans).



By default, the Properties view displays regular properties. To see expert properties, click on the **Show Advanced Properties** ( ) button in the view toolbar and to see read-only properties, click the **Show read-only Properties** ( ) button.

Different font styles are used for the property names. Bold style is used for preferred (often used) properties, plain style for normal properties and italic style for expert properties. Read-only properties are shown using a gray font color.

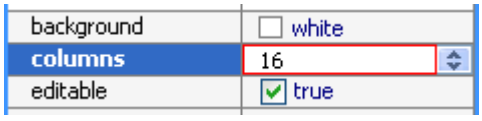
The light gray background indicates unset properties. The shown values are the default values of the component. The white background indicates set properties. Java code will be generated for set properties only. Use **Restore Default Value** ( ) to unset a property. Use **Set Value to null** ( ) from the popup menu to set a property explicitly to `null`.

The left column displays the property names, the right column the property values. Click on a property value to edit it.

You can either edit a value directly in the property table or use a custom property editor by clicking on the ellipsis button ( ) on the right side. The custom editor pops up in a new dialog. The globe button ( ), which is only available for localized forms and string properties, allows you to choose existing strings from the resource bundle of the form.

The type of the editor depends on the data type of the property. JFormDesigner has built-in property editors for all standard data types.

For numbers, a spinner editor makes it easier to increase or decrease the value using the arrow buttons or **Up** and **Down** keys. Press the **Enter** key to confirm the change; or the **Esc** key to cancel it.

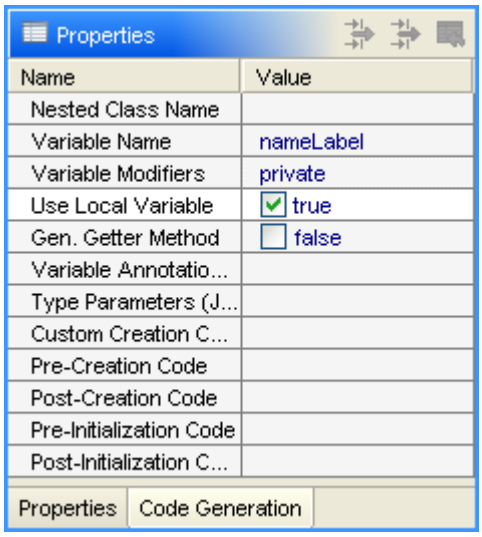


Common properties (at the top of the table):

Property Name	Description
Name	The name of the component. Must be unique within the form. Used as variable name in the generated Java code. It is also possible to specify a different variable name on the Code Generation tab.
Class	The class name of the component. The tooltip displays the full class name and the class hierarchy. Click on the value to morph the component class to another class (e.g. JTextField to JTextArea).
Button Group	The name of the button group assigned to the component. This property is only visible for components derived from `JToggleButton` (e.g. `JRadioButton` and `JCheckBox`).
Layout	Layout properties of the container component. Click on the plus sign to expand it. The list of layout properties depends on the used layout manager. This property is only visible for container components. Click on the value to change the layout manager.
Constraints	Layout constraints properties of the component. Click on the plus sign to expand it. The list of constraints properties depends on the layout manager of the parent component. This property is only visible if the layout manager of the parent component uses constraints.
Client Properties	Client properties of the component. Click on the plus sign to expand it. This property is only visible if there are client properties defined in the Client Properties preferences.

Code Generation tab

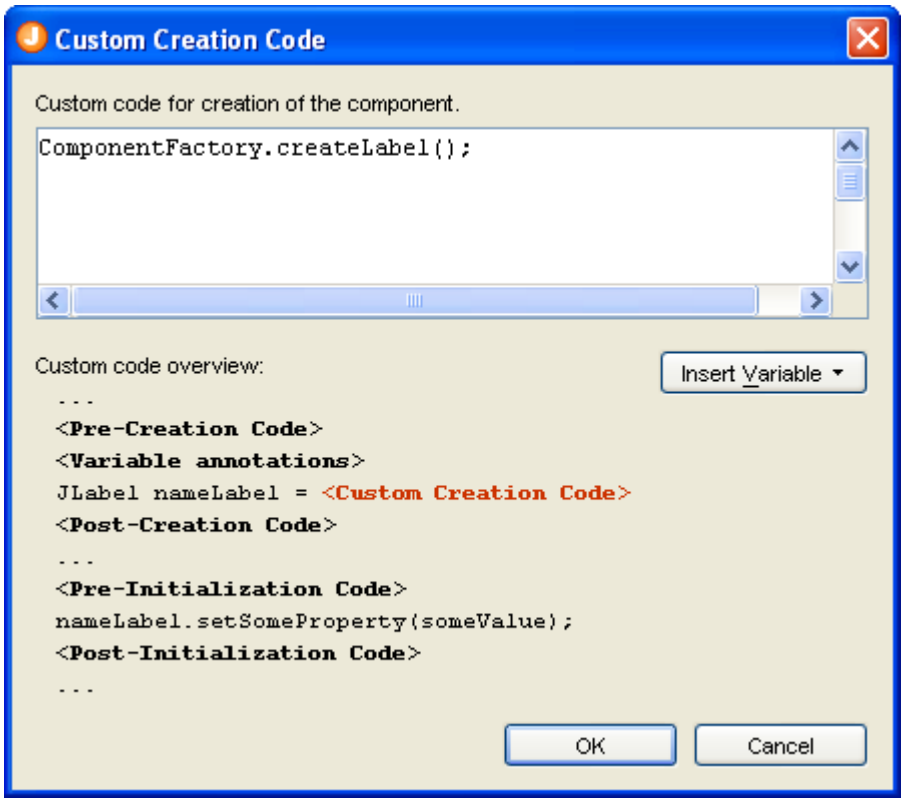
This tab displays properties related to the Java code generator.



Property Name	Description
Nested Class Name	The name of the generated nested Java class. See Nested Classes for details.
Variable Name	The variable name of the component used in the generated Java code. By default, it is equal to the component name.
Variable Modifiers	The modifiers of the variable generated for the component. Allowed modifiers: `public`, `default`, `protected`, `private`, `static` and `transient`. Default is `private`.
Use Local Variable	If `true`, the variable is declared as local in the initialization method. Otherwise at class level. Default is `false`.
Gen. Getter Method	If `true`, generate a public getter method for the component. Default is `false`.

Property Name	Description
Variable Annotations (Java 5)	Annotations of component variable (Java 5).
Type Parameters (Java 5)	Parameters of component type (Java 5). E.g. `MyTypedBean<String>`.
Custom Creation Code	Custom code for creation of the component.
Pre-Creation Code	Code included before creation of the component.
Post-Creation Code	Code included after creation of the component.
Pre-Initialization Code	Code included before initialization of the component.
Post-Initialization Code	Code included after initialization of the component.

This is the dialog for custom code editing:



"(form)" properties

Select the "(form)" node in the Structure view to modify special form properties:

Properties tab

Property Name	Description
Set Component Names	If `true`, invokes java.awt.Component.setName() on all components of the form.

Code Generation tab

Property Name	Description
Generate Java Source Code	If true, generate Java source code for the form. Defaults to "Generate Java source code" option in the Java Code Generator preferences.
Default Variable Modifiers	The default modifiers of the variables generated for components. Allowed modifiers: `public`, `default`, `protected`, `private`, `static` and `transient`. Default is `private`.
Default Use Local Variable	If `true`, the component variables are declared as local in the initialization method. Otherwise at class level. Default is `false`.
Default Gen. Getter Method	If `true`, generate public getter methods for components. Default is `false`.
Default Handler Modifiers	The default modifiers used when generating event handler methods. Allowed modifiers: `public`, `default`, `protected`, `private`, `final` and `static`. Default is `private`.
Member Variable Prefix	Prefix used for component member variables. E.g. "m_".
Use 'this' for member variables	If enabled, the code generator inserts 'this.' before all member variables. E.g. `this.nameLabel.setText("Name:");`
I18n Initialization Method	If enabled, the code generator puts the code to initialize the localized texts into a method initComponentsI18n(). You can invoke this method from your code to switch the locale of a form at runtime.
I18n 'getBundle' Template	Template used by code generator for getting a resource bundle. Default is `ResourceBundle.getBundle(${bundleName})`
I18n 'getString' Template	Template used by code generator for getting a string from a resource bundle. Default is `${bundle}.getString(${key})`
I18n Key Constants Class	The name of a class that contains constants for resource keys.

Property Editors

Property editors are used in the Properties view to edit property values.

You can either edit a value directly in the property table or use a custom property editor by clicking on the ellipsis button (...) on the right side. The custom editor pops up in a new dialog.

The type of the editor depends on the data type of the property. JFormDesigner has built-in property editors for all standard data types. Custom JavaBeans can provide their own property editors. Take a look at the API documentation of `java.beans.PropertyEditor`, `java.beans.PropertyDescriptor` and `java.beans.BeanInfo` and the JavaBeans topic for details.

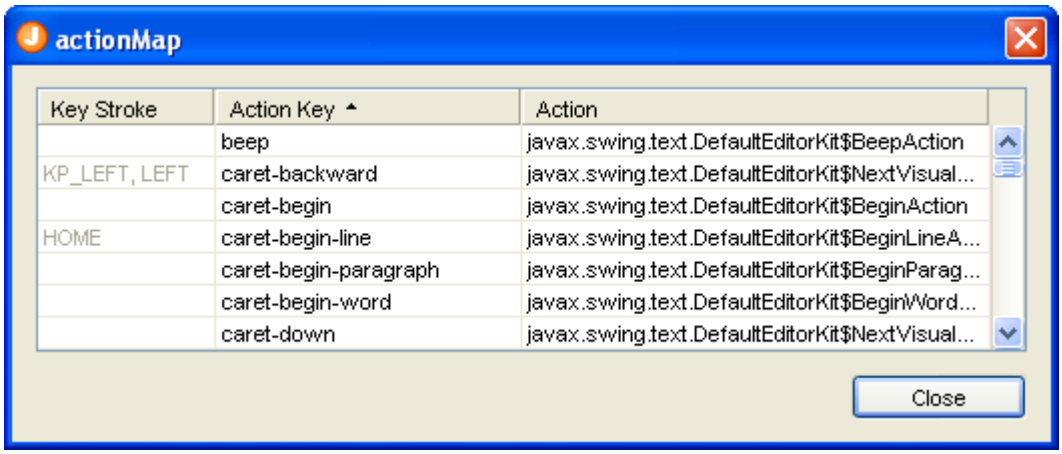
Built-in property editors

JFormDesigner has built-in property editors for following data types:

- String, `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, `short`, `java.lang.Boolean`, `java.lang.Byte`, `java.lang.Character`, `java.lang.Double`, `java.lang.Float`, `java.lang.Integer`, `java.lang.Long`, `java.lang.Short`, `java.math.BigDecimal` and `java.math.BigInteger`
- ActionMap (javax.swing)
- Border (javax.swing)
- Color (java.awt)
- ComboBoxModel (javax.swing)
- Cursor (java.awt)
- Dimension (java.awt)
- Font (java.awt)
- Icon (javax.swing)
- Image (java.awt)
- InputMap (javax.swing)
- Insets (java.awt)
- KeyStroke (javax.swing)
- ListModel (javax.swing)
- Object (java.lang)
- Paint (java.awt)
- Point (java.awt)
- Rectangle (java.awt)
- SpinnerModel (javax.swing)
- TableModel (javax.swing)
- TreeModel (javax.swing)

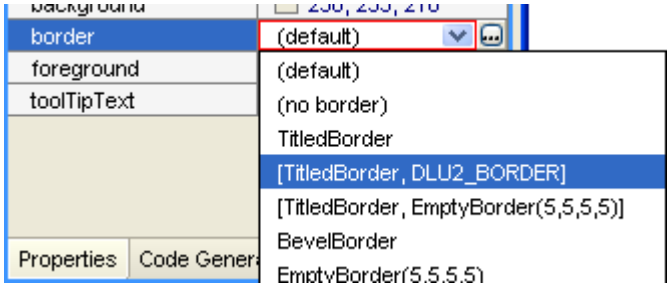
ActionMap (javax.swing)

This (read-only) custom editor allows you to see the actions registered for a component in its action map. The information in the column "Key Stroke" comes from the input map of the component and shows which key strokes are assigned to actions. The JComponent property "actionMap" is read-only. Select the **Show read-only Properties** button in the Properties view toolbar to make it visible.



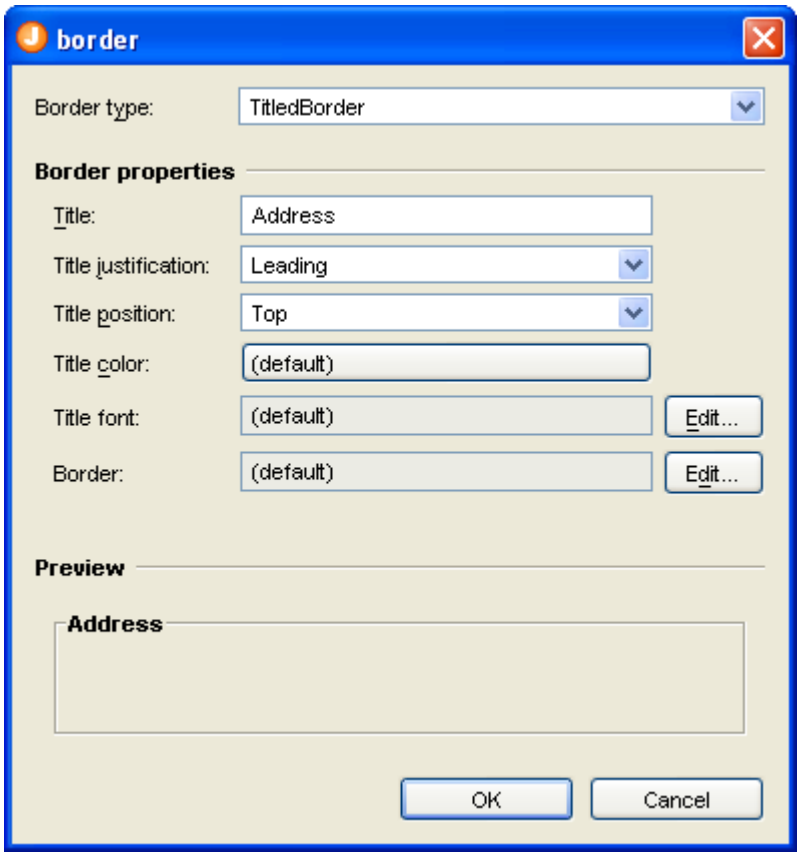
Border (javax.swing)

You can either select a border from the combo box in the properties table or use the custom editor.



In the custom editor you can edit all border properties. Use the combo box at the top of the dialog to choose a border type. In the mid area of the dialog you can edit the border properties. This area is different for each border type. At the bottom, you can see a preview of the border.

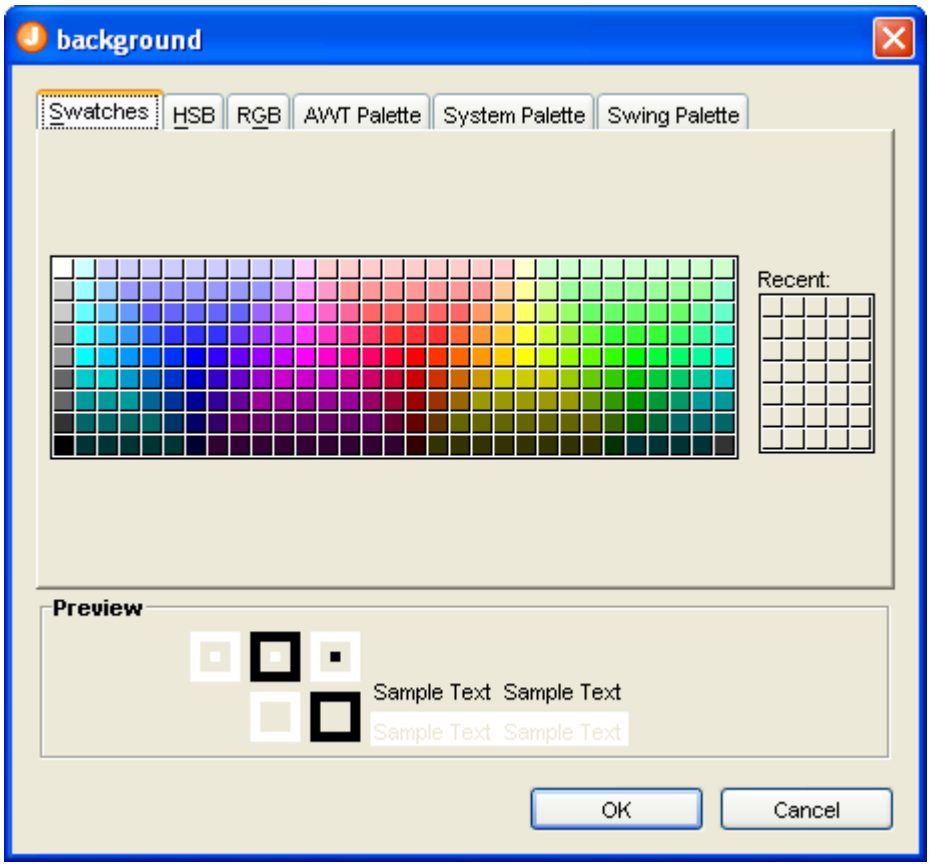
Following border types are supported: `BevelBorder`, `CompoundBorder`, Empty`Border`, `EmptyBorder (JGoodies)`, `EtchedBorder`, Line`Border`, `MatteBorder`, `SoftBevelBorder` and `TitledBorder`.



Color (java.awt)

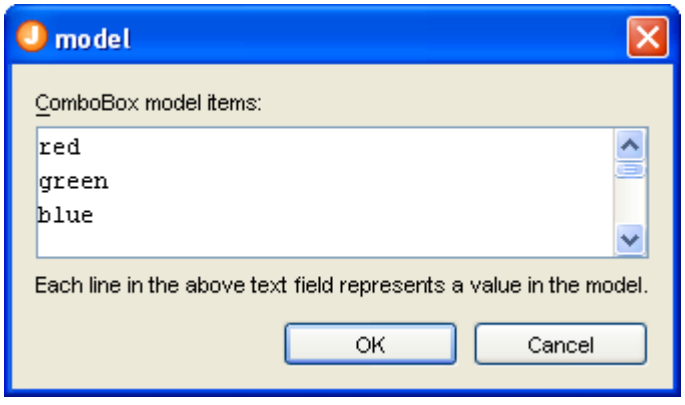
In the properties table, you can either enter RGB values, color names, system color names or Swing UIManager color names. When using a RGB value, you can also specify the alpha value by adding a fourth number.

The custom editor supports various ways to specify a color. Besides RGB, you can select a color from the AWT, System or Swing palettes.



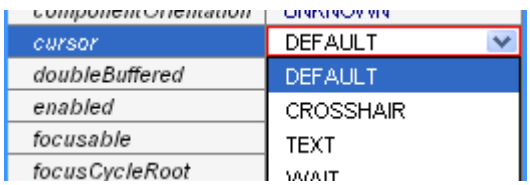
ComboBoxModel (javax.swing)

This custom editor allows you to specify string values for a combo box.



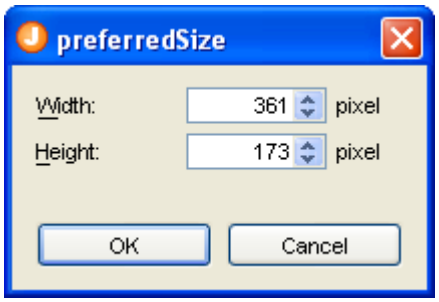
Cursor (java.awt)

This editor allows you to choose a predefined cursor.



Dimension (java.awt)

Either edit the dimension in the property table or use the custom editor.

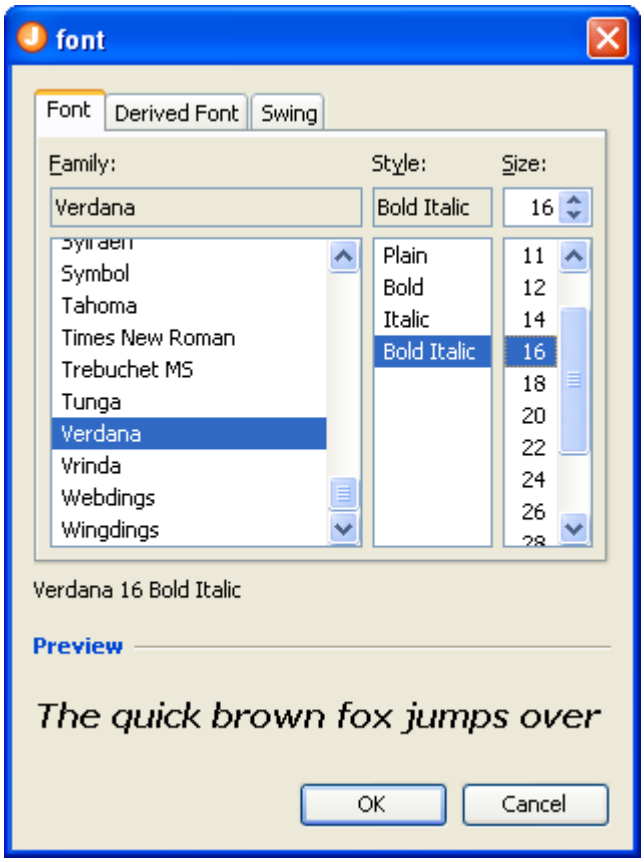


Font (java.awt)

You can either use absolute fonts, derived fonts or predefined fonts of the look and feel. Derived fonts are recommended if you just need a bold/italic or a larger/smaller font (e.g. for titles), because derived fonts are computed based on the current look and feel. If your application runs on several look and feels (e.g. several operating systems), derived fonts ensure that the font family stays consistent.

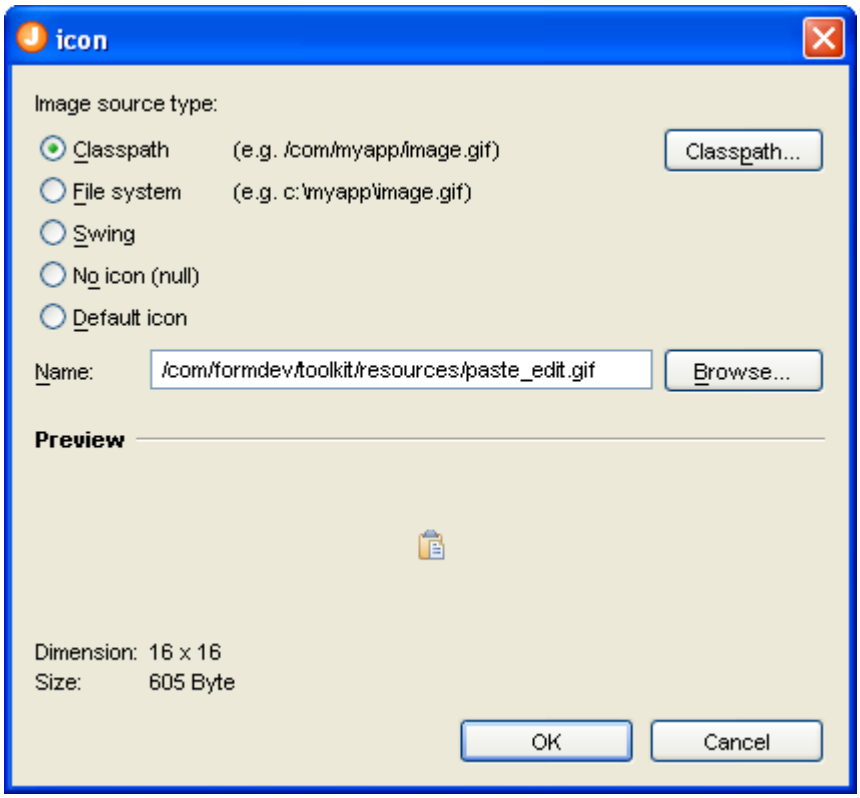
In the properties table, you can quickly change the style (bold and italic) and the size of the font.

In the custom editor you can choose one of the tabs to specify either absolute fonts, derived fonts or predefined fonts.



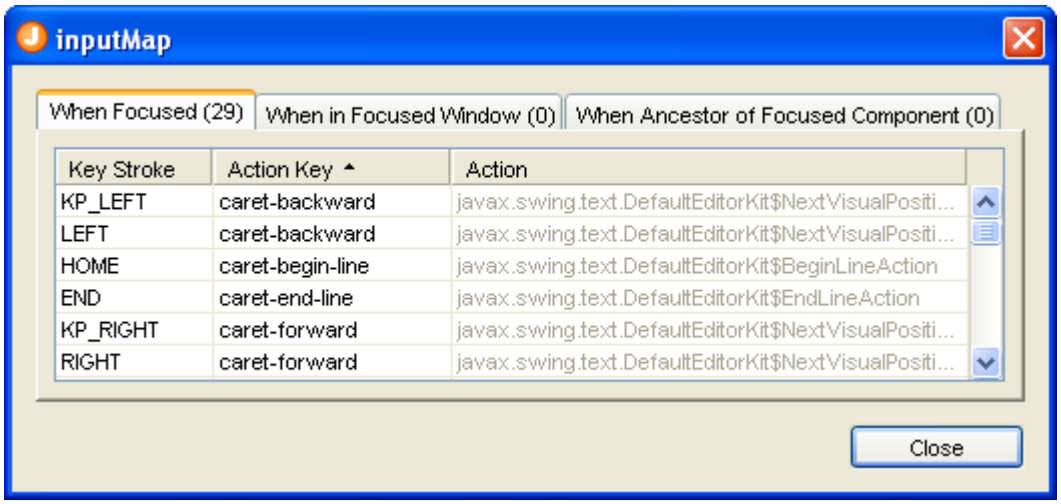
Icon (javax.swing) and Image (java.awt)

This custom editor allows you to choose an icon. Either use an icon from the classpath, from the file system or from the Swing UIManager (look and feel). It is recommended to use the classpath and embed your icons into your application JAR.



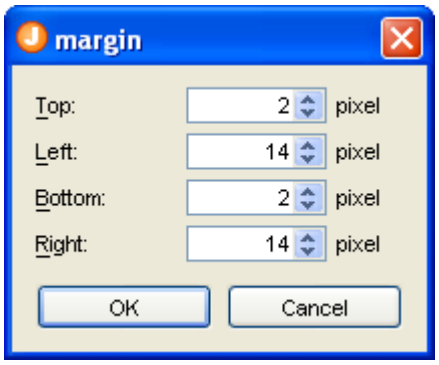
InputMap (javax.swing)

This (read-only) custom editor allows you to see the key strokes registered for a component in its input map. The information in the column "Action" comes from the action map of the component and shows which action classes are assigned to key strokes. The JComponent property "inputMap" is read-only. Select the **Show read-only Properties** button in the Properties view toolbar to make it visible.



Insets (java.awt)

Either edit the insets in the property table or use the custom editor.

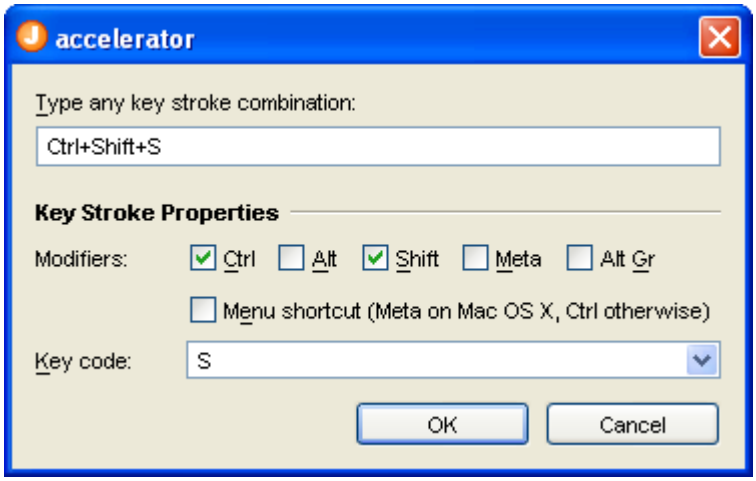


KeyStroke (javax.swing)

In the properties table, you can enter a string representation of the keystroke. E.g. "Ctrl+C" or "Ctrl+Shift+S".

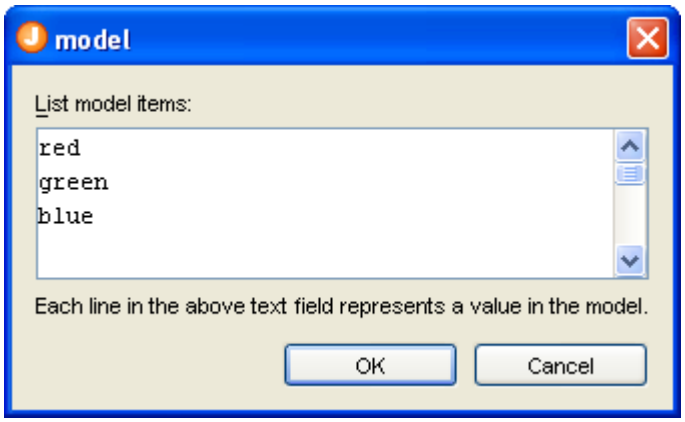
The custom editor supports two ways to specify a keystroke. Either type any key stroke combination if the focus is in the first field or use the controls below.

The KeyStroke editor supports menu shortcut modifier key (**Command** key on Mac OS X, **Ctrl** key otherwise).



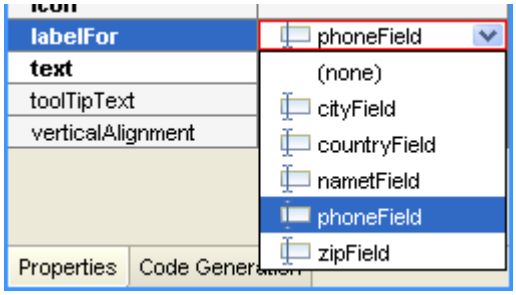
ListModel (javax.swing)

This custom editor allows you to specify string values for a list.



Object (java.lang)

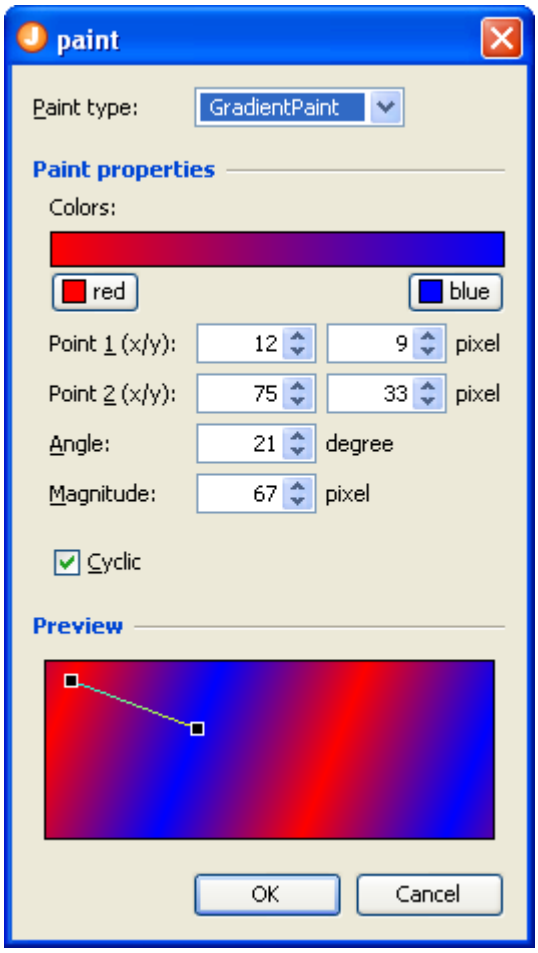
This editor allows you to reference any (non-visual) JavaBean as a property value. Often used for `JLabel.labelFor`.



