JFormDesigner 5.1 Documentation

Version: 5.1

Copyright © 2004-2012 FormDev Software GmbH. All rights reserved.

Contents

1   Introduction

JFormDesigner is a professional **GUI designer** for Java Swing user interfaces. Its outstanding support for **JGoodies FormLayout**, GroupLayout (**Free Design**), TableLayout and GridBagLayout makes it easy to **create professional looking forms**.

Why use JFormDesigner?

JFormDesigner makes Swing GUI design a real pleasure. It **decreases the time** you spend on **hand coding** forms, giving you **more time** to focus on the **real tasks**. You'll find that JFormDesigner **quickly pays back its cost** in improved **GUI quality** and increased **developer productivity**. Even non-programmers can use it, which makes it also ideal for **prototyping**.

Editions

JFormDesigner is available in five editions: as **stand-alone** application and as IDE plug-ins for **Eclipse**, **NetBeans**, **IntelliJ IDEA** and **JBuilder**. This documentation covers all editions.

If there are functional differences between the editions, then they are marked with: **Stand-alone**, **Eclipse plug-in**, **NetBeans plug-in**, **IntelliJ IDEA plug-in**, **JBuilder plug-in** or **IDE plug-ins**.

Key features

- Easy and intuitive to use, powerful and productive
- IDE plug-ins and stand-alone application
- GroupLayout (Free Design) support
- JGoodies FormLayout and TableLayout support
- Advanced GridBagLayout support
- Column and row headers
- Localization support
- Beans Binding (JSR 295) support
- BeanInfo Annotations
- Java code generator or runtime library
- Generation of nested classes

2   User Interface

This is the main window of JFormDesigner **stand-alone** edition:
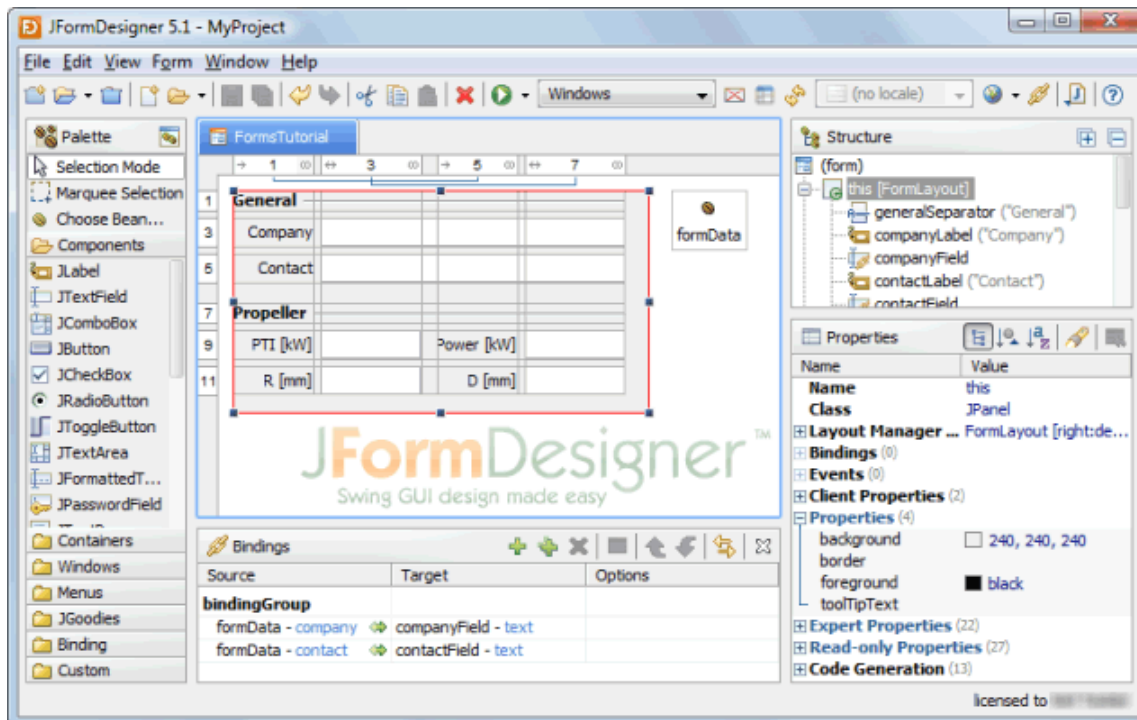
The main window consists of the following areas:

- 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.
- Bindings View: Located below the Design view. This view is not visible by default. Click the **Show Bindings View** button ( ) in the toolbar to make is visible.
- Error Log View: Located below the Design view. This view is not visible in the above screenshot.

2.1   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 Edit View Form Window Help

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 enabled 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 enabled in the Preferences).
	Save All	Saves all open forms and generates the Java source code for the forms (if Java Code Generation is enabled in the Preferences).
	Import	Imports NetBeans, IntelliJ IDEA or Abeille form files and creates new JFormDesigner forms. Use **File > Save** to save the new form in the same folder as the original form file. This also updates the .java file.
	Exit	Exits JFormDesigner. *Mac*: 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 Designer	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. You can use this command, if you changed the code of a component used in the form to reload the component classes. But usually this is not necessary because JFormDesigner automatically reloads 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. Usually 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 Bindings	Activates the Bindings view. By default, the Bindings view is not visible.
	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*: 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.
	Check for Updates	Checks whether a newer version of JFormDesigner is available.
	About	Displays information about JFormDesigner and the system properties. *Mac*: this item is in the JFormDesigner application menu.

Context menus

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

Design view context menu:
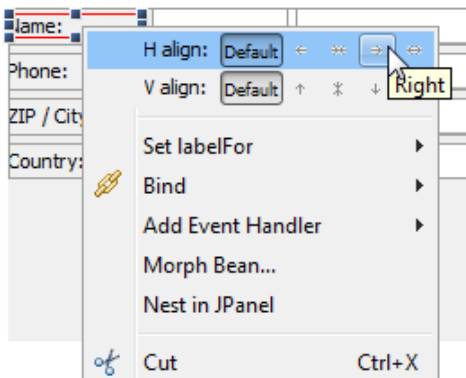
Properties view context menu:
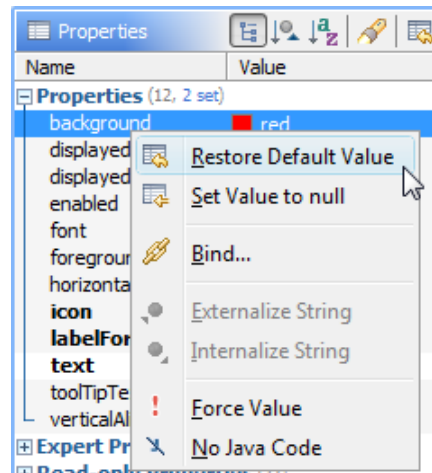
2.2   Toolbars

Toolbars provides shortcuts to often used commands.

Main Toolbar

This is the toolbar of JFormDesigner **stand-alone** edition. Many of the 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 enabled in the Preferences).
	Save All	Saves all open forms and generates the Java source code for the forms (if Java Code Generation is enabled 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 Designer	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. You can use this command, if you changed the code of a component used in the form to reload the component classes. But usually this is not necessary because JFormDesigner automatically reloads component classes.
German (de)		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.
	Show Bindings View	Shows the Bindings view.
	Generate Java Code	Generates the Java code for the active form. Usually 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.

2.3 Design View

This view is the central part of JFormDesigner. It displays the opened forms and lets you edit forms.
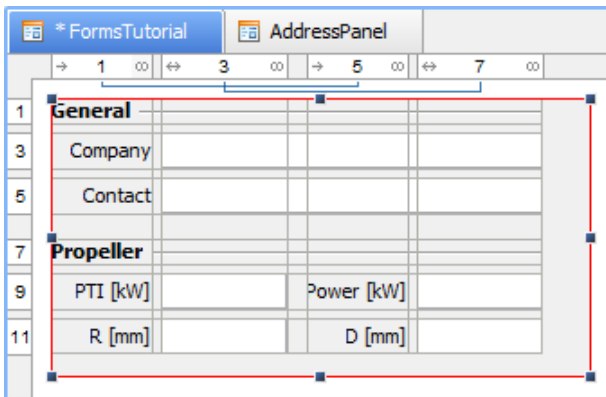
Stand-alone: At top of the view, tabs are displayed for each open form. Click on a tab to activate a form. To close a form, click the ✖ symbol that appears on the right side of a tab if the mouse is over it. An asterisk (*) in front of the form name indicates that the form has been changed.

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

On the top and left sides of the view, 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*: **Command**) or **Shift** key and click on the components. To select the parent of a selected component, hold down the **Alt** key (*Mac*: **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.
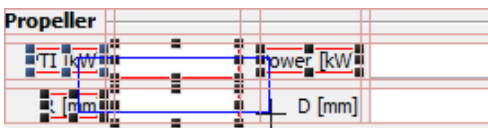
The selection in the Design view and in the Structure view is synchronized both ways.

Drag feedback

JFormDesigner provides four types of drag feedback.
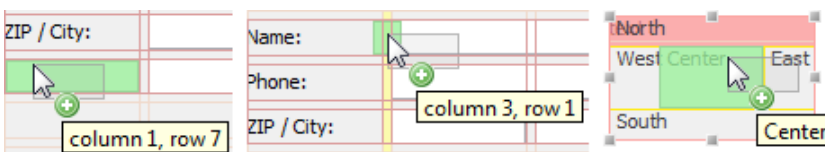
The <u>gray</u> figure shows the outline of the dragged components. It always follows the mouse location. The <u>green</u> figure indicates the drop location, the <u>yellow</u> figure indicates a new column/row and <u>red</u> figures indicate occupied areas.

Cursor feedback

JFormDesigner uses various cursors while dragging components:

The dragged components will be moved to the new location.

Either add a new component to the form or copy existing components.

Add multiple components of the same type to the form.

It is not possible to drop the component at this location.

Add components

To add components, choose a component from the Palette and drop it to the location where you want to add it.

To add multiple instances of a component, hold down the **Ctrl** key (*Mac*: **Command** key) while clicking on the Design view.

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*: **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.
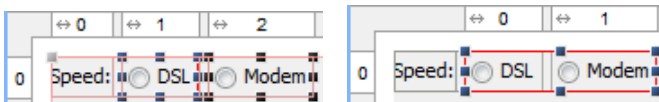
Non-visual beans

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.
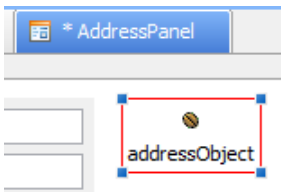
Red beans

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, take a look at the Error Log view and if necessary add required jars to the classpath of your project.

2.3.1   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.
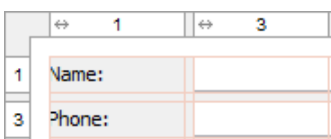
Use them to insert, delete or move columns/rows and change column/row properties. Right-clicking on a column/row displays a popup menu. Double-clicking shows a dialog that allows you to edit the column/row properties.

 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*: **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 <u>before</u> the right-clicked column/row. To add a column/row <u>after</u> 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.
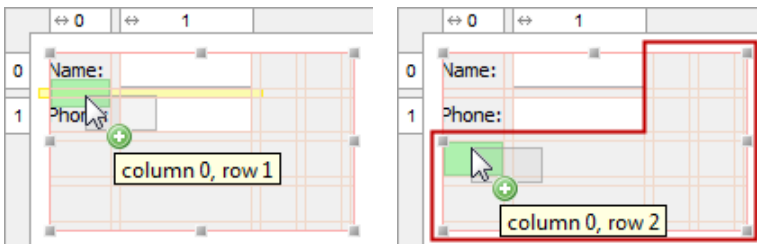
Delete columns/rows

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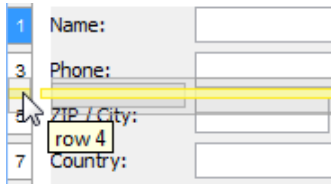
If the layout manager is FormLayout, then existing gap columns/rows are also moved. Hold down the **Shift** key before dropping a column/row to avoid this.

Resize columns/rows

To change the (minimum) size of a column/row, click near the right edge of a column/row and drag it.
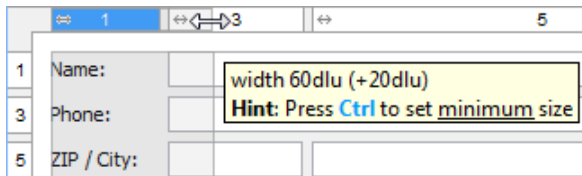
FormLayout supports minimum and constant column/row sizes. Hold down the **Ctrl** key to change the minimum size. TableLayout supports only constant sizes and GridBagLayout supports only minimum sizes.

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.
⍴	Baseline aligns components in the row.
⍴	Aligns components above baseline in the row.
⍴	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.

2.3.2 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.
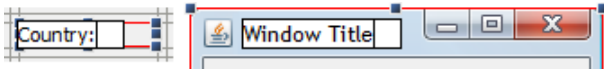
You can also use the **Space** key or double-click on a component to start in-place-editing. Confirm your changes using the **Enter** key, or cancel editing using the **Esc** key.

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

In-place-editing is also supported for the title of `TitledBorder` and the tab titles of JTabbedPane.
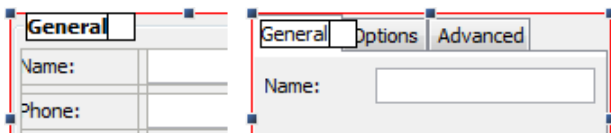
`TitledBorder`: double-click on the title of the `TitledBorder`; or select the component with the `TitledBorder` and start in-place-editing as usual.

`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.

2.3.3 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.

2.3.4 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
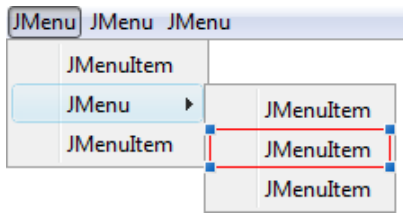
This figure shows the structure of a menu bar. The horizontal bar on top of the image is a `JMenuBar` that contains `JMenu` components. The `JMenu` contains `JMenuItem`, `JCheckBoxMenuItem`, `JRadioButtonMenuItem` or Menu Separator components. To create a sub-menu, put a `JMenu` into a `JMenu`.

The component palette provides 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.
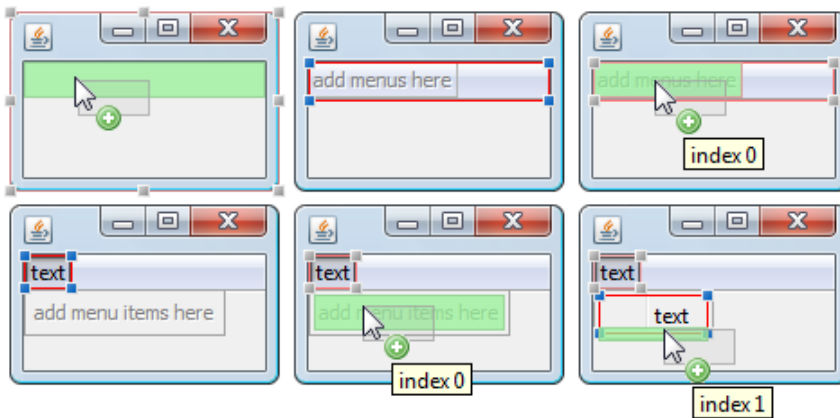
You can freely drag and drop the various menu components to rearrange them.

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`
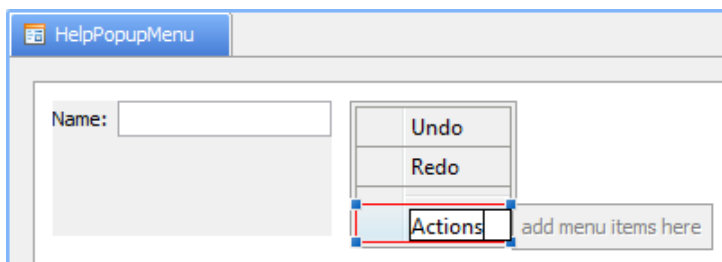
Assign popup menus to components

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 expand the **Expert Properties** category to see the property. Or use search as in the screenshot below.
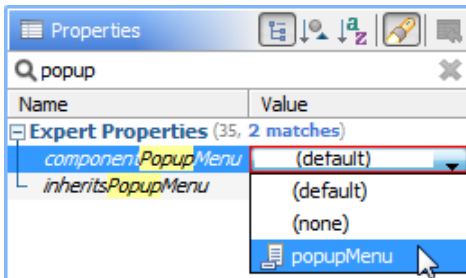
Note that JFormDesigner must run on Java 5 (or later) to use the "componentPopupMenu" property. Open the JFormDesigner About dialog and check whether it displays "Java 1.5.x" (or later).

2.3.5 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.
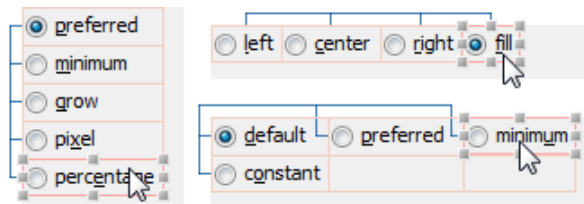
To visualize the grouping, JFormDesigner displays lines connecting the grouped buttons.

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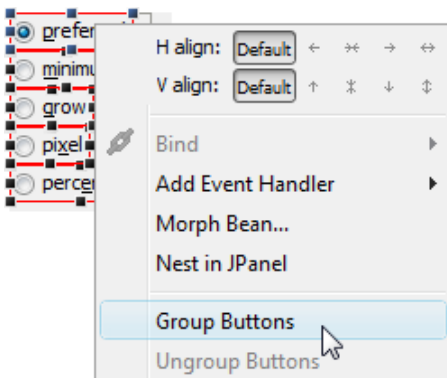
You can extend existing button groups by selecting at least one button of the existing group and the buttons that you want to add to that group, then right-click on a selected button and select **Group Buttons** from the popup menu.

Note that the **Group Buttons** and **Ungroup Buttons** commands are only available in the context menu if the selection contains only components that 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.
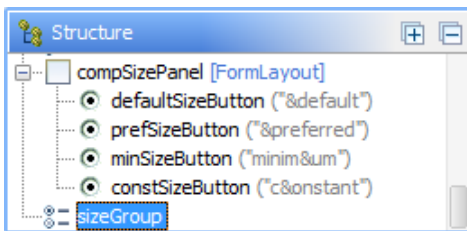
If a grouped button is selected, you can see the association to the button group in the Properties view.
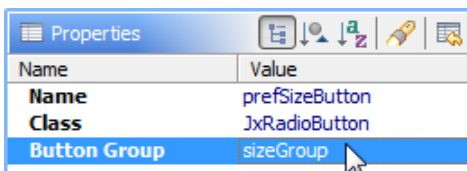
2.3.6 JTabbedPane

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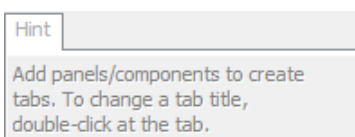
To add pages, select an appropriate component (e.g. JPanel) in the palette, move the cursor over the tabs area of the JTabbedPane and click to add it.

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.
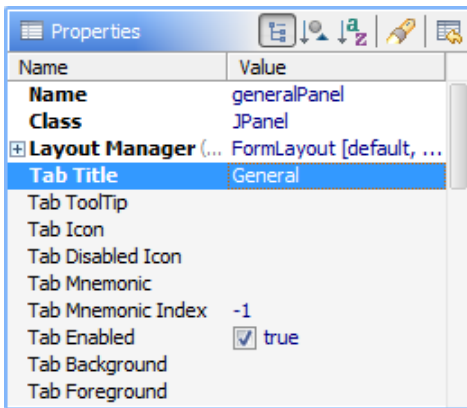
To change the tab order, select a page component (e.g. JPanel) and drag it over the tabs to a new place. You can also drag and drop page components in the Structure view to change its order.
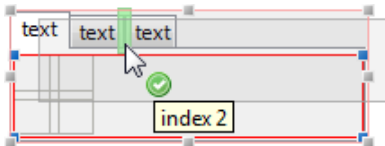
Use an empty border to separate the page contents from the JTabbedPane border. If you are using JGoodies Forms, it's recommended to use `TABBED_DIALOG_BORDER`. Otherwise use an `EmptyBorder`.
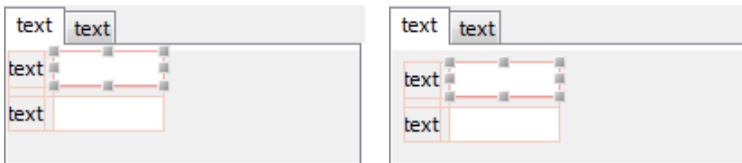
2.3.7 Events

Components can provide events to signal when activity occurs (e.g. button pressed or mouse moved). JFormDesigner shows events in the **Events** category in the Properties view.
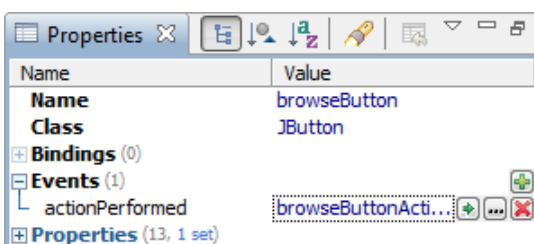
IDE plug-ins: Click on the **Go to Method** button (⬆) to go to the event handler method in the Java editor of the IDE.