Paint (java.awt)

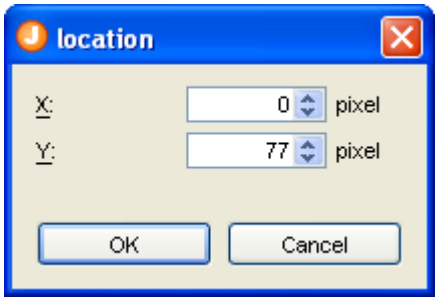
This editor allows you to specify a `java.awt.Paint` object (used by `java.awt.Graphics2D`). Use the combo box at the top of the dialog to choose a paint type. In the mid area of the dialog you can edit the paint properties. This area is different for each paint type. At the bottom, you can see a preview of the paint. For GradientPaint you can click-and-drag the handles in the preview area to move the points.

Following paint types are supported: `Color` and `GradientPaint`.



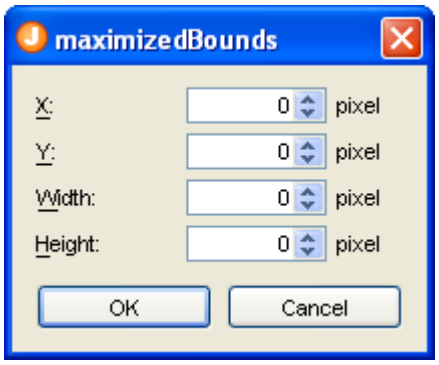
Point (java.awt)

Either edit the point in the property table or use the custom editor.



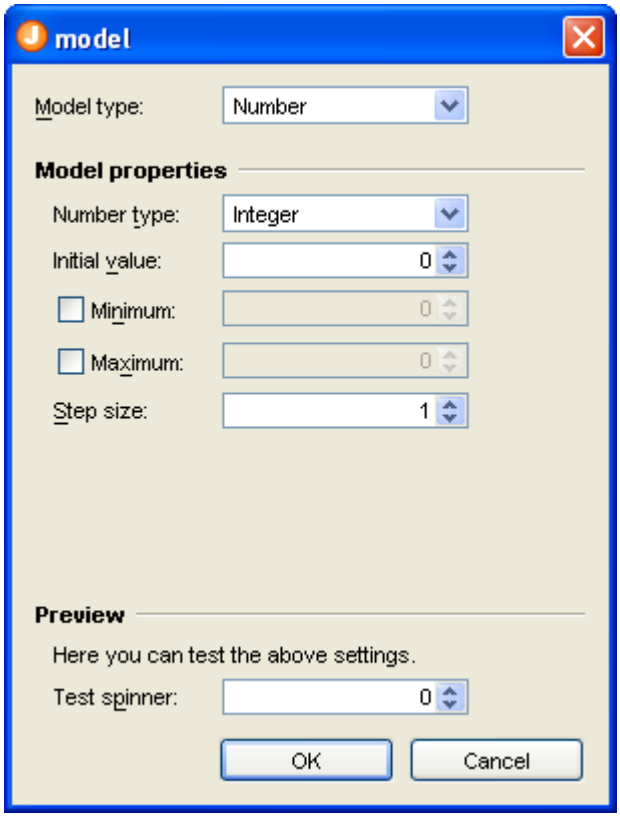
Rectangle (java.awt)

Either edit the rectangle in the property table or use the custom editor.



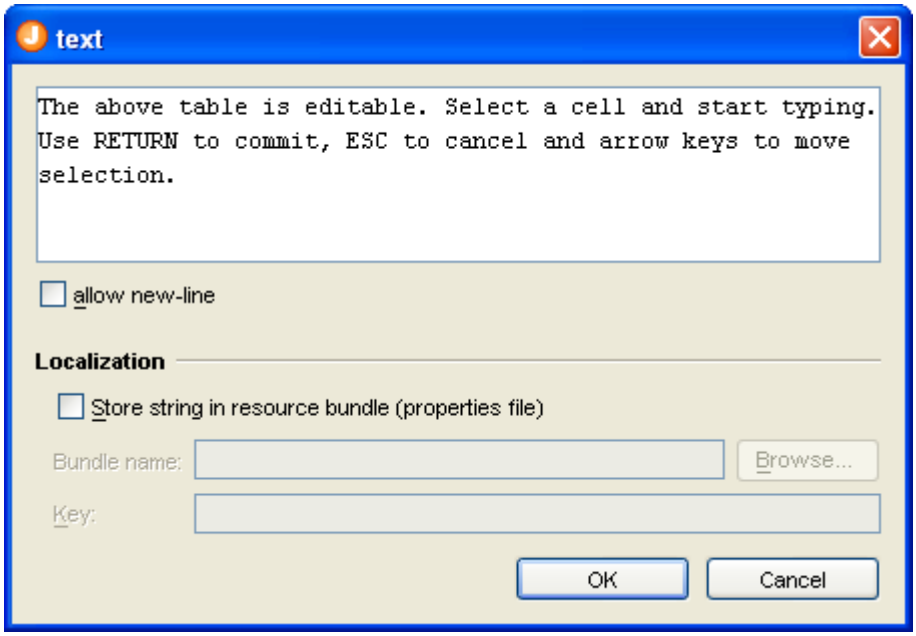
SpinnerModel (javax.swing)

This custom editor allows you to specify a spinner model (used by `JSpinner`). Use the combo box at the top of the dialog to choose a spinner model type (Number, Date or List). In the mid area of the dialog you can edit the model properties. This area is different for each model type. At the bottom, you can see a test spinner where you can test the spinner model.



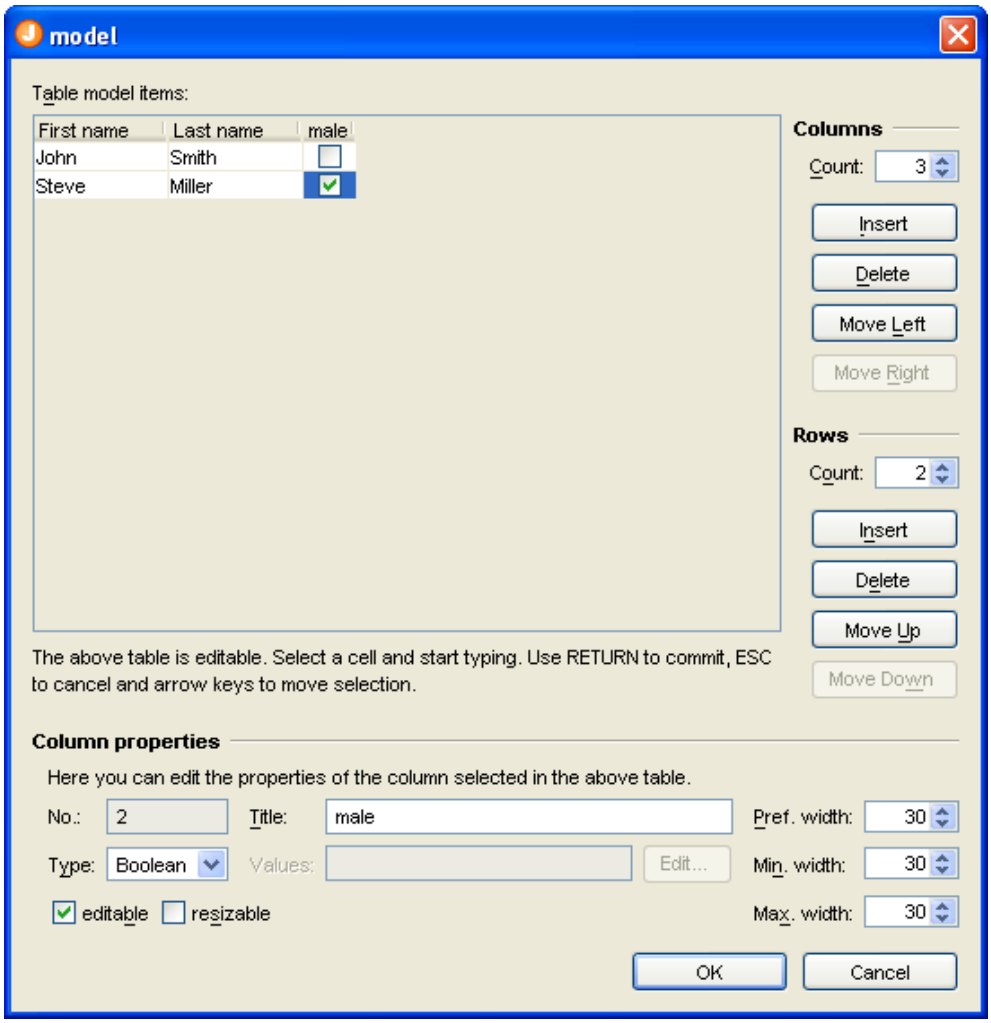
String (java.lang)

Either edit the string in the property table or use the custom editor. Switch the "allow new-line" check box on, if you want enter new lines.



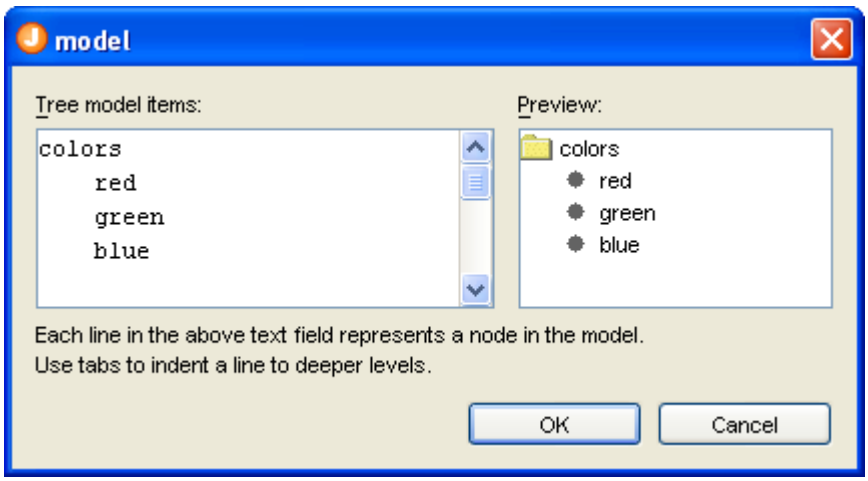
TableModel (javax.swing)

This custom editor allows you to specify values for a table.



TreeModel (javax.swing)

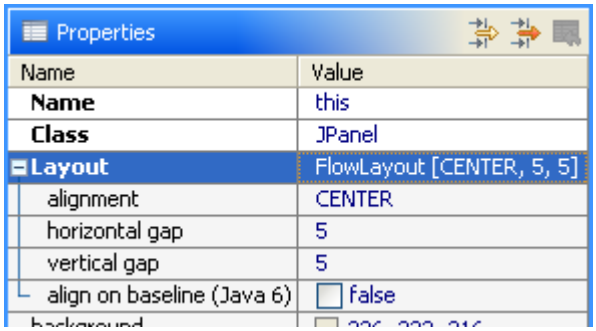
This custom editor allows you to specify string values for a tree.



Layout Properties

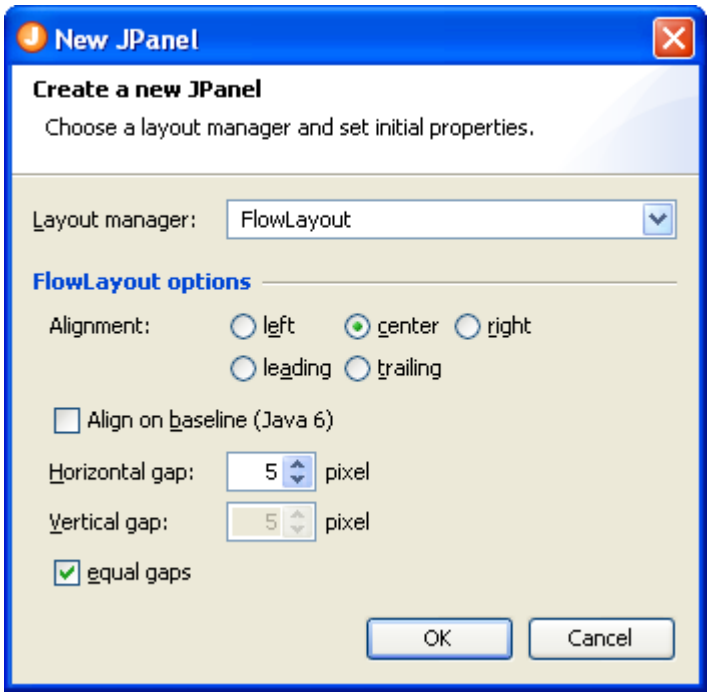
Each container component that has a layout manager has layout properties. The list of layout properties depends on the used layout manager.

Select a container component in the Design or Structure view to see its layout properties in the Properties view.



This screenshot shows layout properties (alignment, horizontal and vertical gap) of a container that has a FlowLayout.

When you add a container component to a form, following dialog appears and you can choose the layout manager for the new container. You can also set the layout properties in this dialog.

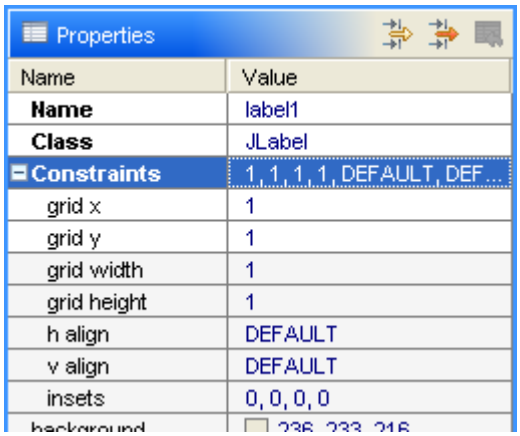


Constraints Properties

Constraints properties are related to layout managers. Some layout managers (FormLayout, TableLayout, GridBagLayout, ...) use constraints to associate layout information to the child components of a container.

The list of constraints properties depends on the layout manager of the parent component.

Select a component in the Design or Structure view to see its constraints properties in the Properties view.



This screenshot shows constraints properties of a component in a FormLayout.

Client Properties

What is a client property?

Swings base class for all components, `javax.swing.JComponent`, provides following methods that allows you to set and get user-defined properties:

```
public final Object getClientProperty(Object key);
public final void putClientProperty(Object key, Object value);
```

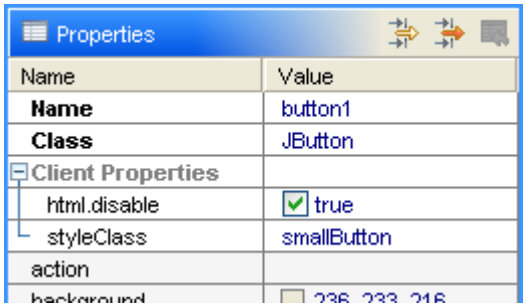
Some Swing components use client properties to change their behavior. E.g. for JLabel you can disable HTML display with `label.putClientProperty("html.disable", Boolean.TRUE);` You can use client properties to store any information in components. Visit Finally... Client Properties You Can Use on Ben Galbraith's Blog for a use case.

Define client properties

You can define client properties on the Client Properties page in the Preferences dialog.

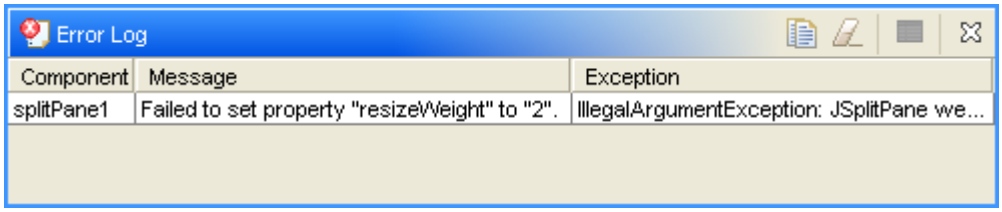
Edit client properties

If you've defined client properties, JFormDesigner shows them in the Properties view, where you can set the values of the client properties.



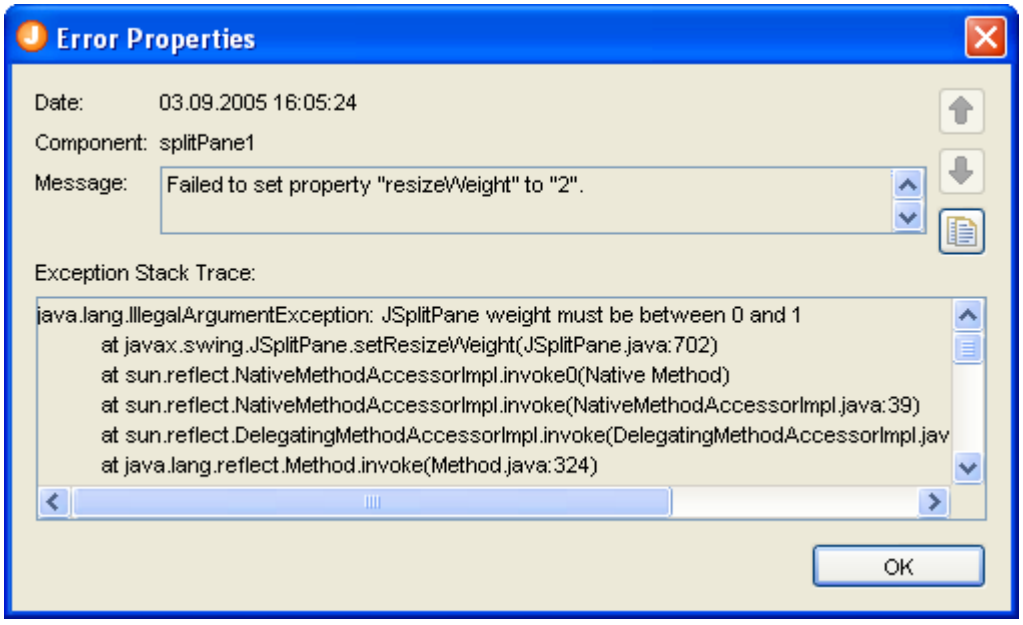
Error Log View

This view appears at the bottom of the main window if an exception is throw by a bean. You can see which bean causes the problem and the stack trace of the exception. This makes it much easier to solve problems when using your own (or 3rd party) beans.



Toolbar commands	
Copy Log	Copies all log records to the clipboard.
Clear Log	Clears the log.
Properties	Displays the properties of the selected log record in a dialog (see below).
Close	Closes the Error Log view.

Double-click on a log entry to see its details:



How to fix errors

This mainly depends on the error. The problem shown in the above screenshots is easy to fix by setting `resizeWeight` to a value between 0 and 1.

If the problem occurs in your own beans, use the stack trace to locate the problem and fix it in your bean's source code. After compiling your bean, click the **Refresh** button in the designer toolbar (or press **F5**) to reload your bean.

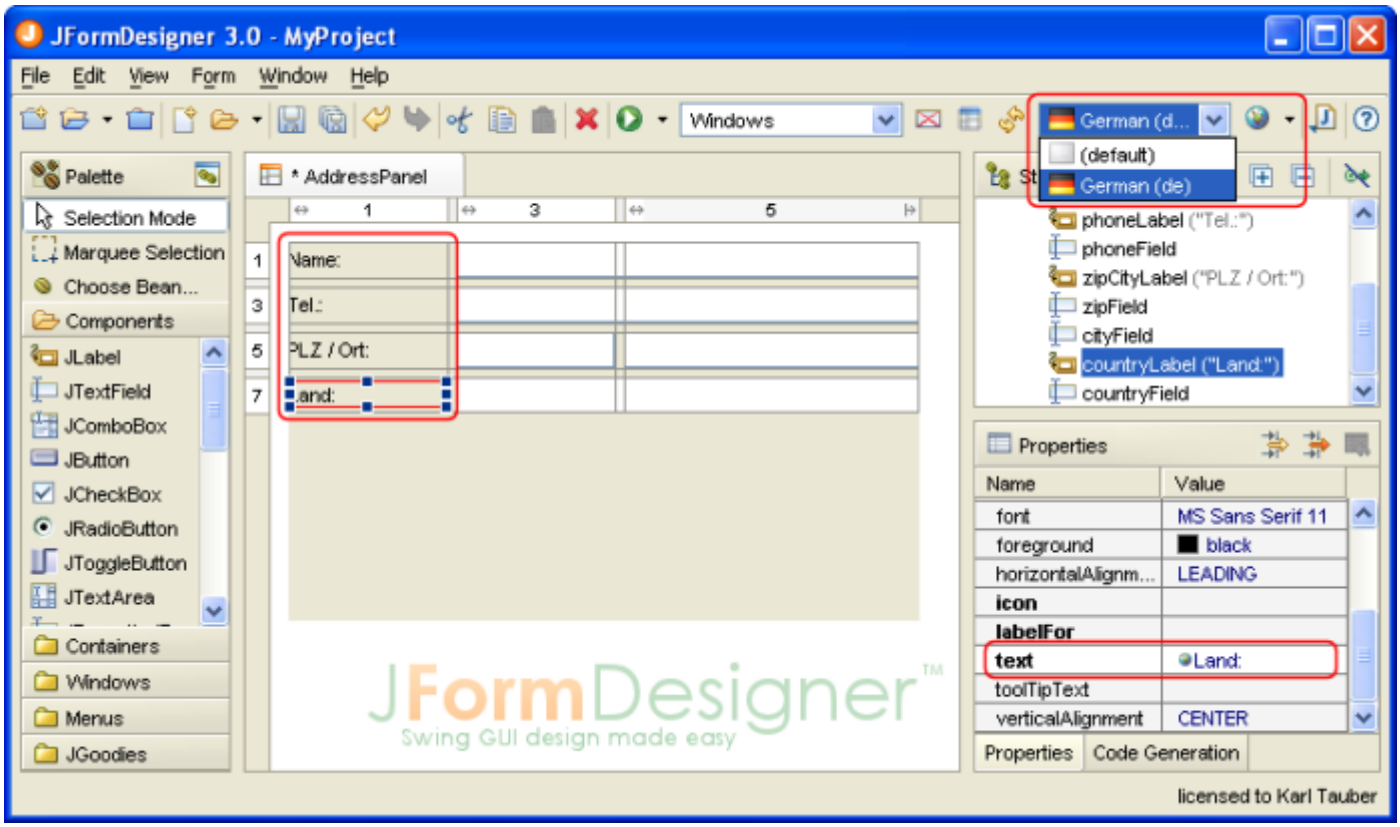
If you are using 3rd party beans, it is possible that you need to add additional libraries to the classpath. You should be able to identify such a problem on the kind of exception. In this case, add the needed libraries to the JFormDesigner classpath of the current Project, and refresh the Design view.

Localization

JFormDesigner provides easy-to-use and powerful localization/internationalization support:

- Externalize and internalize strings.
- Edit resource bundle strings.
- Create new locales.
- Delete locales.
- Switch locale used in Design view.
- In-place-editing strings of current locale.
- Auto-externalize strings.
- Choose existing strings.
- Updates resource keys when renaming components.
- Copies resource strings when copying components.
- Removes resource strings when deleting components.
- Localization preferences.
- Fully integrated in undo/redo.

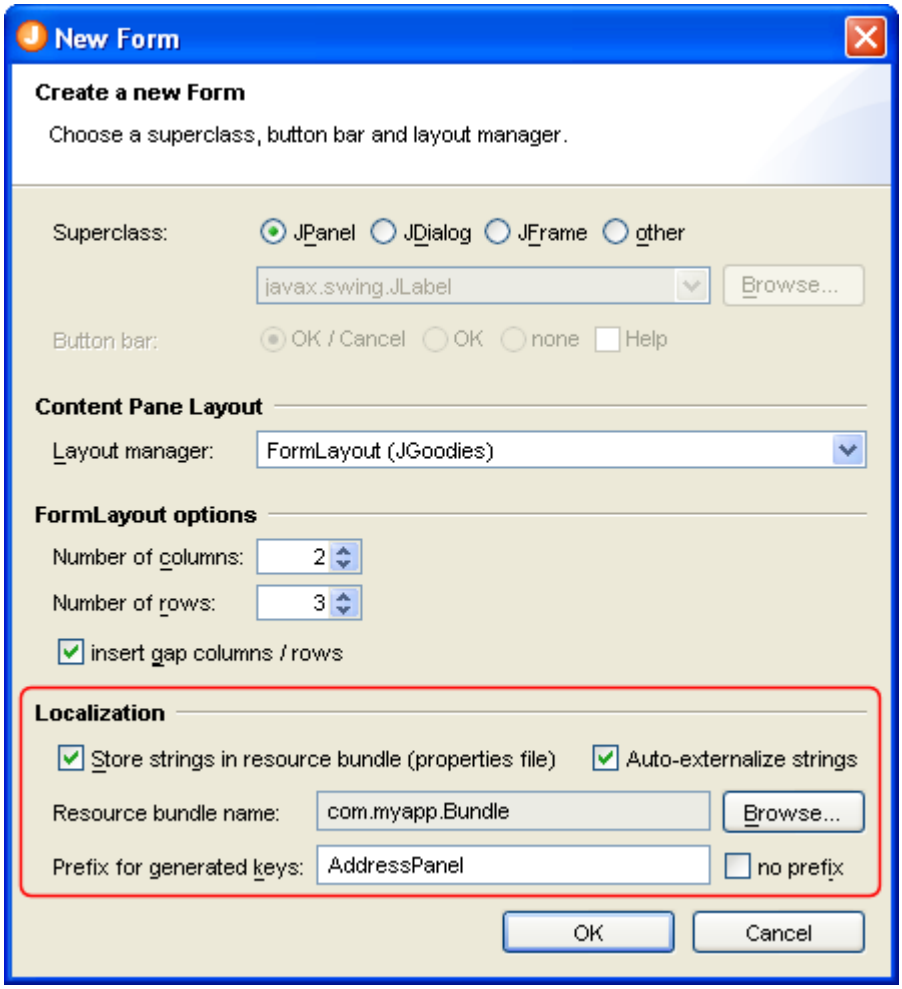
The locales combo box in the toolbar allows you to select the locale used in the Design, Structure and Properties views. If you in-place-edit a localized string in the Design view, you change it in the current locale. A small globe in front of property values in the Properties view indicates that the string is localized (stored in a properties file).



Create a new localized form

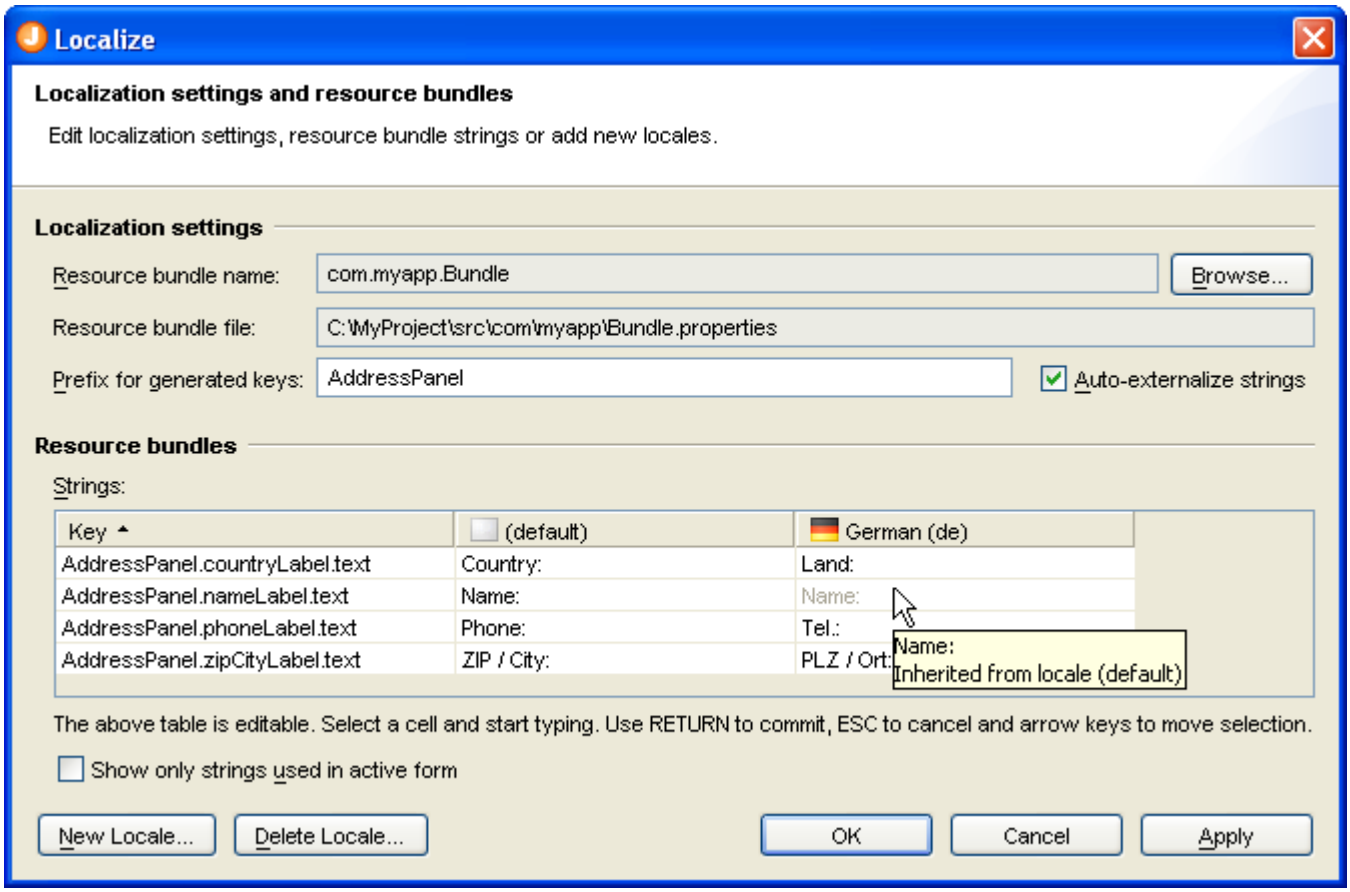
When creating a new form, you can specify that JFormDesigner should put all strings into a resource bundle (.properties file). In the **New Form** dialog select the **Store strings in resource bundle** check box, specify a resource bundle name and a prefix for generated keys. If **Auto-externalize strings** is selected, then JFormDesigner automatically puts all new strings into the properties file (auto-externalize). E.g. when you add a `JLabel` to the form and change the "text" and "toolTipText" properties, both strings will be put into the properties file.

To localize existing forms use Externalize Strings.



Edit localization settings and resource bundle strings

To edit localization settings and resource bundle strings, select **Form > Localize** from the main menu or click the **Localize** button in the toolbar. Here you can create or delete locales and edit strings. The light gray color used to draw the string "Name:" in the table column "German" indicates that the string is inherited from a parent locale.



The **Resource bundle name** field is used to locate the properties files within the Source Folders of the current Project. Use the **Browse** button to choose a resource bundle (.properties file).

In the **Prefix for generated keys** field you can specify a prefix for generated resource bundle keys. The format for generated keys is "<prefix>.<componentName>.<propertyName>". You can change the separator ('.') in the Localization preferences.

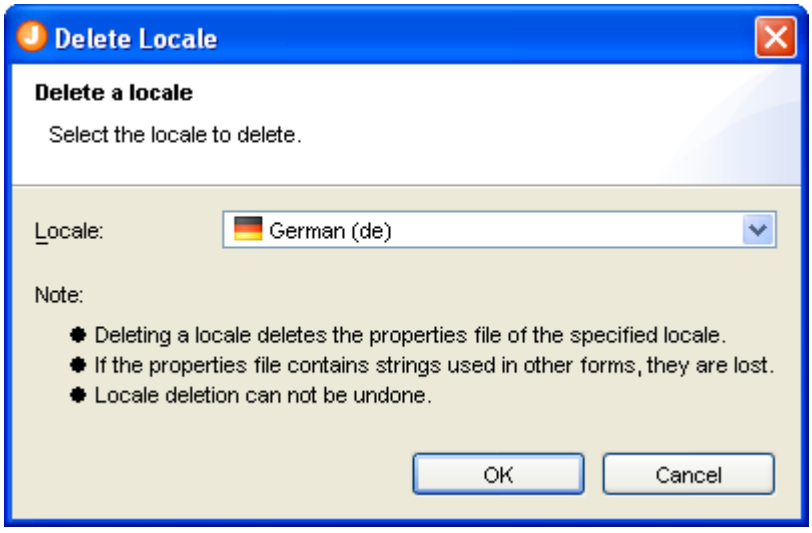
If the **Auto-externalize strings** check box is selected, then JFormDesigner automatically puts all new strings into the properties file. E.g. when you add a `JLabel` to the form and change the "text" and "toolTipText" properties, both strings will be put into the properties file. You can exclude properties from externalization in the Localization preferences.

Create new locale

To create a new locale, either select **Form > New Locale** from the main menu, **New Locale** form the toobar or click the **New Locale** button in the **Localize** dialog. Select a language and an optional country. You can copy strings from an existing locale into the new locale, but JFormDesigner fully supports inheritance in the same way as specified by `java.util.ResourceBundle`. E.g. if a message is not in locale "de_AT" then it will be loaded from locale "de".

Delete a locale

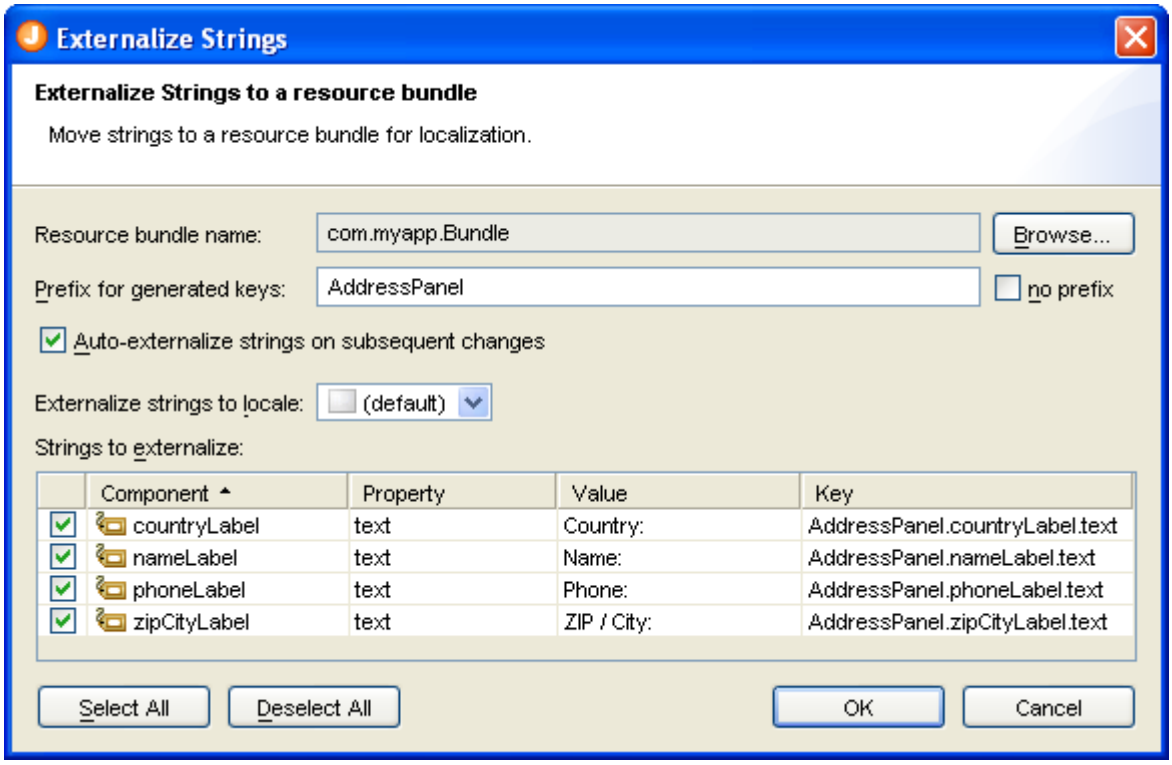
To delete an existing locale, either select **Form > Delete Locale** from the main menu, **Delete Locale** form the toobar or click the **Delete Locale** button in the **Localize** dialog. Select the locale to delete.



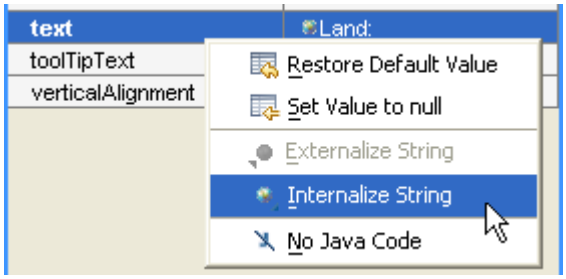
Externalize strings

Externalizing allows you to move strings from a .jfd file to a .properties file. If you want localize existing forms, start here.

Select **Form > Externalize Strings** from the main menu or **Externalize Strings** from the toolbar, specify the resource bundle name, the prefix for generated keys and select/deselect the strings to externalize. You can exclude properties from externalization in the Localization preferences.



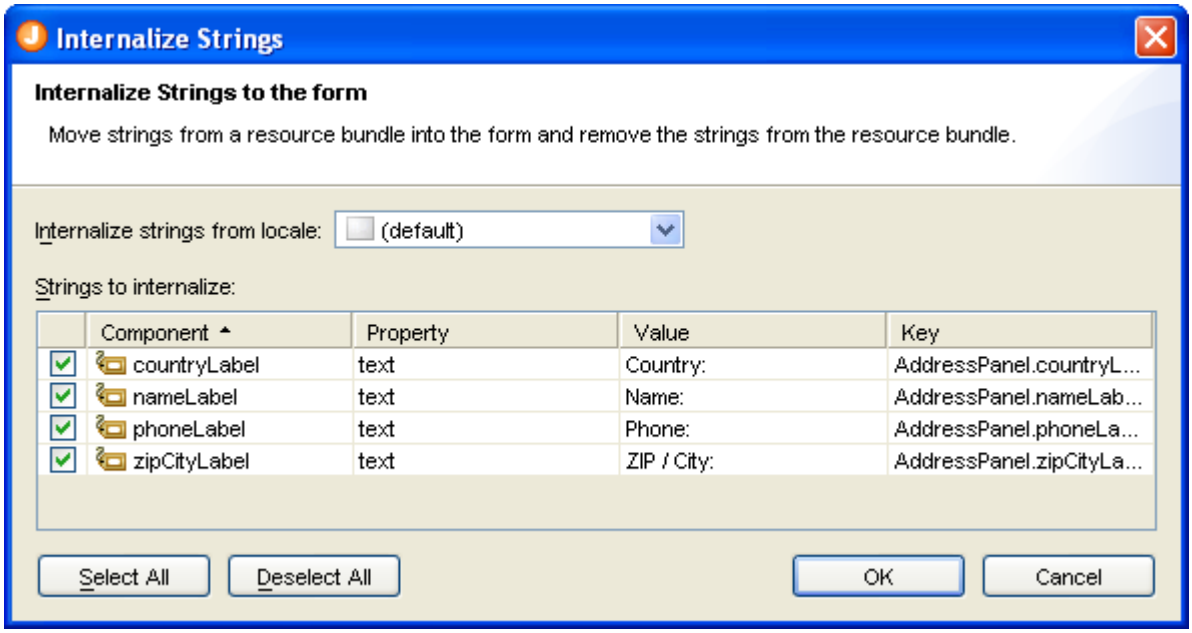
You can also externalize and internalize properties in the Properties view.



Internalize strings

Internalizing allows you to move strings from a .properties file to a .jfd file.

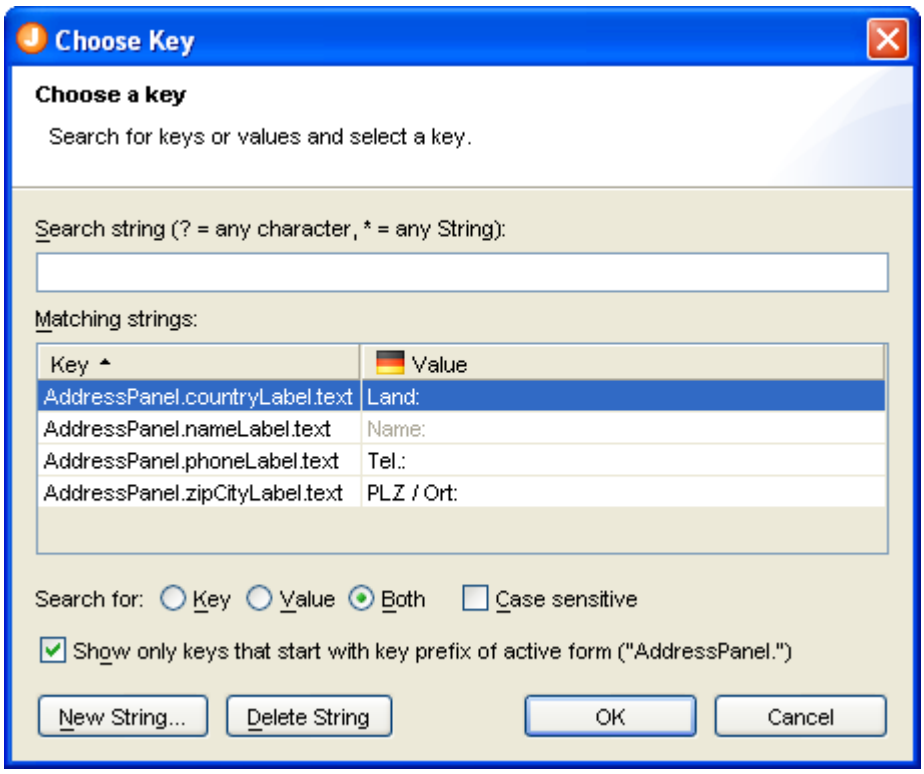
Select **Form > Internalize Strings** from the main menu, specify the locale to internalize from and select/deselect the strings to internalize. If you internalize all strings, JFormDesigner asks you whether you want to disable localization for the form.



Choose existing strings

The globe button () in the Properties view, which is only available for localized forms and string properties, allows you to choose existing strings from the resource bundle of the form.

In the **Choose Key** dialog you can search for keys and/or values. Then select a key in the table and press OK to use its value in the form.



Projects

Stand-alone edition only. The **IDE plug-ins** use the source folders and classpath from the IDE projects.

Projects allow you to store project specific options in project files. You can create new projects or open existing projects using the menubar or toolbar.

When you start JFormDesigner the first time, it creates and opens a default project named DefaultProject.jfdproj in the folder ${user.home}/.jformdesigner, where ${user.home} is your home directory. You can see the value of ${user.home} in the About dialog on the System tab.

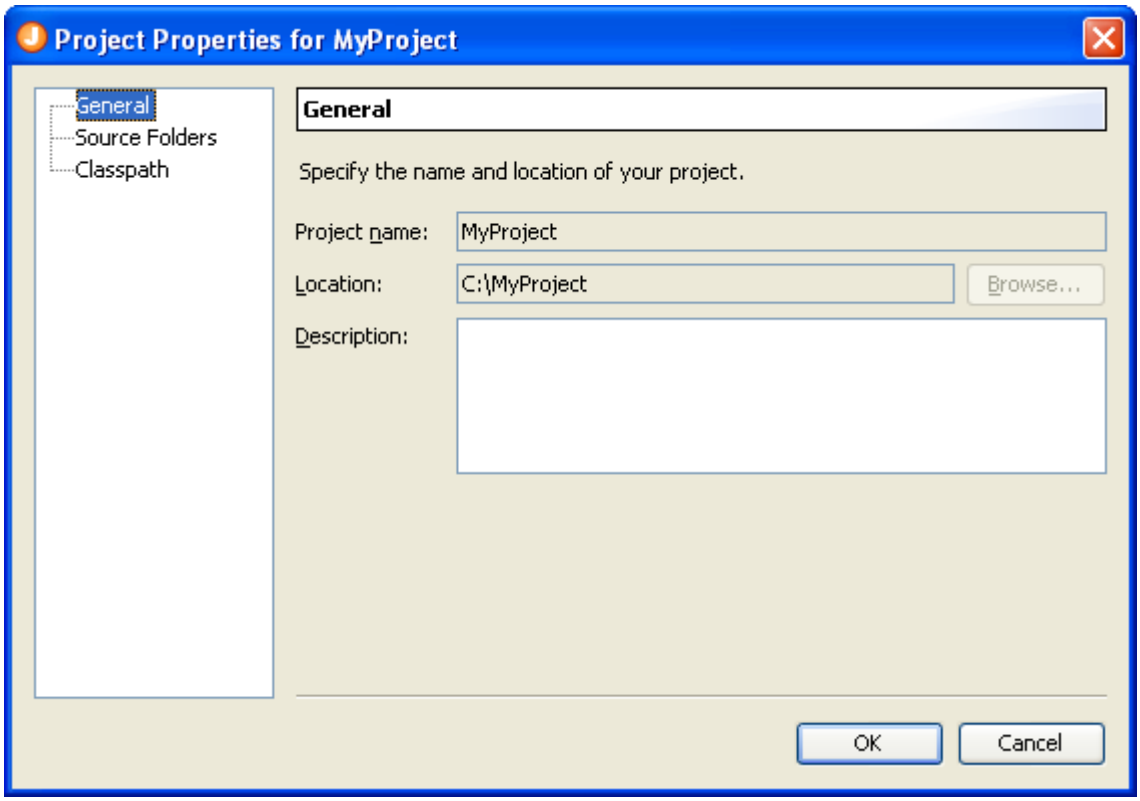
You can use the default project, but it is recommended to create an own JFormDesigner project in your project root folder. Then you can commit the JFormDesigner project file into a version control system and reuse it on other computers. Paths in the project file are stored relative to the location of the project file. Project files have the extension **.jfdproj**

Pages:

- General
- Source Folders
- Classpath

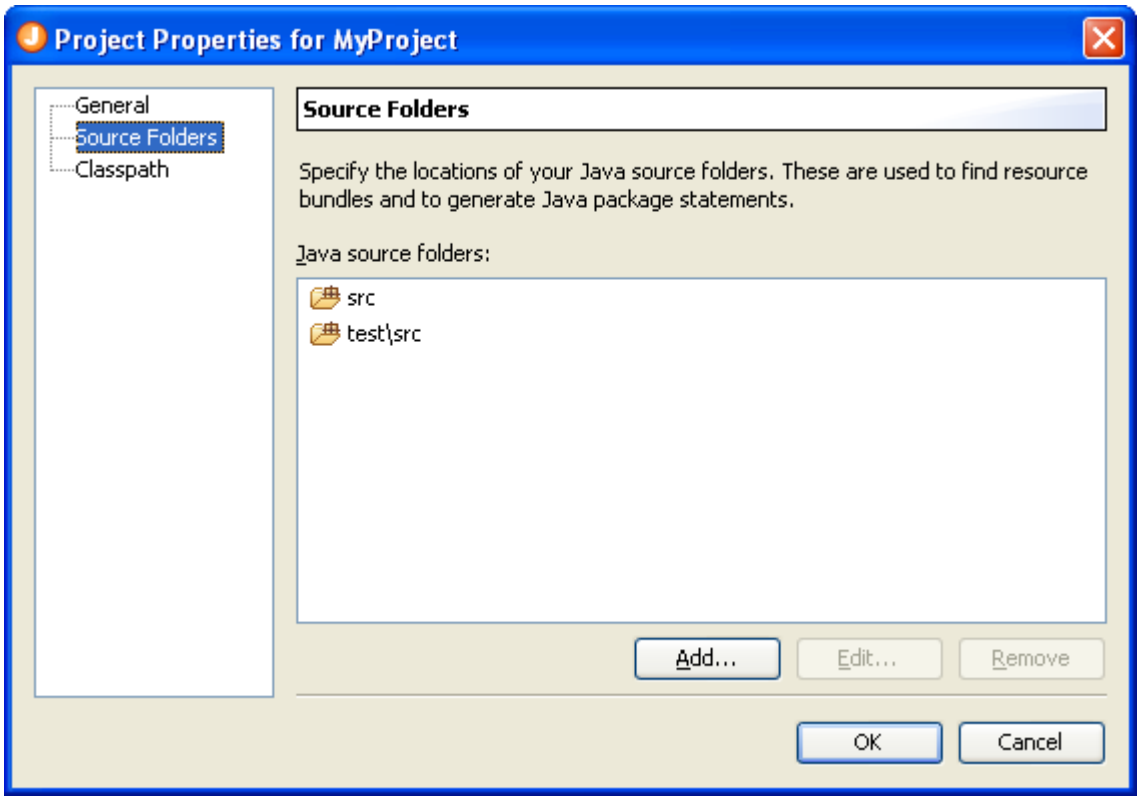
General

When creating a new project, you can specify a project name and the location where to store the project file.



Source Folders

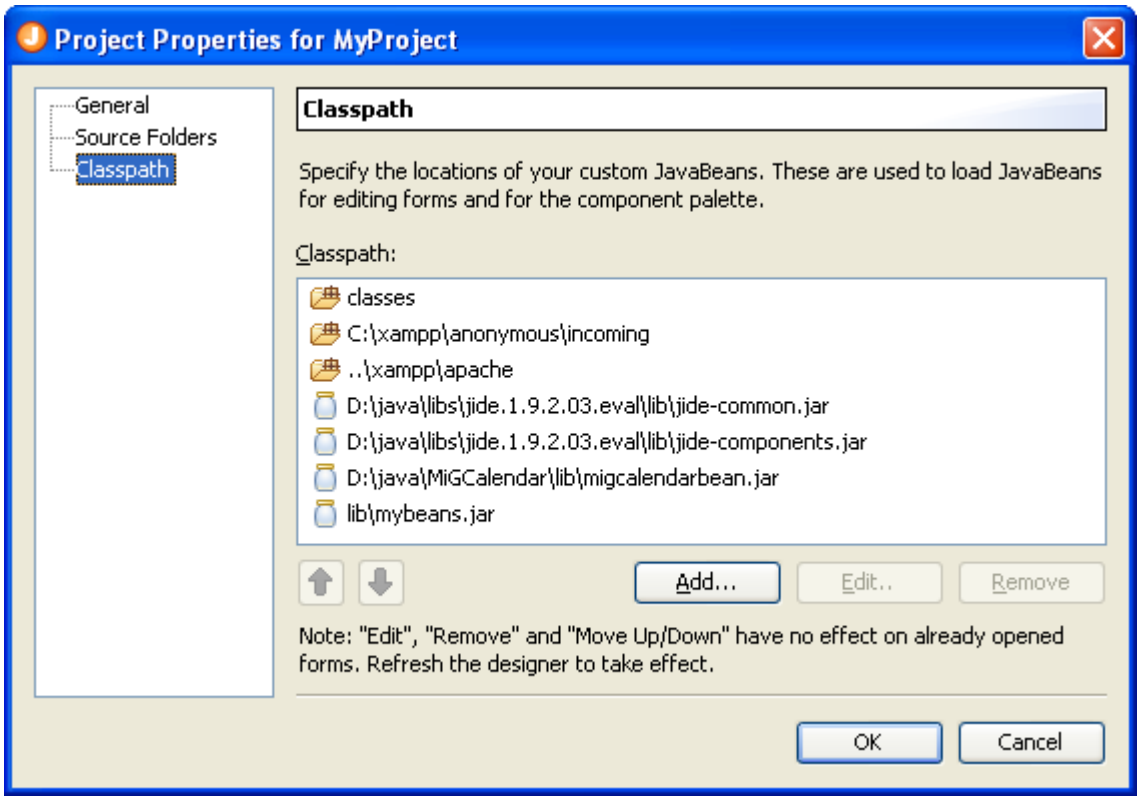
On this page, you can specify the locations of your Java source folders. Source folders are the root of packages containing .java files and are used find resource bundles for localization and are also used by the Java code generator to generate package statements.



If the folders list is focused, you can use the **Insert** key to add folders or the **Delete** key to delete selected folders.

Classpath

To use your custom components (JavaBeans), JFormDesigner needs to know, from where to load the JavaBean classes. Specify the locations of your custom JavaBeans on this page. You can add JAR files or folders containing .class files.



If the classpath list is focused, you can use the **Insert** key to add folders/JAR files, the **Delete** key to delete selected folders/JAR files, **Ctrl+Up** keys to move selected items up or **Ctrl+Down** keys to move selected items down.

Preferences

This dialog is used to set user preferences.

Stand-alone: Select **Window > Preferences** from the menu to open this dialog.

Eclipse plug-in: The JFormDesigner preferences are fully integrated into the Eclipse preferences dialog. Select **Window > Preferences** from the menu to open it and then expand the node "JFormDesigner" in the tree.

IntelliJ IDEA plug-in: IntelliJ IDEA uses the term "Settings" instead of "Preferences". The JFormDesigner preferences are fully integrated into the IntelliJ IDEA settings dialog. Select **File > Settings** from the menu to open it and then click the icon named "JFormDesigner".

JBuilder plug-in: The JFormDesigner preferences are fully integrated into JBuilder preferences dialog. Select **Tools > Preferences** from the menu to open it.

Pages

Import and export preferences

You can use the **Import** button to import preferences from a file and the **Export** button to export preferences to a file. This preferences file is compatible with all JFormDesigner editions. On export, you can specify what parts of the preferences you want export.

Eclipse plug-in: You can use the menu commands **File > Import** and **File > Export** to import and export preferences to/from Eclipse preferences files.

IntelliJ IDEA plug-in: You can use the menu commands **File > Import Settings** and **File > Export Settings** to import and export settings to/from IntelliJ IDEA preferences files.

JBuilder plug-in: Import and export of preferences is not supported.

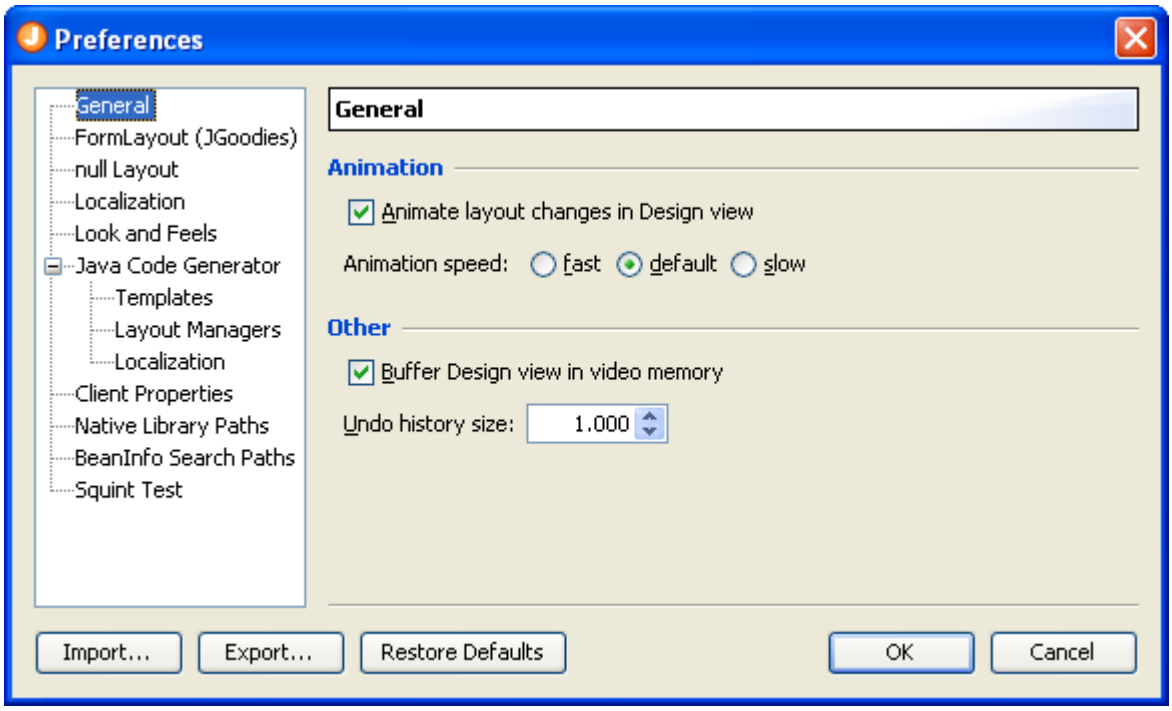
Note: Each IDE uses its own file format for preferences. The only way to transfer preferences between the different JFormDesigner editions is to use JFormDesigner preferences files.

Restore defaults

Use the **Restore Defaults** button to restore the values of the active page to its defaults.

General

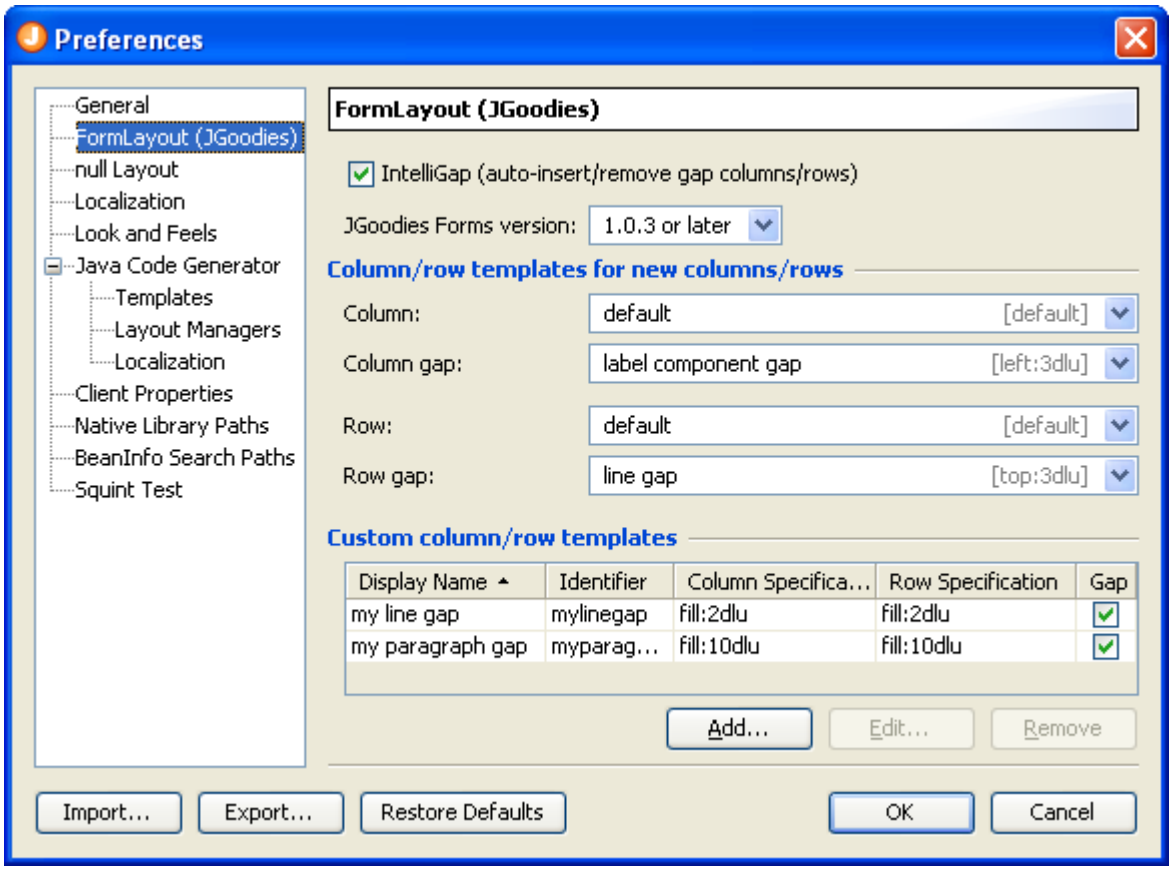
On this page, you can specify general options.



Option	Description	Default
Animate layout changes in Design view	If enabled, changes to the layout in the Design view are done animated.	On
Animation speed	The speed of the animation.	default
Buffer Design view in video memory	If enabled, parts of the Design view are buffered in the video memory of the graphics card to improve painting speed.	On
Undo history size	The maximum number of steps in the undo history of a form.	1000

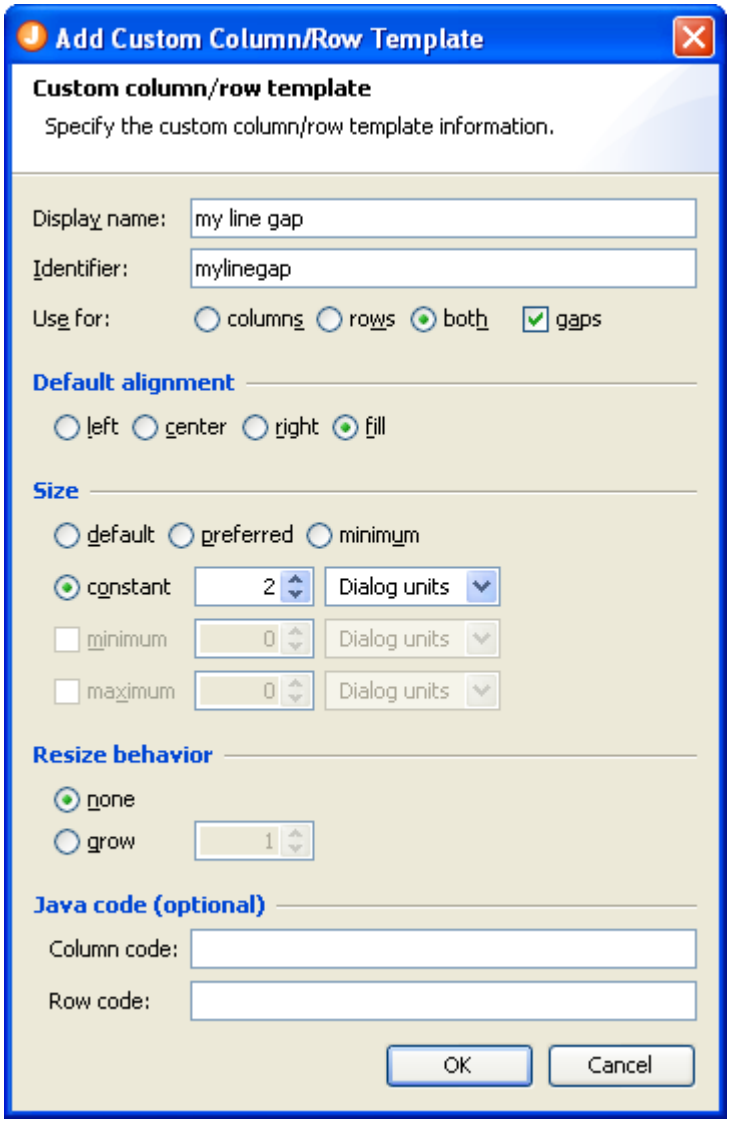
FormLayout (JGoodies)

On this page, you can specify FormLayout related options.