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. Or click the **Add Event** button (🔛) in the Properties view. 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.
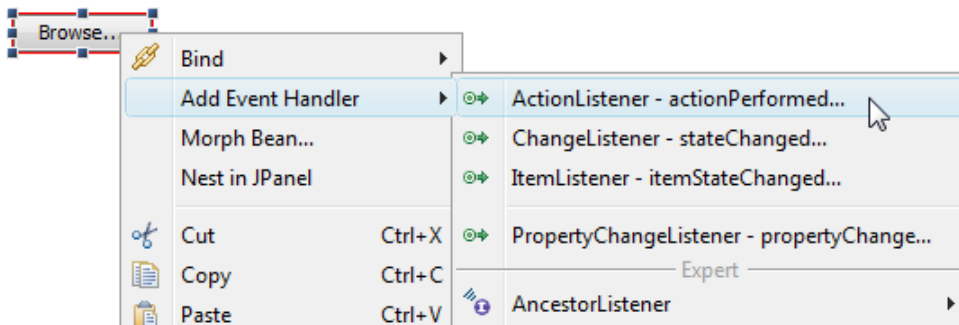
The 🔵 icon in the popup menu indicates that the listener interface will be implemented (e.g. javax.swing.ChangeListener). The 🟢 icon indicates that the listener adapter class will be used (e.g. java.awt.event.FocusAdapter for java.awt.event.FocusListener). The icons 🔵 and 🟢 are used when the listener 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.
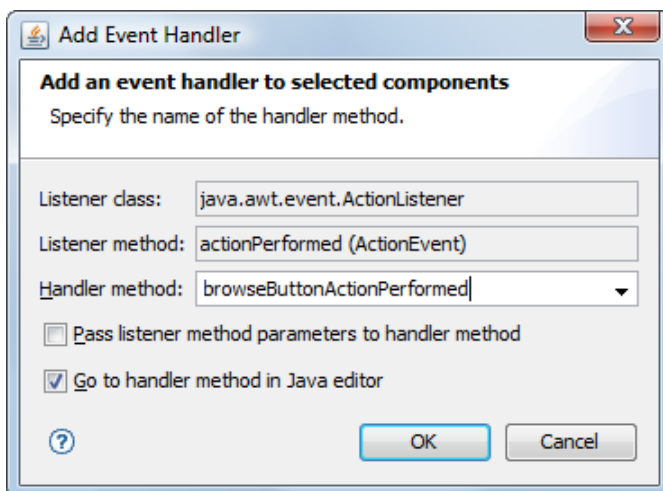
If you add a `PropertyChangeListener`, you can also specify a property name (field is not visible in screenshot). Then the listener is added using the method `addPropertyChangeListener(String propertyName, PropertyChangeListener listener)`.

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, click the **Remove Event** button (❌). Or right-click on the event and select **Remove Event** from the popup menu.

Change Handler Method Name

You can either edit the method name directly in the property table or click the ellipsis button (⋯) to open the **Edit Event Handler** dialog where you can edit all event options.

2.4   Palette

The component palette provides quick access to commonly used components (JavaBeans) available for adding to forms.
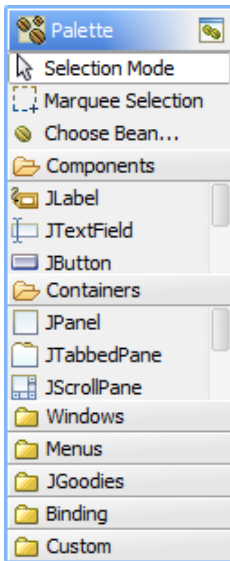
The components are organized in categories. Click on a category header to expand or collapse a category.

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, hold down the **Ctrl** key (*Mac*: **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.

Palette Manager

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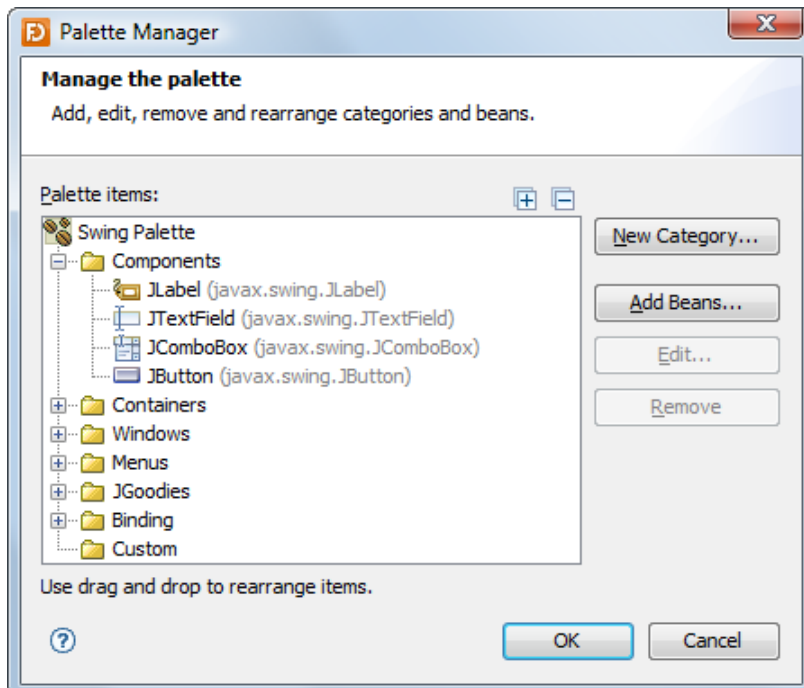
Choose Bean

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.

Following pattern kinds are supported:

- Wildcards: *"*"* for any string; *"?"* for any character; terminating *"<"* or *" "* (space) prevents implicit trailing *"*"*
- Camel case: *"JB"* for classes containing *"J"* and *"B"* as upper-case letters in camel-case notation, e.g. `JButton` or `JideButton`; *"DaPi"* for classes containing *"Da"* and *"Pi"* as parts in camel-case notation, e.g. `DatePicker`

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

- History matches: 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.
- Project matches: classes found in the Classpath specified in the current Project.
- Palette matches: classes found in the palette.
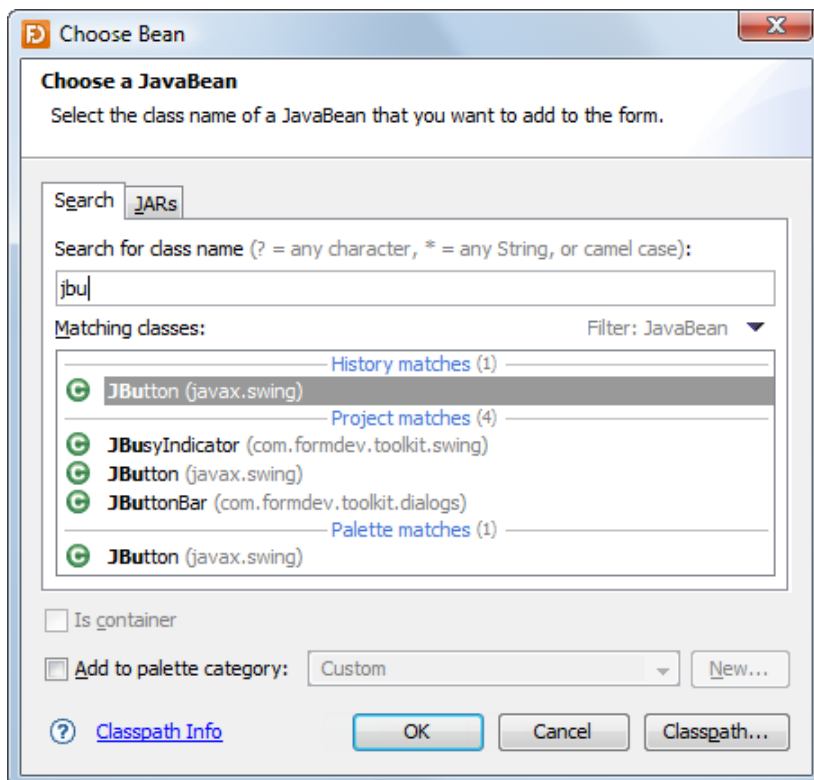
Filter Menu Options	
Use Filter	Classes are hidden if they do not match the filter. E.g. if the JavaBean filter is active and the class is not public or does not have a public constructor.
Show Interfaces	Includes interfaces in the list of matching classes.

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.
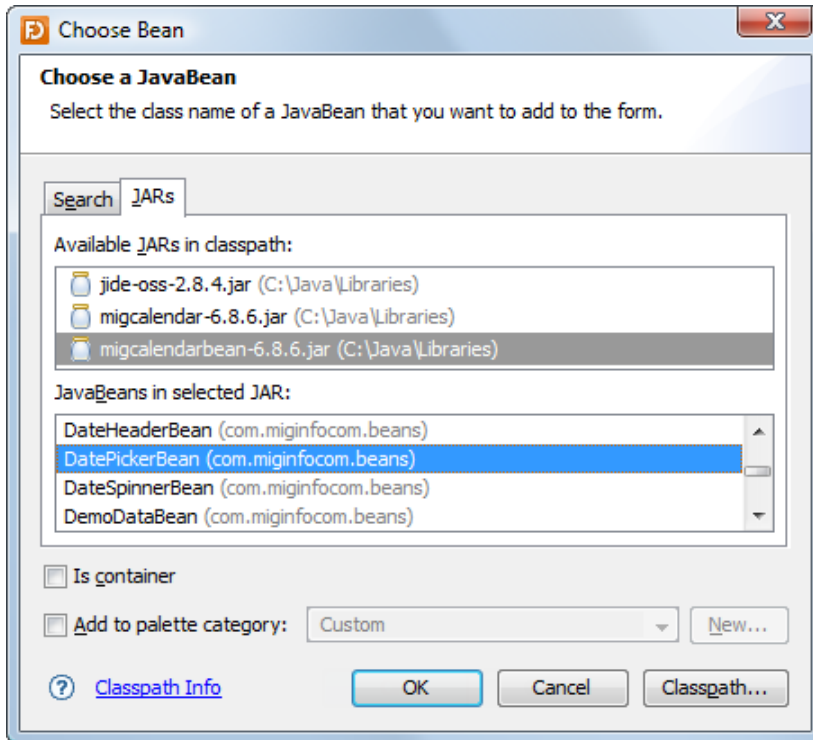
See also http://docs.oracle.com/javase/7/docs/technotes/guides/jar/jar.html#Per-Entry_Attributes

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.

2.5 Structure View

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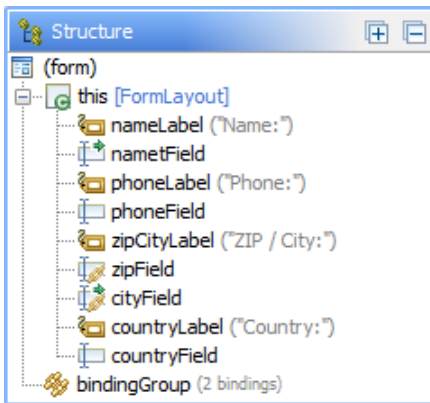
Each component is shown in the tree with an icon, its name and additional information like layout manager class or the text of a label or button. The name must be unique within the form and is used as variable name in the generated Java code.

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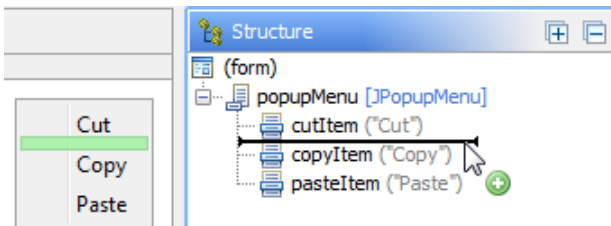
Various overlay icons are used in the tree to indicate additional information:

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 bindings assigned to it. The bindings are shown in Bindings view and in the **Bindings** category in the Properties view.
✦	The component has events assigned to it. The events are shown in the **Events** category in the Properties view.
J	The component has custom code assigned to it. See Code Generation properties.
⊙	The variable modifier of the component is set to `public`. See Code Generation properties.
△	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`.
✖	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.

2.6 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 available in all selected components are shown.

The properties table displays the component name, component class, layout manager and constraints properties, bindings, events, client properties, component properties and code generation properties. The list of component properties comes from introspection of the component class (JavaBeans).
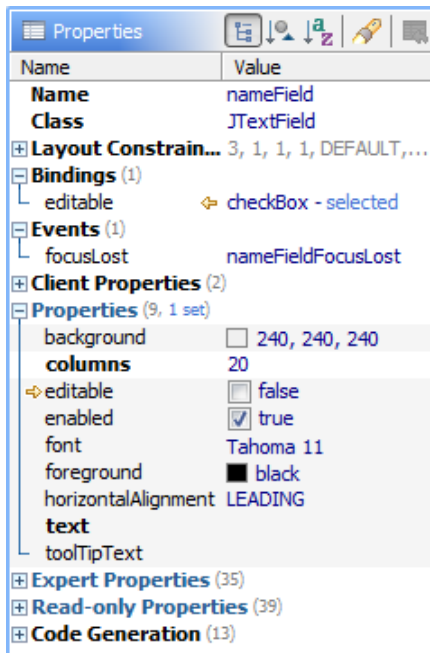
Properties are organized in categories, which you can expand/collapse by clicking on the category name or on the small plus/minus icons. The number of properties in a category and the number of set properties is displayed near the category name.

The category names of component property categories (Properties, Expert Properties, etc) are displayed in blue color.

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.

A small arrow (⮕) near the property name indicates that the property is bound.

Use **Group by Category** ( ) to organize component properties into three predefined categories (normal, expert and read-only) and custom categories (defined in BeanInfo). **Group by Defining Type** ( ) organizes component properties into defining types (e.g. JTextField, JTextComponent, JComponent, Container, Component). **Alphabetical** ( ) shows all component properties in one category.

Changing property values

The left column displays the property names, the right column the property values. Click on a property value to edit it.
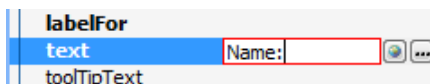
You can either edit a value directly in the property table or use a custom property editor by clicking on the ellipsis button (⬛) on the right side. The custom editor pops up in a new dialog. The globe button (⬛), which is only available for localized forms and string properties, allows you to choose 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.
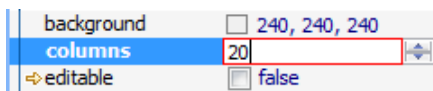
Search for property names

To filter the list of shown properties, select the **Show Filter** (🖊) toolbar button. This shows a text field below the toolbar, where you can enter your filter criteria.
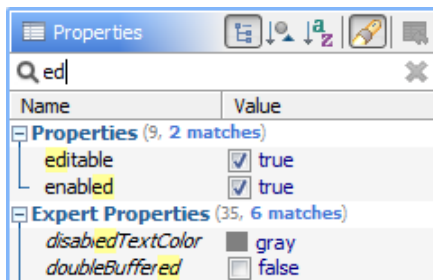
Common properties and categories

Property/Category	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 in the Code Generation category.
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 Manager	Layout manager 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.
Layout 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.
Bindings	Bindings of the component.
Events	Events of the component.
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	Code Generation properties of the component.

"(form)" properties

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

Property Name	Description
Form file format	The format used to persist the form. See also "Form file format" option in General preferences.
Set Component	If true, invokes java.awt.Component.setName() on all components of the form.

Property Name	Description
Names	

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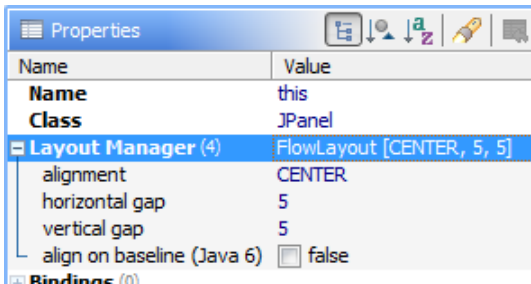
This screenshot shows layout manager properties (alignment, horizontal and vertical gap) of a container that has a FlowLayout.

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.
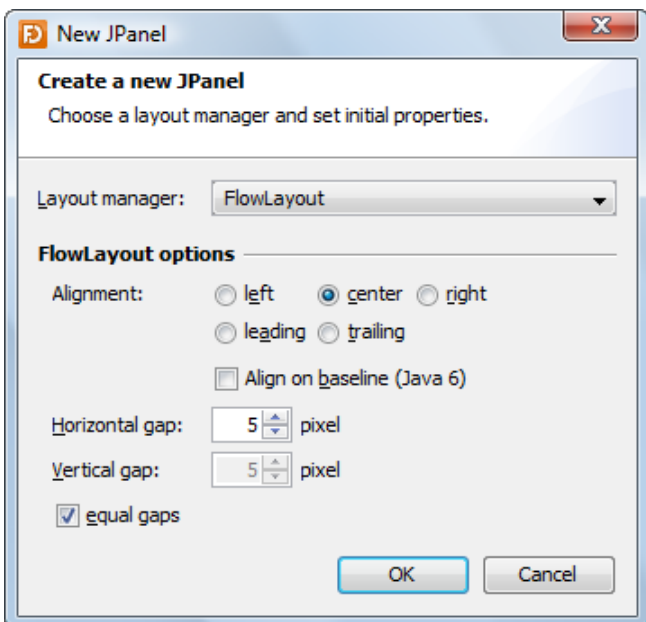
2.6.2   Layout Constraints Properties

Layout Constraints properties are related to layout managers. Some layout managers (FormLayout, TableLayout, GridBagLayout, ...) use constraints to associate layout information (e.g. grid x/y) 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.
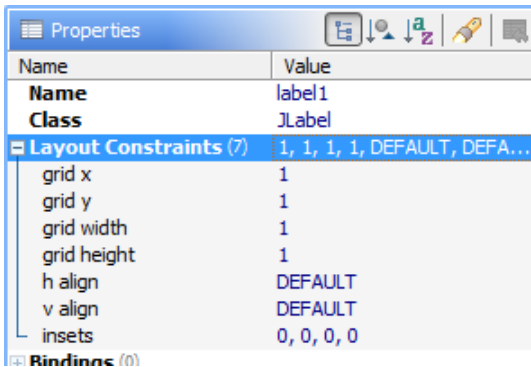
This screenshot shows constraints properties of a component in a FormLayout.

2.6.3   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.
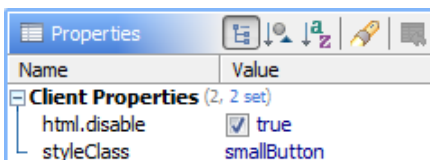
2.6.4   Code Generation Properties

This category contains properties related to the Java code generator.
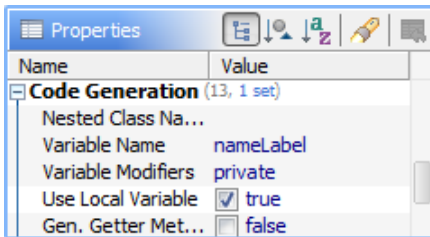
Component

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`.
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 Create	If `true`, create component in createUIComponents() method. Useful if you want use component factories for or non-default constructors. JFormDesigner generates the createUIComponents() method, but no component instantiation code. It is your responsibility to add code to createUIComponents().
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.

"(form)" properties

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

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`.

Property Name	Description
Default Gen. Getter Method	If `true`, generate public getter methods for components. Default is `false`.
Default Event 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.
Binding Initialization Method	If enabled, the code generator puts the code to create bindings into a method initComponentBindings().

2.6.5 Property Editors

Property editors are used in the Properties view to edit property values.
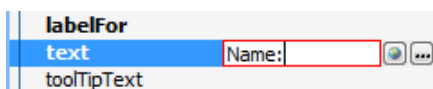
You can either edit a value directly in the property table or use a custom property editor by clicking on the ellipsis button (![...]) on the right side. The custom editor pops up in a new dialog.