Option	Description	Default
IntelliGap	If enabled, JFormDesigner automatically inserts/removes gap columns/rows.	On
JGoodies Forms version	Required JGoodies Forms version for the created forms. JGoodies Forms 1.0.3 and later require Java 1.4 or later. JGoodies Forms 1.0.2 is the last version that supports Java 1.3.	1.0.3 or later
Column/row templates for new columns/rows	Here you can specify the column and row templates that should be used when new columns or rows are inserted.	
Column	The column template used for new columns.	default
Column gap	The column template used for new gap columns.	label component gap
Row	The row template used for new rows.	default
Row gap	The row template used for new gap rows.	line gap
Custom column/row templates	If the predefined templates does not fit to your needs, you can define your own here.	

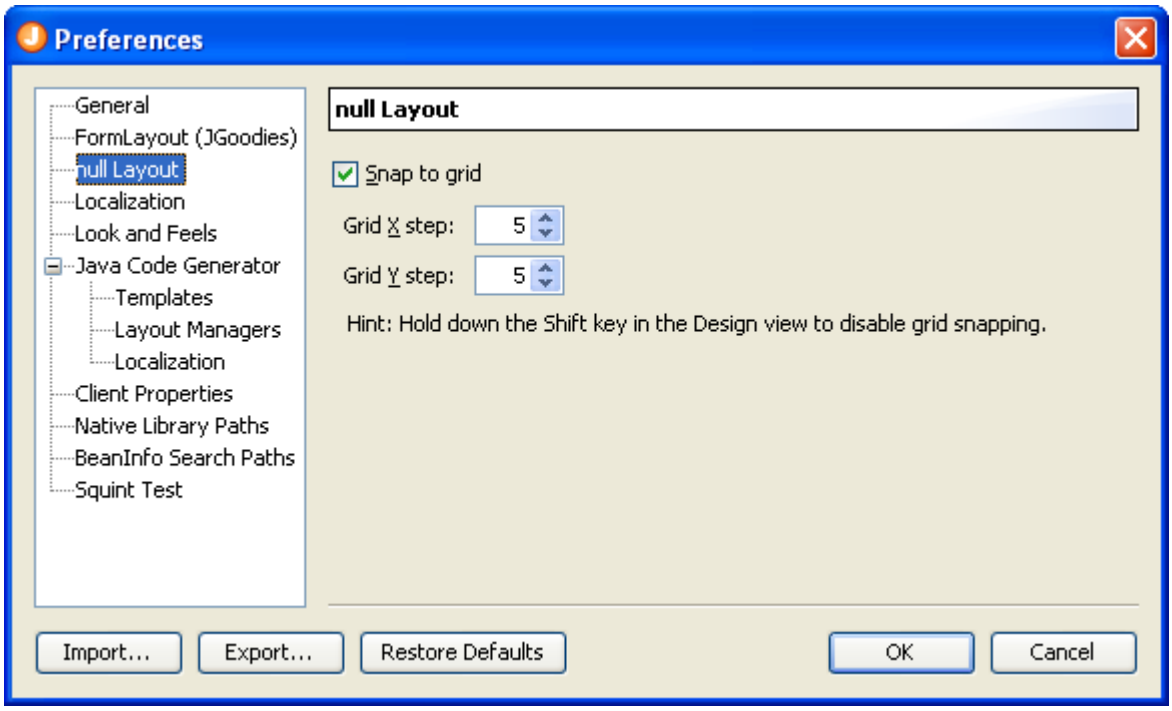
Custom column/row templates



Option	Description
Display name	The display name is used within JFormDesigner whenever the template is shown in combo boxes or popup menus.
Identifier	The (unique) identifier is stored in form files. Choose a short string. Only letters and digits are allowed.
Use for	Specifies whether the template should be used for columns, rows or both. Also specifies whether it represents a gap column/row.
Default alignment	The default alignment of the components within a column/row. Used if the value of the component constraint properties "h align" or "v align" are set to DEFAULT.
Size	The width of a column or height of a row. You can use default, preferred or minimum component size. Or a constant size. It is also possible to specify a minimum and a maximum size. Note that the maximum size does not limit the column/row size if the column/row can grow (see resize behavior).
Resize behavior	The resize weight of the column/row.
Java code	Optional Java code used by the Java code generator. Useful if you have factory classes for ColumnSpecs and RowSpecs.

null Layout

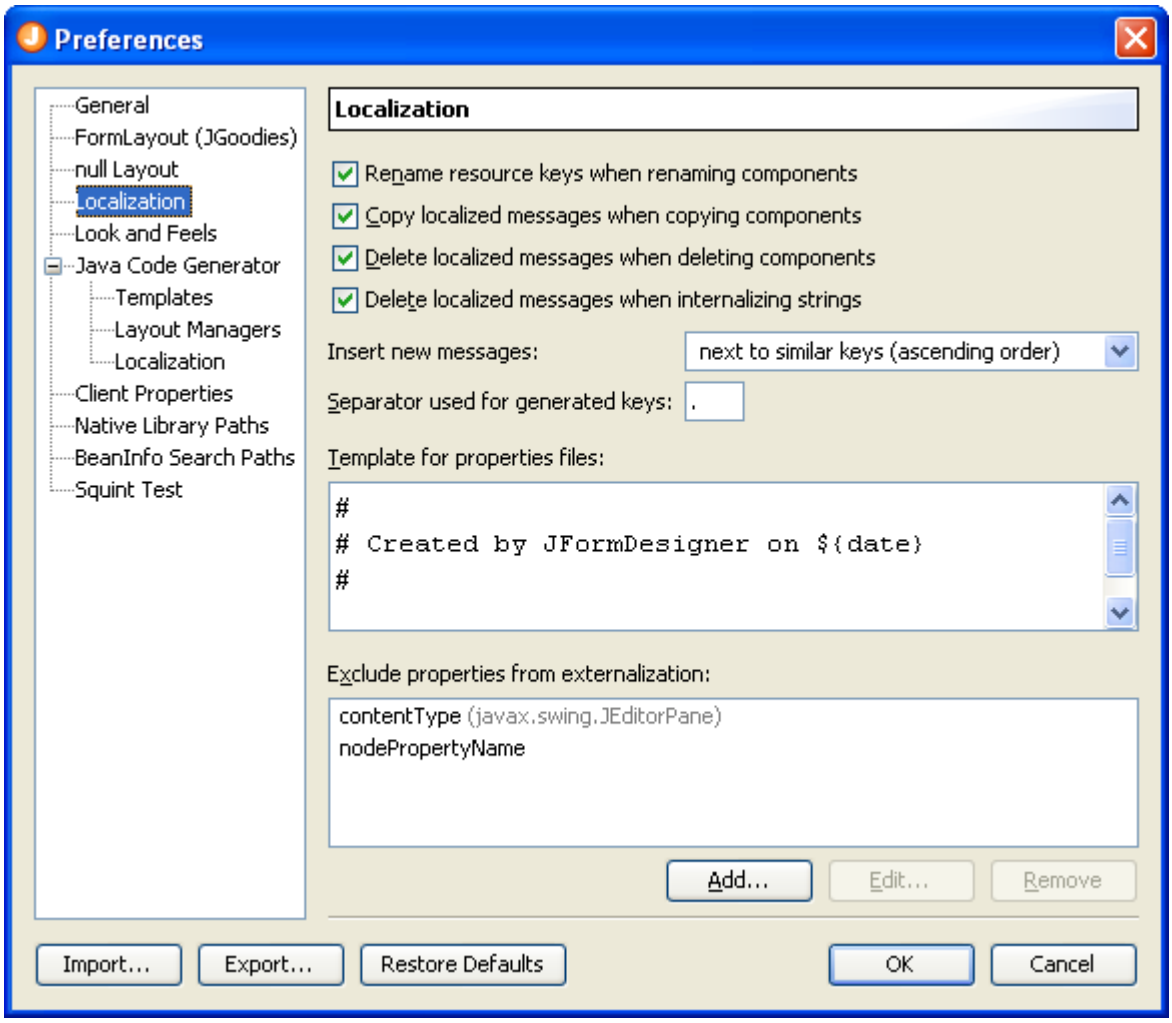
On this page, you can specify null layout related options.



Option	Description	Default
Snap to grid	If enabled, snap to the grid specified below when moving or resizing a component in null layout.	On
Grid X step	The horizontal step size of the grid.	5
Grid Y step	The vertical step size of the grid.	5

Localization

On this page, you can specify localization related options.



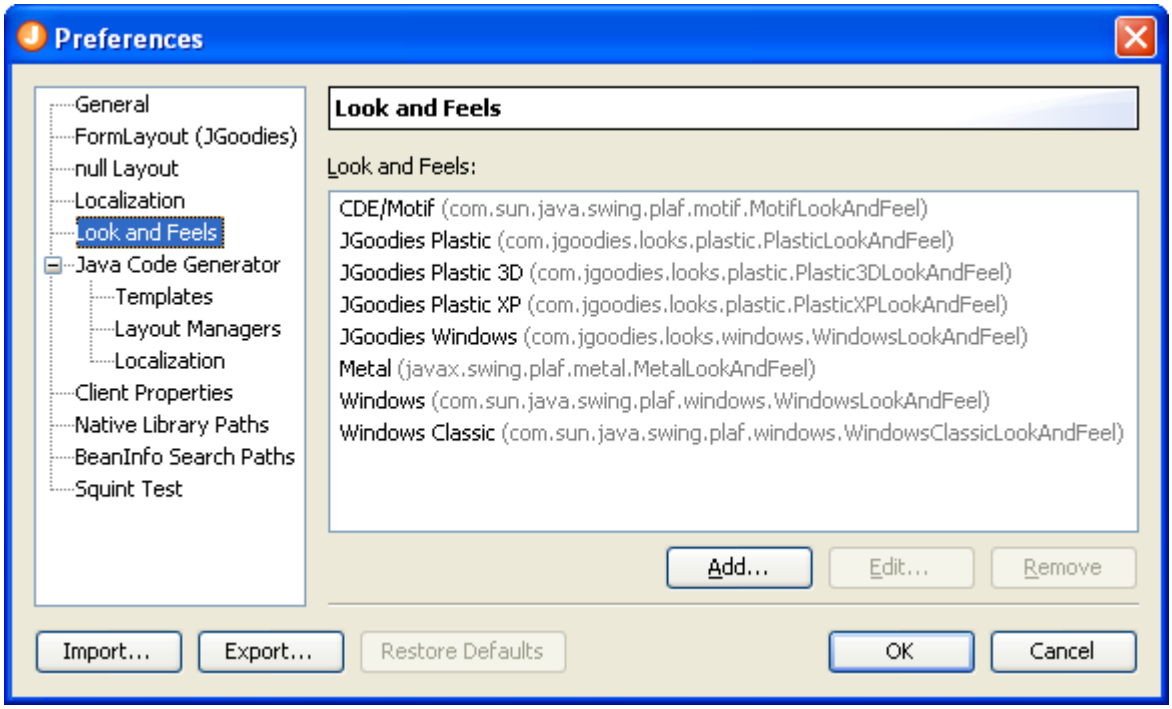
Option	Description	Default
Rename resource keys when renaming components	If enabled, auto-rename resource keys when renaming components and the resource key contains the old component name.	On
Copy localized messages when copying components	If enabled, duplicate localized strings in all locales when copying components.	On
Delete localized messages when deleting components	If enabled, auto-delete localized strings, that were used by the deleted components, from all locales.	On
Delete localized messages when internalizing strings	If enabled, auto-delete localized strings, that were internalized, from all locales.	On
Insert new messages	Specifies where new messages will be inserted into properties files. "next to similar keys" inserts new messages next to other similar keys so that messages that belong together are automatically at the same location in the properties file. "at the end of the properties file" always appends new messages to the end of the properties file.	next to similar keys (ascending order)
Separator used for generated keys	Separator used when generating a resource key.	'.'

Option	Description	Default
Template for properties files	Template used when creating new properties files.	
Exclude properties from externalization	Specify properties that should be excluded from externalization. Useful when using auto-externalization to ensure that some string property values stay in the Java code. If the list is focused, you can use the **Insert** key to add a property or the **Delete** key to delete selected properties.	

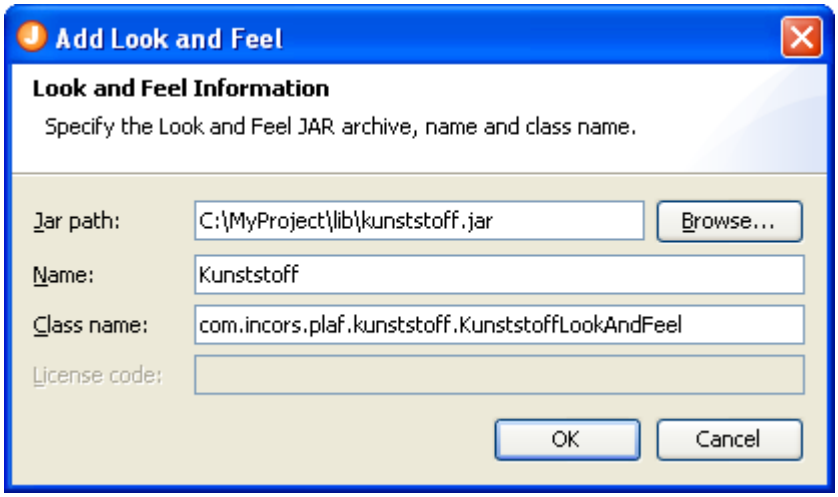
Look and Feels

On this page, you can add Swing look and feels for use in the Design view.

Note: Because Swing is not designed to use two look and feels at the same time (application and Design view), it can not guaranteed that each look and feel works without problems.



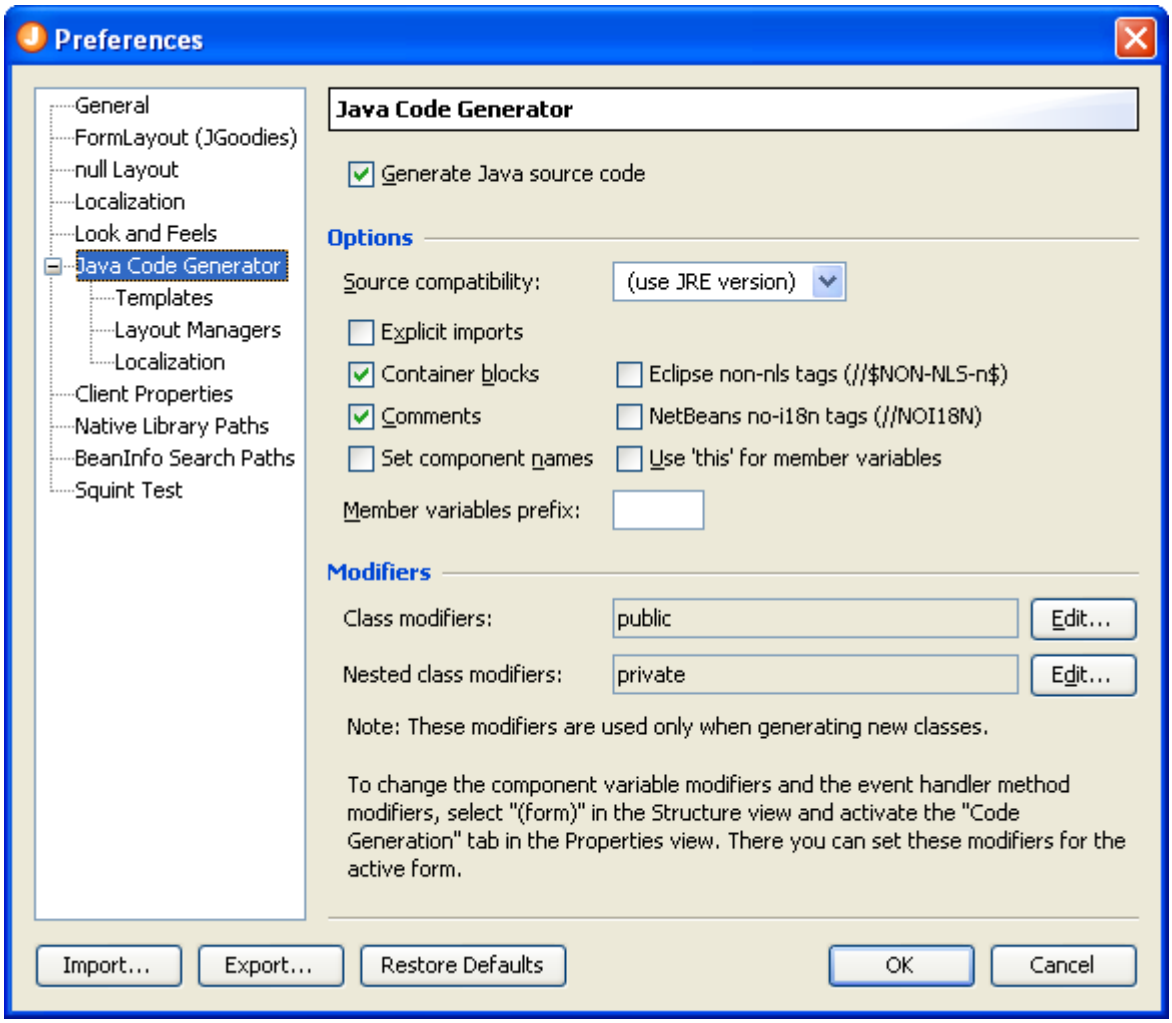
If the look and feels list is focused, you can use the **Insert** key to add a look and feel or the **Delete** key to delete selected look and feels.



Option	Description
Jar path	Full path name of the jar file that contains the look and feel classes. Use the **Browse** button to select a jar.
Name	Name of the look and feel used in the look and feel combo box in the Main Toolbar.
Class name	Class name of the look and feel class (derived from `javax.swing.LookAndFeel`).
License code	License code for the commercial Alloy Look and Feel.

Java Code Generator

On this page, you can turn off the Java code generator and specify other code generation options.



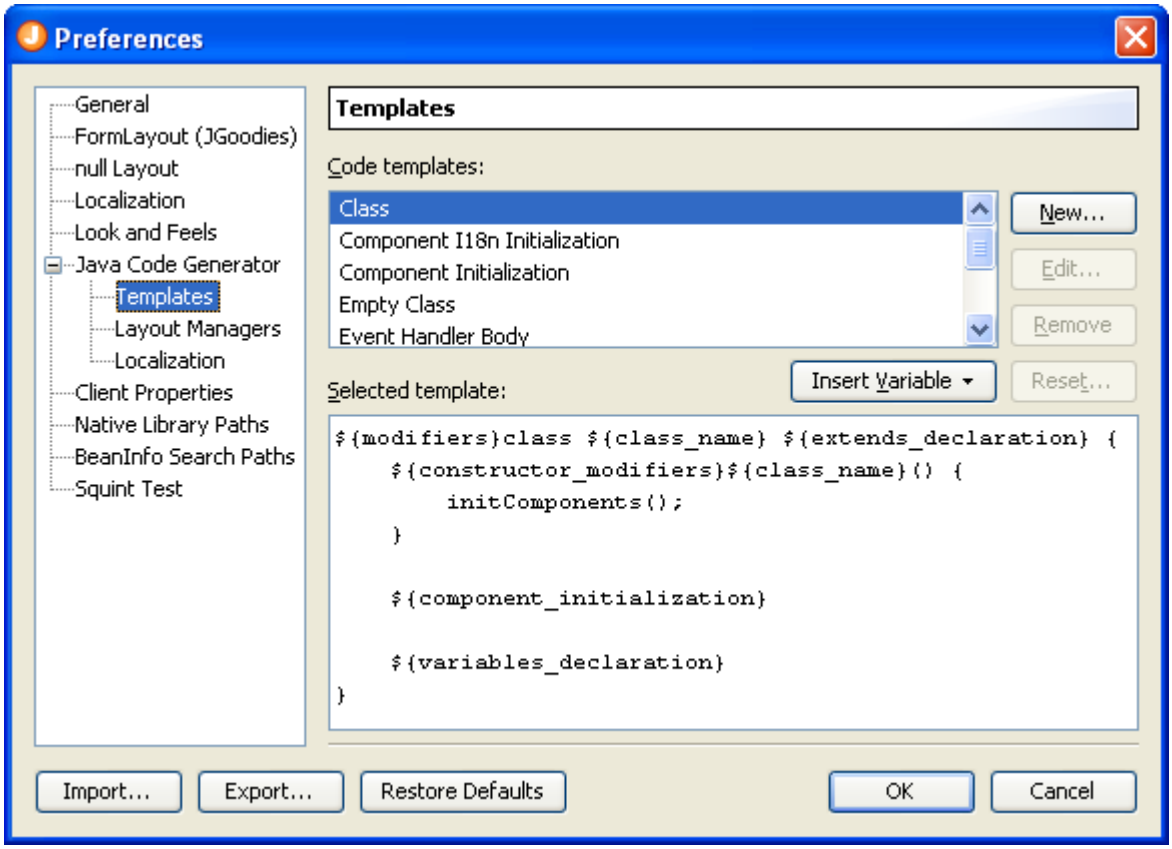
Option	Description	Default
Generate Java source code	If enabled, JFormDesigner generates Java source code when you save a form.	On
Source compatibility	Specifies the compatibility of the generated source code. Besides generating Java 1.x compatible source code, JFormDesigner can also use Java 5 (or later) features in the generated source code (e.g. auto-boxing, @Override, etc).	**Stand-alone**: use JRE version **IDE plug-ins**: use project setting
Explicit imports	If enabled, the code generator adds explicit import statements (without '*') for used classes.	Off

Option	Description	Default
Container blocks	If enabled, the code generator puts the initialization code for each container into a block (enclosed in curly braces).	On
Comments	If enabled, the code generator puts a comment line above the initialization code for each component.	On
Set component names	If enabled, the code generator inserts `java.awt.Component.setName()` statements for all components of the form.	Off
Eclipse non-nls tags (//$NON-NLS-n$)	If enabled, the code generator appends non-nls comments to lines containing strings. These comments are used by the Eclipse IDE to denote strings that should not be externalized.	Off
NetBeans no-i18n tags (//NOI18N)	If enabled, the code generator appends non-nls comments to lines containing strings. These comments are used by the NetBeans IDE to denote strings that should not be externalized.	Off
Use 'this' for member variables	If enabled, the code generator inserts 'this.' before all member variables. E.g. `this.nameLabel.setText("Name:");`	Off
Member variables prefix	Prefix used for component member variables. E.g. "m_".	
Class modifiers	Class modifiers used when generating a new class. Allowed modifiers: `public`, `default`, `abstract` and `final`.	`public`
Nested class modifiers	Class modifiers used when generating a new nested class. Allowed modifiers: `public`, `default`, `protected`, `private`, `abstract`, `final` and `static`.	`private`

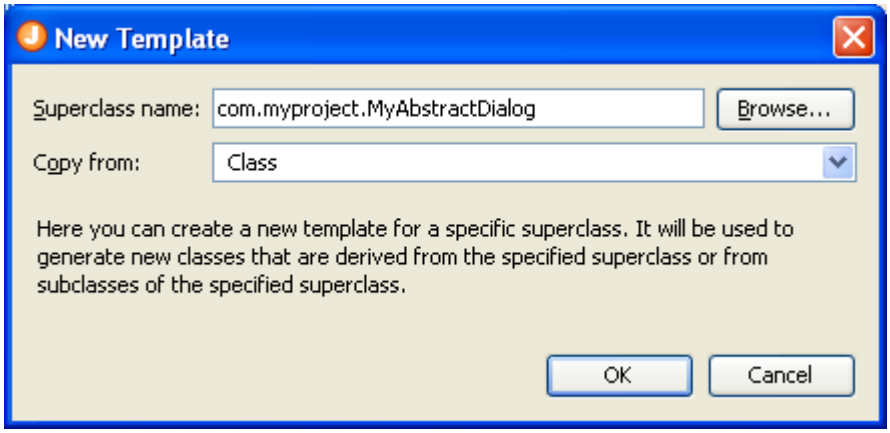
You can set additional options per form in the "(form)" properties.

Templates (Java Code Generator)

This page contains templates that are used by the code generator when generating a new class. See Code Templates for details about templates.

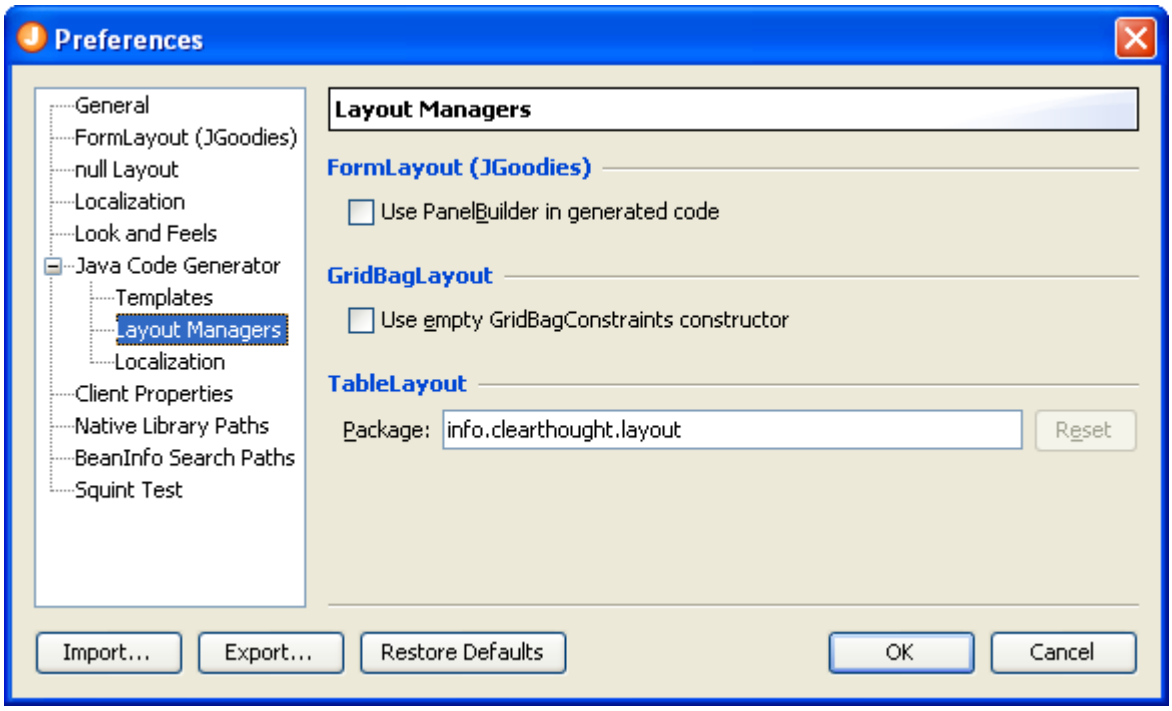


New: Create a new template for a specific superclass.
Edit: Edit the superclass of the selected user-defined template.
Remove: Remove the selected template. Only allowed for user-defined templates.
Reset: Reset the selected predefined template to the default.
Insert Variable: Insert a variable at the current cursor location into the selected template.



Layout Managers (Java Code Generator)

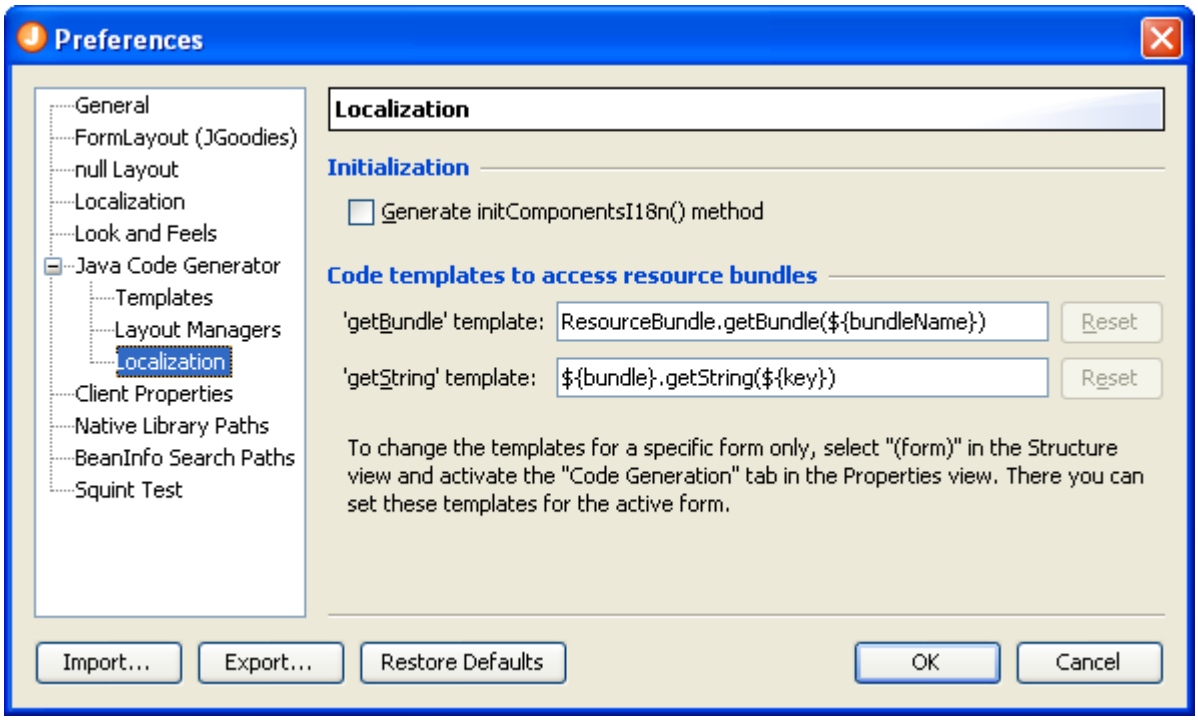
On this page, you can specify code generation options for some layout managers.



Option	Description	Default
Use PanelBuilder in generated code	If enabled, the PanelBuilder class of JGoodies Forms is used for FormLayout.	Off
Use empty GridBagConstraints constructor	If enabled, the empty GridBagConstraints constructor is used in the generated code, which is necessary for Java 1.0 and 1.1 compatibility. Since Java 1.2, GridBagConstraints has a constructor with parameters, which is used by default.	Off
TableLayout package	Package name used by the Java code generator for TableLayout. Change this only if you have a copy of the original TableLayout in another package.	info.clearthought.layout

Localization (Java Code Generator)

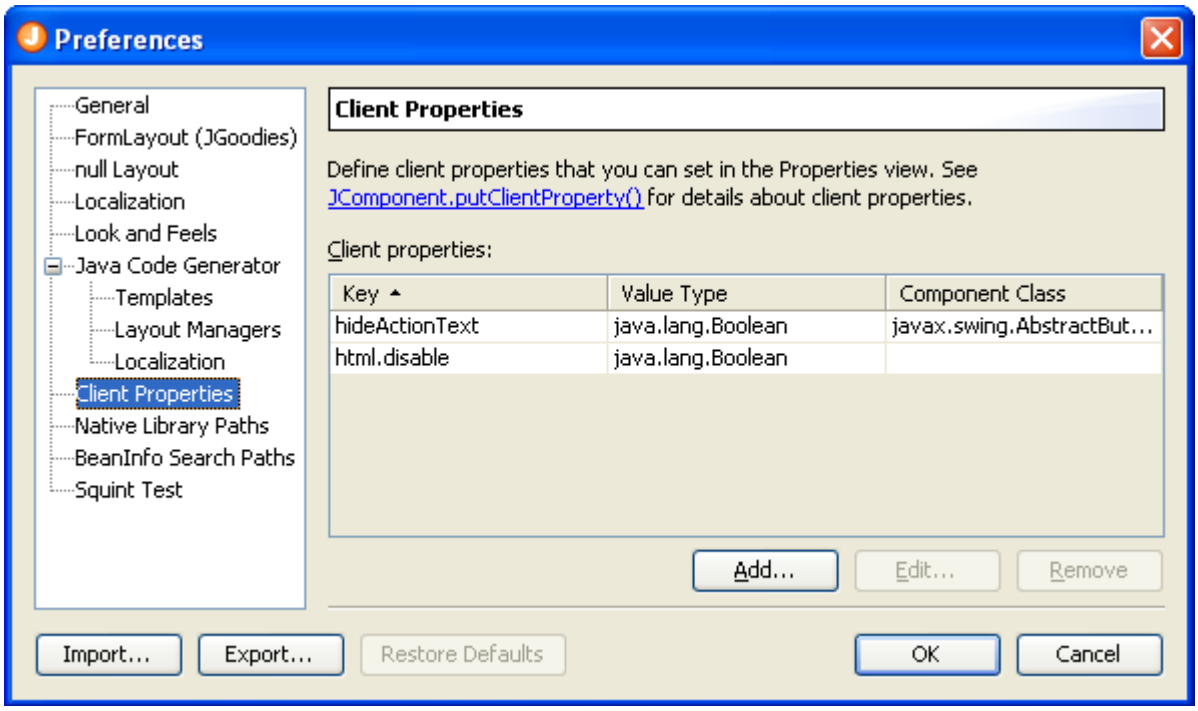
On this page, you can specify code generation options for localization.



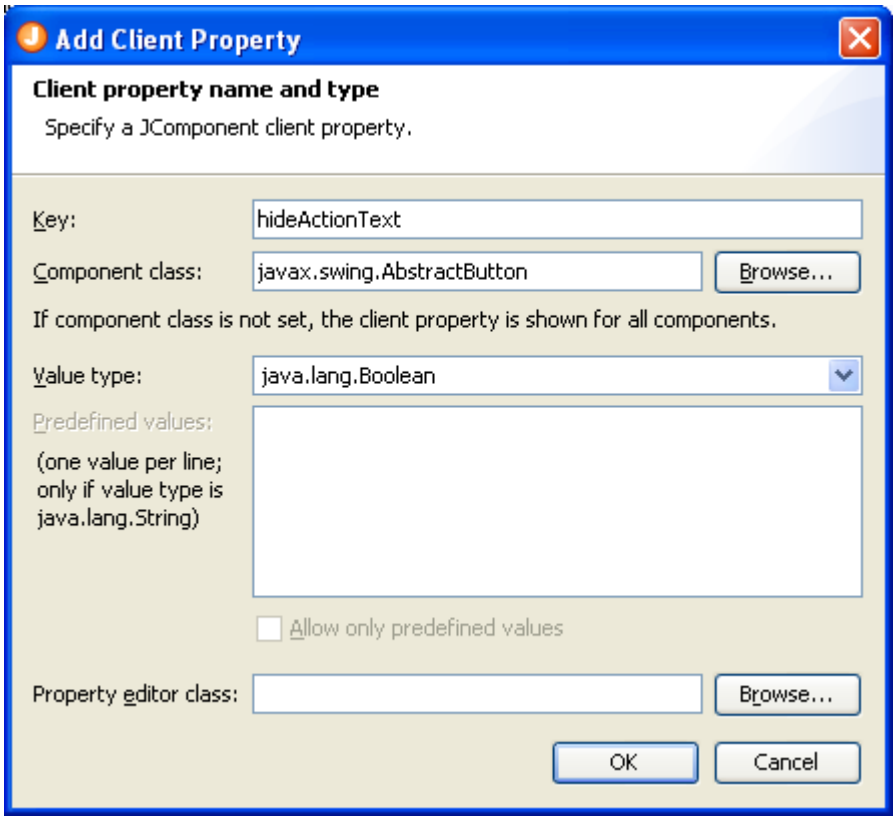
Option	Description	Default
Generate initComponentsI18n() method	If enabled, the code generator puts the code to initialize the localized texts into a method initComponentsI18n(). You can invoke this method from your code to switch the locale of a form at runtime. You can set this options also per form in the "(form)" properties.	Off
'getBundle' template	Template used by code generator for getting a resource bundle.	ResourceBundle.getBundle(${bundleName})
'getString' template	Template used by code generator for getting a string from a resource bundle.	${bundle}.getString(${key})

Client Properties

On this page, you can can define client properties, which can be set in the Properties view.



If the client properties list is focused, you can use the **Insert** key to add a client property or the **Delete** key to delete selected client properties.



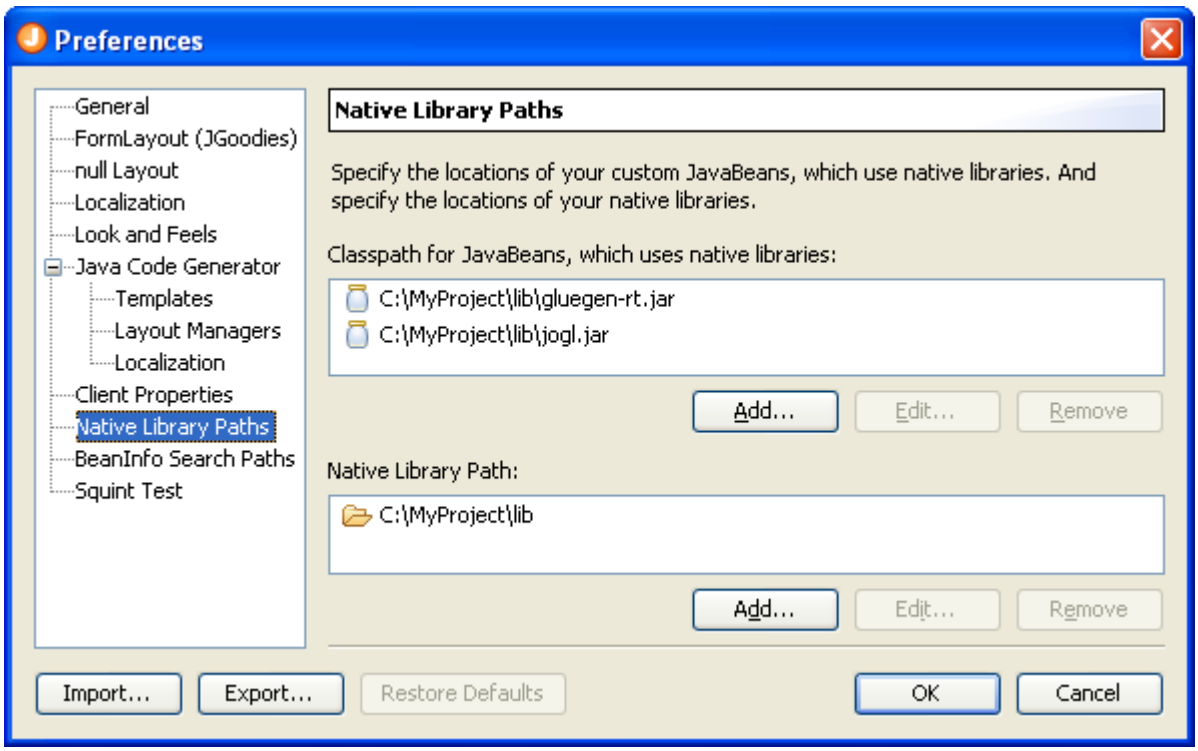
Option	Description
Key	The key that identifies the client property.

Option	Description
Component class	The component class to which the client property belongs. E.g. if set to javax.swing.JButton, then the client property is shown in the Properties view for buttons and for subclasses of JButton. If not specified, the client property is shown for all components.
Value type	The type of the client property value. You can select one of the common types (String, Boolean, Integer, etc) from the combo box or enter the class name of a custom type.
Predefined values	If the value type is java.lang.String, then you can specify predefined values for the client property. When editing the client property in the Properties view, a combo box that contains these values is shown. The combo box is editable by default. Select the "Allow only predefined values" check box to make the combo box not-editable.
Property editor class	Optional class name of a property editor that should be used when editing the client property in the Properties view.

Native Library Paths

On this page, you can specify the locations of custom JavaBeans that use native libraries and you can specify the folders where to search for the native libraries.

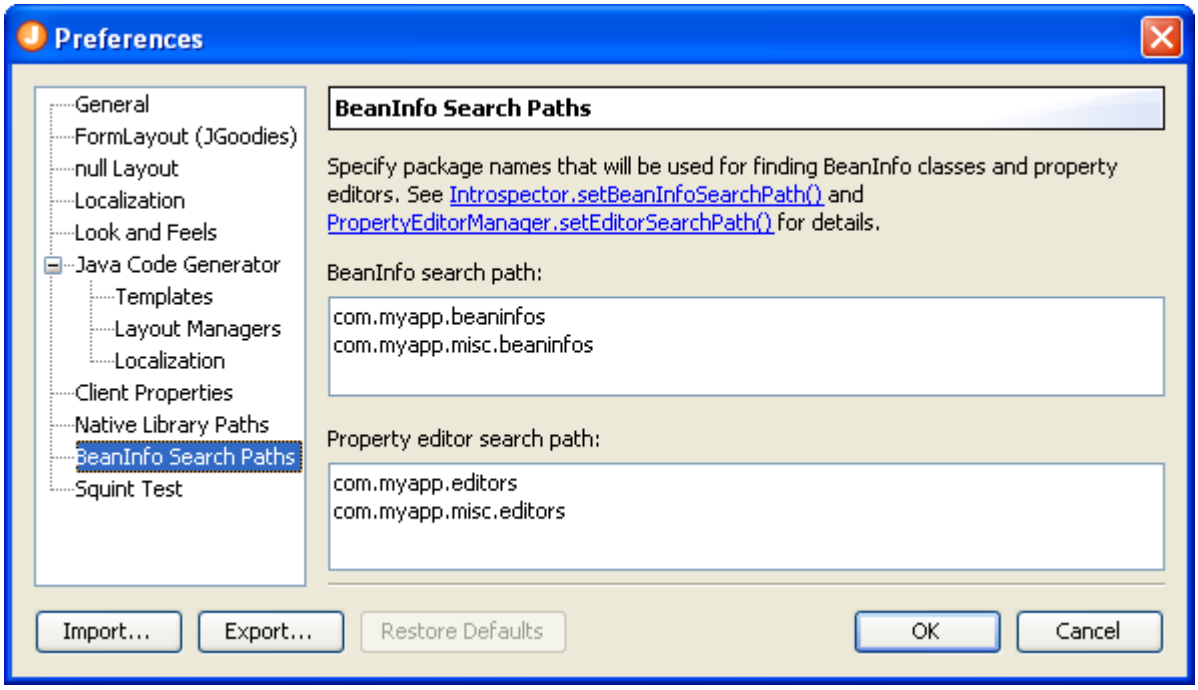
Note: When removing or changing paths, a restart of JFormDesigner (or the IDE) is probably necessary to make the changes work.



Option	Description
Classpath for JavaBeans, which use native libraries	JAR files or folders containing .class files, which are using native libraries. They must be specified here to ensure that the native libraries are loaded from a special class loader only once.
Native Library Path	Folders used to search for native libraries.

BeanInfo Search Paths

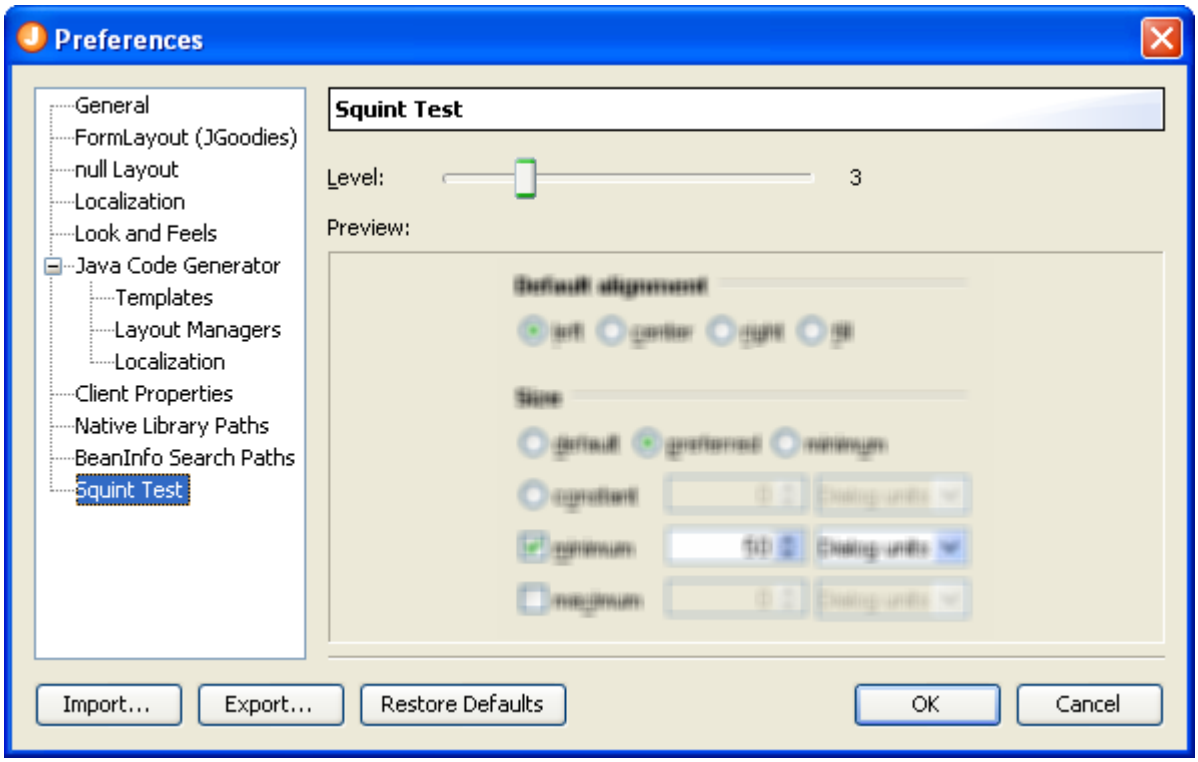
On this page, you can specify package names that will be used for finding BeanInfo classes and property editors.



Option	Description
BeanInfo search path	Package names that will be used for finding BeanInfo classes. Only necessary if the BeanInfo class is not in the same package as the component class to which it belongs. See java.beans.Introspector and Introspector.setBeanInfoSearchPath() for details.
Property editor search path	Package names that will be used for finding property editors. Only necessary if the property editor is not in the same package as the property type to which it belongs. See java.beans.PropertyEditorManager and PropertyEditorManager.setEditorSearchPath() for details.

Squint Test

The page allows you to specify the squint level for the squint test (menu **View** > **Squint Test**).



IDE Integrations

JFormDesigner is available as stand-alone application and as plug-ins for various IDEs. The IDE plug-ins completely integrate JFormDesigner into the IDEs.

Eclipse plug-in

This plug-in integrates JFormDesigner into Eclipse and other Eclipse based IDEs (e.g. JBuilder 2007).

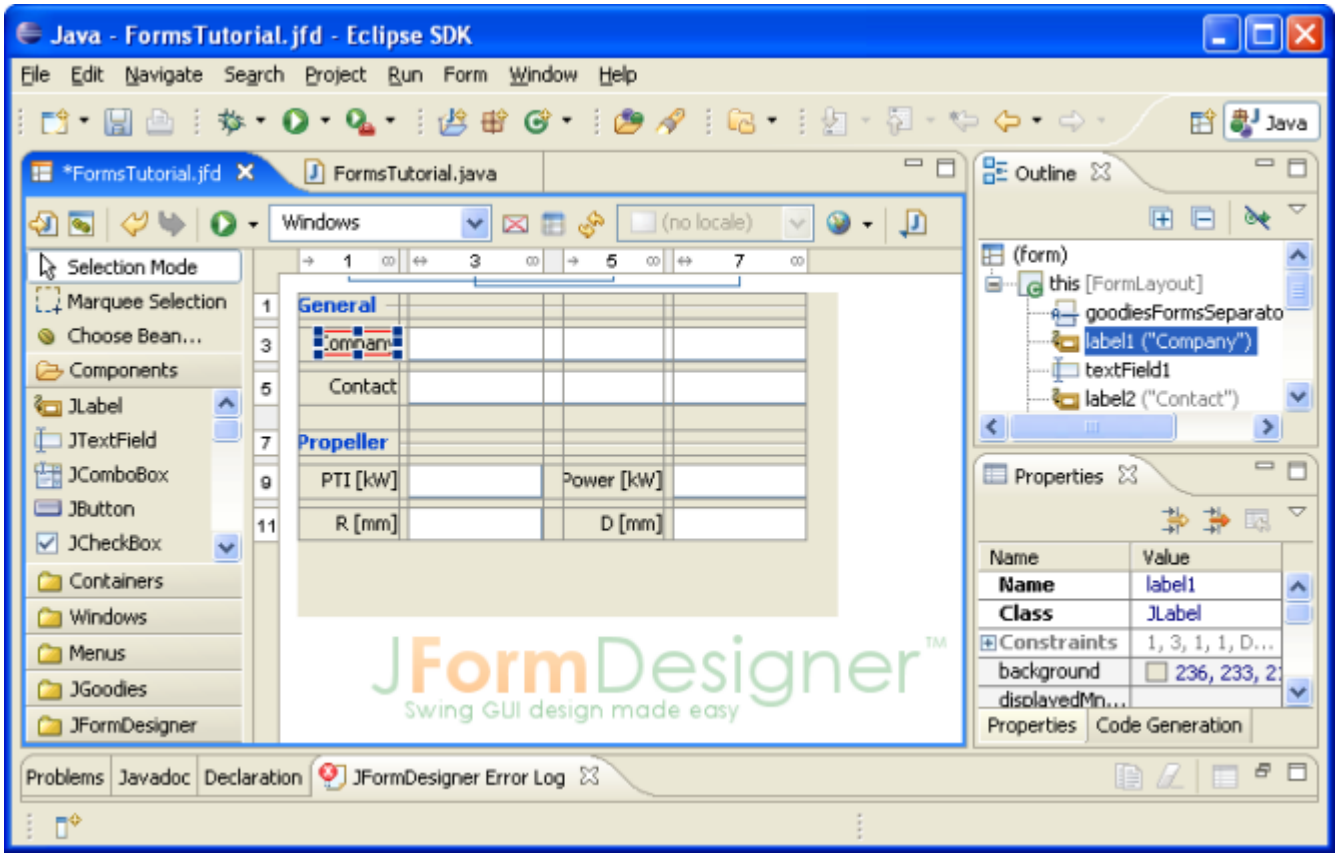
Benefits

Using this plug-in has following benefits compared to JFormDesigner stand-alone edition:

- Fully integrated as editor for JFormDesigner .jfd files. Create and design forms within Eclipse. No need to switch between applications.
- Uses the source folders and classpath of the current Eclipse project. No need to specify them twice.
- The Java code generator updates the .java file in-memory on each change in the designer. You can design forms and edit its source code without the need to save them (as necessary when using JFormDesigner stand-alone edition).
- Folding of generated GUI code in Java editor.
- Go to event handler method in Java editor. Double-click on the event in the Structure view to go to the event handler method in the Java editor of Eclipse.
- Two-way synchronization of localized strings in designer and in properties file editors. Changing localized strings in the designer immediately updates the .properties file in-memory and changing the .properties file updates the designer.
- Copy needed libraries (JGoodies Forms, TableLayout, etc) to the project and add them to the classpath of the current Eclipse project. Optionally include source code and javadoc.
- Integrated with Eclipse's Version Control Systems.
- Integrated into refactoring: Copy, rename, move or delete .jfd files when coping, renaming, moving or deleting .java files.

User interface

The screenshot below shows the Eclipse main window editing a JFormDesigner form. JFormDesigner adds the menu **Form** to the main menu.

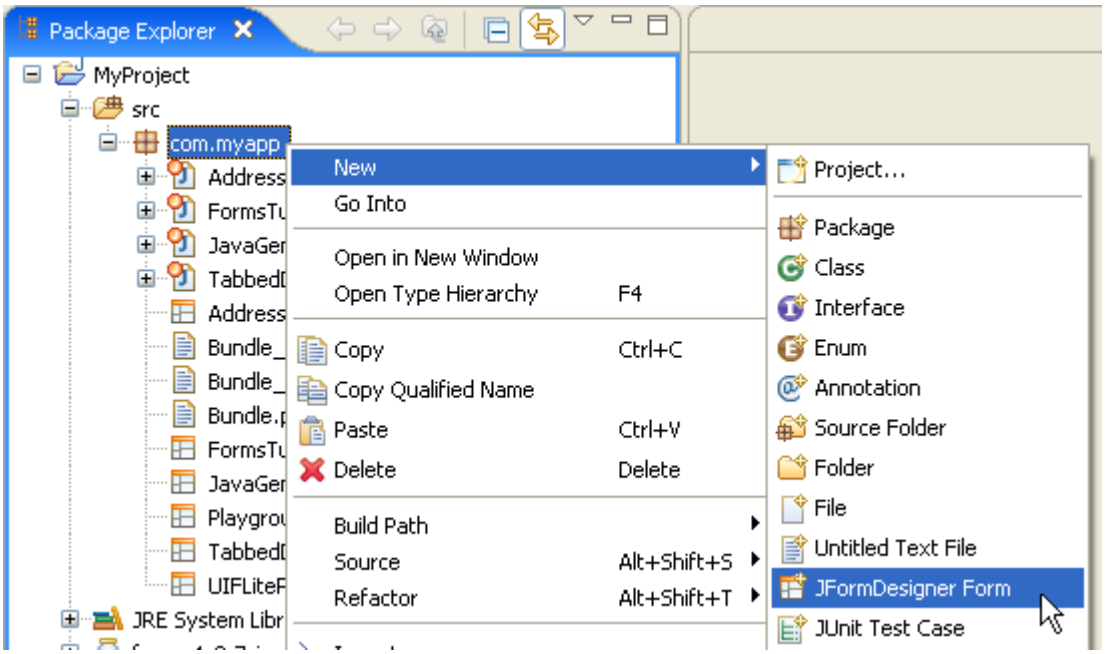


A JFormDesigner editor consists of:

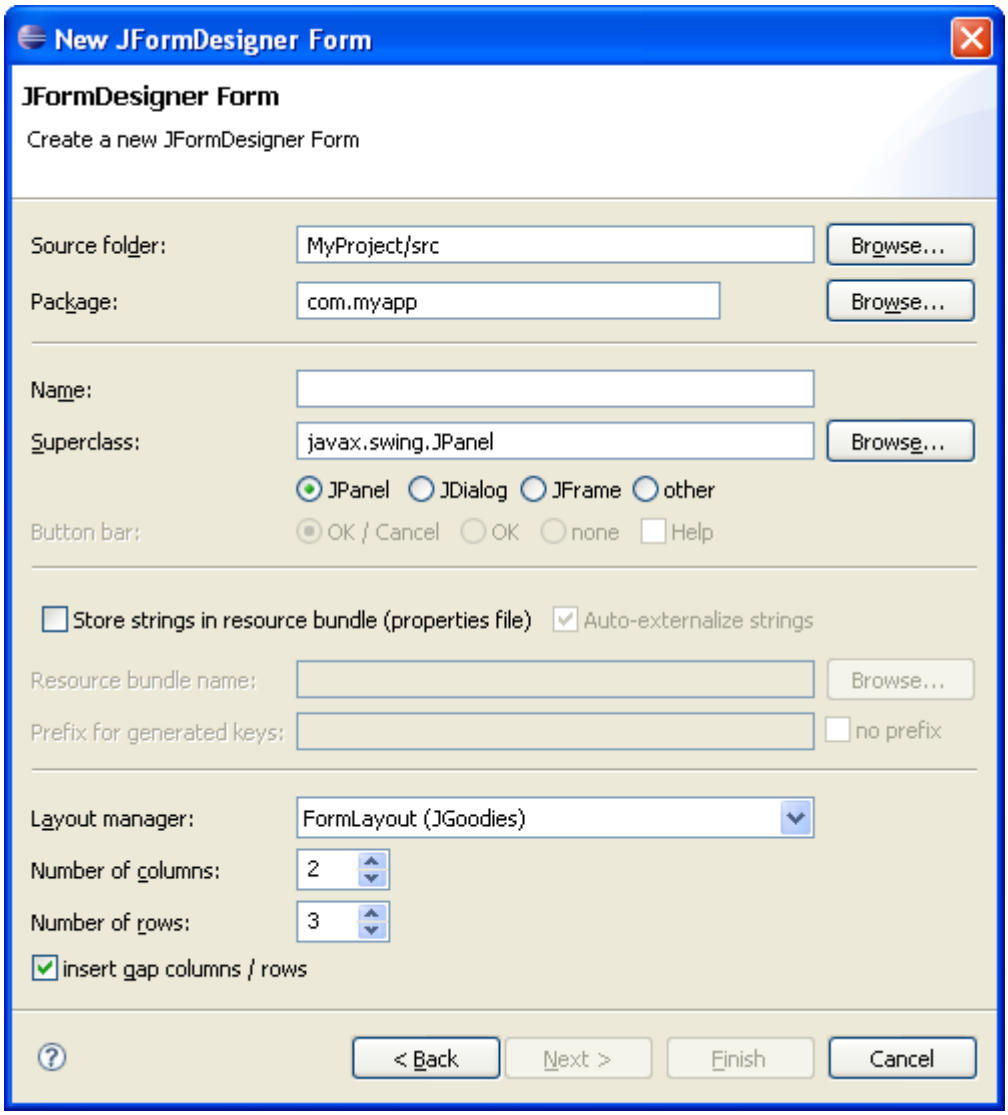
- Toolbar: Located at top of the editor area.
- Palette: Located at the left side.
- Design View: Located at the center.
- Structure View: Located in Eclipse's Outline view.
- Properties View: Located in Eclipse's Properties view.
- Error Log View: Automatically opens on errors in a view at the bottom. This view is minimized in the above screenshot.

Creating new forms

You can create new forms in Eclipse's Package Explorer view. First select the destination package or folder, then invoke Eclipse's **New** command and choose **JFormDesigner Form**. If it is not in the sub menu, select **Other**, which opens Eclipse's **New** dialog and choose **JFormDesigner Form** form the list of wizards.



In the **New JFormDesigner Form** dialog, enter the form name (which is also used as class name), choose a superclass, a layout manager and set localization options.



After clicking OK, the form will be created and opened.

Open forms for editing

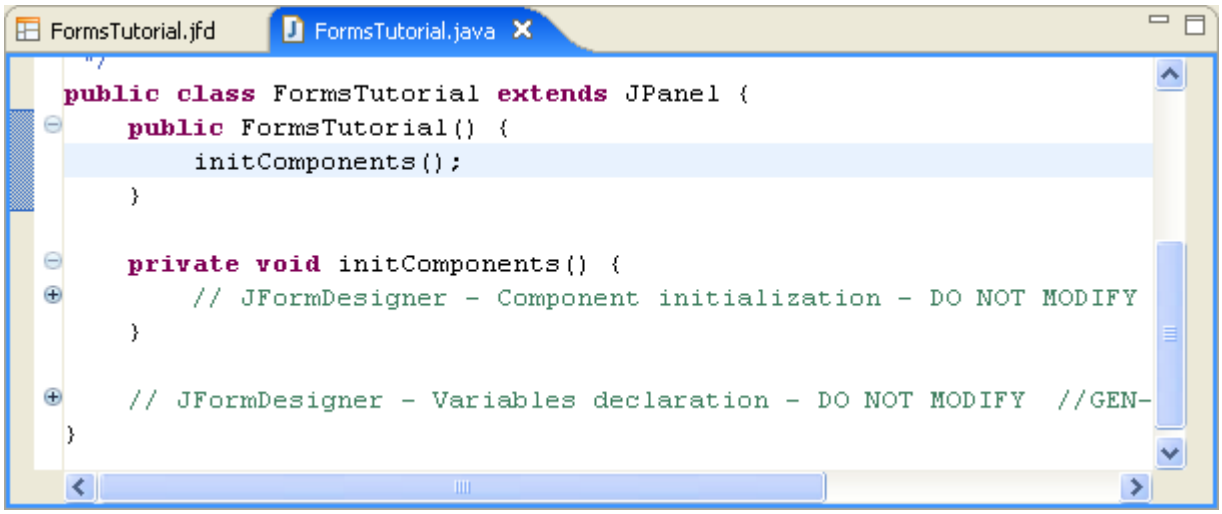
You can open existing forms the same way as opening any other file in Eclipse. Locate it in Eclipse's Package Explorer view and double-click it.

Go to Java code

JFormDesigner adds a button to its toolbar that enables you to switch quickly from a JFormDesigner form editor to its Java editor.