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, String[], `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, `short`, `java.lang.Boolean`, `java.lang.Byte`, `java.lang.Character`, `java.lang.Class`, `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. Expand the **Read-only Properties** category in the Properties view to make it visible.
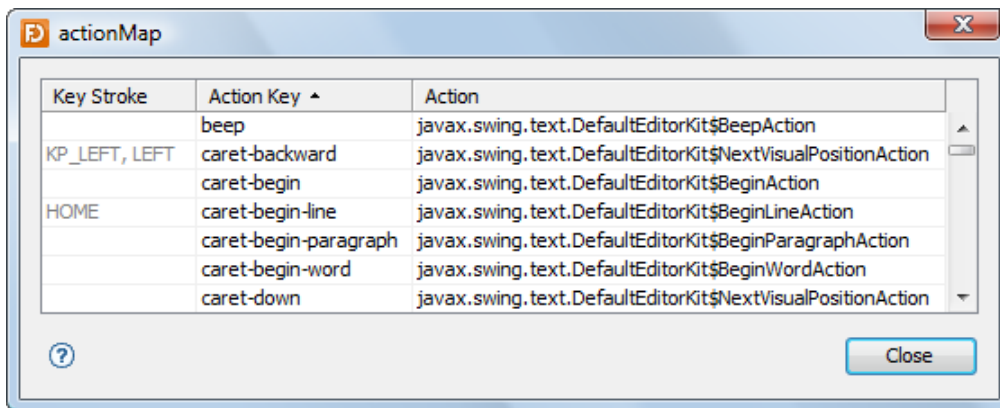
Border (javax.swing)

You can either select a border from the combo box in the properties table or use the custom editor.
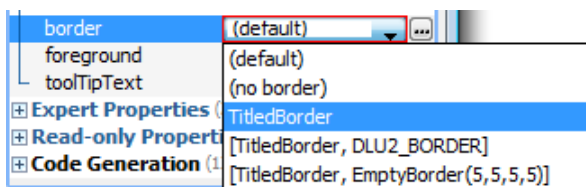
In the custom editor you can edit all border properties. Use the combo box at the top of the dialog to choose a border type. In the mid area of the dialog you can edit the border properties. This area is different for each border type. At the bottom, you can see a preview of the border.
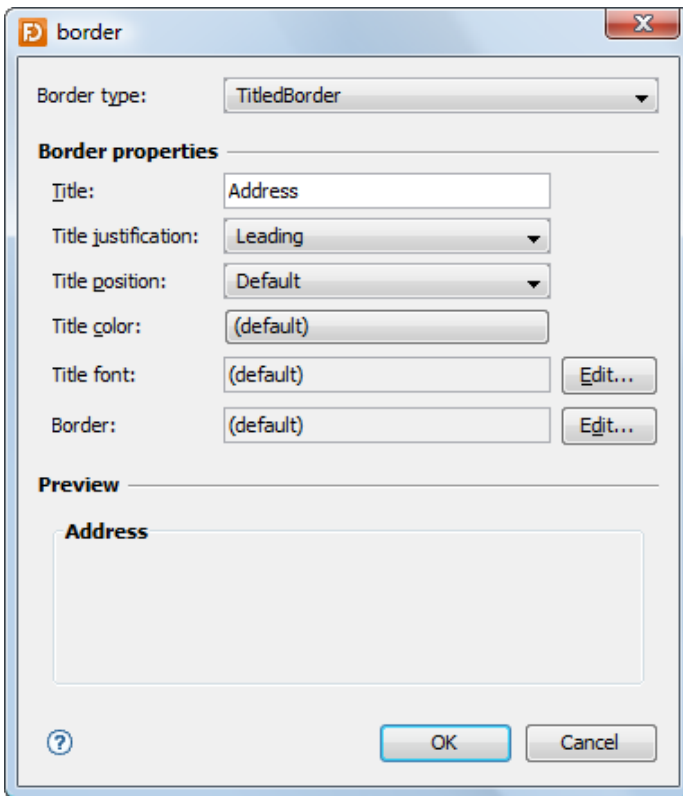
Following border types are supported:

- `BevelBorder`
- `CompoundBorder`
- `DropShadowBorder` (SwingX)
- `EmptyBorder`
- `EmptyBorder` (JGoodies)
- `EtchedBorder`
- `LineBorder`
- `MatteBorder`
- `SoftBevelBorder`
- `TitledBorder`
- Swing look and feel
- custom borders

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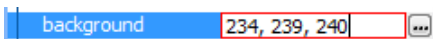
The custom editor supports various ways to specify a color. Besides RGB, you can select a color from the AWT, System or Swing palettes.
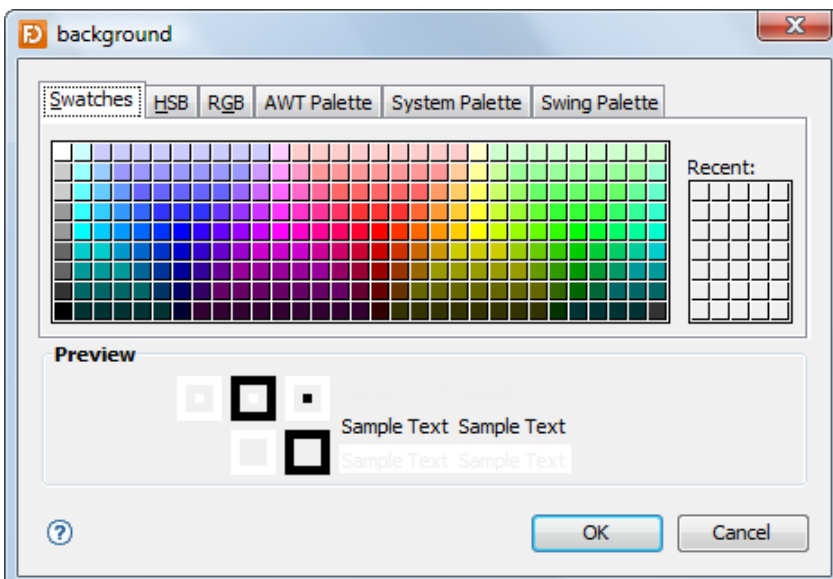
ComboBoxModel (javax.swing)

This custom editor allows you to specify string values for a combo box.
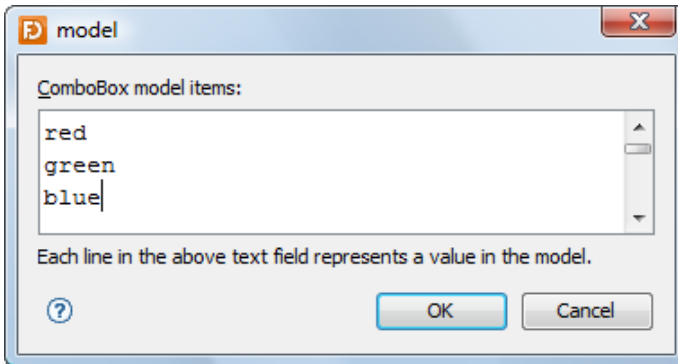
Cursor (java.awt)

This editor allows you to choose a predefined cursor.
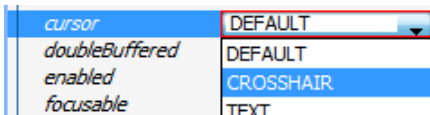
Dimension (java.awt)

Either edit the dimension in the property table or use the custom editor.
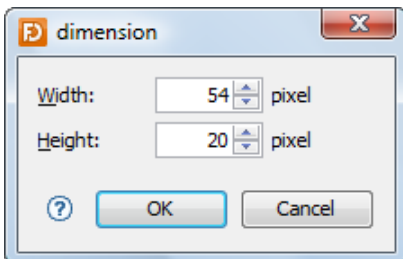
Font (java.awt)

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.
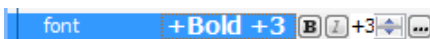
In the custom editor you can choose one of the tabs to specify either absolute fonts, derived fonts or predefined fonts.
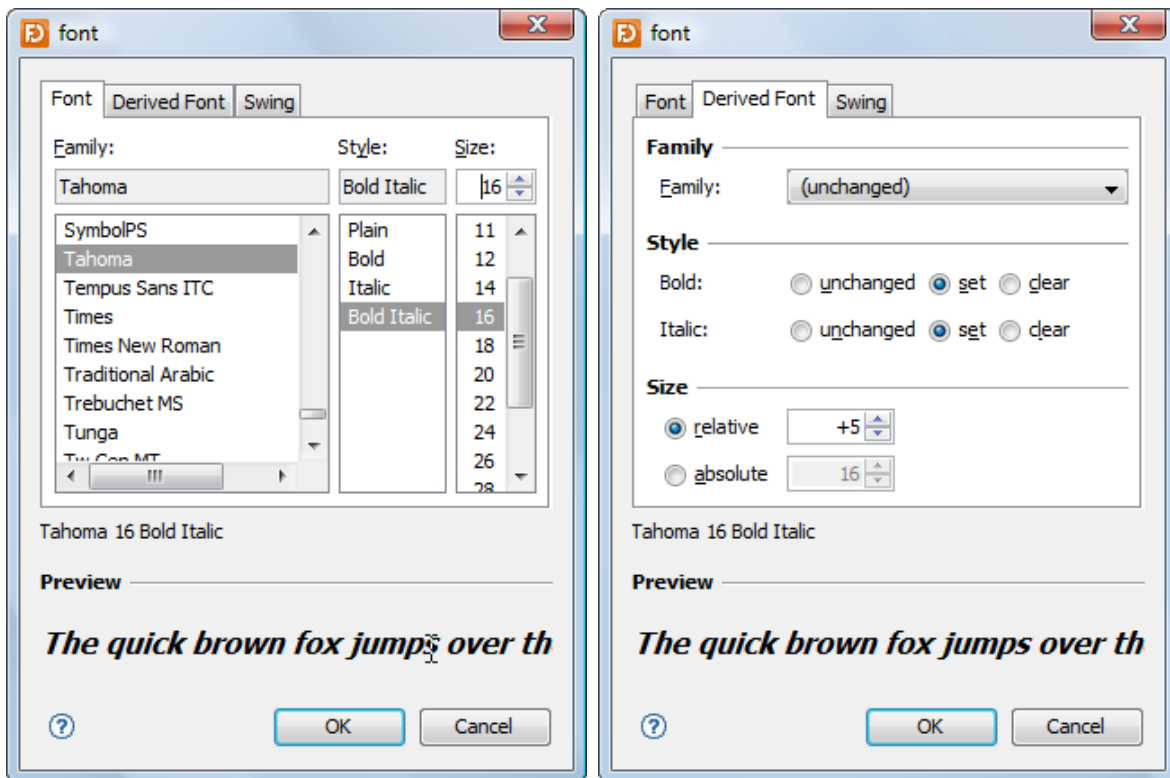
Icon (javax.swing) and Image (java.awt)

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.
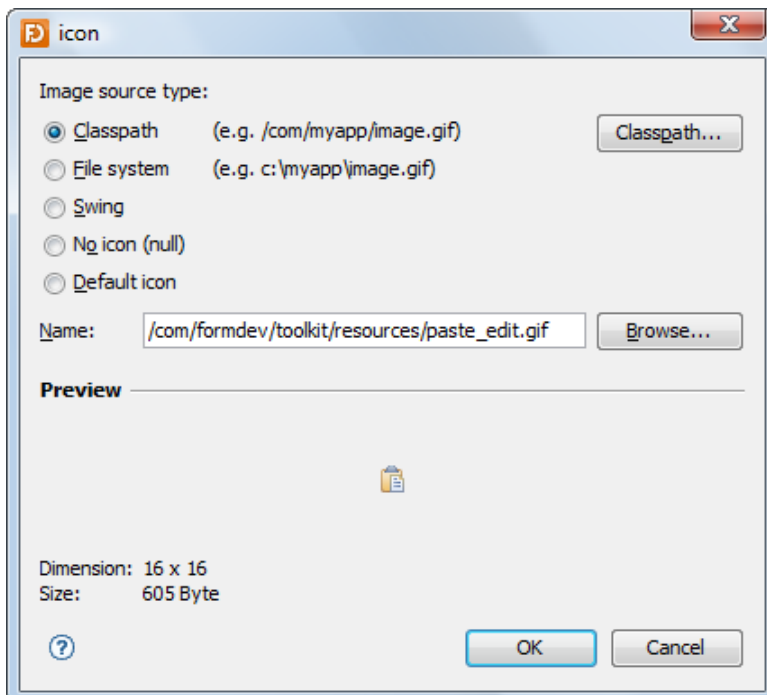
InputMap (javax.swing)

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. Expand the **Read-only Properties** category in the Properties view to make it visible.
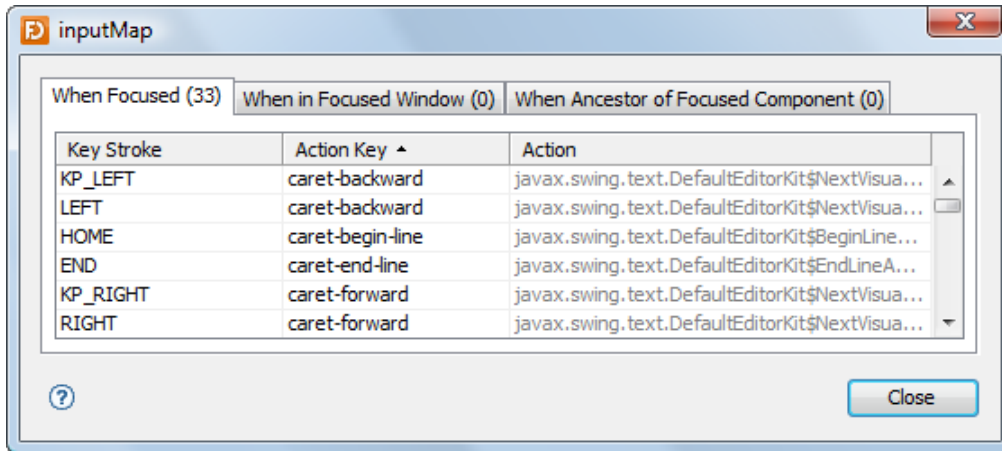
Insets (java.awt)

Either edit the insets in the property table or use the custom editor.
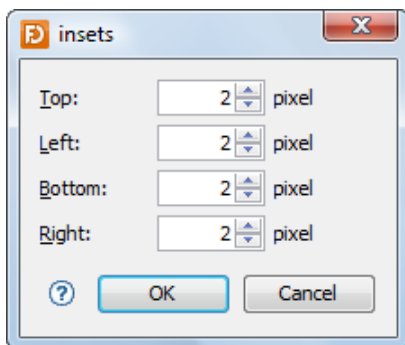
KeyStroke (javax.swing)

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 the Mac, **Ctrl** key otherwise).
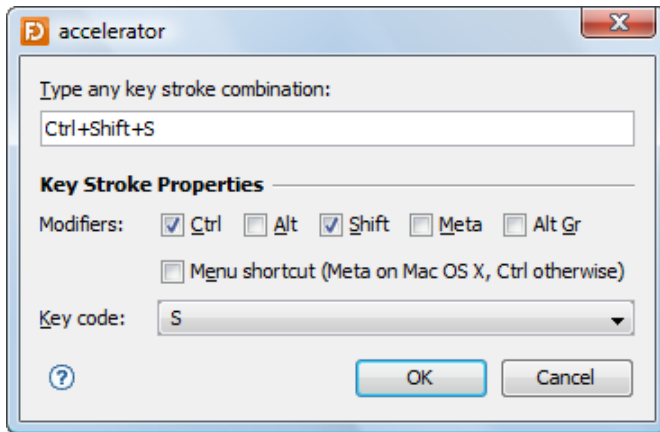
ListModel (javax.swing)

This custom editor allows you to specify string values for a list.
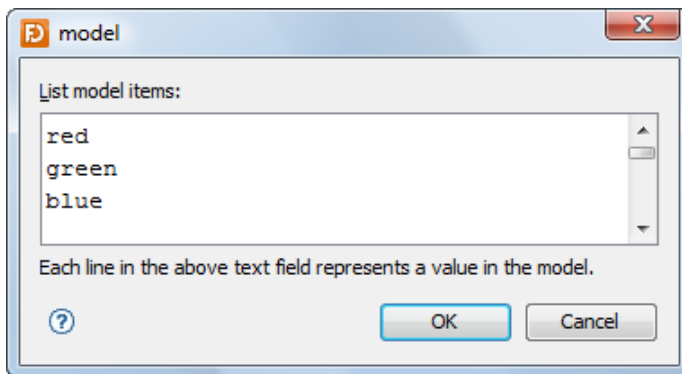
Object (java.lang)

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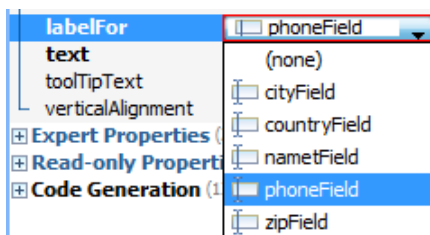
Paint (java.awt)

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.
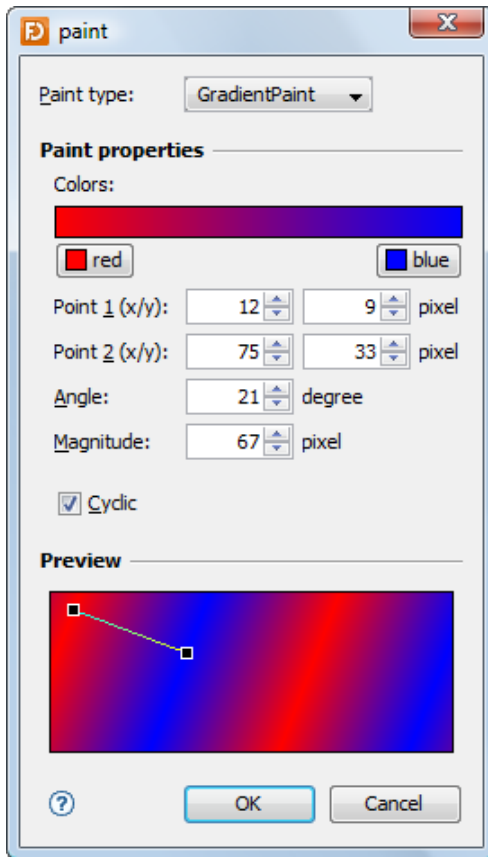
Following paint types are supported:

- Color
- GradientPaint

Point (java.awt)

Either edit the point in the property table or use the custom editor.
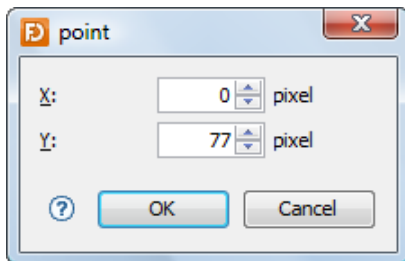
Rectangle (java.awt)

Either edit the rectangle in the property table or use the custom editor.
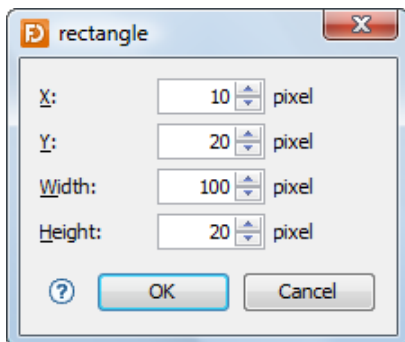
SpinnerModel (javax.swing)

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.
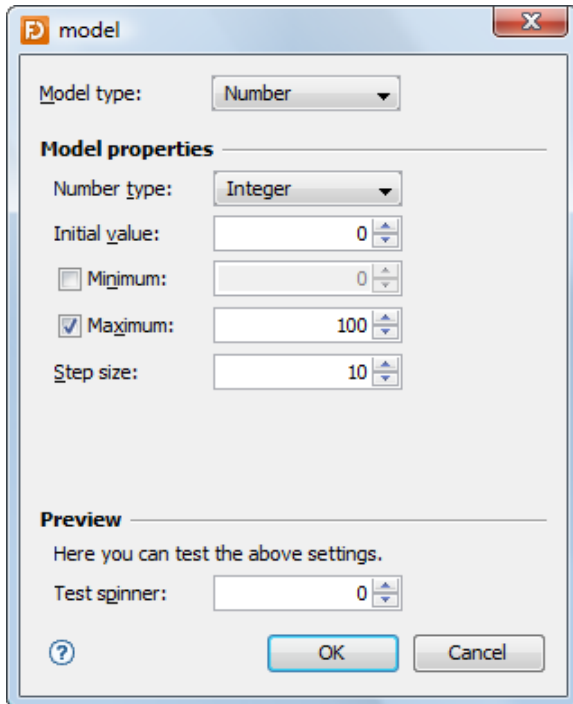
String (java.lang)

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.

- 36 -
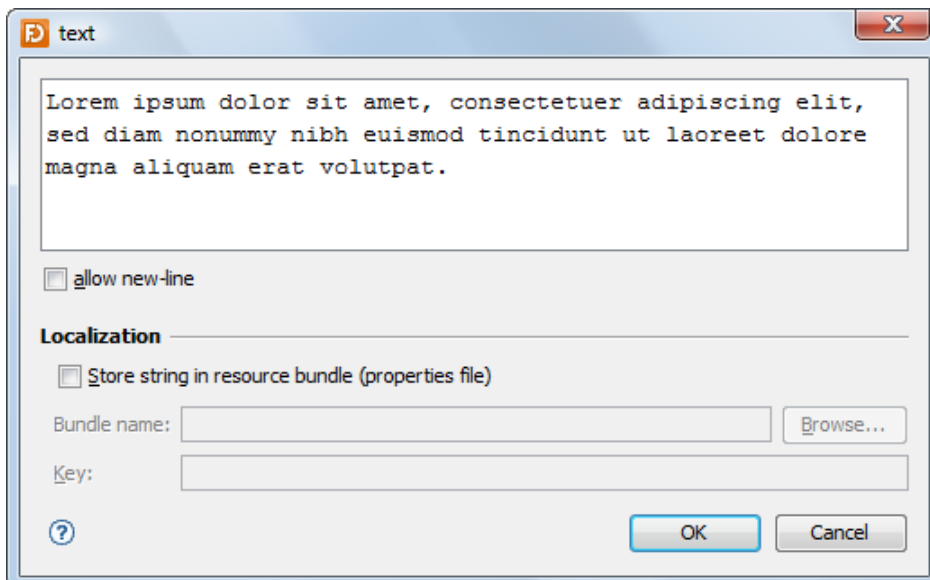
String[] (java.lang)

This custom editor allows you to specify string values for a string array.
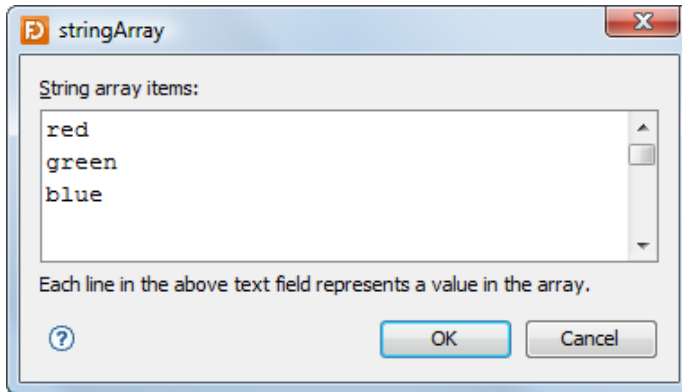
TableModel (javax.swing)