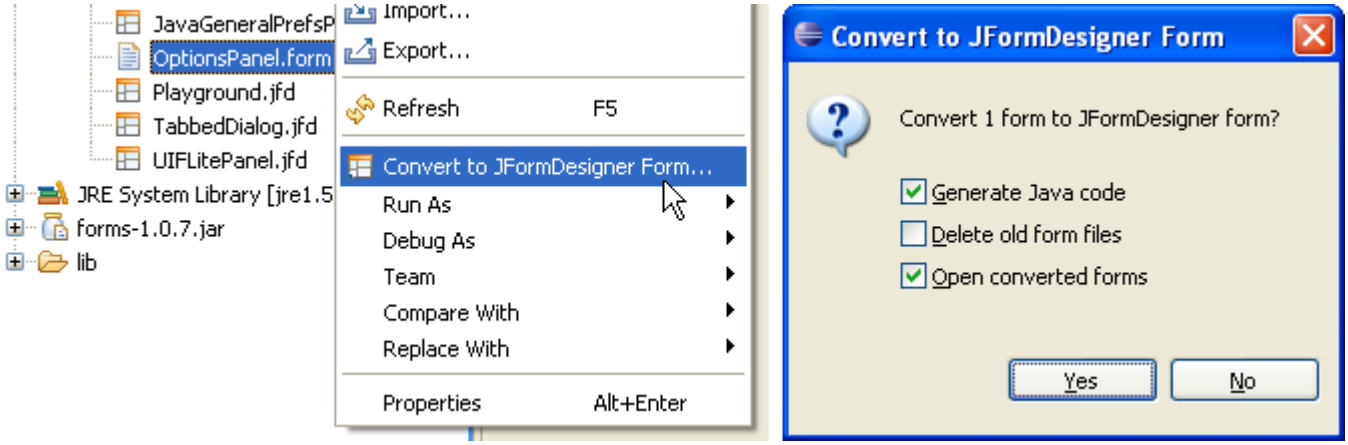
Code folding

To move the generated code out of the way, JFormDesigner folds it in the Java editor.



Convert NetBeans and IntelliJ IDEA forms

You can convert existing NetBeans and IntelliJ IDEA forms to JFormDesigner forms. Right-click on the form file and select **Convert to JFormDesigner Form**.



Note: When converting an IntelliJ IDEA form, JFormDesigner inserts its own generated GUI code into the existing Java class, but does not remove IDEA's GUI code. You have to remove IDEA's component variables and initialization code yourself.

Preferences

The JFormDesigner preferences are fully integrated into the Eclipse preferences dialog. Select **Window > Preferences** from the menu to open it and then expand the node "JFormDesigner" in the tree. See Preferences for details.

IntelliJ IDEA plug-in

This plug-in integrates JFormDesigner into Jetbrains IntelliJ IDEA.

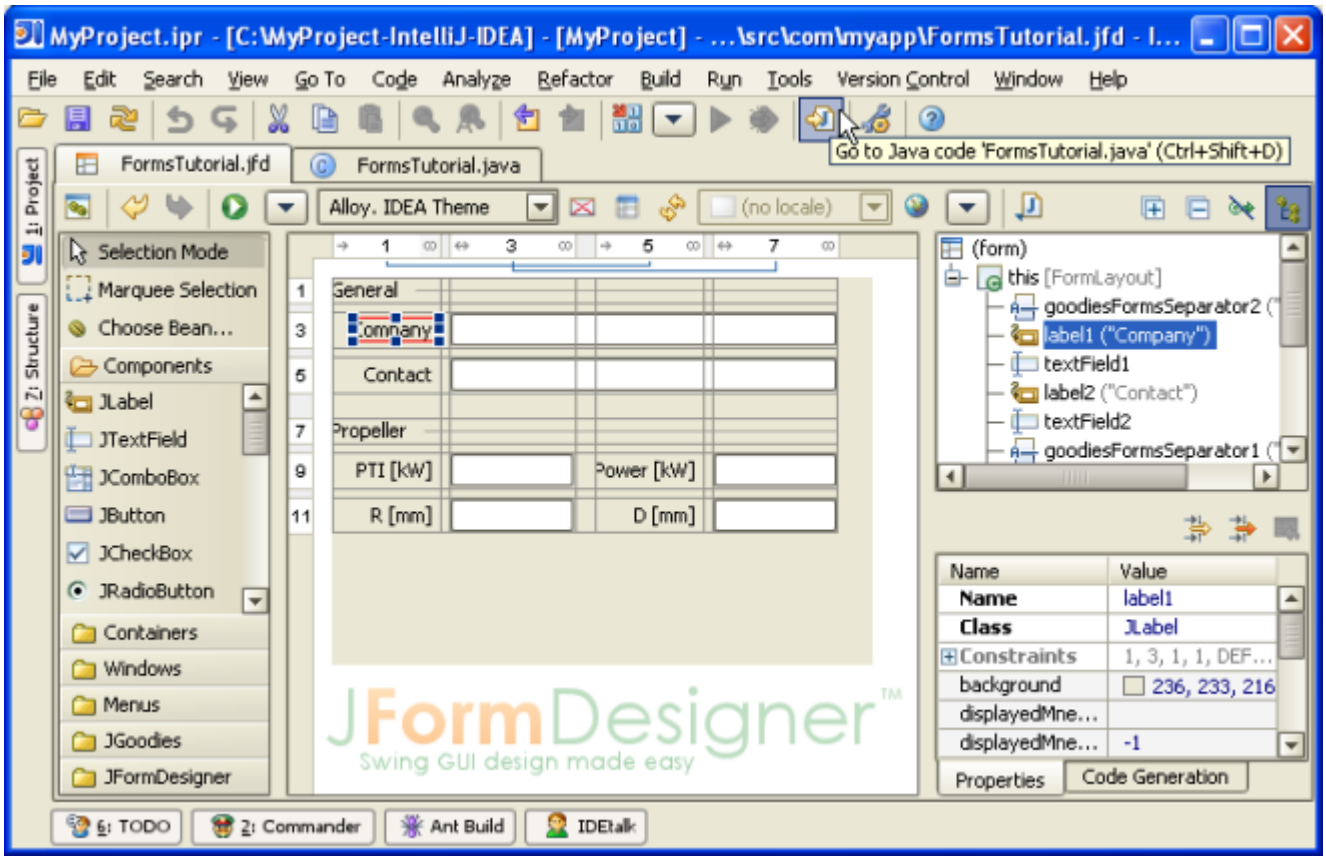
Benefits

Using this plug-in has following benefits compared to JFormDesigner stand-alone edition:

- Fully integrated as editor for JFormDesigner .jfd files. Create and design forms within IntelliJ IDEA. No need to switch between applications.
- Uses the source folders and classpath of the current IntelliJ IDEA project/module. No need to specify them twice.
- The Java code generator updates the .java file in-memory on each change in the designer. You can design forms and edit its source code without the need to save them (as necessary when using JFormDesigner stand-alone edition).
- Folding of generated GUI code in Java editor.
- Go to event handler method in Java editor. Double-click on the event in the Structure view to go to the event handler method in the Java editor of IntelliJ IDEA.
- Two-way synchronization of localized strings in designer and in properties file editors. Changing localized strings in the designer immediately updates the .properties file in-memory and changing the .properties file updates the designer.
- Copy needed libraries (JGoodies Forms, TableLayout, etc) to the project and add them to the classpath of the current IntelliJ IDEA project/module. Optionally include source code and javadoc.
- Assign shortcut keys to most JFormDesigner commands in IntelliJ IDEA's keymap settings.
- Integrated with IntelliJ IDEA's Version Control Systems.

User interface

The screenshot below shows the IntelliJ IDEA main window editing a JFormDesigner form.

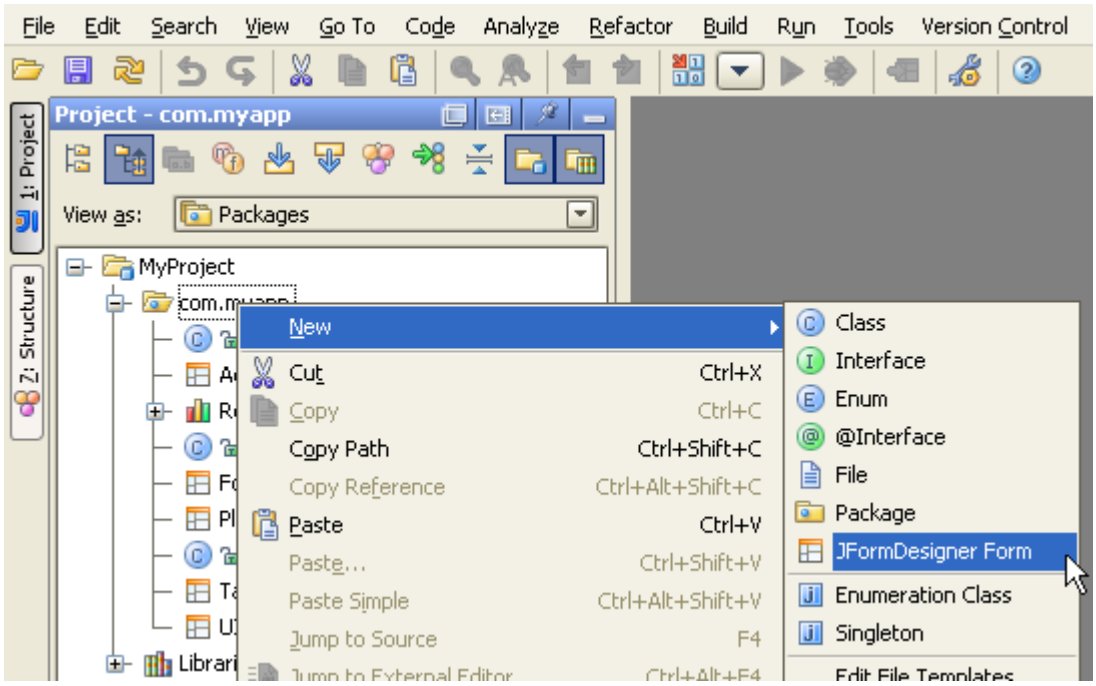


A JFormDesigner editor consists of:

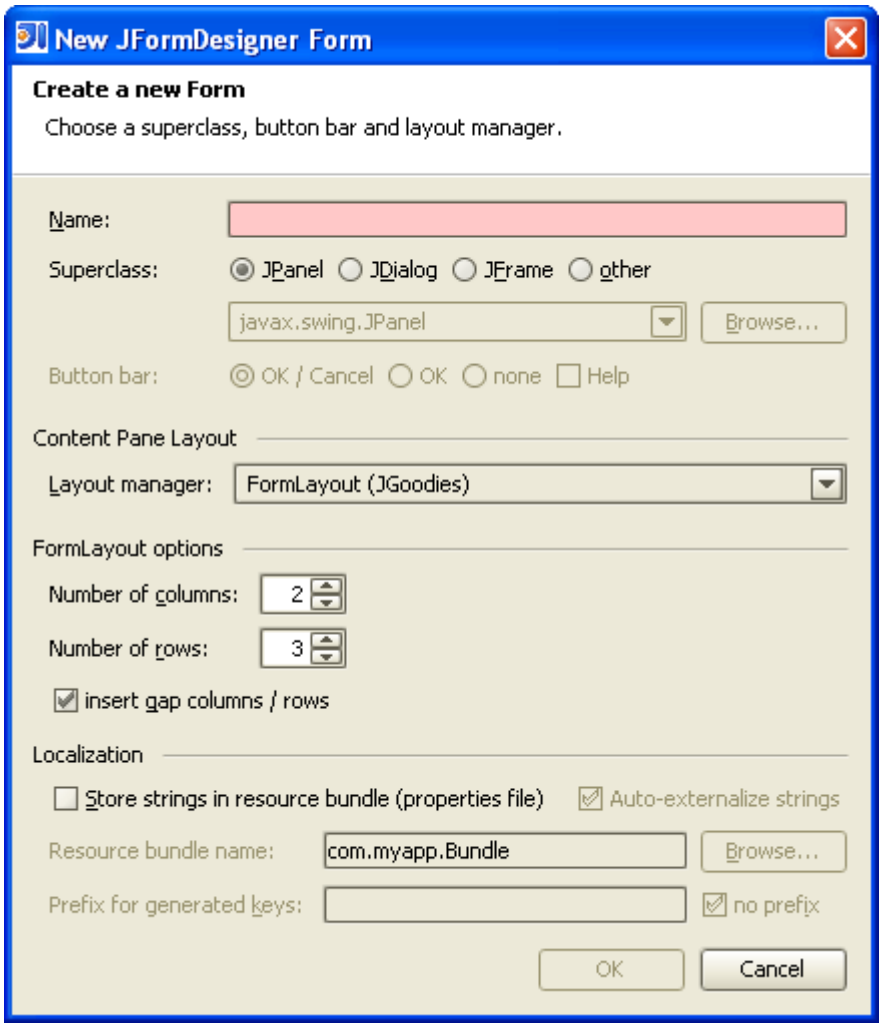
- Toolbar: Located at top of the editor area.
- Palette: Located at the left side.
- Design View: Located at the center.
- Structure View: Located at the upper right. You can hide this view in the editor and show it instead in IntelliJ IDEA's Structure tool window by unselecting **Show Structure in Editor** ( ).
- Properties View: Located at the lower right.
- Error Log View: Automatically opens on errors in a tool window at the bottom. This view is not visible in the above screenshot.

Creating new forms

You can create new forms in any of IntelliJ IDEA's project views. First select the destination package or folder, then invoke IDEA's **New** command and choose **JFormDesigner Form**.



In the **New JFormDesigner Form** dialog, enter the form name (which is also used as class name), choose a superclass, a layout manager and set localization options.



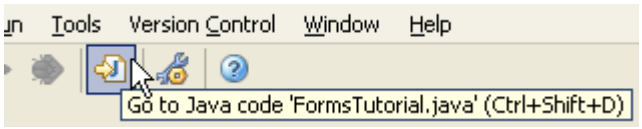
After clicking OK, the form will be created and opened.

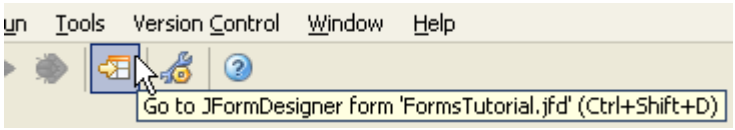
Open forms for editing

You can open existing forms the same way as opening any other file in IntelliJ IDEA. Locate it in any of IDEA's project views and double-click it.

Go to Java code / go to form

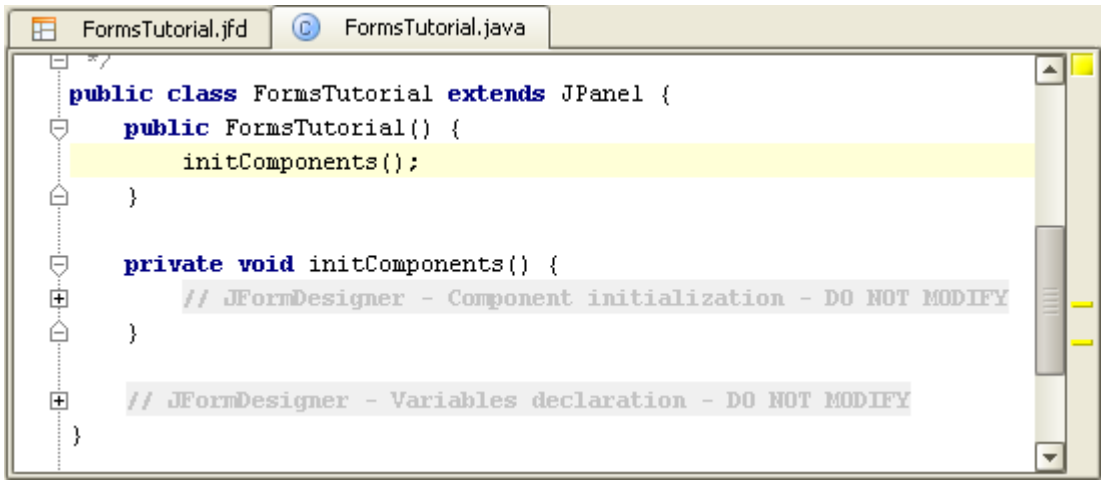
JFormDesigner adds a button to IntelliJ IDEA's main toolbar that enables you to switch quickly from a JFormDesigner form editor to its Java editor and vice versa. If a form editor is active, then the button is named **Go to Java code** (). If a Java editor is active, then it is named **Go to JFormDesigner form** (). You can also use **Ctrl+Shift+D**.





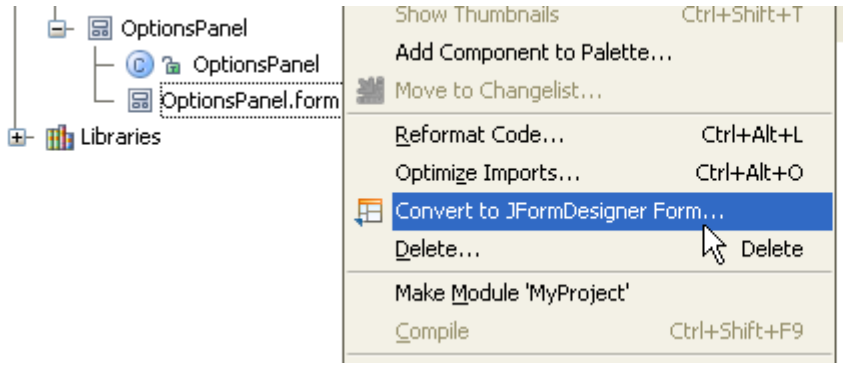
Code folding

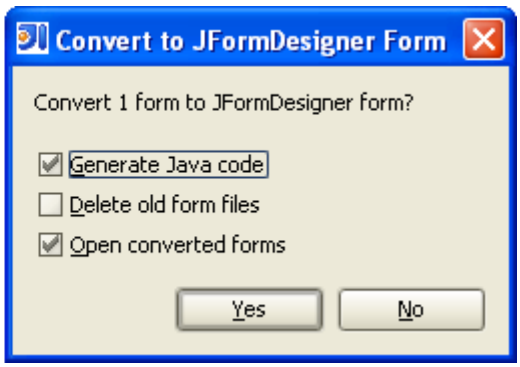
To move the generated code out of the way, JFormDesigner folds it in the Java editor.



Convert IntelliJ IDEA and NetBeans forms

You can convert existing IntelliJ IDEA and NetBeans forms to JFormDesigner forms. Right-click on the form file and select **Convert to JFormDesigner Form**.





Note: When converting an IntelliJ IDEA form, JFormDesigner inserts its own generated GUI code into the existing Java class, but does not remove IDEA's GUI code. You have to remove IDEA's component variables and

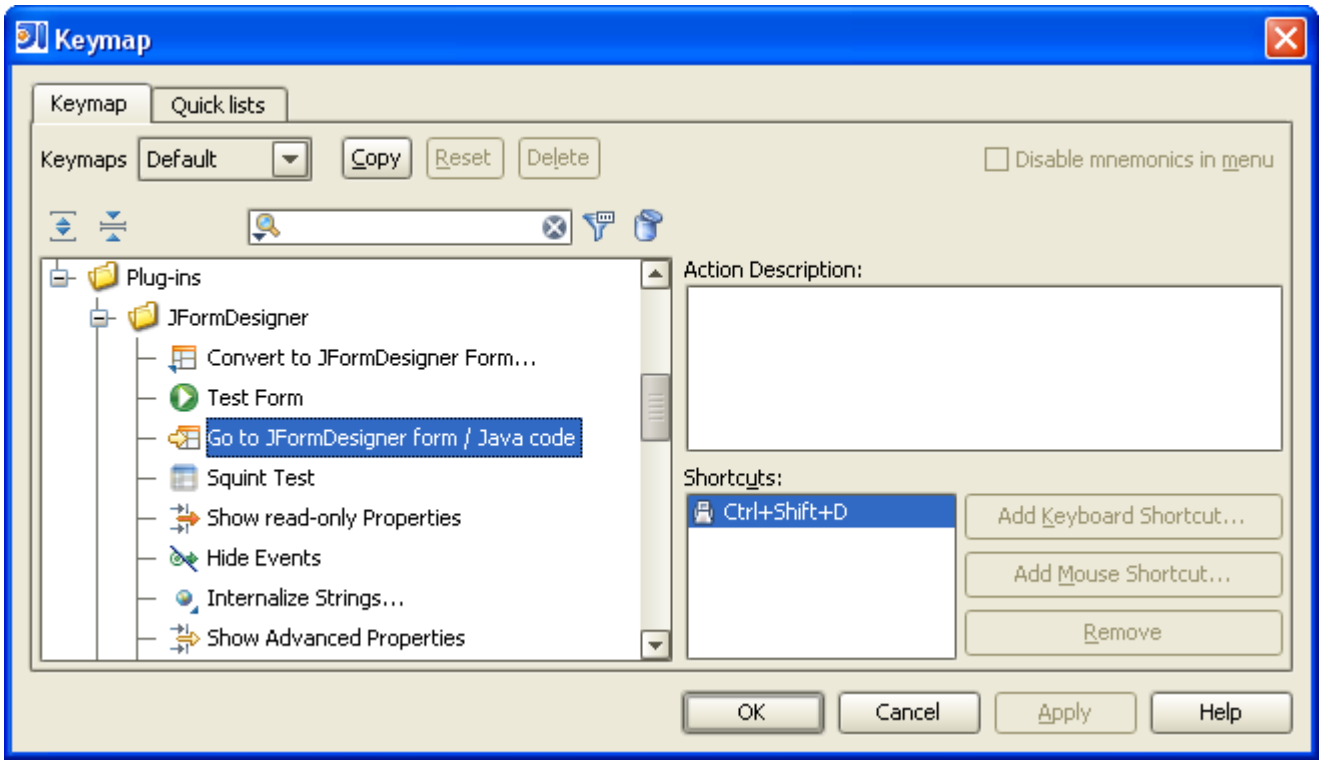
initialization code yourself.

Settings

JFormDesigner uses the term "Preferences" instead of IntelliJ IDEA's "Settings". The JFormDesigner preferences are fully integrated into the IntelliJ IDEA settings dialog. Select **File > Settings** from the menu to open it and then click the icon named "JFormDesigner". See Preferences for details.

Keyboard shortcuts

You can assign shortcut keys to most JFormDesigner commands in IntelliJ IDEA's keymap settings dialog. Select **File > Settings > Keymap** to open it. In the actions tree expand **All Actions > Plug-ins > JFormDesigner**.



JBuilder plug-in

This plug-in integrates JFormDesigner into JBuilder 2005 and 2006. For JBuilder 2007 use the Eclipse plug-in.

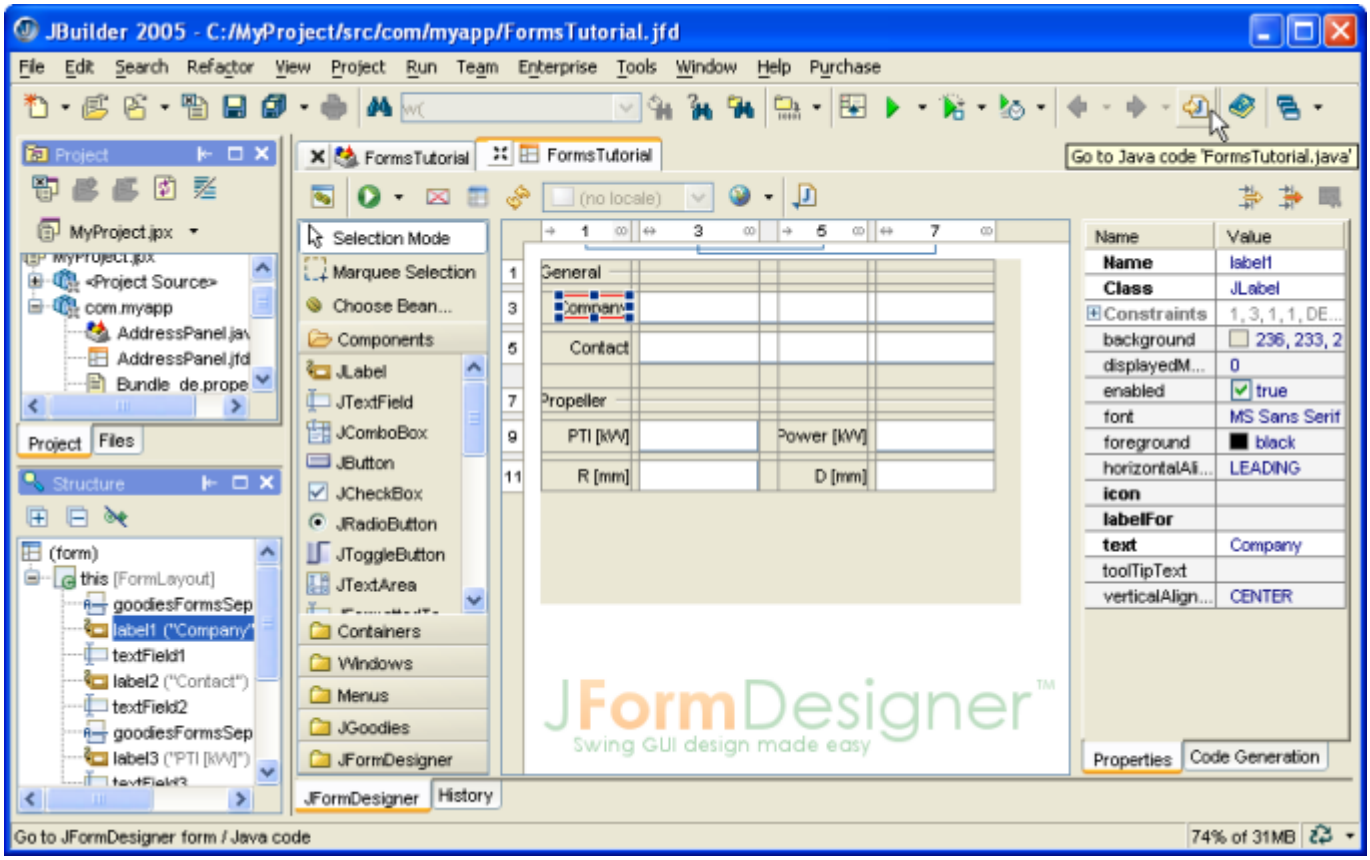
Benefits

Using this plug-in has following benefits compared to JFormDesigner stand-alone edition:

- Fully integrated as editor for JFormDesigner .jfd files. Create and design forms within JBuilder. No need to switch between applications.
- Uses the source folders and classpath of the current JBuilder project. No need to specify them twice.
- The Java code generator updates the .java file in-memory on each change in the designer. You can design forms and edit its source code without the need to save them (as necessary when using JFormDesigner stand-alone edition).
- Folding of generated GUI code in Java editor.
- Go to event handler method in Java editor. Double-click on the event in the Structure view to go to the event handler method in the Java editor of JBuilder.
- Two-way synchronization of localized strings in designer and in properties file editors. Changing localized strings in the designer immediately updates the .properties file in-memory and changing the .properties file updates the designer.
- Copy needed libraries (JGoodies Forms, TableLayout, etc) to the project and add them to the classpath of the current JBuilder project. Optionally include source code and javadoc.
- Convert JBuilder forms (jbInit() methods) to JFormDesigner .jfd files.

User interface

The screenshot below shows the JBuilder main window editing a JFormDesigner form.

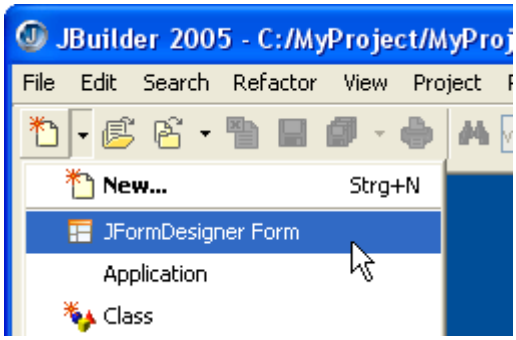


A JFormDesigner editor consists of:

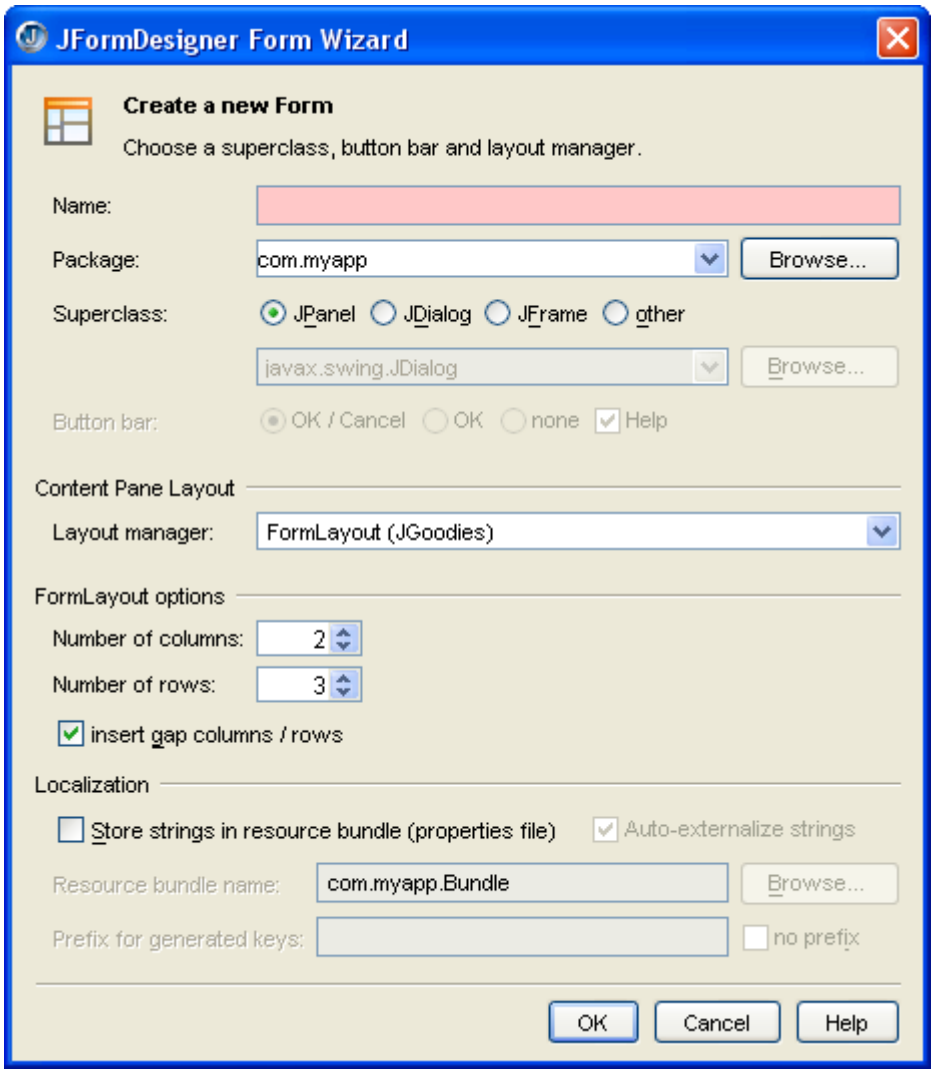
- Toolbar: Located at top of the editor area.
- Palette: Located at the left side.
- Design View: Located at the center.
- Structure View: Located at the lower left.
- Properties View: Located at the right side.
- Error Log View: Automatically opens on errors in a tool window at the bottom. This view is not visible in the above screenshot.

Creating new forms

You can create new forms using JBuilder's object gallery. Click the **New** arrow in the toolbar and choose **JFormDesigner Form**.



In the **New JFormDesigner Form** dialog, enter the form name (which is also used as class name), choose a superclass, a layout manager and set localization options.



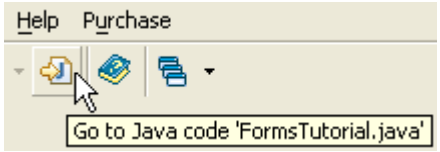
After clicking OK, the form will be created and opened.