This custom editor allows you to specify values for a table.
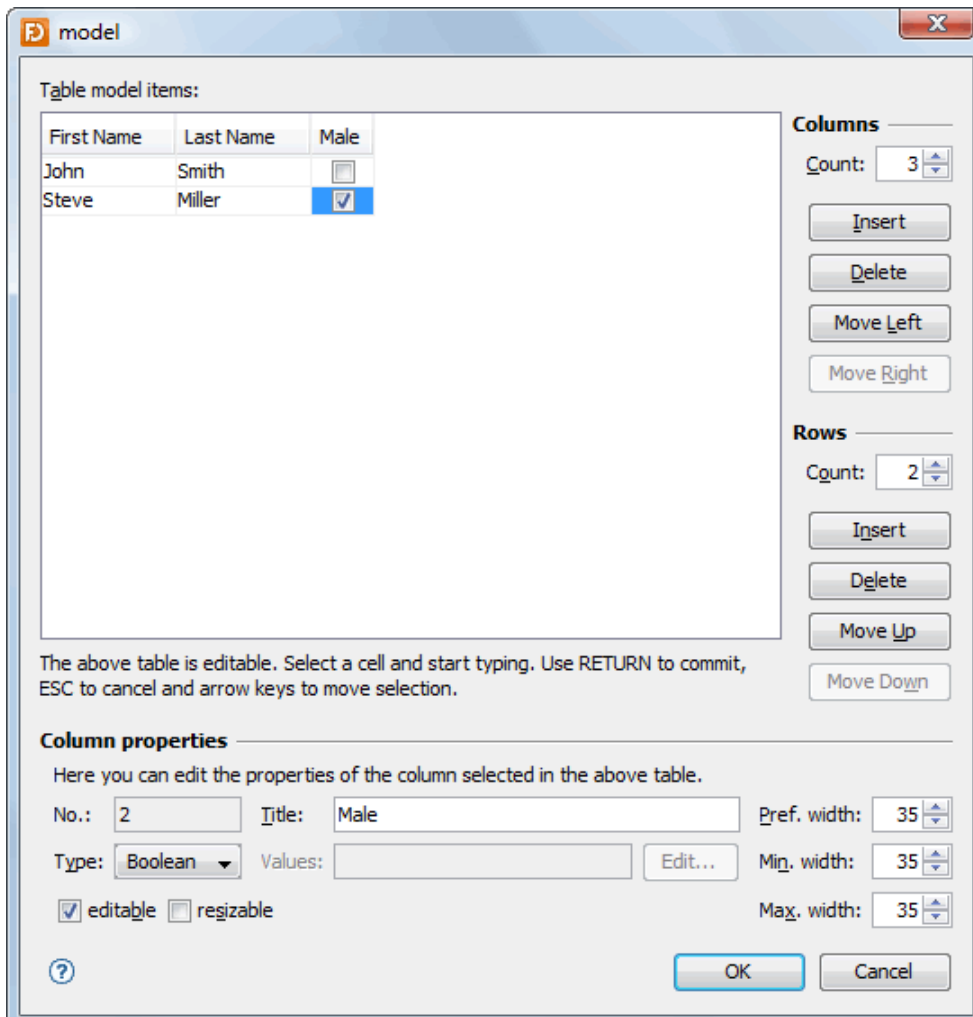
TreeModel (javax.swing)
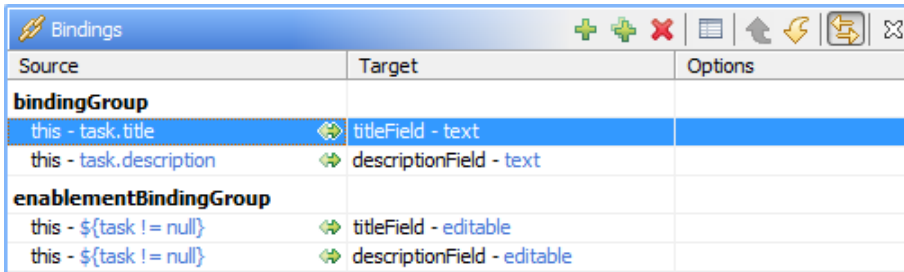
This custom editor allows you to specify string values for a tree.
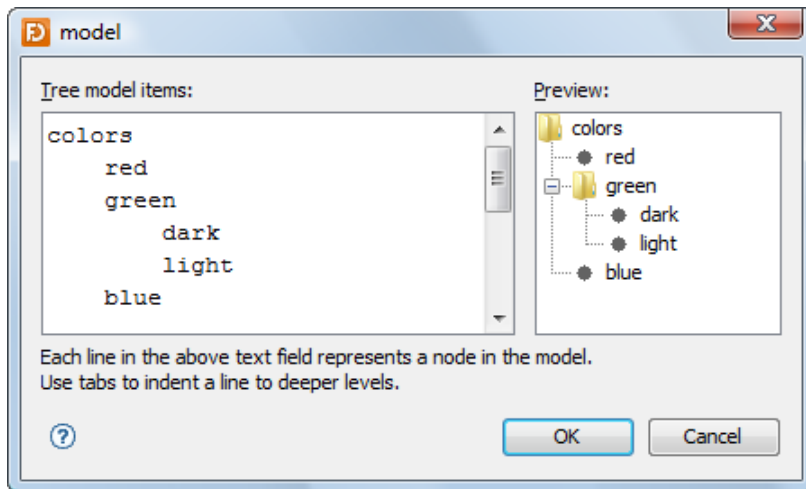
2.7   Bindings View

The Bindings view displays and lets you edit all bindings of the form. The bindings and binding groups are shown in the order they will be bound.

This view is not visible by default. It appears at the bottom of the main window when you click the **Show Bindings View** button (✐) in the toolbar.

The icon between the source and the target columns indicate the update strategy used by the binding:

- ⇨ Always sync (read-write)
- ➡ Only read from source (read-only)
- ₁➡ Read once from source (read-once)

	Toolbar and context menu commands	
➕	Add	Create a new binding.
➕	Add Group	Create a new binding group.
✖	Remove	Remove the selected bindings.
▦	Properties	Displays the properties of the selected binding in the Binding dialog.
↰	Move Up	Move the selected bindings up.
↳	Move Down	Move the selected bindings down.
⇆	Link with Designer	Links the bindings selection to the active designer.
✕	Close	Closes the Bindings view.

Double-click on a binding item to see its details in the Binding dialog.

2.8   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.
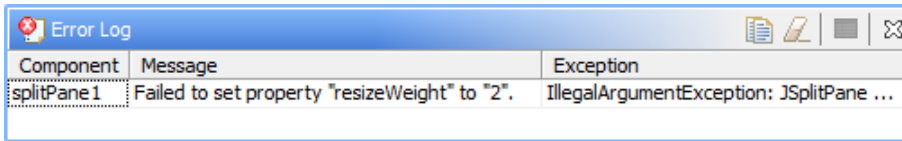
Toolbar commands

	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:
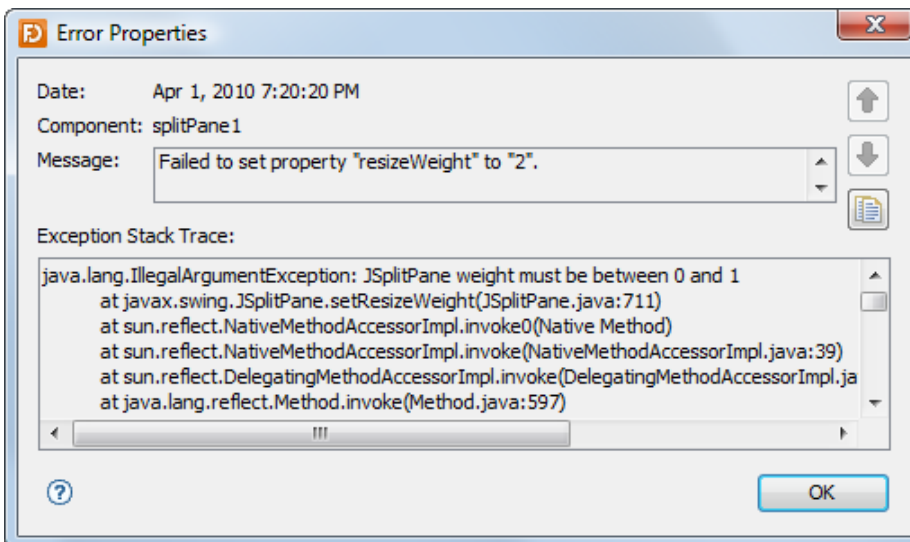
How to fix errors

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 Designer** button ( ) in the designer toolbar 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.

3 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.
- Use .properties or .xml files.
- Fully integrated in undo/redo.

The locales combo box ❶ 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 ❷, you change it in the current locale. A small globe ❸ in front of property values in the Properties view indicates that the string is localized (stored in a properties file).
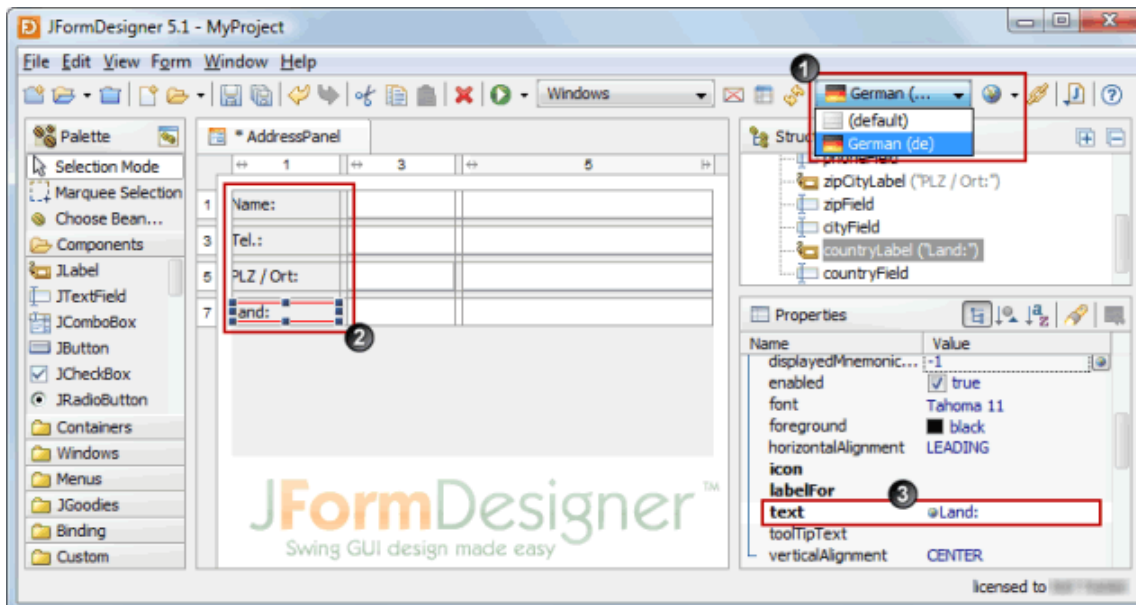
Create a new localized form

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.
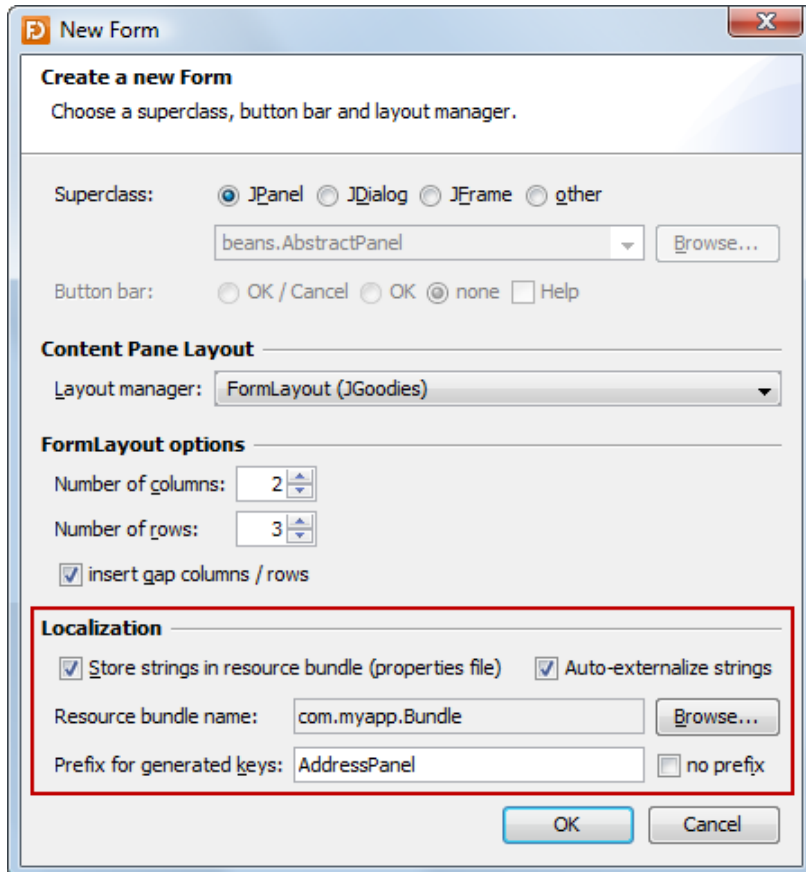
Edit localization settings and resource bundle strings

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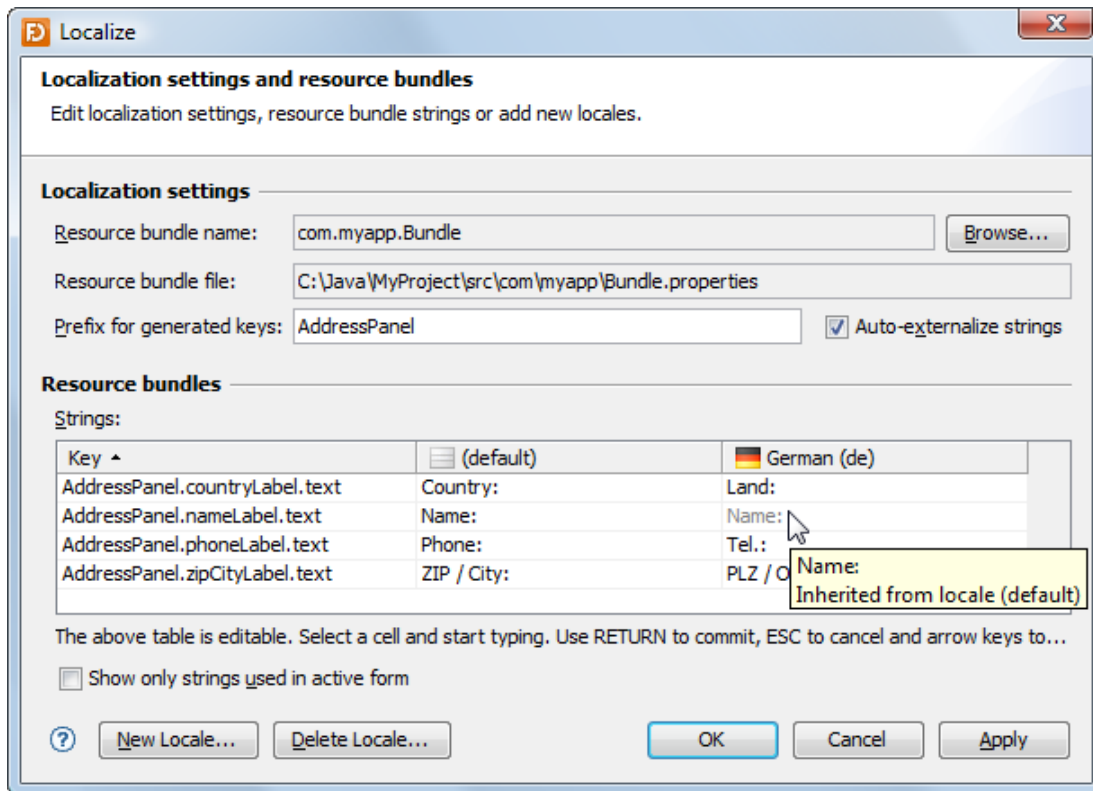
The **Resource bundle name** field is used to locate the properties files within the Source Folders of the current Project. Use the **Browse** button to choose a resource bundle (.properties file).

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** ( ) from 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** ( ) from the toobar or click the **Delete Locale** button in the **Localize** dialog. Select the locale to delete.
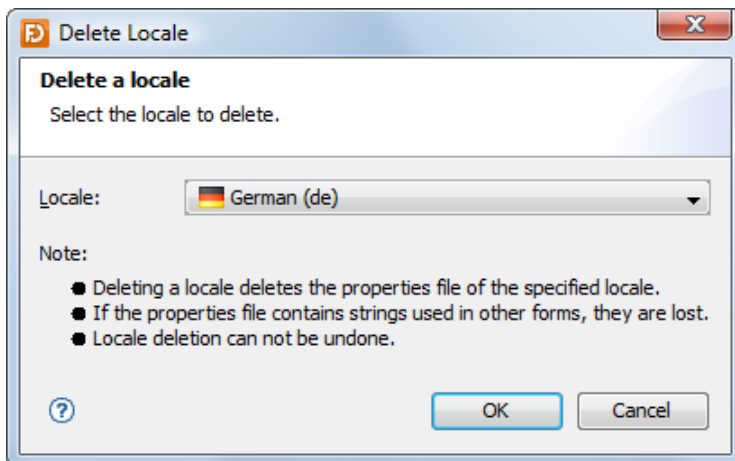
Externalize strings

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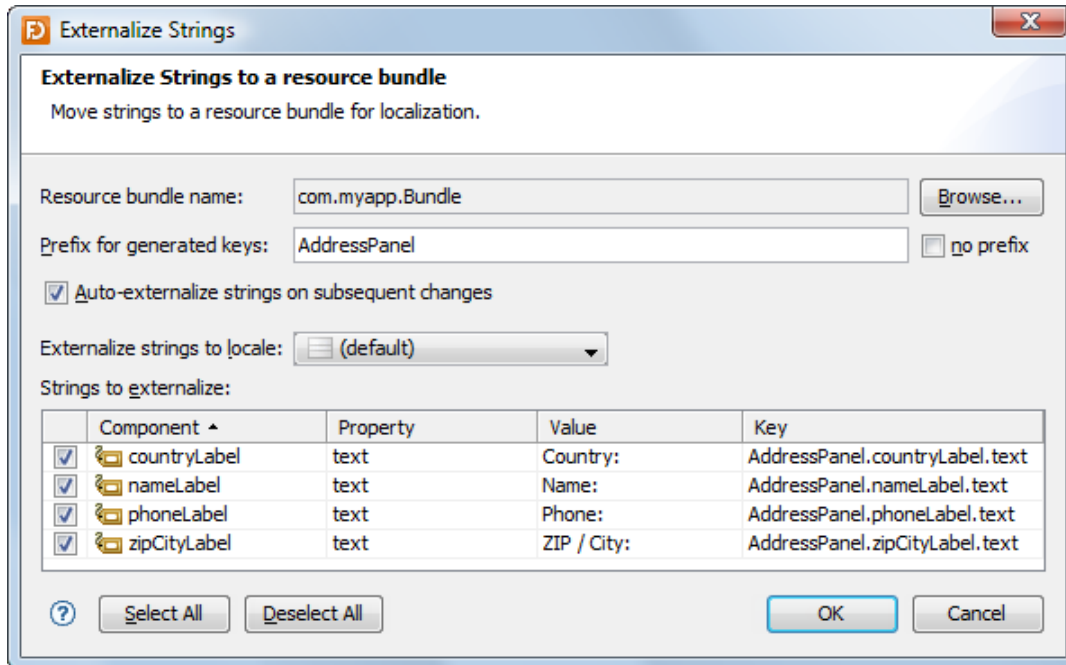
You can also externalize and internalize properties in the Properties view.
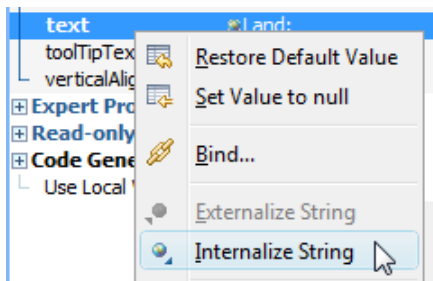
Internalize strings

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

Select **Form > Internalize Strings** from the main menu or **Internalize Strings** (🌐) from the toolbar, 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.
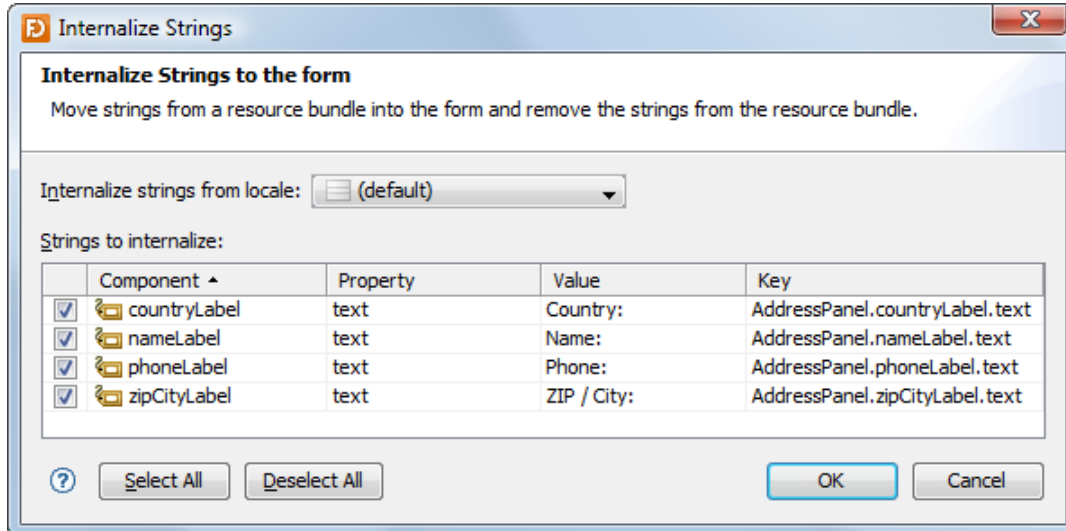
Choose existing strings

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.
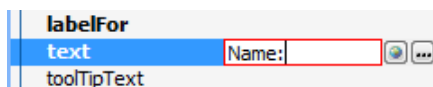
In the **Choose Key** dialog you can search for keys and/or values. Then select a key in the table and press OK to use its value in the form.
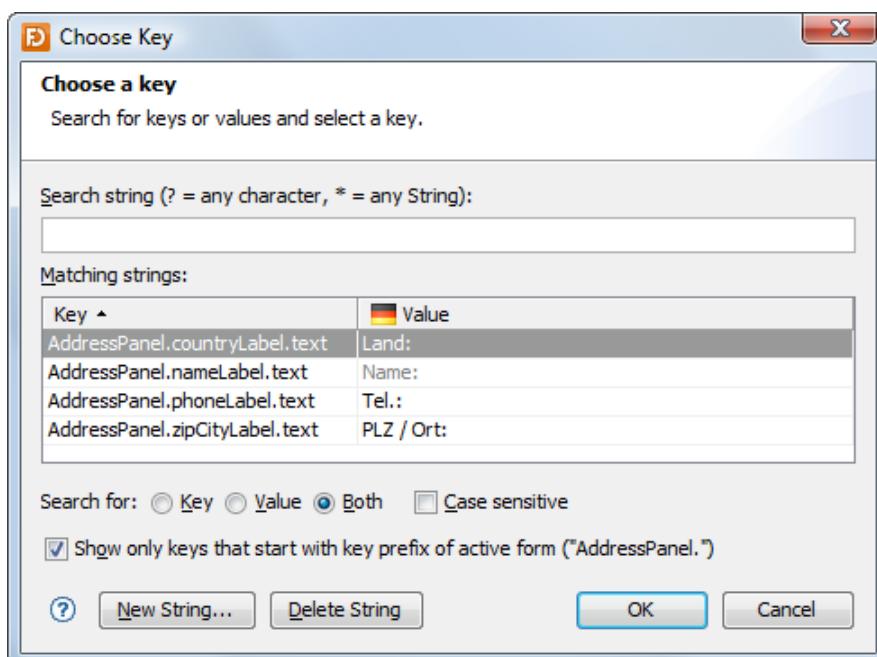
4   Beans Binding (JSR 295)

JFormDesigner supports the Beans Binding specification (JSR 295).