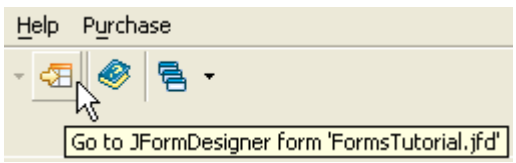
Open forms for editing

You can open existing forms the same way as opening any other file in JBuilder. Locate it in JBuilder's project view and double-click it.

Go to Java code / go to form

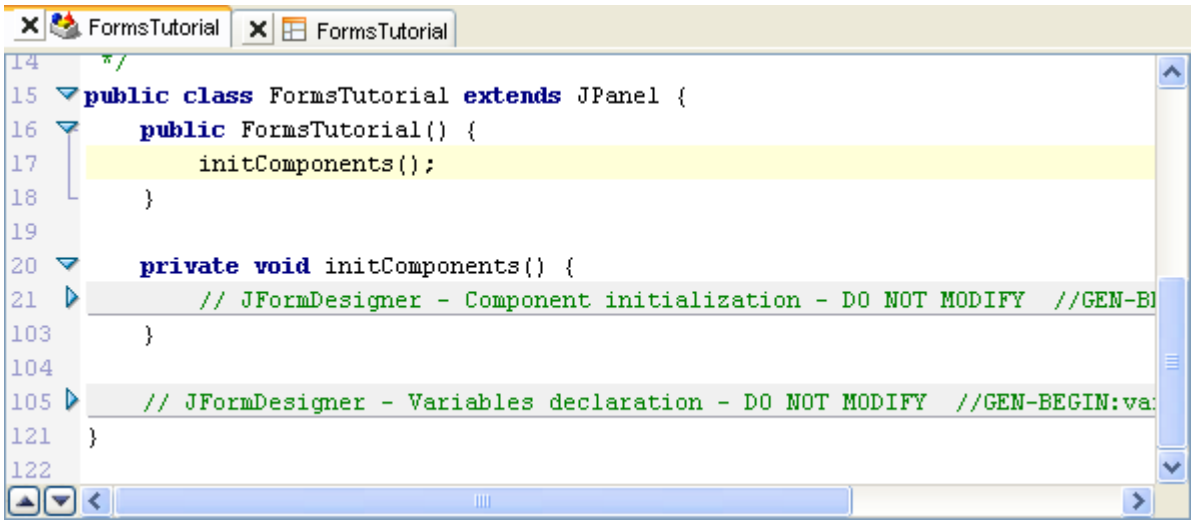
JFormDesigner adds a button to JBuilder's main toolbar that enables you to switch quickly from a JFormDesigner form editor to its Java editor and vice versa. If a form editor is active, then the button is named **Go to Java code** (). If a Java editor is active, then it is named **Go to JFormDesigner form** ().





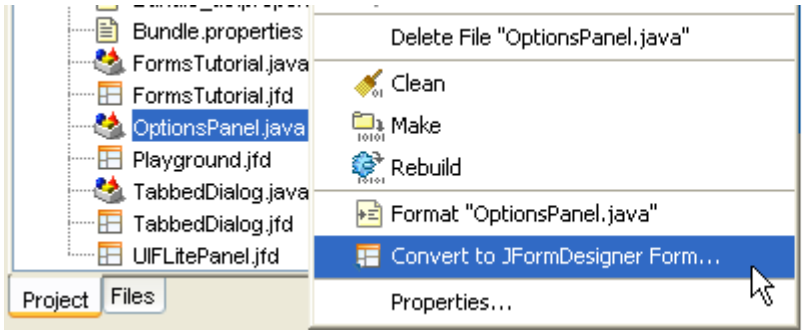
Code folding

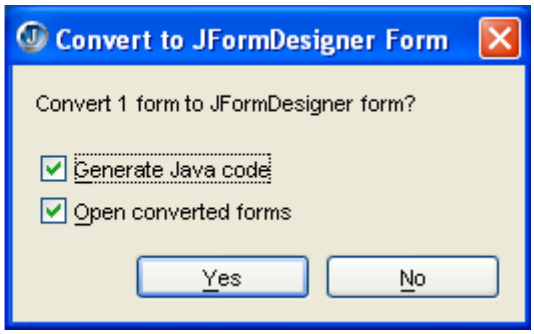
To move the generated code out of the way, JFormDesigner folds it in the Java editor.



Convert JBuilder forms

You can convert existing JBuilder forms to JFormDesigner forms. Right-click on the Java file and select **Convert to JFormDesigner Form**.





Note: JFormDesigner inserts its own generated GUI code into the existing Java class, but does not remove JBuilder's GUI code. You have to remove JBuilder's component variables and initialization code yourself.

Preferences

The JFormDesigner preferences are fully integrated into the JBuilder preferences dialog. Select **Tools > Preferences** from the menu to open it. See Preferences for details.

Unsupported features

Following features from other editions are not supported by the JBuilder plug-in:

- Convert NetBeans and IntelliJ IDEA forms to JFormDesigner forms.
- Use look and feels in Design view.
- Import and export of preferences.

Other IDEs

If there is no JFormDesigner plug-in for your favorite IDE, you can use the stand-alone edition of JFormDesigner side by side with your IDE.

IDE plug-ins for JDeveloper and NetBeans are planed for a future release.

IDE interworking

Care must be taken because you edit the Java source in the IDE and JFormDesigner also modifies the Java source file when generating code for the form. As long as you follow the following rule, you will never have a problem:

Save the Java file in the IDE **before** saving the form in JFormDesigner.

Your IDE will recognize that the Java file was modified outside of the IDE and will reload it. Some IDEs ask the user before reloading files, other IDEs silently reload files.

If you have not saved the Java file in the IDE, then you should prevent the IDE from reloading it. In this case save the Java file in the IDE and then use **Generate Java Code** in JFormDesigner.

JFormDesigner generates Java code when you either **Save** the form or select **Generate Java Code**. JFormDesigner does not hold a copy of the Java source in memory. Every time JFormDesigner generates Java code, it first reads the Java source file, parses it, updates it and writes it back to the disk.

Layout Managers

Layout managers are an essential part of Swing forms. They lay out components within a container. JFormDesigner provides support for following layout managers:

- BorderLayout
- BoxLayout
- CardLayout
- FlowLayout
- FormLayout (JGoodies)
- GridBagLayout
- GridLayout
- IntelliJ IDEA GridLayout
- null Layout
- TableLayout

How to choose a layout manager?

For "normal" forms use one of the grid based layout managers FormLayout, TableLayout or GridBagLayout. Each has its advantages and disadvantages. FormLayout and TableLayout are open source and require that you ship an additional library with your application.

- FormLayout has the most features (dialog units, column/row alignment, column/row grouping), but may have problems if a component span multiple columns or rows and can not handle right-to-left component orientation.
- TableLayout does not have these limitations, but has fewer features than FormLayout.
- GridBagLayout is the weakest of these three layout managers, but JFormDesigner hides its complexity and adds additional features like gaps. Use GridBagLayout if you cannot use FormLayout or TableLayout.

For button bars use FormLayout, TableLayout, GridBagLayout or FlowLayout.

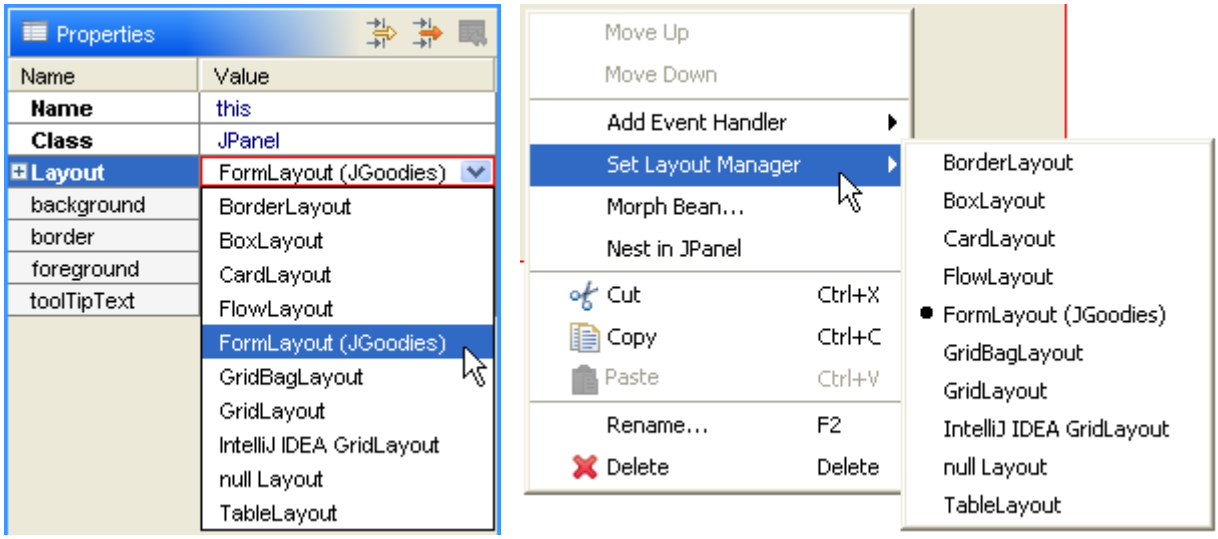
To layout a main window, use BorderLayout. Place the toolbar to the north, the status bar to the south and the content to the center.

For toolbars use `JToolBar`, which has its own layout manager (based on BoxLayout).

For radio button groups, BoxLayout may be a good choice. Mainly because `JRadioButton` has a gap between its text and its border and therefore the gaps provided by FormLayout, TableLayout and GridBagLayout are not necessary.

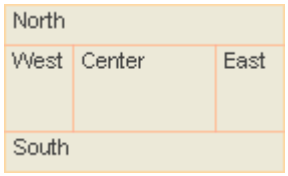
Change layout manager

You can change the layout manager at any time. Either in the Properties view or by right-clicking on a container in the Design or Structure view and selecting the new layout manager from the popup menu.



BorderLayout

The border layout manager places components in up to five areas: center, north, south, east and west. Each area can contain only one component.



The components are laid out according to their preferred sizes. The north and south components may be stretched horizontally. The east and west components may be stretched vertically. The center component may be stretched horizontally and vertically to fill any space left over.

BorderLayout is part of the standard Java distribution.

Layout properties

A container with this layout manager has following layout properties:

Property Name	Description
horizontal gap	The horizontal gap between components. Default is 0.
vertical gap	The vertical gap between components. Default is 0.

Constraints properties

A component contained in a container with this layout manager has following constraints properties:

Property Name	Description
constraints	Specifies where the component will be placed. Possible values: CENTER, NORTH, SOUTH, EAST and WEST.

BoxLayout

The box layout manager places components either vertically or horizontally. The components will not wrap as in FlowLayout.

This layout manager is used rarely. Take a look at the BoxLayout API documentation for more details about it.

BoxLayout is part of the standard Java distribution.

Layout properties

A container with this layout manager has following layout properties:

Property Name	Description
axis	The axis to lay out components along. Possible values: X_AXIS, Y_AXIS, LINE_AXIS and PAGE_AXIS.

CardLayout

The card layout manager treats each component in the container as a card. Only one card is visible at a time. The container acts as a stack of cards. The first component added to a CardLayout object is the visible component when the container is first displayed.

CardLayout is part of the standard Java distribution.

Layout properties

A container with this layout manager has following layout properties:

Property Name	Description
horizontal gap	The horizontal gap at the left and right edges. Default is 0.
vertical gap	The vertical gap at the top and bottom edges. Default is 0.

Constraints properties

A component contained in a container with this layout manager has following constraints properties:

Property Name	Description
Card Name	Identifier that can be used to make a card visible. See API documentation for `CardLayout.show(Container, String)`.

FlowLayout

The flow layout manager arranges components in a row from left to right, starting a new row if no more components fit into a row. Flow layouts are typically used to arrange buttons in a panel.

FlowLayout is part of the standard Java distribution.

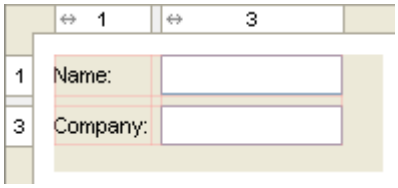
Layout properties

A container with this layout manager has following layout properties:

Property Name	Description
alignment	The alignment of the layout. Possible values: LEFT, RIGHT, LEADING and TRAILING. Default is CENTER.
horizontal gap	The horizontal gap between components and between the component and the border of the container. Default is 5.
vertical gap	The vertical gap between components and between the component and the border of the container. Default is 5.
align on baseline (Java 6)	Specifies whether components are vertically aligned along their baseline. Components that do not have a baseline are centered. Default is false.

FormLayout (JGoodies)

FormLayout is a powerful, flexible and precise general purpose layout manager. It places components in a grid of columns and rows, allowing specified components to span multiple columns or rows. Not all columns/rows necessarily have the same width/height.



Unlike other grid based layout managers, FormLayout uses 1-based column/row indices. And it uses "real" columns/rows as gaps. Therefore the unusual column/row numbers in the above screenshot. Using gap columns/rows has the advantage that you can give gaps different sizes.

Use the column and row headers to insert or delete columns/rows and change column/row properties. JFormDesigner automatically adds/removes gap columns if you add/remove a column/row.

Compared to other layout managers, FormLayout provides following outstanding features:

- Default alignment of components in a column/row.
- Specification of minimum and maximum column width or row height.
- Supports different units: Dialog units, Pixel, Point, Millimeter, Centimeter and Inch. Especially Dialog units are very useful to create layouts that scale with the screen resolution.
- Column/row templates.
- Column/row grouping.

FormLayout is open source and **not** part of the standard Java distribution. You must ship an additional library with your application. JFormDesigner includes `forms.jar`, `forms-javadoc.zip` and `forms-src.zip` in its `redist` folder. For more documentation and tutorials, visit forms.dev.java.net.

IDE plug-ins: If you use FormLayout the first time, the JFormDesigner IDE plug-in ask you whether it should copy the required library (and its source code and documentation) to the IDE project and add it to the classpath of the IDE project.

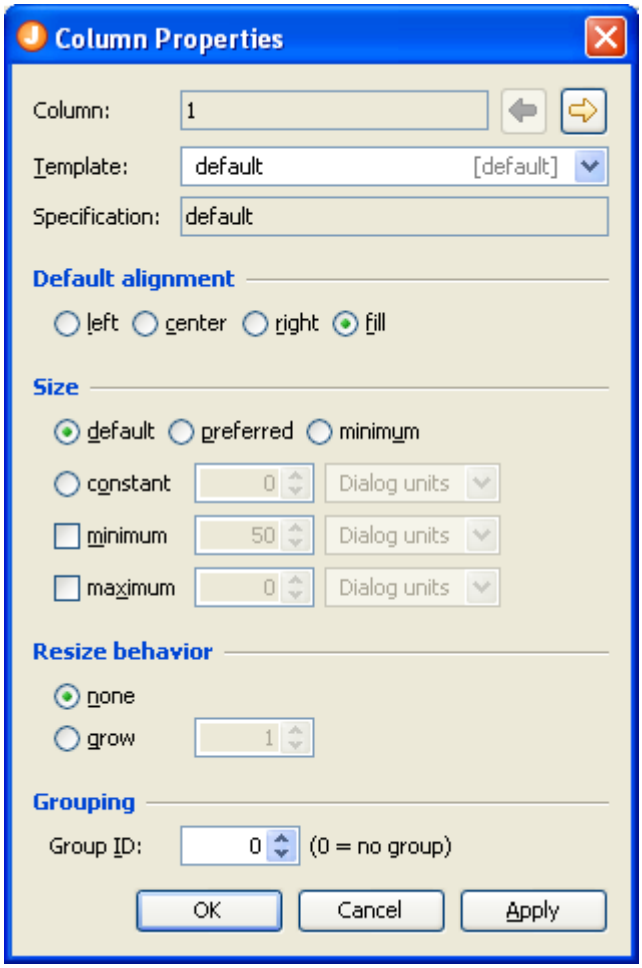
Layout properties

A container with this layout manager has following layout properties:

Property Name	Description
columnSpecs	Comma separated encoded column specifications. This property is for experts only. Use the column header instead of editing this property.
rowSpecs	Comma separated encoded row specifications. This property is for experts only. Use the row headers instead of editing this property.

Column/row properties

Each column and row has its own properties. Use the column and row headers to change column/row properties.



Property Name	Description
Column/Row	The index of the column/row. Use the arrow buttons (or **Alt+Left**, **Alt+Right**, **Alt+Up**, **Alt+Down** keys) to edit the properties of the previous or next column/row.
Template	FormLayout provides several predefined templates for columns and rows. Here you can choose one.
Specification	The column/row specification. This is a string representation of the options below.
Default alignment	The default alignment of the components within a column/row. Used if the value of the component constraint properties "h align" or "v align" are set to DEFAULT.
Size	The width of a column or height of a row. You can use default, preferred or minimum component size. Or a constant size. It is also possible to specify a minimum and a maximum size. Note that the maximum size does not limit the column/row size if the column/row can grow (see resize behavior).
Resize behavior	The resize weight of the column/row.
Grouping	See column/row grouping for details.

Tip: The column/row context menu allows you to alter many of these options for multi-selections.

Constraints properties

A component contained in a container with this layout manager has following constraints properties:

Property Name	Description
grid x	Specifies the component's horizontal grid origin (column index).
grid y	Specifies the component's vertical grid origin (row index).
grid width	Specifies the component's horizontal grid extend (number of columns). Default is 1.
grid height	Specifies the component's vertical grid extend (number of rows). Default is 1.
h align	The horizontal alignment of the component within its cell. Possible values: DEFAULT, LEFT, CENTER, RIGHT and FILL. Default is DEFAULT.
v align	The vertical alignment of the component within its cell. Possible values: DEFAULT, TOP, CENTER, BOTTOM and FILL. Default is DEFAULT.
insets	Specifies the external padding of the component, the minimum amount of space between the component and the edges of its display area. Default is [0,0,0,0]. Note that the insets do not increase the column width or row height (in contrast to the GridBagConstraints.insets).

Tip: The component context menu allows you to alter the alignment for multi-selections.

Column/Row Templates

FormLayout provides several predefined templates for columns and rows. You can also define custom column/row templates in the Preferences dialog.

Column templates

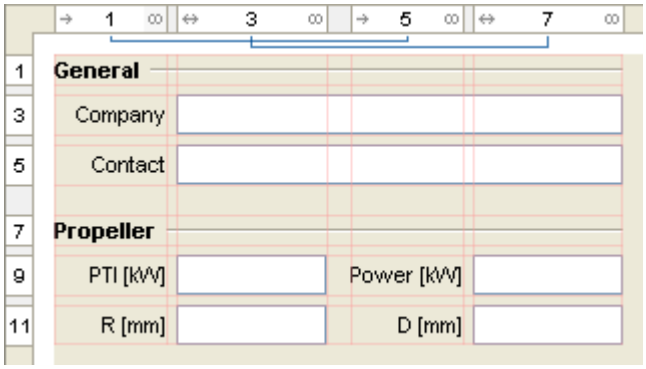
Name	Description	Gap
default	Determines the column width by computing the maximum of all column component preferred widths. If there is not enough space in the container, the column can shrink to the minimum width.	no
preferred	Determines the column width by computing the maximum of all column component preferred widths.	no
minimum	Determines the column width by computing the maximum of all column component minimum widths.	no
related gap	A logical horizontal gap between two related components. For example the OK and Cancel buttons are considered related.	yes
unrelated gap	A logical horizontal gap between two unrelated components.	yes
label component gap	A logical horizontal gap between a label and an associated component.	yes
glue	Has an initial width of 0 pixels and grows. Useful to describe *glue* columns that fill the space between other columns.	yes
button	A logical horizontal column for a fixed size button.	no
growing button	A logical horizontal column for a growing button.	no

Row templates

Name	Description	Gap
default	Determines the row height by computing the maximum of all row component preferred heights. If there is not enough space in the container, the row can shrink to the minimum height.	no
preferred	Determines the row height by computing the maximum of all row component preferred heights.	no
minimum	Determines the row height by computing the maximum of all row component minimum heights.	no
related gap	A logical vertical gap between two related components.	yes
unrelated gap	A logical vertical gap between two unrelated components.	yes
narrow line gap	A logical vertical narrow gap between two rows. Useful if the vertical space is scarce or if an individual vertical gap shall be smaller than the default line gap.	yes
line gap	A logical vertical default gap between two rows. A little bit larger than the narrow line gap.	yes
paragraph gap	A logical vertical default gap between two paragraphs in the layout grid. This gap is larger than the default line gap.	yes
glue	Has an initial height of 0 pixels and grows. Useful to describe *glue* rows that fill the space between other rows.	yes

Column/Row Groups

Column and row groups are used to specify that a set of columns or rows will get the same width or height. This is an essential feature for symmetric, and more generally, balanced design.

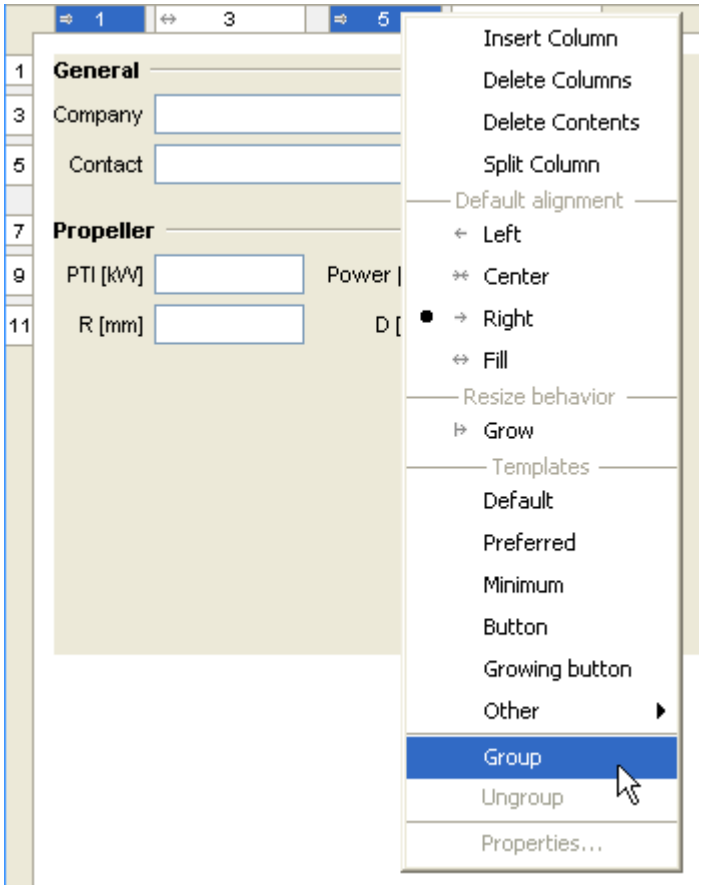


In the above example, columns [1 and 5] and columns [3 and 7] have the same width.

To visualize the grouping, JFormDesigner displays lines connecting the grouped columns/rows near to the column and row headers.

Group columns/rows

To create a new group, select the columns/rows you want to group in the header, right-click on a selected column/row in the header and select **Group** from the popup menu.



Note that selected gap columns/rows will be ignored when grouping.

You can extend existing groups by selecting at least one column/row of the existing group and the columns/rows that you want to add to that group, then right-click on a selected column/row and select **Group** from the popup menu.

Ungroup columns/lines

To remove a group, select all columns/rows of the group, right-click on a selected column/row and select **Ungroup** from the popup menu.

To remove a column/row from a group, right-click on it and select **Ungroup** from the popup menu.

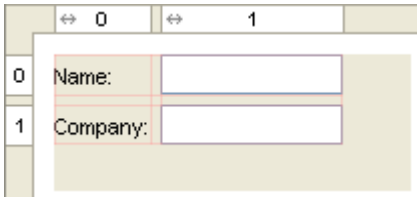
Group IDs

A unique group ID identifies each group. When using the header context menu to group/ungroup, you don't have to care about those IDs. JFormDesigner manages the group IDs automatically.

However it is possible to edit the group ID in the Column/row properties dialog.

GridBagLayout

The grid bag layout manager places components in a grid of columns and rows, allowing specified components to span multiple columns or rows. Not all columns/rows necessarily have the same width/height. Essentially, GridBagLayout places components in rectangles (cells) in a grid, and then uses the components' preferred sizes to determine how big the cells should be.

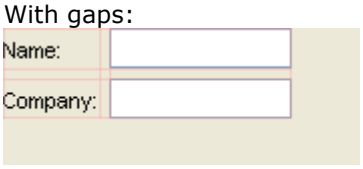


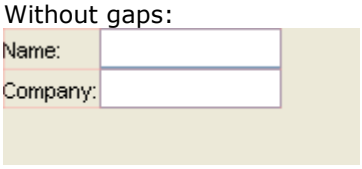
Use the column and row headers to insert or delete columns/rows and change column/row properties.

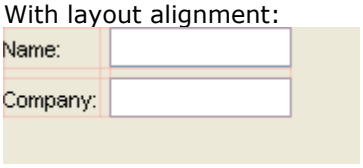
GridBagLayout is part of the standard Java distribution.

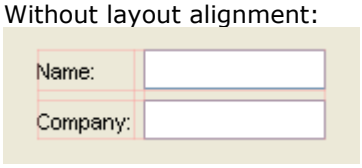
Extensions

JFormDesigner extends the original GridBagLayout with following features:

- **Horizontal and vertical gaps**
  Simply specify the gap size and JFormDesigner automatically computes the `GridBagConstraints.insets` for all components. This makes designing a form with consistent gaps using GridBagLayout much easier. No longer wrestling with `GridBagConstraints.insets`.
  With gaps:
  

  Without gaps:
  

- **Left/top layout alignment**
  The pure GridBagLayout centers the layout within the container if there is enough space. JFormDesigner easily allows you to fix this problem by switching on two options: align left and align top.
  With layout alignment:
  

  Without layout alignment:
  

- **Default component alignment**
  Allows you to specify a default alignment for components within columns/rows. This is very useful for columns with right aligned labels because you specify the alignment only once for the column and all added labels will automatically aligned to the right.

Layout properties

A container with this layout manager has following layout properties:

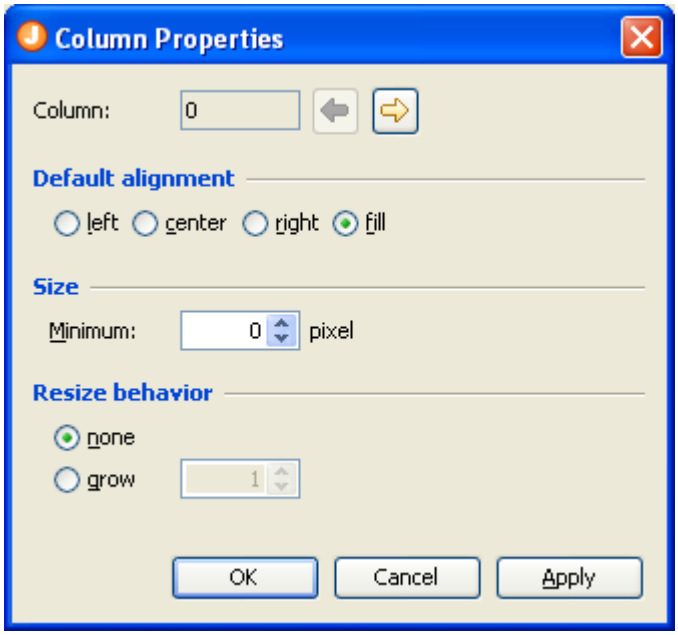
Property Name	Description
horizontal gap	The horizontal gap between components. Default is 5.
vertical gap	The vertical gap between components. Default is 5.
align left	If true, aligns the layout to the left side of the container. If false, then the layout is centered horizontally. Default is true.

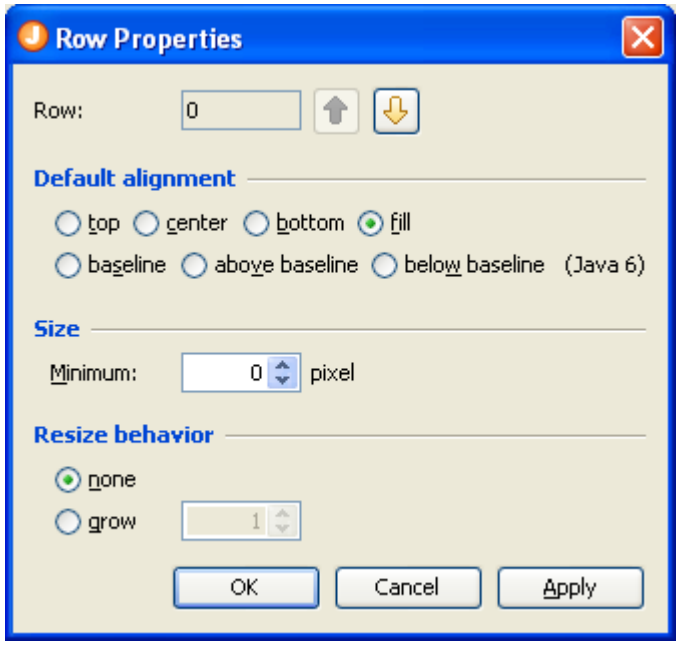
Property Name	Description
align top	If true, aligns the layout to the top side of the container. If false, then the layout is centered vertically. Default is true.

These four properties are JFormDesigner extensions to the original GridBagLayout. However, no additional library is required.

Column/row properties

Each column and row has its own properties. Use the column and row headers to change column/row properties.





Property Name	Description
Column/Row	The index of the column/row. Use the arrow buttons (or **Alt+Left**, **Alt+Right**, **Alt+Up**, **Alt+Down** keys) to edit the properties of the previous or next column/row.
Default alignment	The default alignment of the components within a column/row. Used if the value of the constraints properties "h align" or "v align" is DEFAULT.
Size	The minimum width of a column or height of a row.
Resize behavior	The resize weight of the column/row.

Tip: The column/row context menu allows you to alter many of these options for multi-selections.

Constraints properties

A component contained in a container with this layout manager has following constraints properties:

Property Name	Description
grid x	Specifies the component's horizontal grid origin (column index).
grid y	Specifies the component's vertical grid origin (row index).
grid width	Specifies the component's horizontal grid extend (number of columns). Default is 1.

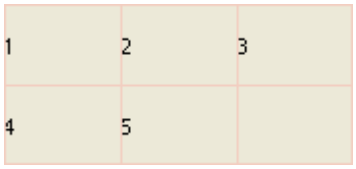
Property Name	Description
grid height	Specifies the component's vertical grid extend (number of rows). Default is 1.
h align	The horizontal alignment of the component within its cell. Possible values: DEFAULT, LEFT, CENTER, RIGHT and FILL. Default is DEFAULT.
v align	The vertical alignment of the component within its cell. Possible values: DEFAULT, TOP, CENTER, BOTTOM, FILL, BASELINE (Java 6), ABOVE_BASELINE (Java 6) and BELOW_BASELINE (Java 6). Default is DEFAULT.
weight x	Specifies how to distribute extra horizontal space. Default is 0.0.
weight y	Specifies how to distribute extra vertical space. Default is 0.0.
insets	Specifies the external padding of the component, the minimum amount of space between the component and the edges of its display area. Default is [0,0,0,0].
ipad x	Specifies the internal padding of the component, how much space to add to the minimum width of the component. Default is 0.
ipad y	Specifies the internal padding, that is, how much space to add to the minimum height of the component. Default is 0.