A binding syncs two properties: the source property with the target property. The source is usually a (non-visual) data model object and the target is usually an UI component (e.g. a JTextField). Initially the value of the source property is copied to the target property. Depending on the "Update strategy", a binding tracks changes on both properties and syncs the properties.
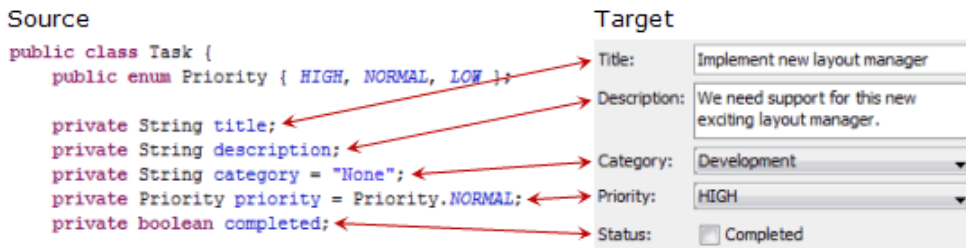
> Beans Binding is open source and **not** part of the standard Java distribution. You must ship an additional library with your application. JFormDesigner includes beansbinding.jar, beansbinding-doc.zip and beansbinding-src.zip in its redist folder. For more documentation and tutorials, visit beansbinding.java.net.
>
> The API documentation is also available here: doc.formdev.com/beansbinding/.

The Bindings view ❶ gives a good overview of all bindings in the form. The **Show Bindings View** button ❷ makes this view visible. The **Bindings** property category ❸ in the Properties view shows the bindings of the selected component and you can add (➕), edit (…) and remove (❌) bindings. Small arrows ❹ indicate that the property is bound. Binding groups are also shown in the Structure view ❺. The **Binding** palette category ❻ provides useful components.

Add/Edit Bindings

There are several ways to add/edit bindings:

- Right-click on a component in the Design or Structure view and select **Bind** from the popup menu. To edit an existing binding, select a bound property from the **Bind** submenu.
- Click the **Add/Edit Binding** button (![icon]/![icon]) in the **Bindings** property category in Properties view.
- Right-click on a component property in the Properties view and select **Bind** from the popup menu.
- Use the **Add**/**Properties** command in the Bindings view.

Remove Bindings

To remove existing bindings do one of:

- Click the **Remove Binding** button (![icon]) in the **Bindings** property category in Properties view.
- Use the **Remove** command in the Bindings view.

Binding Dialog

This dialog enables you to edit all options of one binding.

General tab
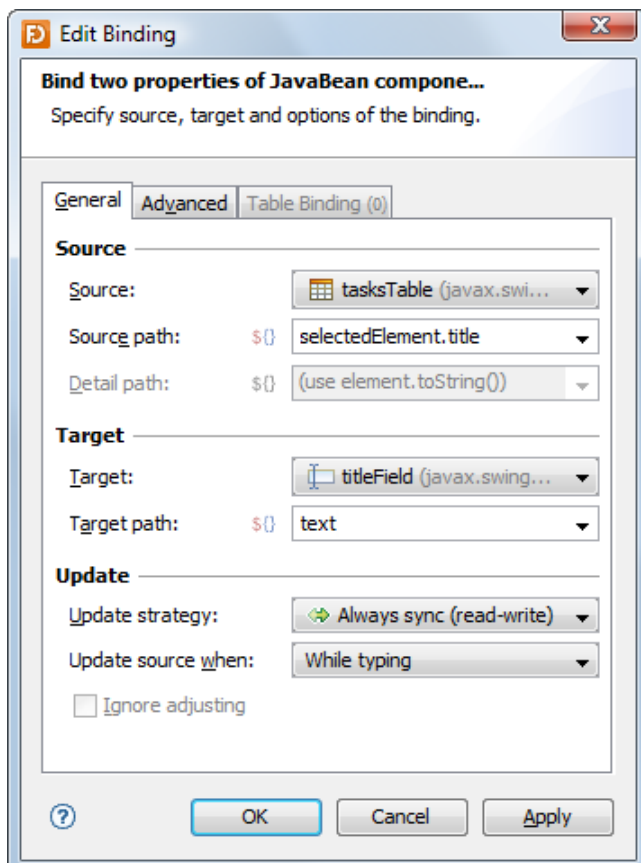
Field	Description
Source	The source object.
Source path	The path (or expression) that identifies the source property.
Detail path	The path (or expression) that determines what is displayed to the user in the target JList. (only if target is JList.elements)
Target	The target object.
Target path	The path (or expression) that identifies the target property.
Update strategy	Specifies how the properties are kept synchronized. Possible values: "Always sync (read-write)", "Only read from source (read-only)" and "Read once from source (read-once)".
Update source when	Specifies when the source is updated from the target. Possible values: "While typing", "On focus lost" and "On focus lost or Enter key pressed". (only if target is JTextComponent.text)
Ignore adjusting	If enabled, do not update properties until the user finished adjusting. E.g. while a slider is adjusting its value or while the list selection is being updated. (only if target is JSlider.value, JList.selectedElement(s) or JTable.selectedElement(s))

Advanced tab
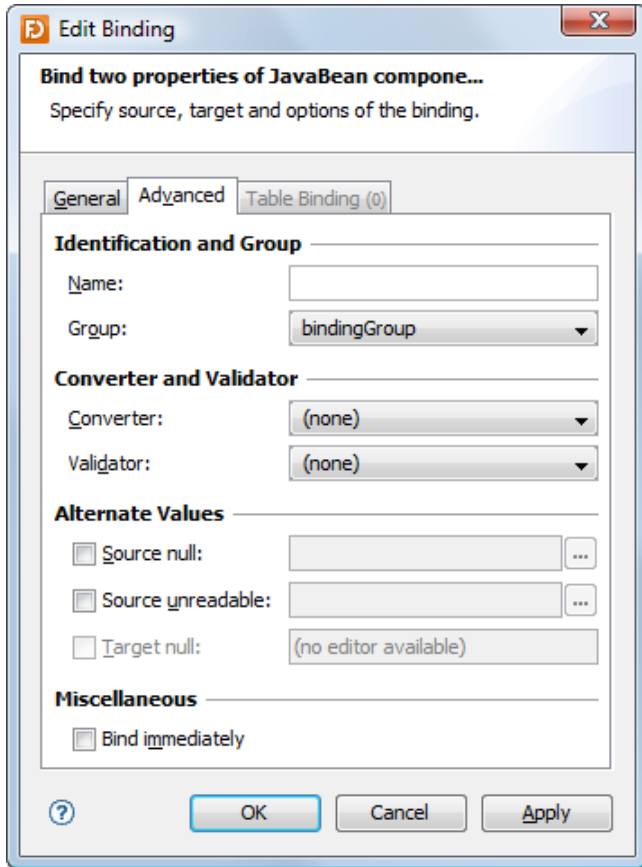
Field	Description
Name	The binding's name. Useful for BindingGroup.getBinding(name).
Group	The group this binding belongs to.
Converter	The Converter that converts the value from source type to target type and vice versa.
Validator	The Validator that validates the value before passing it from the target back to the source property.
Source null	Used if the value of the source property is null.
Source unreadable	Used if the source property is unreadable.
Target null	Used if the value of the target property is null.
Bind immediately	Bind this binding immediately after creation. Otherwise bind when the group is bound.

Table Binding tab

On this tab you can bind `List<E>` element properties to `JTable` columns. Each item in the source `List<E>` represents a row in the `JTable`. See JTableBinding for details about table binding.

This tab is enabled if source is an instance of `java.util.List<E>`, target an instance of `javax.swing.JTable` and target property is `elements`.
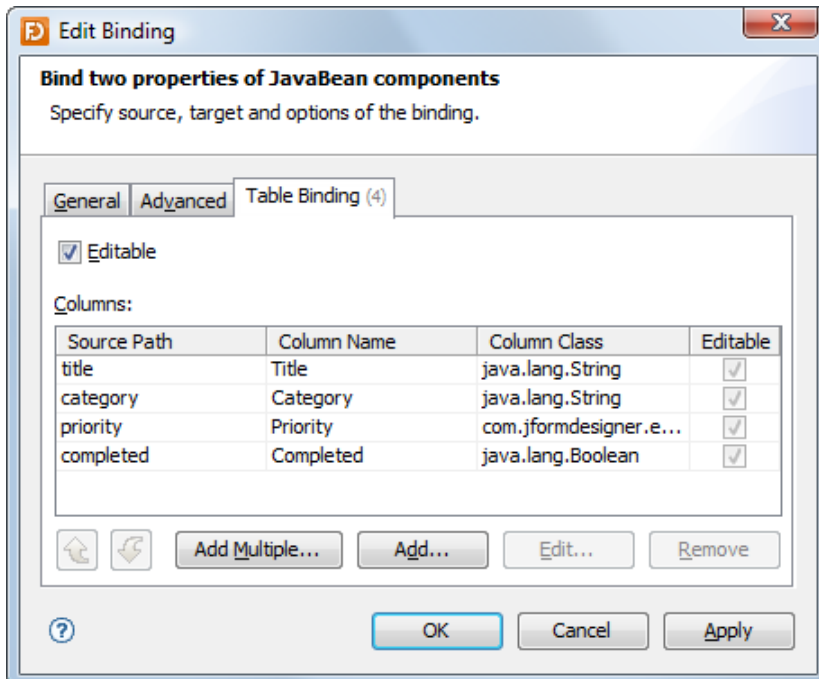
Field	Description
Editable	Specifies whether the table cells are editable or not.
Columns	The column bindings. The Source Path identifies the source property in `<E>`. The Column Name is shown in the JTable column header. Each column binding may have its own Converter, Validator and Alternative Values.

Path or Expression

To address source or target properties you can either use a path or an expression. Select the **Expression Language** button (`${}`) left to the input field to enter an expression.

A path (implemented by BeanProperty) uses a dot-separated path syntax. E.g. `task.title` addresses the `title` property of an object's `task` property. This is equivalent to `source.getTask().getTitle()`.

An expression (implemented by ELProperty) uses the Expression Language (EL) also known from JSP and JSF. Besides a dot-separated path syntax to address properties (e.g. "`${task.title}`") it also supports following operators:

- Arithmetic: `+`, `-`, `*`, `/` and `div`, `%` and `mod`
- Logical: `and`, `&&`, `or`, `||`, `not`, `!`
- Relational: `==`, `eq`, `!=`, `ne`, `<`, `lt`, `>`, `gt`, `<=`, `ge`, `>=`, `le`
- Empty: `empty`
- Conditional: `A ? B : C`

EL expression examples:

EL expression	Result
`${task.title}`	The `title` property of an object's `task` property.
`${firstName} ${lastName}`	Concatenation of `firstName` and `lastName` properties.
`${mother.age > 65}`	`true` if mother is older than 65, `false` otherwise.
`${image.width * image.height}`	Computes the number of pixels of an image.
`${image.width * image.height * 4}`	Computes the number of bytes of an 32 bit image.

Following words are reserved for the EL and should not be used as identifiers:

```
and    or    not   div   mod
eq     ne    lt    gt    ge     le
true   false null  empty instanceof
```

Data model

The data model used by Beans Binding (JSR 295) is based on the JavaBeans specification. Getters are necessary to read property values and setters to modify property values. On modifications, property change events should be fired so that beans binding can update the UI components. E.g.:

```java
public class Task {
    private String title;

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        String oldTitle = this.title;
        this.title = title;
        changeSupport.firePropertyChange("title", oldTitle, title);
    }

    private final PropertyChangeSupport changeSupport = new PropertyChangeSupport(this);

    public void addPropertyChangeListener(PropertyChangeListener listener) {
        changeSupport.addPropertyChangeListener(listener);
    }

    public void removePropertyChangeListener(PropertyChangeListener listener) {
        changeSupport.removePropertyChangeListener(listener);
    }
}
```

Data model access

The source and target combo boxes in the Binding dialog offer only the components added to the form. To bind your data model to form components, you could add an instance of your data object to the form (using Choose Bean), but this requires that the data object is a JavaBean with public null constructor, which is not always possible.

The preferred way to access the data model for binding is to add a getter for the data model to the form class. E.g.:

```java
public class TaskViewForm extends JPanel {
    private Task task;

    public Task getTask() {
        return task;
    }
}
```

After compiling the form class, you can use `this` as binding source and `task.someProperty` as binding source path.

Add a setter to the form class, if the whole data model may change. E.g.:

```java
public class TaskViewForm extends JPanel {
    public void setTask(Task task) {
        Task oldTask = this.task;
        this.task = task;
        firePropertyChange("task", oldTask, task);
    }
}
```

How to bind data to a JTable

Beans Binding requires that the data is in a `java.util.List` (or ObservableList). The type of each data row should be specified as type parameter to the list. E.g. `java.util.List<MyData>`. The data class should have getters and setters for its values, which can bound to table columns.

Steps to bind a table:

1. Add a `java.util.List` component from the **Bindings** palette category to the form. JFormDesigner creates a variable for the list in the Java code, but does not assign a value to it. Its up to you, to assign data to the list before invoking `initComponents()`.
2. Set the **Type Parameters** property (expand the **Class** property in Properties view) of the `List` to your data class (e.g. `MyData`). Make sure that the data class is compiled and in the classpath of the project.
3. Add a `JTable` to the form.
4. Right-click on the table and select **Bind > elements** from the popup menu, which opens the Binding dialog.
5. On the **General** tab, set the source to your `List` object and leave the source path empty.
6. Switch to the **Table Bindings** tab.
7. Click the **Add Multiple** button and add columns.

Examples

For examples that use Beans Binding, take a look at the package `com.jformdesigner.examples.beansbinding` in the examples.

5   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

Project specific preference pages:

- FormLayout (JGoodies)
- null Layout
- Localization
- Java Code Generator
  - Templates
  - Layout Managers
  - Localization
  - Binding
  - Code Style
- Client Properties

General

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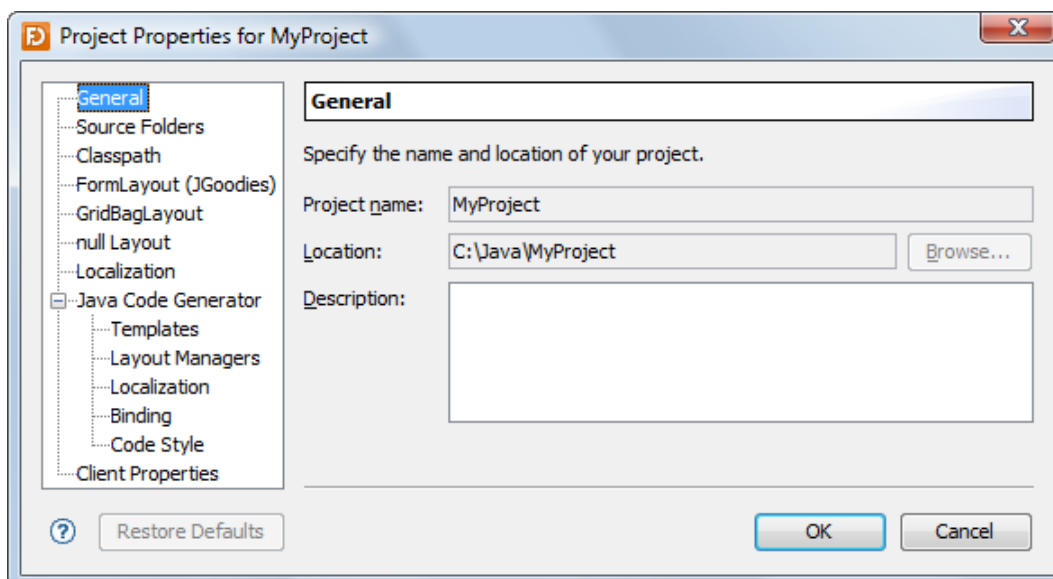
Source Folders

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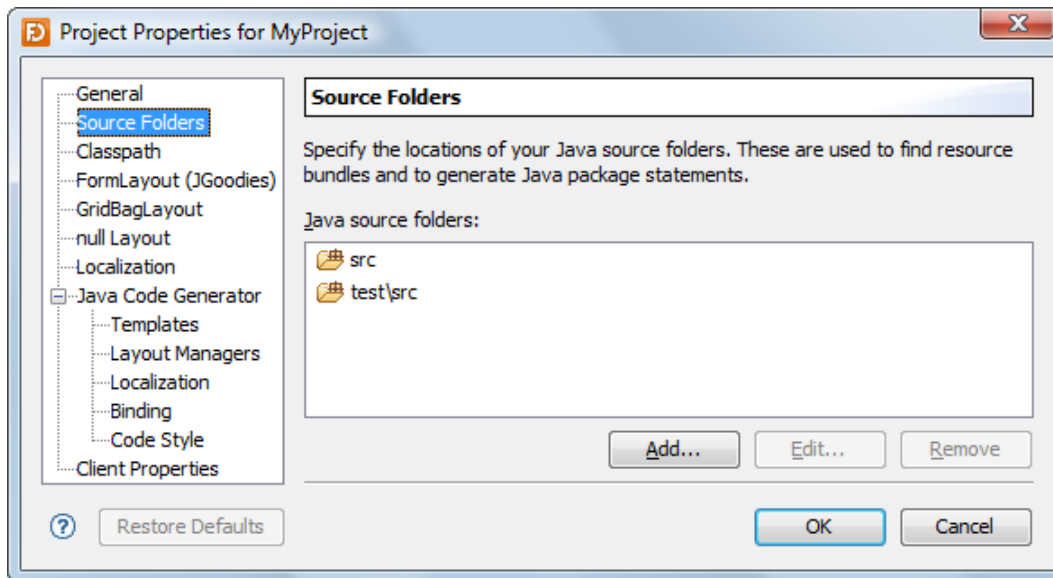
If the folders list is focused, you can use the **Insert** key to add folders or the **Delete** key to delete selected folders.

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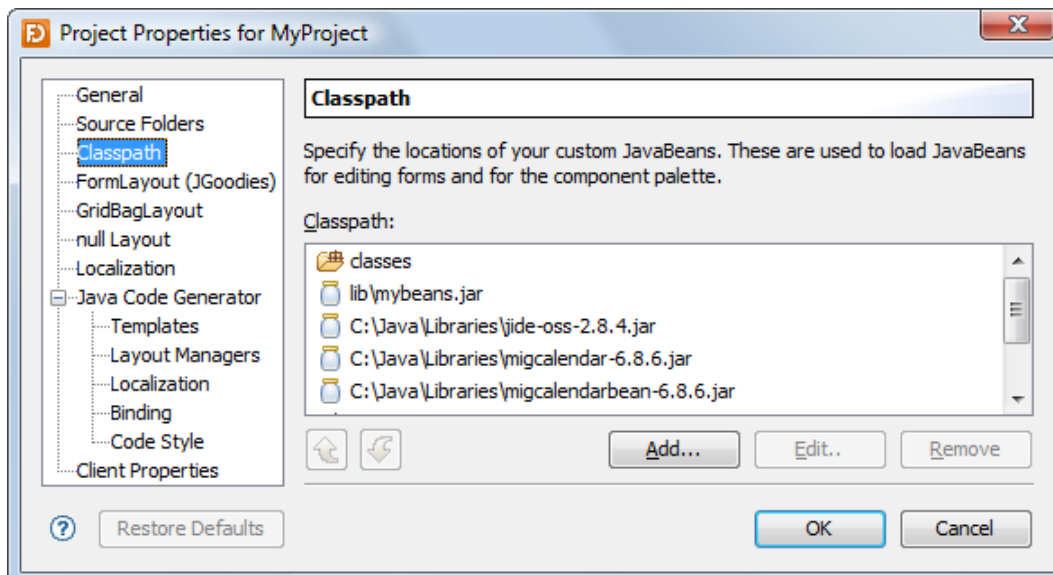
If the classpath list is focused, you can use the **Insert** key to add folders/JAR files, the **Delete** key to delete selected folders/JAR files, **Ctrl+Up** keys to move selected items up or **Ctrl+Down** keys to move selected items down.

6 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.
- **NetBeans plug-in**: NetBeans uses the term "Options" instead of "Preferences". The JFormDesigner preferences are fully integrated into the NetBeans options dialog. Select **Tools > Options** from the menu to open it and then select the "JFormDesigner" page.
- **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 item named "JFormDesigner" in the "IDE Settings" area.
- **JBuilder plug-in**: The JFormDesigner preferences are fully integrated into JBuilder preferences dialog. Select **Tools > Preferences** from the menu to open it.

Pages

- General
- FormLayout (JGoodies)
- GridBagLayout
- null Layout
- Localization
- Look and Feels
- Java Code Generator
  - Templates
  - Layout Managers
  - Localization
  - Binding
  - Code Style (**Stand-alone** only)
- Client Properties
- Native Library Paths
- BeanInfo Search Paths
- Squint Test
- Check for Updates

Import and export preferences

In the Preferences dialog, 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. You can also use IDE specific commands:

- **Eclipse plug-in**: You can use the menu commands **File > Import** and **File > Export** to import and export preferences to/from Eclipse preferences files.
- **NetBeans plug-in**: You can use the **Import** and **Export** buttons in the Options dialog to import and export options to/from NetBeans options 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
Form file format	The format used to persist the form. Since version 5.1, JFormDesigner supports the compact, easy-to-merge and fast-to-load persistence format JFDML. To change the persistence format of an existing form, open the form, select the "(form)" node in the Structure view and change the "Form file format" property in the Properties view. Or use the JFormDesigner command-line tool to convert the format of many forms.	JFDML

FormLayout (JGoodies)

On this page, you can specify FormLayout related options.
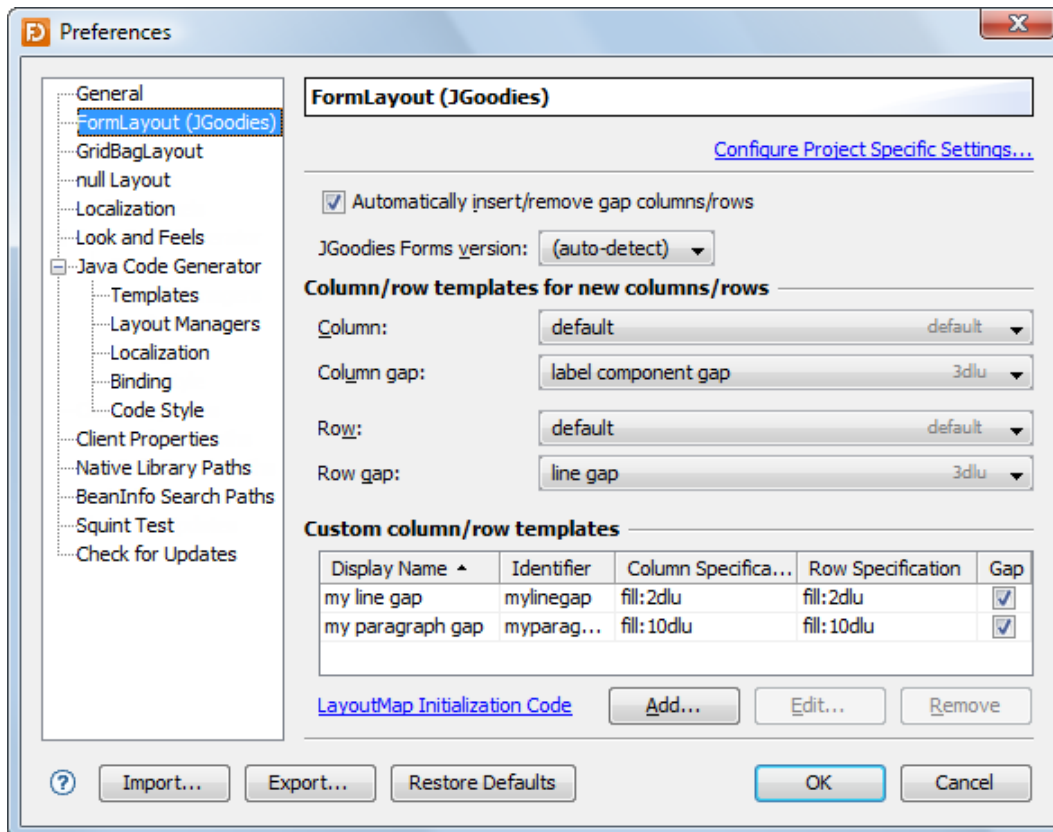
Option	Description	Default
Automatically insert/remove gap columns/rows	If enabled, JFormDesigner automatically inserts/removes gap columns/rows.	On
JGoodies Forms version	Required JGoodies Forms version for the created forms.	auto-detect
Column/row templates for new columns/rows	Here you can specify 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. Since JGoodies Forms 1.2 you can add these custom column/row templates to the global LayoutMap using the "LayoutMap Initialization Code" link.	

Custom column/row templates
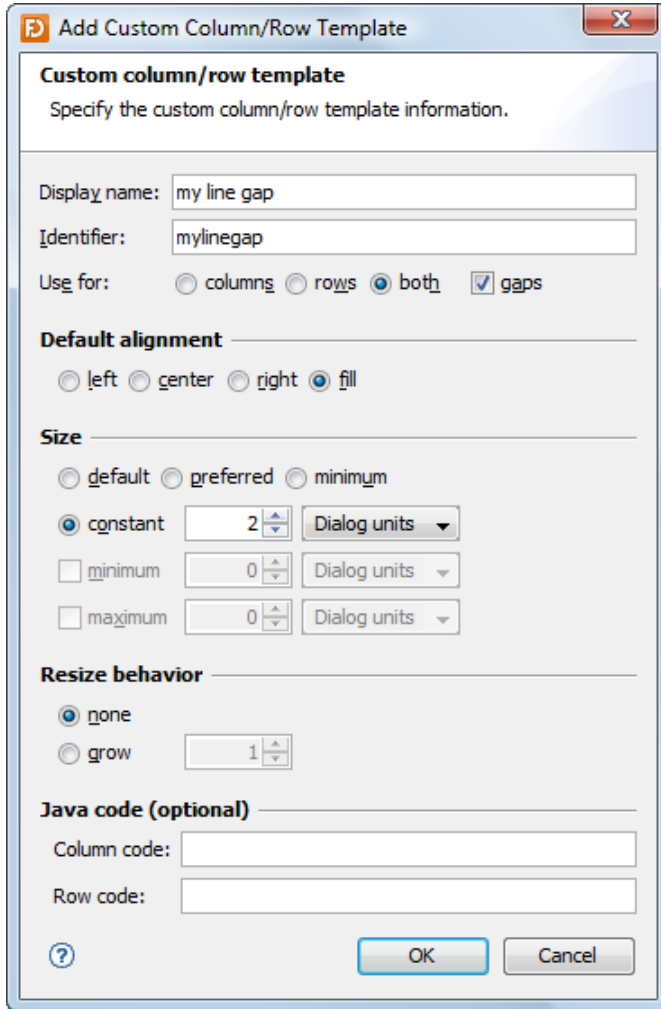
Option	Description
Display name	The display name is used within JFormDesigner whenever the template is shown in combo boxes or popup menus.
Identifier	The (unique) identifier is stored in form files. Choose a short string. Only letters and digits are allowed.
Use for	Specifies whether the template should 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. Not available for JGoodies Forms 1.2 and later.

GridBagLayout

On this page, you can specify GridBagLayout related options.
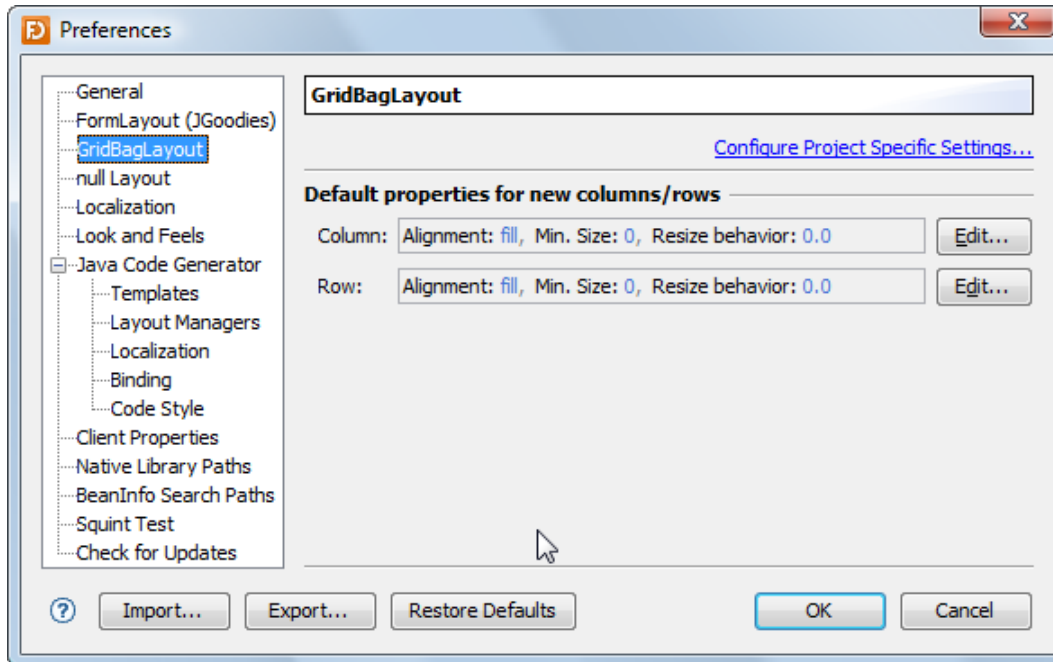
Option	Description	Default
Default properties for new columns/rows	Here you can specify the column and row properties that should be used when new columns or rows are inserted.	
Column	The column properties used for new columns.	fill:0:0.0
Row	The row properties used for new rows.	fill:0:0.0

null Layout

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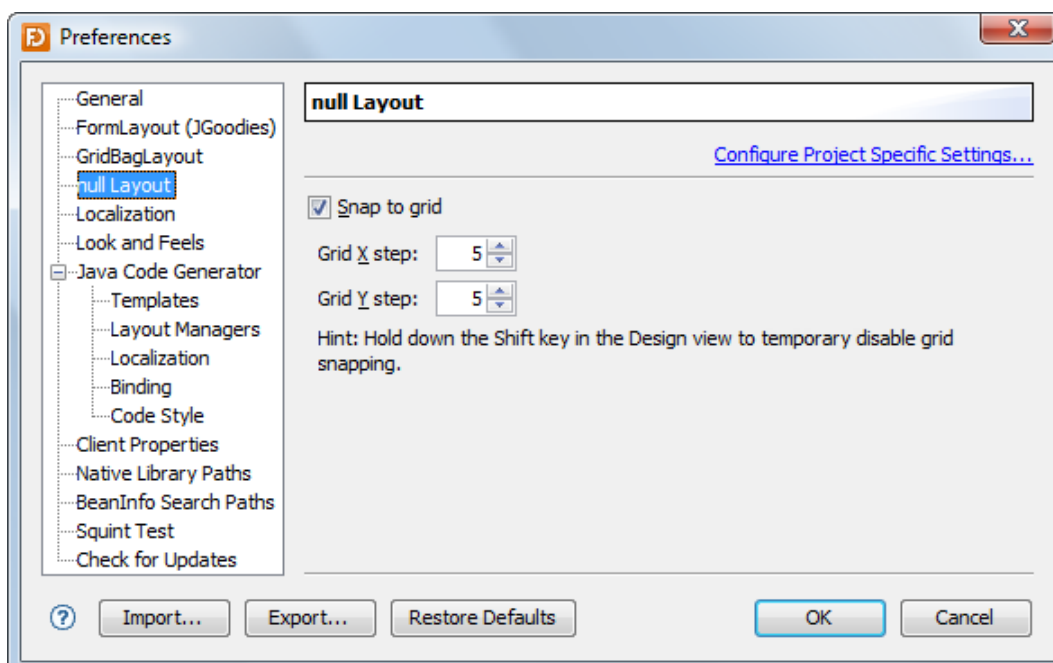
Option	Description	Default
Snap to grid	If enabled, snap to the grid specified below when moving or resizing a component in null layout.	On
Grid X step	The horizontal step size of the grid.	5
Grid Y step	The vertical step size of the grid.	5

Localization

On this page, you can specify localization related options.
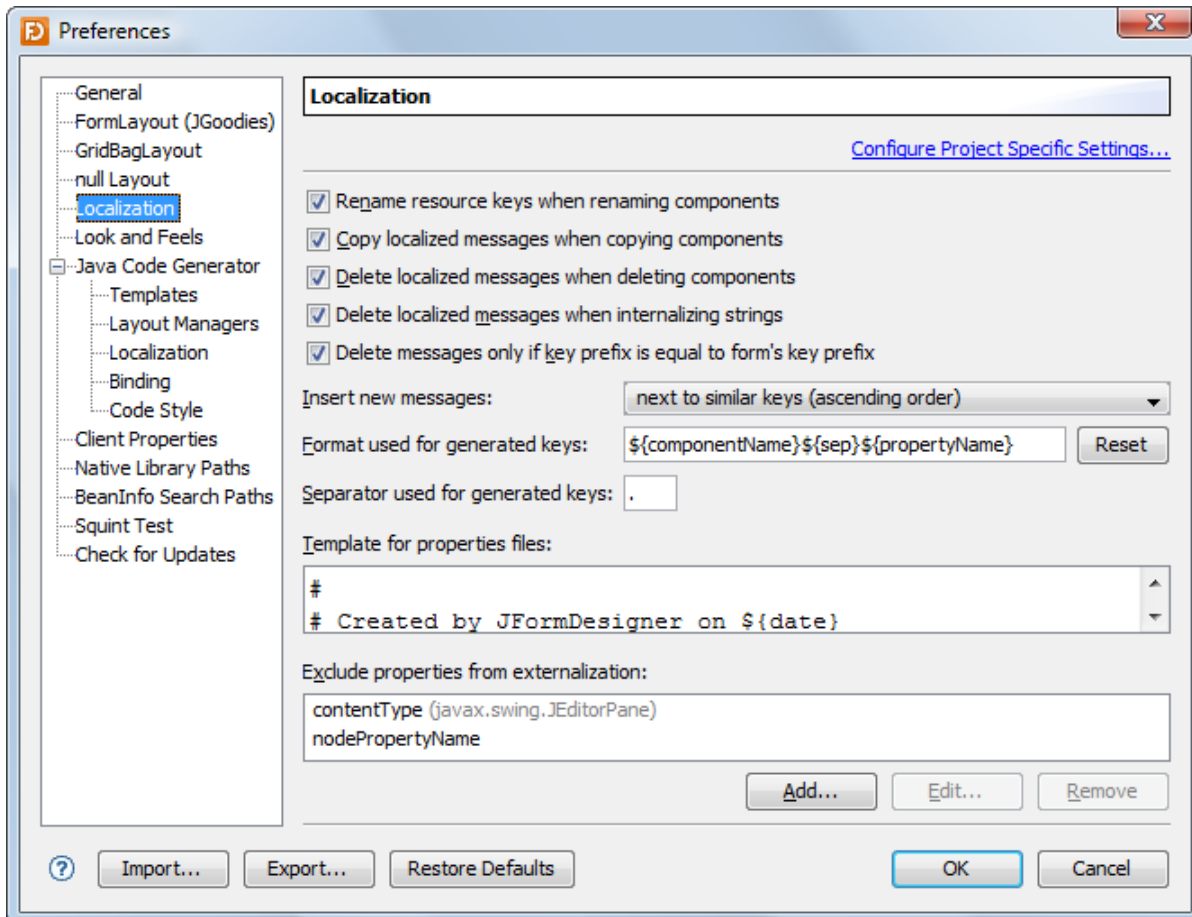
Option	Description	Default
Rename resource keys when renaming components	If enabled, auto-rename resource keys when renaming components and the resource key contains the old component name.	On
Copy localized messages when copying components	If enabled, duplicate localized strings in all locales when 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
Delete messages only if key prefix is equal	If enabled, messages will be auto-deleted only if their key prefix	On

Option	Description	Default
to form's key prefix	is equal to the key prefix of the form.	
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)
Format used for generated keys	Format used when generating a resource key.	${componentName}${sep}${propertyName}
Separator used for generated keys	Separator used when generating a resource key.	'.'
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. The popular Substance and Synthetica look and feels are currently not supported.
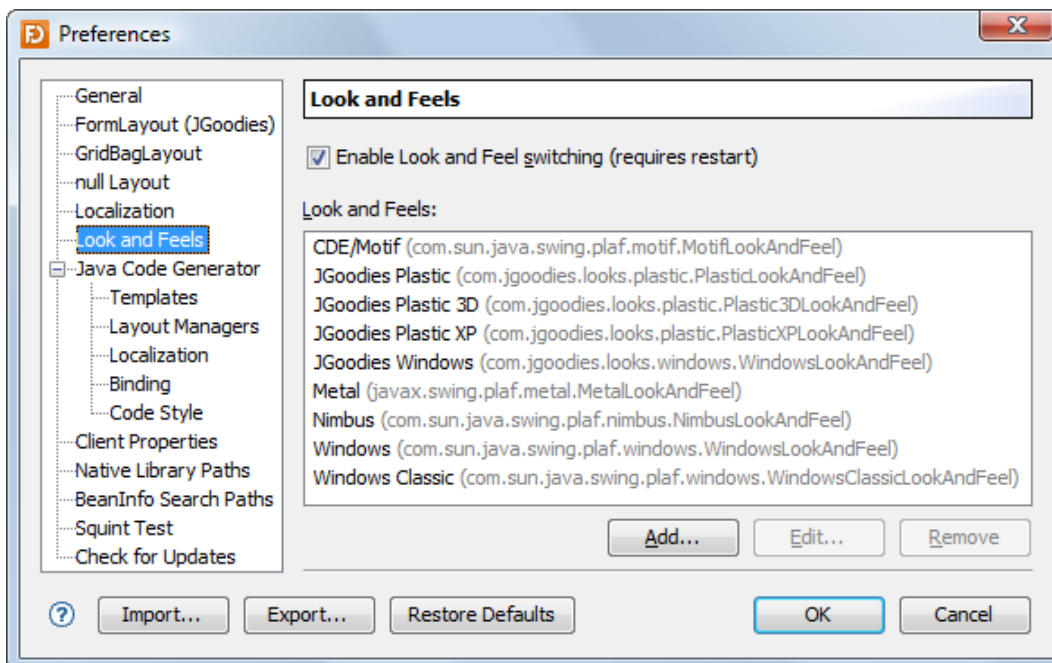
If the look and feels list is focused, you can use the **Insert** key to add a look and feel or the **Delete** key to delete selected look and feels.

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 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
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
Use Eclipse code formatter	If enabled, the Eclipse code formatter is used to format the generated code. (**Eclipse plug-in** only)	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

Option	Description	Default
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`
Variable modifiers	The default modifiers of the variables generated for components. Allowed modifiers: `public`, `default`, `protected`, `private`, `static` and `transient`.	`private`
Event handler modifiers	The default modifiers used when generating event handler methods. Allowed modifiers: `public`, `default`, `protected`, `private`, `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.
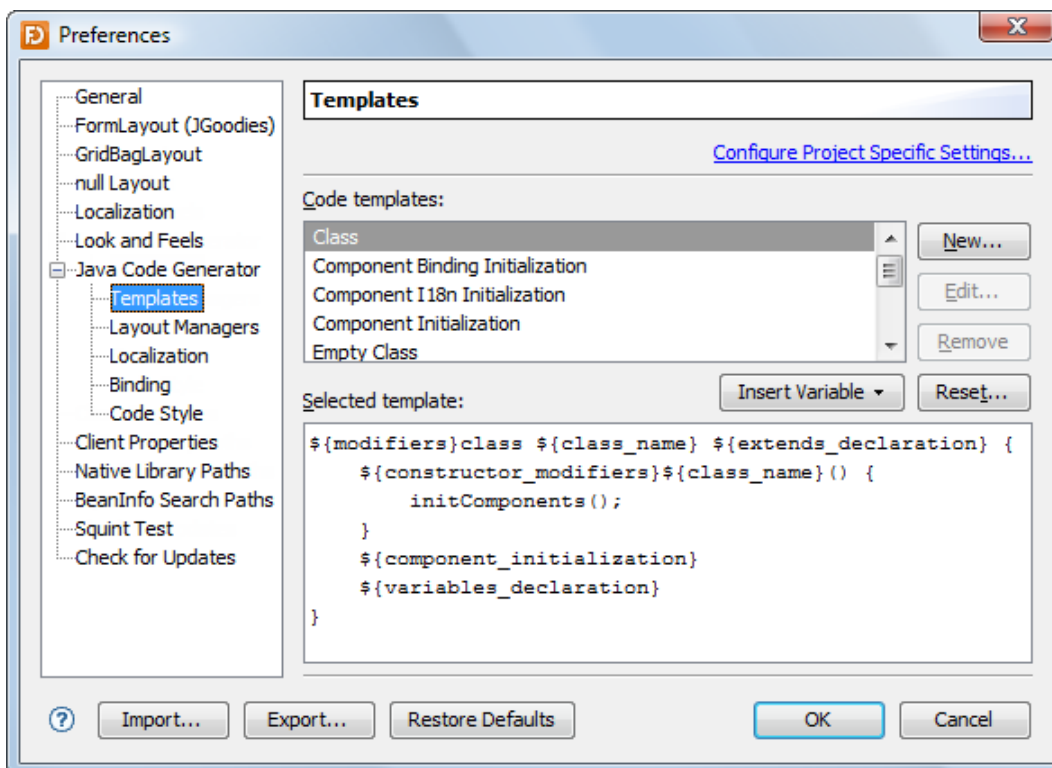
New: Create a new template for a specific superclass.
Edit: Edit the superclass of the selected user-defined template.
Remove: Remove the selected template. Only allowed for user-defined templates.
Reset: Reset the selected predefined template to the default.
Insert Variable: Insert a variable at the current cursor location into the selected template.
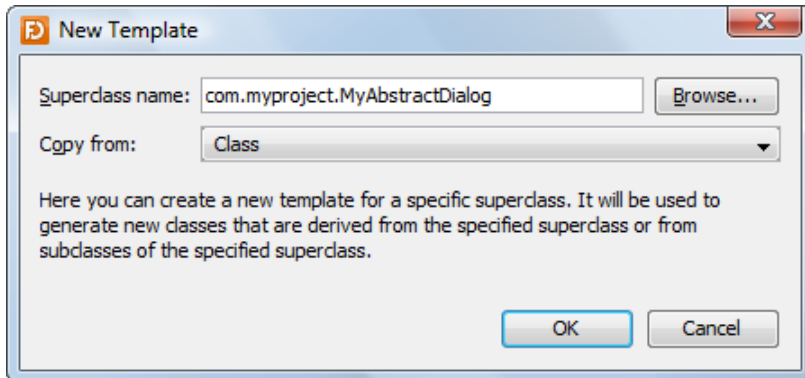
Layout Managers (Java Code Generator)

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
GroupLayout Generation Style	Specifies whether class javax.swing.GroupLayout is used, which is part of Java 6 and later. Or whether org.jdesktop.layout.GroupLayout from the Open Source Swing Layout library swing-layout.jar is used, which is also available for Java 1.4 and 5.	use source compatibility (see Java Code Generator preferences page)
TableLayout package	Package name used by the Java code generator for TableLayout.	info.clearthought.layout

Option	Description	Default
	Change this only if you have a copy of the original TableLayout in another package.	