In contrast to the GridBagConstraints API, which uses `anchor` and `fill` to specify the alignment and resize behavior of a component, JFormDesigner uses the usual `h/v align` notation.

Tip: The component context menu allows you to alter the alignment for multi-selections.

GridLayout

The grid layout manager places components in a grid of cells. Each component takes all the available space within its cell, and each cell is exactly the same size.



This layout manager is used rarely.

GridLayout is part of the standard Java distribution.

Layout properties

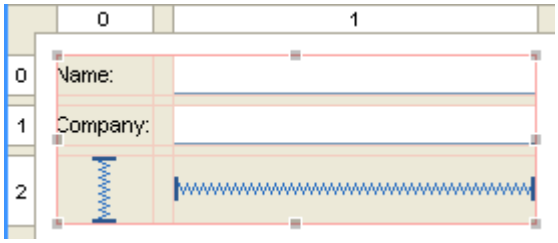
A container with this layout manager has following layout properties:

Property Name	Description
columns	The number of columns. Zero means any number of columns.
rows	The number of rows. Zero means any number of rows. **Note**: If the number of rows is non-zero, the number of columns specified is ignored. Instead, the number of columns is determined from the specified number or rows and the total number of components in the layout.
horizontal gap	The horizontal gap between components. Default is 0.
vertical gap	The vertical gap between components. Default is 0.

IntelliJ IDEA GridLayout

The IntelliJ IDEA grid layout manager places components in a grid of columns and rows, allowing specified components to span multiple columns or rows. Not all columns/rows necessarily have the same width/height.

Note: The IntelliJ IDEA grid layout manager is supported to make it easier to migrate forms, which were created with IntelliJ IDEA's GUI builder. If you never used it, it is recommended to use one of the other grid-based layout managers.



Use the column and row headers to insert or delete columns/rows and change column/row properties. Use horizontal and vertical spacers, which are available in the Palette, to define space between components.

IntelliJ IDEA GridLayout is open source and **not** part of the standard Java distribution. You must ship an additional library with your application. JFormDesigner includes `intellij_forms_rt.jar` and `intellij_forms_rt_src.zip` in its `redist` folder. For more documentation and tutorials, visit www.jetbrains.com/idea/.

IDE plug-ins: If you use IntelliJ IDEA GridLayout the first time, the JFormDesigner IDE plug-in ask you whether it should copy the required library (and its source code) to the IDE project and add it to the classpath of the IDE project.

Layout properties

A container with this layout manager has following layout properties:

Property Name	Description
horizontal gap	The horizontal gap between components. If -1, then inherits gap from parent container that also uses IntelliJ IDEA GridLayout, or uses 10 pixel. Default is -1.
vertical gap	The vertical gap between components. If -1, then inherits gap from parent container that also uses IntelliJ IDEA GridLayout, or uses 5 pixel. Default is -1.
same size horizontally	If true, all columns get the same width. Default is false.
same size vertically	If true, all rows get the same height. Default is false.
margin	Size of the margin between the containers border and its contents. Default is 0, 0, 0, 0.

Constraints properties

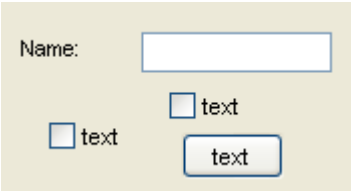
A component contained in a container with this layout manager has following constraints properties:

Property Name	Description
grid x	Specifies the component's horizontal grid origin (column index).
grid y	Specifies the component's vertical grid origin (row index).
grid width	Specifies the component's horizontal grid extend (number of columns). Default is 1.

Property Name	Description
grid height	Specifies the component's vertical grid extend (number of rows). Default is 1.
fill	Specifies how the component fills its cell. Possible values: None, Horizontal, Vertical and Both. Default is None.
anchor	Specifies how the component is aligned within its cell. Possible values: Center, North, North East, East, South East, South, South West, West and North West. Default is Center.
indent	The indent of the component within its cell. In pixel multiplied by 10. Default is 0.
align grid with parent	If true, align the grid of nested containers, which use IntelliJ IDEA GridLayout, with the grid of this container. Default is false.
horizontal size policy	Specifies how the component affects horizontal resizing behavior. Possible values: Fixed, Can Shrink, Can Grow, Want Grow and combinations. Default is Can Shrink and Can Grow.
vertical size policy	Specifies how the component affects vertical resizing behavior. Possible values: Fixed, Can Shrink, Can Grow, Want Grow and combinations. Default is Can Shrink and Can Grow.
minimum size	The minimum size of the component. Default is -1, -1.
preferred size	The preferred size of the component. Default is -1, -1.
maximum size	The maximum size of the component. Default is -1, -1.

null Layout

null layout is not a real layout manager. It means that no layout manager is assigned and the components can be put at specific x,y coordinates.



It is useful for making quick prototypes. But it is not recommended for production because it is not portable. The fixed locations an sizes do not change with the environment (e.g. different fonts on various platforms).

Preferred sizes

JFormDesigner supports preferred sizes of child components. This solves one common problem of null layout: the component sizes change with the environment (e.g. different fonts on various platforms). Unlike other GUI designers, no additional library is required.

Grid

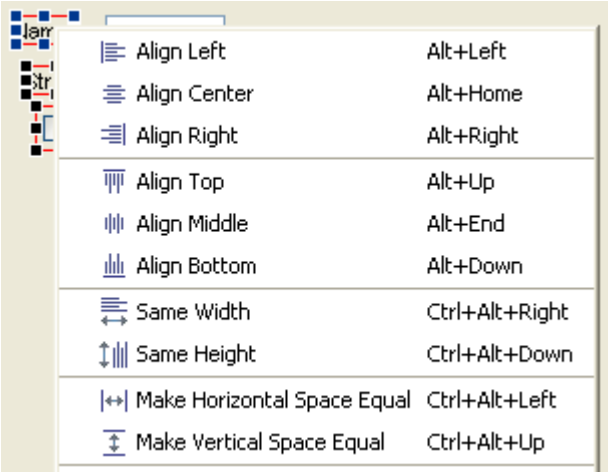
To make it easier to align components, the component edges snap to an invisible grid when moving or resizing components. You can specify the grid step size in the Preferences dialog. To temporary disable grid snapping, hold down the **Shift** key while moving or resizing components.

Keyboard

You can move selected components with **Ctrl+ArrowKey** and change size with **Shift+ArrowKey**.

Aligning components

The align commands help you to align a set of components or make them same width or height.



The dark blue handles in the above screenshot indicate the first selected component.

Align Left	Line up the left edges of the selected components with the left edge of the first selected component.
Align Center	Horizontally line up the centers of the selected components with the center of the first selected component.
Align Right	Line up the right edges of the selected components with the right edge of the first selected component.
Align Top	Line up the top edges of the selected components with the top edge of the first selected component.
Align Middle	Vertically line up the centers of the selected components with the center of the first selected component.
Align Bottom	Line up the bottom edges of the selected components with the bottom edge of the first selected component.
Same Width	Make the selected components all the same width as the first selected component.
Same Height	Make the selected components all the same height as the first selected component.
Make Horizontal Space Equal	Makes the horizontal space between 3 or more selected components equal. The leftmost and rightmost components stay unchanged. The other components are horizontally distributed between the leftmost and rightmost components.
Make Vertical Space Equal	Makes the vertical space between 3 or more selected components equal. The topmost and bottommost components stay unchanged. The other components are vertically distributed between the topmost and bottommost components.

Layout properties

A container with this layout manager has following layout properties:

Property Name	Description
auto-size	If true, computes the size of the container so that all children are entire visible. If false, the size of the container in the Design view is used. Default is true.

Constraints properties

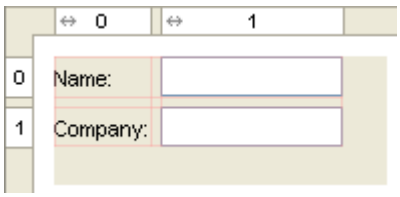
A component contained in a container with this layout manager has following constraints properties:

Property Name	Description
x	The x coordinate of the component relative to the left corner of the container.
y	The y coordinate of the component relative to the upper corner of the container.
width	The width of the component in pixel or Preferred. If set to Preferred, the component's preferred width is used. Default is Preferred.
height	The height of the component in pixel or Preferred. If set to Preferred, the component's preferred width is used. Default is Preferred.

TableLayout

The table layout manager places components in a grid of columns and rows, allowing specified components to span multiple columns or rows. Not all columns/rows necessarily have the same width/height.

A column/row can be given an absolute size in pixels, a percentage of the available space, or it can grow and shrink to fill the remaining space after other columns/rows have been resized.



Use the column and row headers to insert or delete columns/rows and change column/row properties.

TableLayout is open source and **not** part of the standard Java distribution. You must ship an additional library with your application. JFormDesigner includes `TableLayout.jar`, `TableLayout-javadoc.jar` and `TableLayout-src.zip` in its `redist` folder. For more documentation and tutorials, visit tablelayout.dev.java.net.

IDE plug-ins: If you use TableLayout the first time, the JFormDesigner IDE plug-in ask you whether it should copy the required library (and its source code and documentation) to the IDE project and add it to the classpath of the IDE project.

Extensions

JFormDesigner extends the original TableLayout with following features:

- **Default component alignment**
  Allows you to specify a default alignment for components within columns/rows. This is very useful for columns with right aligned labels because you specify the alignment only once for the column and all added labels will automatically aligned to the right.

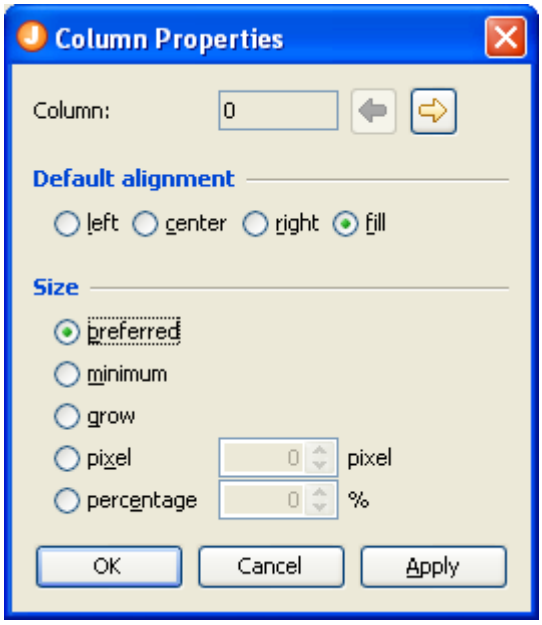
Layout properties

A container with this layout manager has following layout properties:

Property Name	Description
horizontal gap	The horizontal gap between components. Default is 5.
vertical gap	The vertical gap between components. Default is 5.

Column/row properties

Each column and row has its own properties. Use the column and row headers to change column/row properties.



Property Name	Description
Column/Row	The index of the column/row. Use the arrow buttons (or **Alt+Left**, **Alt+Right**, **Alt+Up**, **Alt+Down** keys) to edit the properties of the previous or next column/row.
Default alignment	The default alignment of the components within a column/row. Used if the value of the constraints properties "h align" or "v align" is DEFAULT.
Size	Specifies how TableLayout computes the width/height of a column/row.

Tip: The column/row context menu allows you to alter many of these options for multi-selections.

Constraints properties

A component contained in a container with this layout manager has following constraints properties:

Property Name	Description
grid x	Specifies the component's horizontal grid origin (column index).
grid y	Specifies the component's vertical grid origin (row index).
grid width	Specifies the component's horizontal grid extend (number of columns). Default is 1.
grid height	Specifies the component's vertical grid extend (number of rows). Default is 1.
h align	The horizontal alignment of the component within its cell. Possible values: DEFAULT, LEFT, CENTER, RIGHT and FILL. Default is DEFAULT.
v align	The vertical alignment of the component within its cell. Possible values: DEFAULT, TOP, CENTER, BOTTOM and FILL. Default is DEFAULT.

In contrast to the TableLayoutConstraints API, which uses [column1,row1,column2,row2] to specify the location and size of a component, JFormDesigner uses the usual [x,y,width,height] notation.

Tip: The component context menu allows you to alter the alignment for multi-selections.

Java Code Generator

JFormDesigner can generate and update Java source code. It uses the same name for the Java file as for the Form file. E.g.:

```
C:\MyProject\src\com\myproject\WelcomeDialog.jfd (form file)
C:\MyProject\src\com\myproject\WelcomeDialog.java (java file)
```

Stand-alone: Before creating new forms, you should specify the locations of your Java source folders in the Project dialog. Then JFormDesigner can generate valid `package` statements. For the above example, you should add `C:\MyProject\src`.

IDE plug-ins: The source folders of the IDE projects are used.

If the Java file does not exist, JFormDesigner generates a new one. Otherwise it parses the existing Java file and inserts/updates the code for the form and adds import statements if necessary.

Stand-alone: The Java file will be updated when saving the form file.

IDE plug-ins: If the Java file is opened in the IDE editor, it will be immediately updated in-memory on each change in JFormDesigner. Otherwise it will be updated when saving the form file.

JFormDesigner uses special comments to identify the code sections that it will generate/update. E.g.:

```
// JFormDesigner - ... //GEN-BEGIN:initComponents
// JFormDesigner - ... //GEN-END:initComponents
```

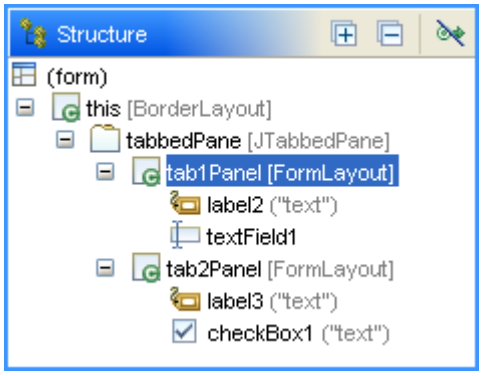
The starting comment must contain `GEN-BEGIN:<keyword>`, the ending comment `GEN-END:<keyword>`. These comments are NetBeans compatible. The text before `GEN-BEGIN` and `GEN-END` (in the same line) does not matter. JFormDesigner uses the following keywords:

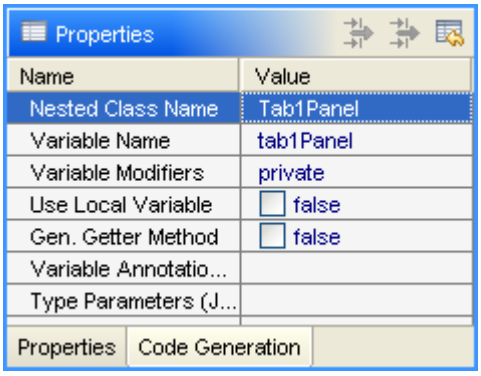
Keyword name	Description
initComponents	Used for code that instantiates and initializes the components of the form.
variables	Used for code that declares the class level variables for components.
initI18n	Used for code that initializes localized component properties.

Nested Classes

One of the advanced features of JFormDesigner is the generation of nested classes. Normally, all code for a form is generated into one class. If you have forms with many components, e.g. a `JTabbedPane` with some tabs, it is not recommended to have only one class. If you hand-code such a form, you would create a class for each tab.

In JFormDesigner you can specify a nested class for each component. You do this on the **Code Generation** tab in the Properties view. JFormDesigner automatically generates/updates the specified nested classes. This allows you to program more object-oriented and makes your code easier to read and maintain.





Components having a nested class are marked with a overlay symbol in the Structure view.

Example source code:

```
public class NestedClassDemo
    extends JPanel
{
    public NestedClassDemo() {
        initComponents();
    }

    private void initComponents() {
        // JFormDesigner - Component initialization - DO NOT MODIFY  //GEN-BEGIN:initComponents
        tabbedPane = new JTabbedPane();
        tab1Panel = new Tab1Panel();
        tab2Panel = new Tab2Panel();

        //======== this ========
        setLayout(new BorderLayout());

        //======== tabbedPane ========
        {
            tabbedPane.addTab("tab 1", tab1Panel);
            tabbedPane.addTab("tab 2", tab2Panel);
        }
        add(tabbedPane, BorderLayout.CENTER);
        // JFormDesigner - End of component initialization  //GEN-END:initComponents
    }

    // JFormDesigner - Variables declaration - DO NOT MODIFY  //GEN-BEGIN:variables
    private JTabbedPane tabbedPane;
    private Tab1Panel tab1Panel;
    private Tab2Panel tab2Panel;
    // JFormDesigner - End of variables declaration  //GEN-END:variables


    private class Tab1Panel
        extends JPanel
    {
        private Tab1Panel() {
            initComponents();
```

```
        }

        private void initComponents() {
            // JFormDesigner - Component initialization - DO NOT MODIFY  //GEN-
BEGIN:initComponents
            label2 = new JLabel();
            textField1 = new JTextField();
            CellConstraints cc = new CellConstraints();

            //======== this ========
            setBorder(Borders.TABBED_DIALOG_BORDER);
            setLayout(new FormLayout( ... ));

            //---- label2 ----
            label2.setText("text");
            add(label2, cc.xy(1, 1));

            //---- textField1 ----
            add(textField1, cc.xy(3, 1));
            // JFormDesigner - End of component initialization  //GEN-END:initComponents
        }

        // JFormDesigner - Variables declaration - DO NOT MODIFY  //GEN-BEGIN:variables
        private JLabel label2;
        private JTextField textField1;
        // JFormDesigner - End of variables declaration  //GEN-END:variables
    }


    private class Tab2Panel
        extends JPanel
    {
        private Tab2Panel() {
            initComponents();
        }

        private void initComponents() {
            // JFormDesigner - Component initialization - DO NOT MODIFY  //GEN-
BEGIN:initComponents
            label3 = new JLabel();
            checkBox1 = new JCheckBox();
            CellConstraints cc = new CellConstraints();

            //======== this ========
            setBorder(Borders.TABBED_DIALOG_BORDER);
            setLayout(new FormLayout( ... ));

            //---- label3 ----
            label3.setText("text");
            add(label3, cc.xy(1, 1));

            //---- checkBox1 ----
            checkBox1.setText("text");
            add(checkBox1, cc.xy(3, 1));
            // JFormDesigner - End of component initialization  //GEN-END:initComponents
        }

        // JFormDesigner - Variables declaration - DO NOT MODIFY  //GEN-BEGIN:variables
        private JLabel label3;
        private JCheckBox checkBox1;
        // JFormDesigner - End of variables declaration  //GEN-END:variables
    }
}
```

When changing the nested class name on the **Code Generation** tab (Properties view), JFormDesigner also renames the nested class in the Java source code. When removing the nested class name, then JFormDesigner does not remove the nested class in the Java source code to avoid loss of own source code.

Code Templates

When generating new Java files or classes, JFormDesigner uses the templates specified in the Preferences dialog.

Template name	Description
File header	Used when creating new Java files. Contains a header comment and a `package` statement.
Class	Used when generating a new (nested) class. Contains a class declaration, a constructor, a component initialization method and variable declarations.
Empty Class	Used when generating a new empty class. This can happen, if all form components are contained in nested classes.
Event Handler Body	Used for event handler method bodies.
Component initialization	Replaces the variable *${component_initialization}* used in other templates. Contains a method named `initComponents`. Invoke this method from your code to instantiate the components of your form. Feel free to change the method name if you don't like it.
Variables declaration	Replaces the variable *${variables_declaration}* used in other templates.
java.awt.Dialog	A template for classes derived from `java.awt.Dialog`. Compared to the "Class" template, this has special constructors, which are necessary for `java.awt.Dialog` derived classes.
javax.swing.AbstractAction	Used for nested action classes.

You can change the existing templates or create additional templates in the Preferences dialog. It is possible to define your own templates for specific superclasses.

Following variables can be used in the templates:

Variable name	Description	Context
${date}	Current date.	global
${user}	User name.	global
${package_declaration}	`package` statement. If the form is not saved under one of the source folders specified in the Project dialog, the variable is empty (no `package` statement will be generated).	file header
${class_name}	Name of the (nested) class.	class
${component_initialization}	See template "Component initialization".	class
${constructor_modifiers}	Modifiers of the constructor. Based on the class modifiers.	class
${extends_declaration}	The `extends` declaration of the class; empty if the class has no superclass.	class
${modifiers}	Modifiers of the (nested) class. You can specify the default modifiers in the Preferences dialog.	class
${variables_declaration}	See template "Variables declaration".	class

Runtime Library

Note: If you use the Java code generator, you don't need this library.

The open-source (BSD license) runtime library allows you to load JFormDesigner XML files at runtime within your applications. Turn off the Java code generation in the Preferences dialog if you use this library.

You'll find the library `jfd-loader.jar` in the `redist` folder (or plug-in) of the JFormDesigner installation; the source code is in jfd-loader-src.zip; the documentation is in `jfd-loader-javadoc.zip` and an example in the `examples` folder or `examples.zip` archive.

Classes

- `FormLoader` provides methods to load JFormDesigner .jfd files into in-memory form models.
- `FormCreator` creates instances of Swing components from in-memory form models and provides methods to access components.
- `FormSaver` saves in-memory form models to JFormDesigner .jfd files. Can be used to convert proprietary form specifications to JFormDesigner .jfd files: first create a in-memory form model from your form specification, then save the model to a .jfd file.

Example

The following example demonstrates the usage of FormLoader and FormCreator. It is included in the examples distributed with all JFormDesigner editions.

```
public class LoaderExample
{
    private JDialog dialog;

    public static void main(String[] args) {
        new LoaderExample();
    }

    LoaderExample() {
        try {
            // load the .jfd file into memory
            FormModel formModel = FormLoader.load(
                "com/jformdesigner/examples/LoaderExample.jfd");

            // create a dialog
            FormCreator formCreator = new FormCreator(formModel);
            formCreator.setTarget(this);
            dialog = formCreator.createDialog(null);

            // get references to components
            JTextField nameField = formCreator.getTextField("nameField")
;
            JCheckBox checkBox = formCreator.getCheckBox("checkBox");

            // set values
            nameField.setText("enter name here");
            checkBox.setSelected(true);

            // show dialog
            dialog.setModal(true);
            dialog.pack();
            dialog.show();

            System.out.println(nameField.getText());
            System.out.println(checkBox.isSelected());
            System.exit(0);
        } catch(Exception ex) {
```

```
            ex.printStackTrace();
        }
    }

    // event handler for checkBox
    private void checkBoxActionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(dialog, "check box clicked");
    }

    // event handler for okButton
    private void okButtonActionPerformed() {
        dialog.dispose();
    }
}
```

JavaBeans

What is a Java Bean?

> A Java Bean is a reusable software component that can be manipulated visually in a builder tool.

JavaBean (components) are self-contained, reusable software units that can be visually composed into composite components and applications. A bean is a Java class that has:

- a "null" constructor (without parameters)
- properties defined by getter and setter methods.

JFormDesigner supports:

- Visual beans (must inherit from `java.awt.Component`).
- Non-visual beans.

BeanInfo

JFormDesigner supports/uses following classes/interfaces specified in the `java.beans` package:

- BeanInfo
- BeanDescriptor
- EventSetDescriptor
- PropertyDescriptor
- PropertyEditor (incl. support for custom and paintable editors)
- Customizer

If you are writing BeanInfo classes for your custom components, you can specify additional information needed by JFormDesigner using the `java.beans.FeatureDescriptor` extension mechanism.

For example implementations of BeanInfos and PropertyEditors, take a look at the examples in the `examples` folder or `examples.zip` archive.

Attribute Name	Description
isContainer (BeanDescriptor)	Specifies whether a component is a container or not. A container can have child components. The value must be a `Boolean`. Default is false. E.g. `beanDesc.setValue("isContainer", Boolean.TRUE);`
containerDelegate (BeanDescriptor)	If components should be added to a descendant of a container, then it is possible to specify a method that returns the container for the children. `JFrame.getContentPane()` is a example for such a method. The value must be a `String` and specifies the name of a method that takes no arguments and returns a `java.awt.Container`. E.g. `beanDesc.setValue("containerDelegate", "getContentPane");`
layoutManager (BeanDescriptor)	Allows the specification of a layout manager, which is used when adding the component to a form. If specified, then JFormDesigner does not allow the selection of a layout manager. The value must be a `Class`. E.g. `beanDesc.setValue("layoutManager", BorderLayout.class);`

Attribute Name	Description

enumerationValues (PropertyDescriptor)	Specifies a list of valid property values. The value must be a `Object[]`. For each property value, the `Object[]` must contain three items:
	• Name: A displayable name for the property value.
	• Value: The actual property value.
	• Java Initialization String: A Java code piece used when generating code.
	
	`propDesc.setValue("enumerationValues", new Object[] {`
	`  "horizontal", new Integer( JSlider.HORIZONTAL ),`
	`  "JSlider.HORIZONTAL",`
	`  "vertical", new Integer( JSlider.VERTICAL ),`
	`  "JSlider.VERTICAL" });`

extraPersistenceDelegates (PropertyDescriptor)	Specifies a list of persistence delegates for classes. The value must be a `Object[]`. For each class, the `Object[]` must contain two items:
	• Class: The class for which the persistence delegate should be used.
	• Persistence delegate: Instance of a class, which extends `java.beans.PersistenceDelegate`, that should be used to persist an instance of the specified class.
	
	Use the attribute "persistenceDelegate" (see below) to specify a persistence delegate for a property value. Use this attribute to specify persistence delegates for classes that are referenced by a property value. E.g. if a property value references classes MyClass1 and MyClass2:
	
	`propDesc.setValue("extraPersistenceDelegates", new Object[] {`
	`  MyClass1.class, new MyClass1PersistenceDelegate(),`
	`  MyClass2.class, new MyClass2PersistenceDelegate(),`
	`});`

imports (PropertyDescriptor)	Specifies one or more class names for which import statements should be generated by the Java code generator. This is useful if you don't use full qualified class names in `enumerationValues` or `PropertyEditor.getJavaInitializationString()`. The value must be a `String` or `String[]`. E.g.
	
	`propDesc.setValue("imports", "com.mycompany.MyConstants");`
	`propDesc.setValue("imports", new String[] {`
	`  "com.mycompany.MyConstants",`
	`  "com.mycompany.MyExtendedConstants" });`

notMultiSelection (PropertyDescriptor)	Specifies whether the property is not shown in the Properties view when multiple components are selected. The value must be a `Boolean`. Default is false. E.g. `propDesc.setValue("notMultiSelection", Boolean.TRUE);`
notNull (PropertyDescriptor)	Specifies that a property can not set to `null` in the Properties view. If true, the **Set Value to null** command is disabled. The value must be a `Boolean`. Default is false. E.g. `propDesc.setValue("notNull", Boolean.TRUE);`
notRestoreDefault (PropertyDescriptor)	Specifies that a property value can not restored to the default in the Properties view. If true, the **Restore Default Value** command is disabled. The value must be a `Boolean`. Default is false. E.g. `propDesc.setValue("notRestoreDefault", Boolean.TRUE);`

Attribute Name	Description
persistenceDelegate (PropertyDescriptor)	Specifies an instance of a class, which extends `java.beans.PersistenceDelegate`, that can be used to persist an instance of a property value. E.g. `propDesc.setValue("persistenceDelegate", new MyPropPersistenceDelegate());`
readOnly (PropertyDescriptor)	Specifies that a property is read-only in the Properties view. The value must be a `Boolean`. Default is false. E.g. `propDesc.setValue("readOnly", Boolean.TRUE);`
transient (PropertyDescriptor)	Specifies that the property value should not persisted and no code should generated. The value must be a `Boolean`. Default is false. E.g. `propDesc.setValue("transient", Boolean.TRUE);`
variableDefault (PropertyDescriptor)	Specifies whether the default property value depends on other property values. The value must be a `Boolean`. Default is false. E.g. `propDesc.setValue("variableDefault", Boolean.TRUE);`

Design time

JavaBeans support the concept of "design"-mode, when JavaBeans are used in a GUI design tool, and "run"-mode, when JavaBeans are used in an application.

You can use the method `java.beans.Beans.isDesignTime()` in your JavaBean to determine whether it is running in JFormDesigner or in your application.

Reload beans

JFormDesigner supports reloading of JavaBeans.

Stand-alone: Just select **View > Refresh** from the menu or press **F5**.

IDE plug-ins: Click the **Refresh** button in the designer tool bar.

Refresh does following:

1. Create a new class loader for loading JavaBeans, BeanInfos and Icons.
2. Recreates the form in the active Design view.

So you can change the source code of the used JavaBeans, compile them in your IDE and use them in JFormDesigner without restarting.

Unsupported standard components

- all AWT components

JGoodies Forms & Looks

JFormDesigner supports and uses software provided by JGoodies Karsten Lentzsch.

The **JGoodies Forms** support is very extensive. Not only the layout manager FormLayout is supported, also some important helper classes are supported: `Borders`, `ComponentFactory` and `FormFactory` (com.jgoodies.forms.factories).

JGoodies Looks look and feels are built-in so that you can preview your forms using those popular look and feels. JGoodies Looks examples contains some useful components to build Eclipse like panels: JGoodies UIF lite.

JGoodies Forms ComponentFactory

The JGoodies Forms ComponentFactory (com.jgoodies.forms.factories) defines three factory methods, which create components. You find these components in the palette category JGoodies.

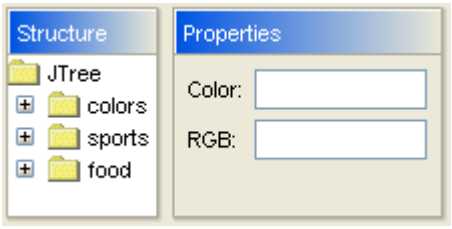
- `Label`: A label with an optional mnemonic. The mnemonic and mnemonic index are defined by a single ampersand (&). For example "&Save" or "Save &As". To use the ampersand itself duplicate it, for example "Look&&Feel".
- `Title`: A label that uses the foreground color and font of a `TitledBorder` with an optional mnemonic. The mnemonic and mnemonic index are defined by a single ampersand (&).
- `Titled Separator`: A labeled separator. Useful to separate paragraphs in a panel, which is often a better choice than a `TitledBorder`.

  text

JGoodies UIF lite

JFormDesigner supports `SimpleInternalFrame` and `UIFSplitPane` from the JGoodies UIF lite package, which is part of the JGoodies Looks examples. You find both components in the palette category JGoodies.

`SimpleInternalFrame` is an Eclipse like frame. `UIFSplitPane` is a subclass of `JSplitPane` that hides the divider border. Use `UIFSplitPane` if you want to put two `SimpleInternalFrame`s into a split pane. See example `examples/UIFLitePanel.jfd`.



When using one of these components, you have to add the library `redist/jgoodies-uif-lite.jar` to the classpath of your application. Or add the source code to your repository and compile it into your application. The source code is in `redist/jgoodies-uif-lite-src.zip`.

IDE plug-ins: If you use one of the UIF lite components the first time, the JFormDesigner IDE plug-in ask you whether it should copy the required library (and its source code and documentation) to the IDE project and add it to the classpath of the IDE project.

To add a toolbar to a `SimpleInternalFrame`, add a `JToolBar` to the Design view, select the `SimpleInternalFrame`, select the "toolBar" property in the Properties view and assign the toolbar to it.