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})

Binding (Java Code Generator)

On this page, you can specify code generation options for Beans Binding (JSR 295).
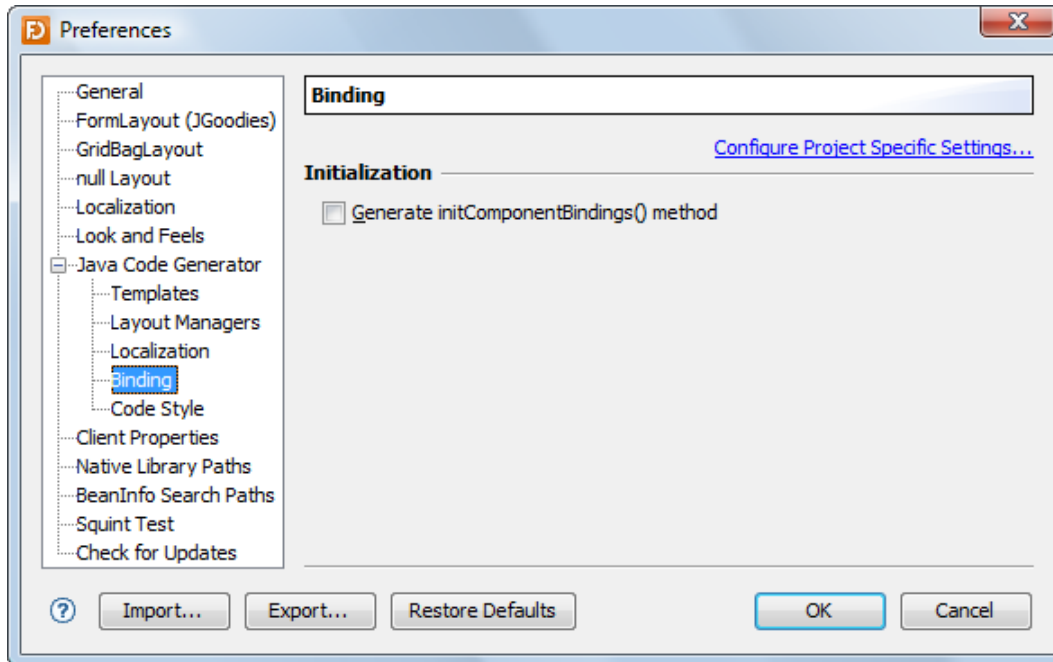
Option	Description	Default
Generate initComponentBindings() method	If enabled, the code generator puts the code to create bindings into a method initComponentBindings(). You can set this options also per form in the "(form)" properties.	Off

Code Style (Java Code Generator)

Stand-alone: On this page, you can specify code style options, which are used for code generation.

IDE plug-ins: This page is not available in IDE plug-ins because IDEs already have preferences that control code style. JFormDesigner uses the code style settings from IDE projects or preferences.
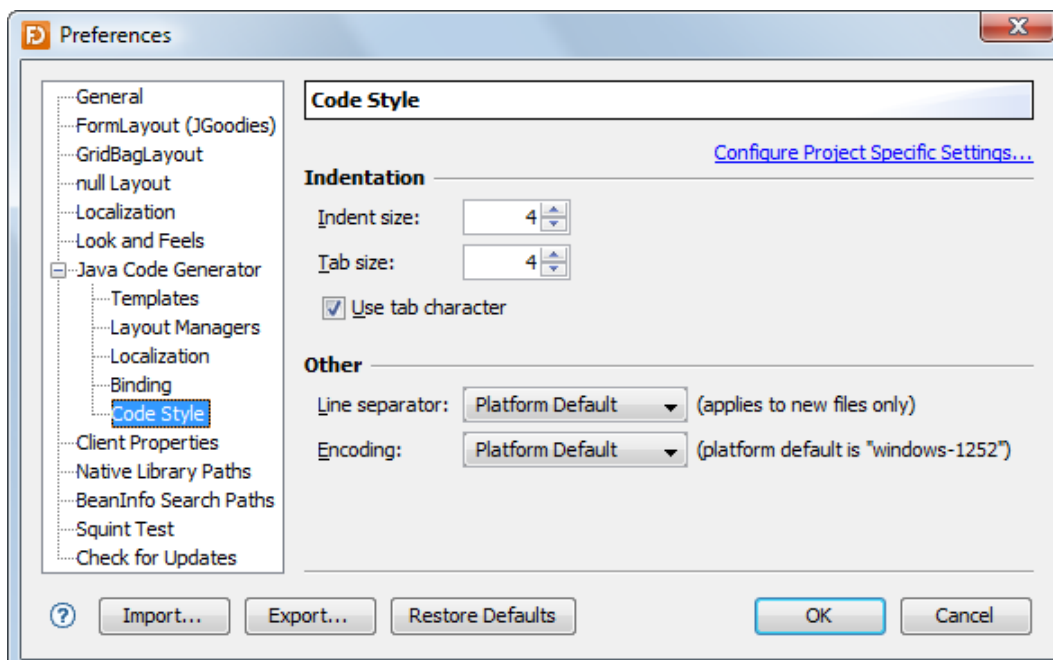
Option	Description	Default
Indent size	The number of spaces used for one indentation level.	4
Tab size	The number of spaces that represents one tabulation.	4
Use tab character	Specifies whether the tab character (\t) is used for indentation or only space characters.	On
Line separator	The line separator used for newly created .java and .properies files.	Platform default
Encoding	The character encoding used for reading and writing Java files.	Platform default

Client Properties

On this page, you can can define client properties, which can be set in the Properties view.
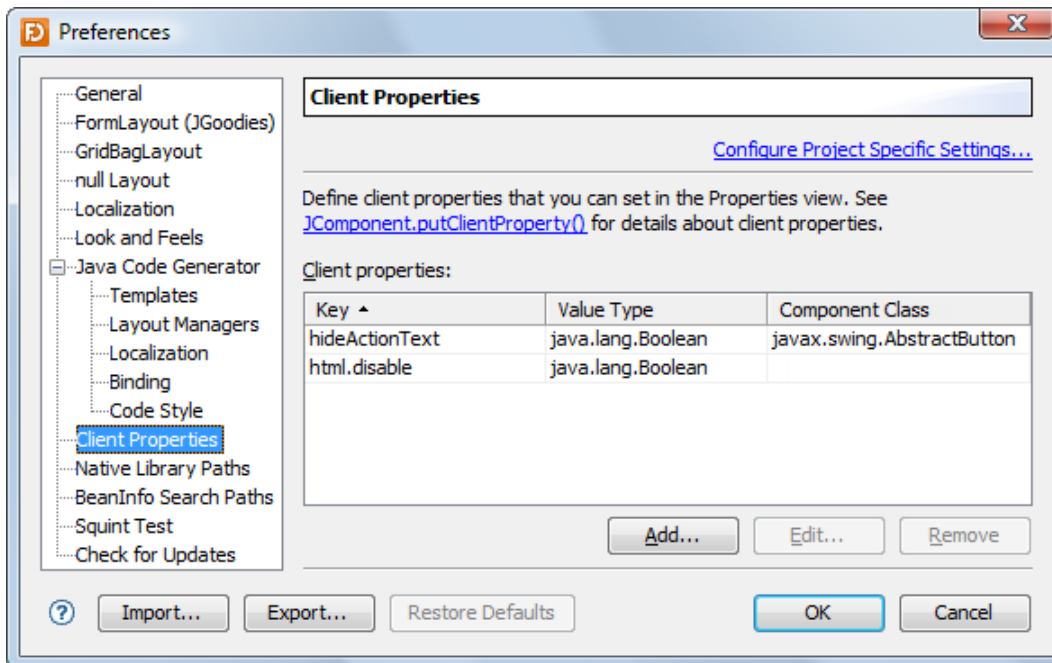
If the client properties list is focused, you can use the **Insert** key to add a client property or the **Delete** key to delete selected client properties.

Option	Description
Key	The key that identifies the client property.
Component class(es)	The component class(es) 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. To specify multiple classes, separate them with commas. If not specified, the client property is shown for all JComponent 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.
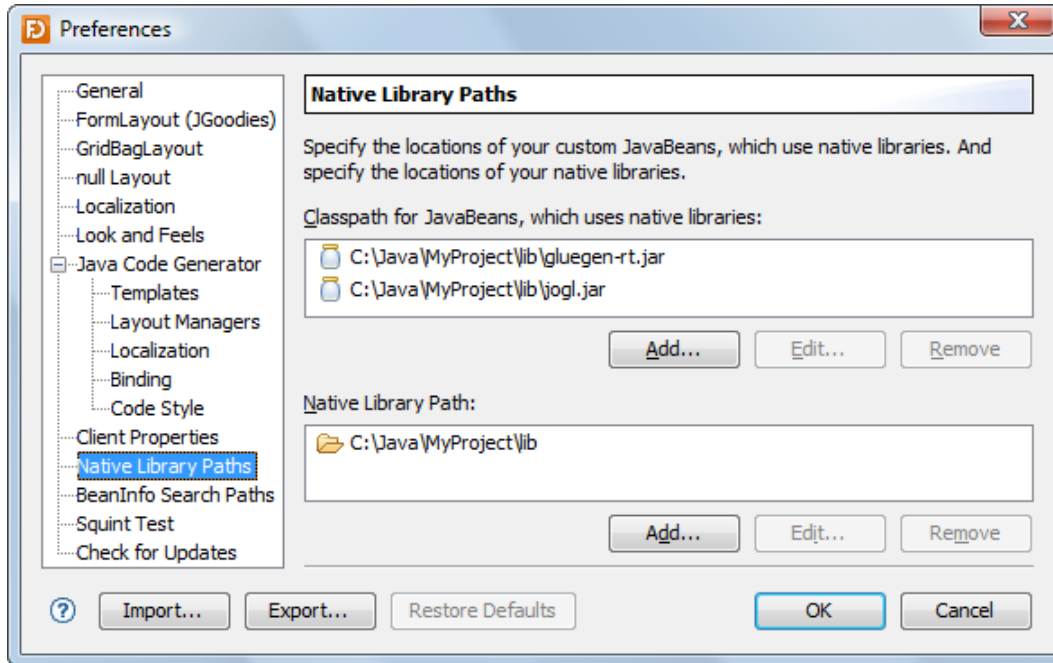
Option	Description
Classpath for JavaBeans, which use native libraries	JAR files or folders containing .class files, which are using native libraries. They must be specified here to ensure that the native libraries are loaded from a special class loader only once.
Native Library Path	Folders used to 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.
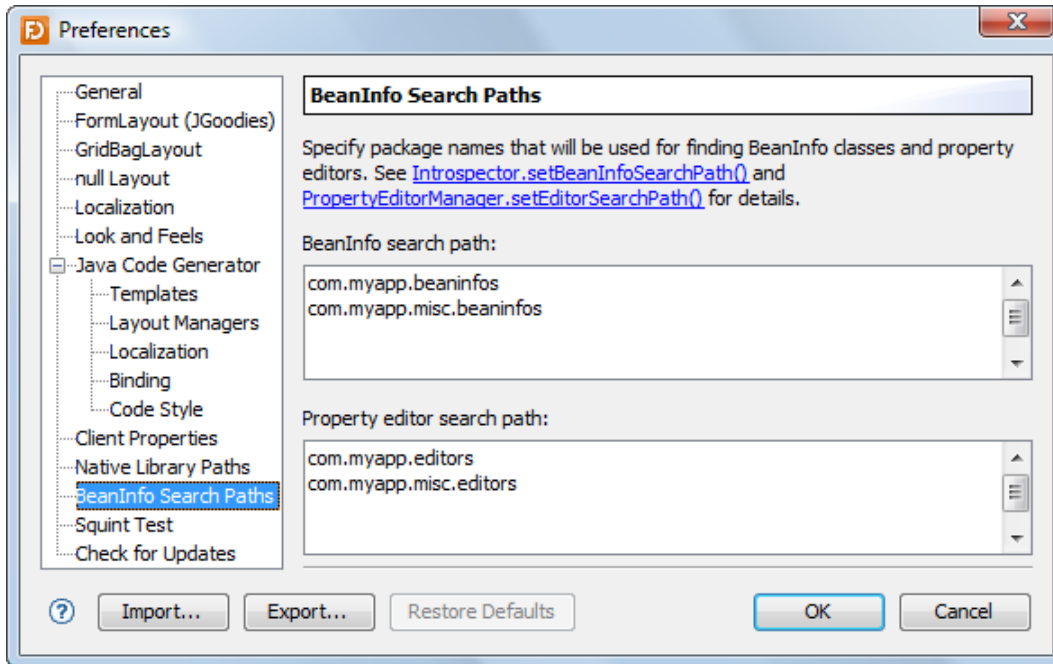
Option	Description
BeanInfo search path	Package names that will be used for finding BeanInfo classes. Only necessary if the BeanInfo class is not in the same package as the component class to which it belongs. See java.beans.Introspector and Introspector.setBeanInfoSearchPath() for details.
Property 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

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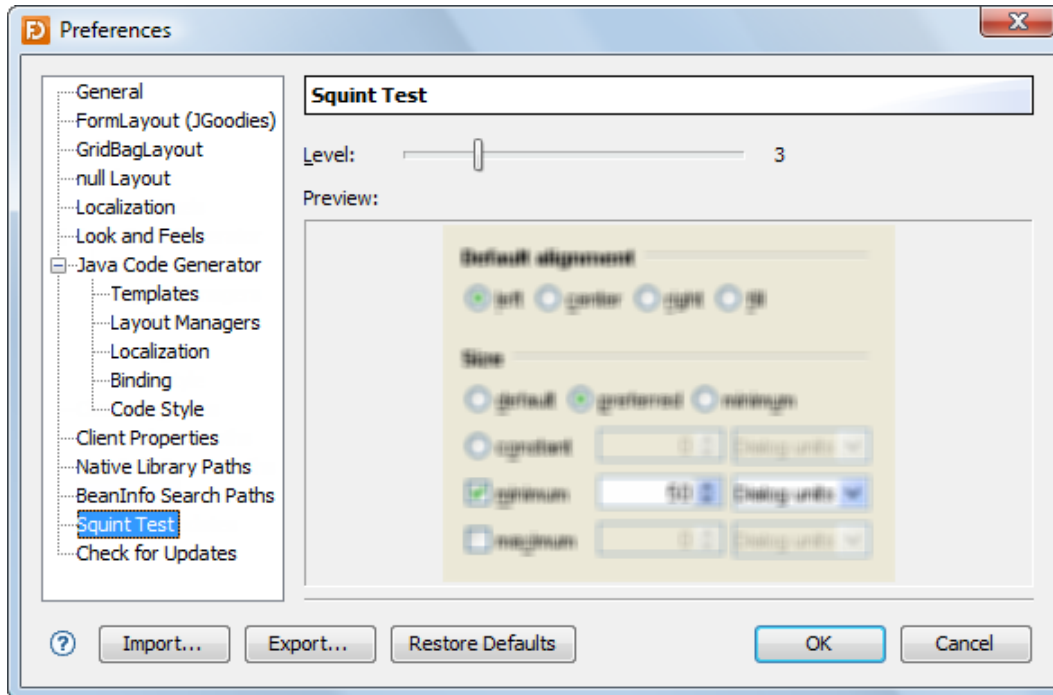
Check for Updates

This page allows you to specify whether JFormDesigner should check for updates and new versions. Click the "Check Now" button to check for updates immediately.
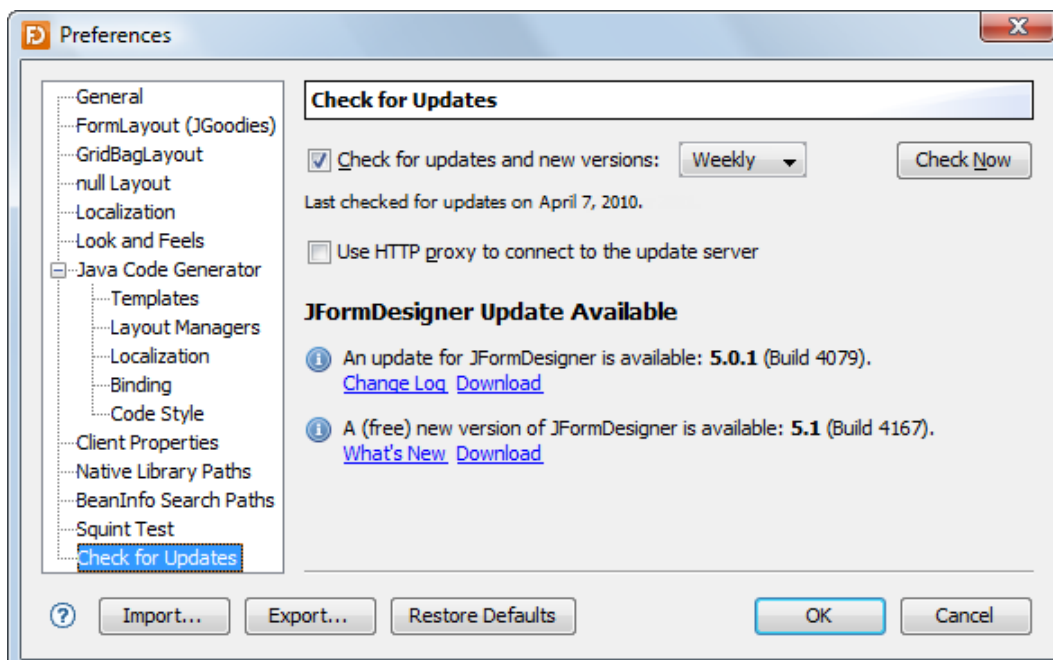
7 IDE Integrations

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

Following IDE plug-ins are available:

- Eclipse plug-in
- NetBeans plug-in
- IntelliJ IDEA plug-in
- JBuilder plug-in

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-in for JDeveloper is already under development.

IDE interworking with stand-alone edition

Care must be taken because you edit the Java source in the IDE and JFormDesigner stand-alone 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 stand-alone.

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 stand-alone.

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.

7.1 Eclipse plug-in

This plug-in integrates JFormDesigner into Eclipse and other Eclipse based IDEs.

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.
- 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, which is only visible if a JFormDesigner form editor is active.
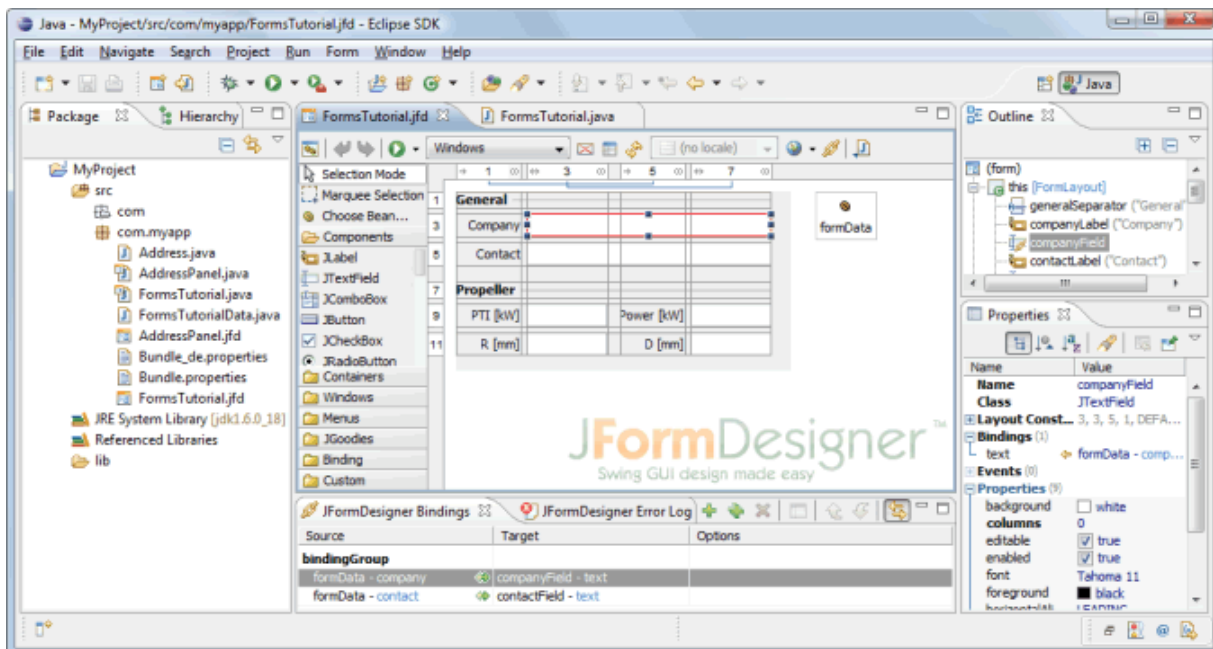
A JFormDesigner editor consists of:

- 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.
- **Bindings View**: Located below the Design view. This view is not visible by default. Click the **Show Bindings View** button (🖉) in the toolbar to make is visible.
- **Error Log View**: Automatically opens on errors in a view at the bottom.

Creating new forms

To create a new form, click the **New JFormDesigner Form** (🖼) button in the Eclipse toolbar. You can also use **Ctrl+Shift+V** (*Mac*: **Shift+Command+V**).
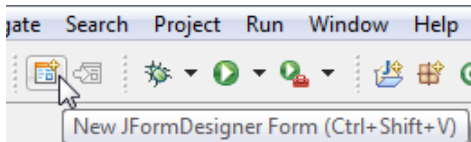
You can also create new forms in Eclipse's Package Explorer view. First select the destination package or folder, then invoke Eclipse's **New** command and select **Other**, which opens Eclipse's **New** dialog. Then choose **JFormDesigner Form** from the list of wizards and click Next to proceed.
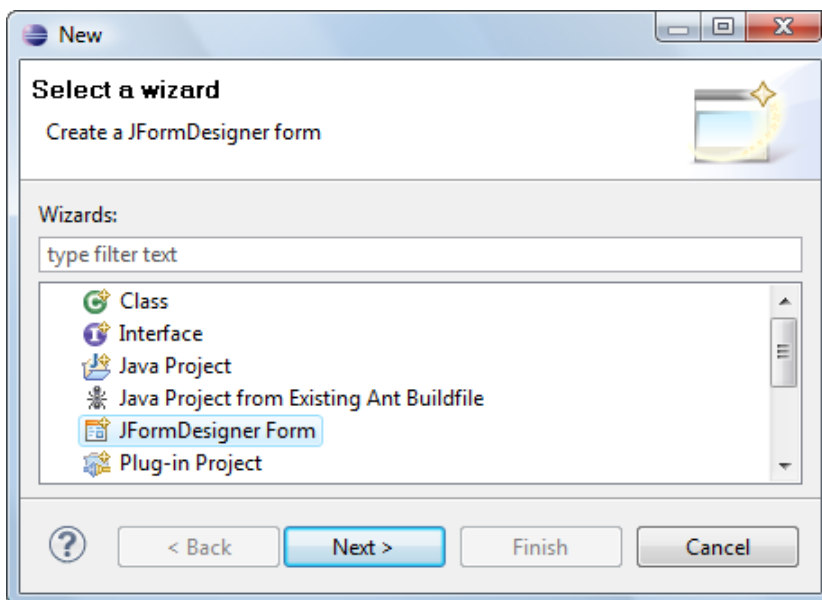
If **JFormDesigner Form** is in the **New** sub menu, you can choose it directly without the need to open Eclipse's **New** dialog.
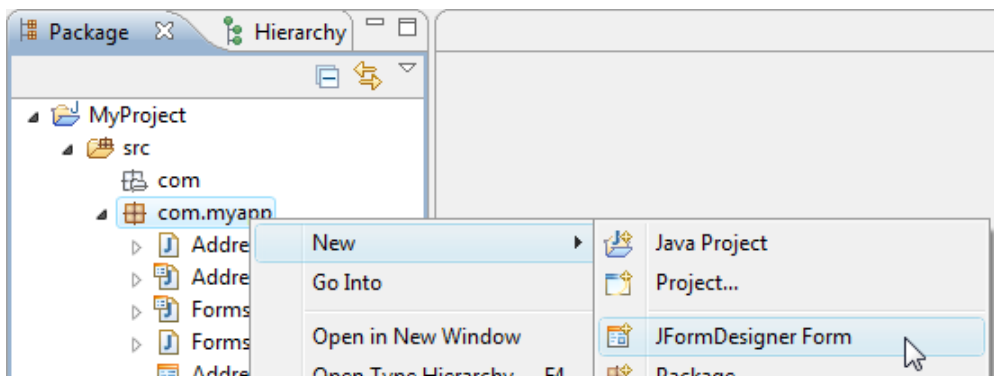
In the **New JFormDesigner Form** dialog, enter the form name (which is also used as class name), choose a superclass, a layout manager and set localization options.

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 / Go to form

JFormDesigner adds a button to Eclipse'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** (*Mac*: **Shift+Command+D**).
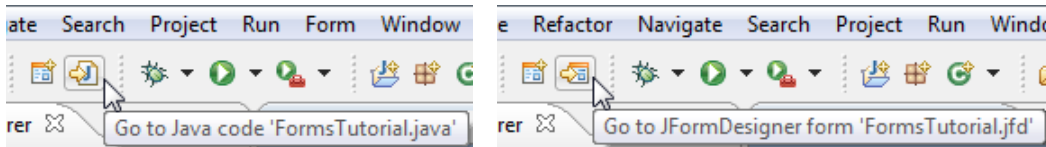
Code folding

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

Convert NetBeans, IntelliJ IDEA and Abeille forms

You can convert existing NetBeans, IntelliJ IDEA and Abeille forms to JFormDesigner forms. Right-click on the form file (or any container) and select **Convert to JFormDesigner Form**.
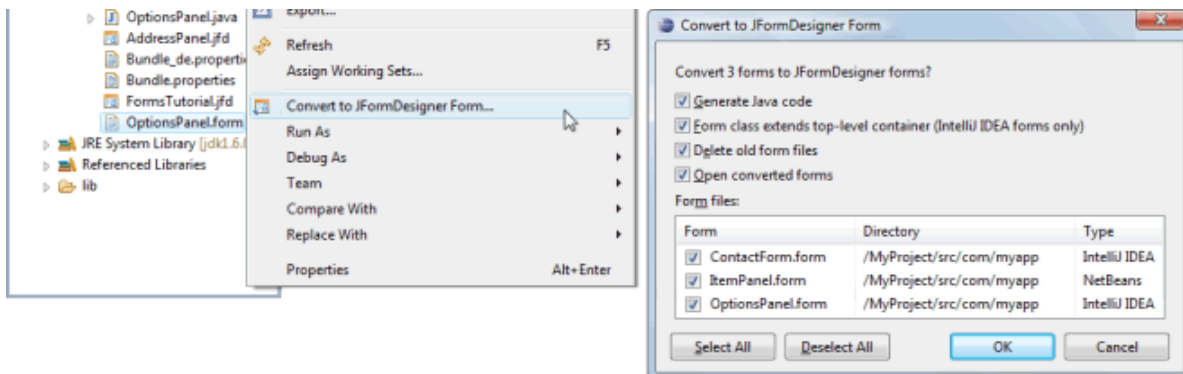
When converting an IntelliJ IDEA form, JFormDesigner inserts its own generated GUI code into the existing Java class and removes IntelliJ IDEA's GUI code.

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.

You can also set project specific settings in the Eclipse project dialog. Select **Project > Properties** from the menu to open it and then expand the node "JFormDesigner" in the tree. See Preferences for details.

Keyboard shortcuts

You can assign shortcut keys to some JFormDesigner commands in Eclipse's keys preferences. Select **Window > Preferences > General > Keys** to open it. Search for "JFormDesigner" to find JFormDesigner commands.

7.2   NetBeans plug-in

This plug-in integrates JFormDesigner into NetBeans.

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 NetBeans. No need to switch between applications.
- Uses the source folders and classpath of the current NetBeans 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 and guarding of generated GUI code in Java editor.
- Go to event handler method in Java editor.
- 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.
- Automatically add needed libraries (JGoodies Forms, TableLayout, etc) to the project.
- Integrated with NetBeans'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 NetBeans main window editing a JFormDesigner form.
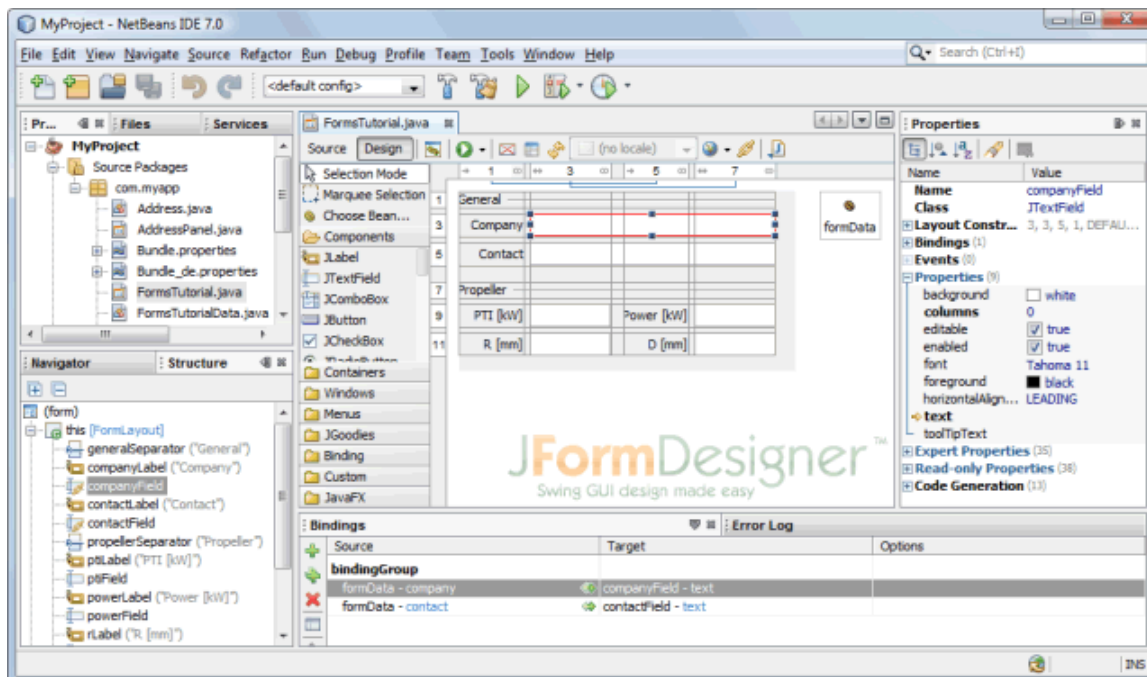
A JFormDesigner editor consists of:

- 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.
- Bindings View: Located below the Design view. This view is not visible by default. Click the **Show Bindings View** button ( ) in the toolbar to make is visible.
- Error Log View: Automatically opens on errors in a view at the bottom.

Creating new forms

You can create new forms using NetBeans's **New File** command. In the category **Swing GUI Forms** choose **JFormDesigner Form** and click Next to proceed.
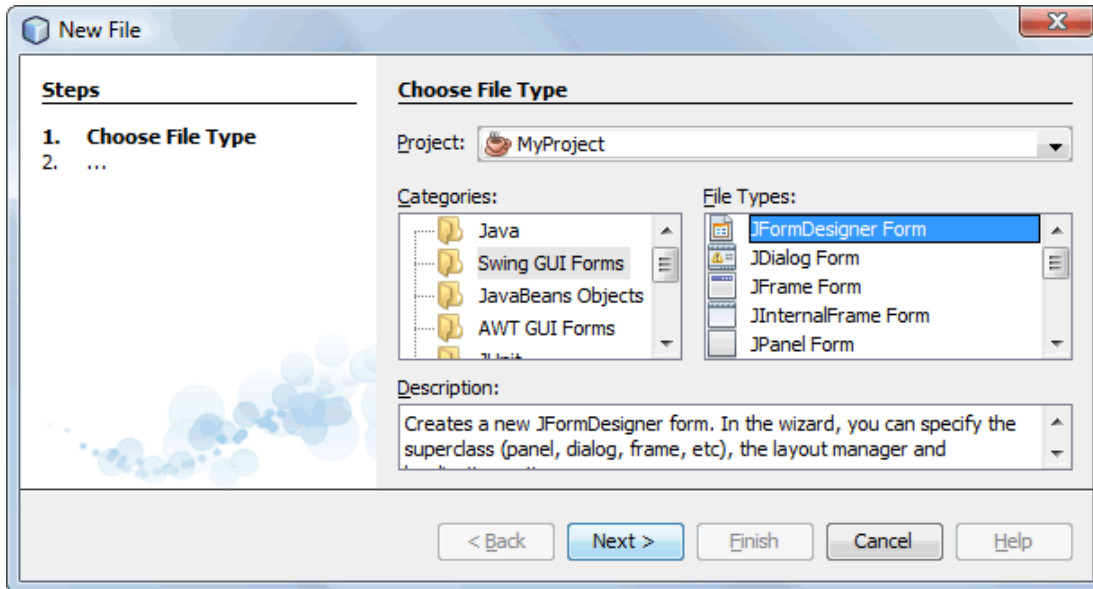
Open forms for editing

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

Source / Design

The **Source** and **Design** toggle buttons in the editor toolbar enable you to switch from a JFormDesigner form editor to its Java editor and vice versa.

Convert NetBeans, IntelliJ IDEA and Abeille forms

You can convert existing NetBeans, IntelliJ IDEA and Abeille forms to JFormDesigner forms. Right-click on the form file (or any container) and select **Tools > Convert to JFormDesigner Form**.
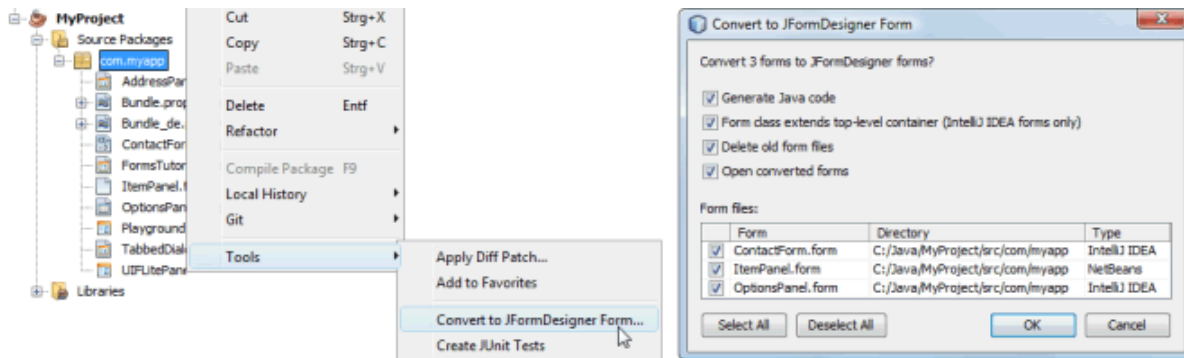
When converting an IntelliJ IDEA form, JFormDesigner inserts its own generated GUI code into the existing Java class and removes IntelliJ IDEA's GUI code.

Options

JFormDesigner uses the term "Preferences" instead of NetBeans "Options". The JFormDesigner preferences are fully integrated into the NetBeans options dialog. Select **Tools > Options** from the menu to open it and then select the "JFormDesigner" page. See Preferences for details.

You can also set project specific options in the NetBeans project dialog. Select **File > Project Properties** from the menu to open it and then expand the node "JFormDesigner" in the tree. See Preferences for details.

Keyboard shortcuts

You can assign shortcut keys to some JFormDesigner commands in NetBeans keymap options. Select **Tools > Options > Keymap** to open it. Click on the Category column to sort key bindings by category name and scroll to the JFormDesigner category.

Unsupported features

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

- Use look and feels in Design view.

7.3   IntelliJ IDEA plug-in

This plug-in integrates JFormDesigner into Jetbrains IntelliJ IDEA (Community and Ultimate Editions).

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.
- 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.
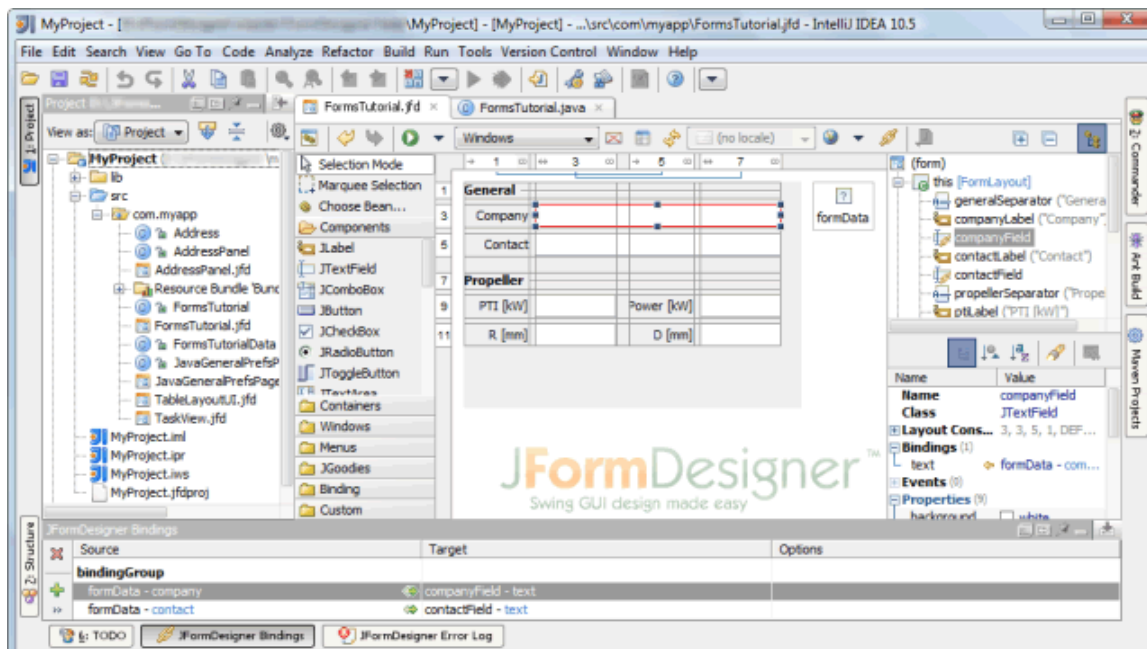

A JFormDesigner editor consists of:

- 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.

- **Bindings View**: Located below the Design view. This view is not visible by default. Click the **Show Bindings View** button ( ) in the toolbar to make is visible.

- **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**.
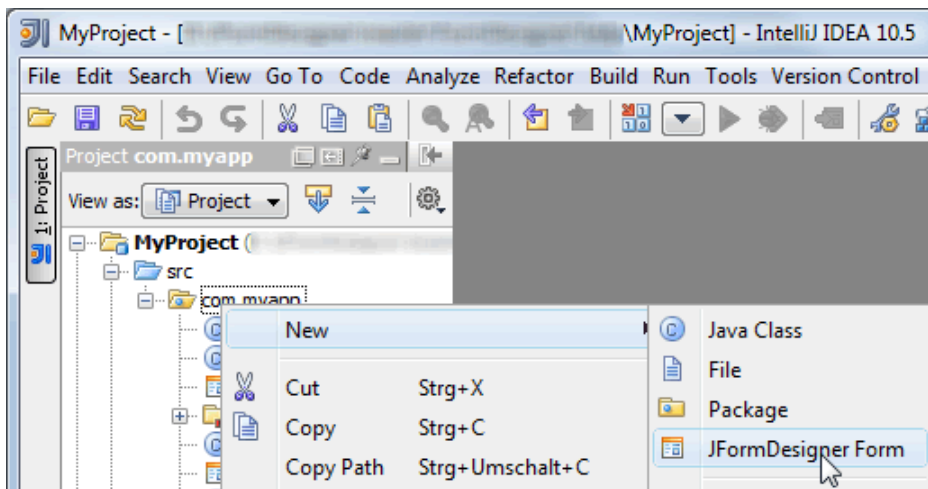

In the **New JFormDesigner Form** dialog, enter the form name (which is also used as class name), choose a superclass, a layout manager and set localization options.

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 IntelliJ IDEA's project views and double-click it.
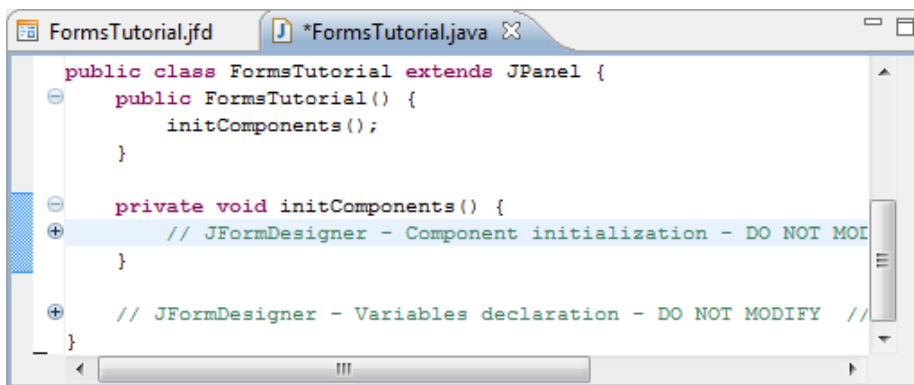
Go to Java code / Go to form

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** (*Mac*: **Shift+Command+D**).
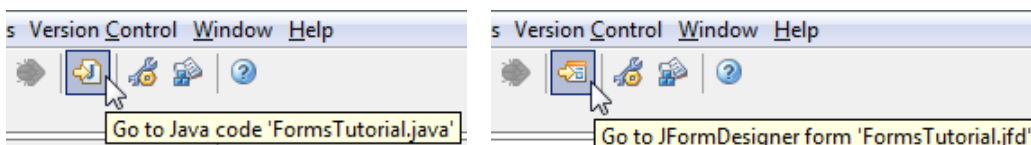
Code folding

To move the generated code out of the way, JFormDesigner folds it in the Java editor.
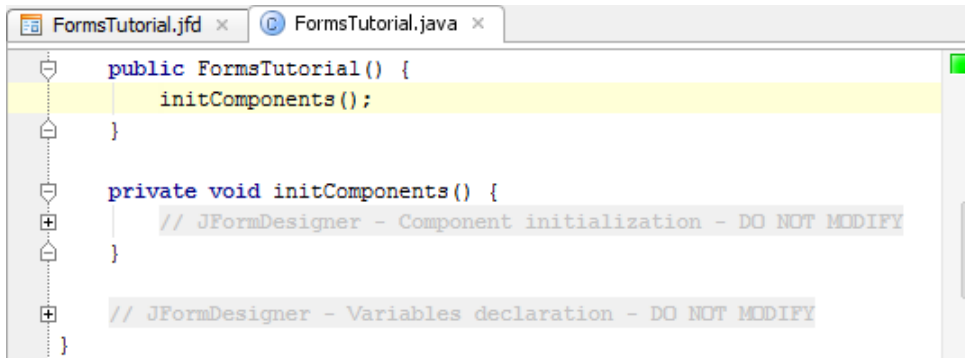
Convert IntelliJ IDEA, NetBeans and Abeille forms

You can convert existing IntelliJ IDEA, NetBeans and Abeille forms to JFormDesigner forms. Right-click on the form file (or any container) and select **Convert to JFormDesigner Form**.
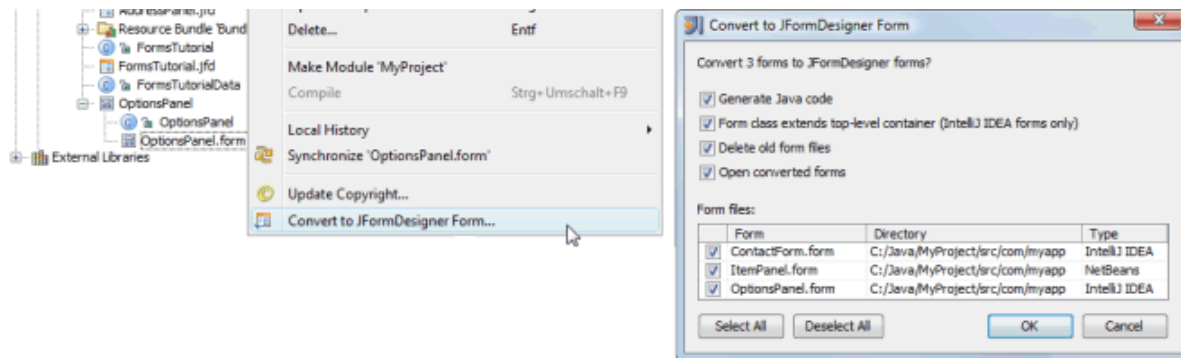
When converting an IntelliJ IDEA form, JFormDesigner inserts its own generated GUI code into the existing Java class and removes IntelliJ IDEA's GUI code.

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 item named "JFormDesigner" in the "IDE Settings" area. To set project specific settings, select the item named "JFormDesigner (Project)" in the "Project Settings" area. See Preferences for details.

Keyboard shortcuts

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

7.4 JBuilder plug-in

This plug-in integrates JFormDesigner into JBuilder 2006. For JBuilder 2007 (or later) use the Eclipse plug-in. For JBuilder 2005 use JFormDesigner 4.

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.
- 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.
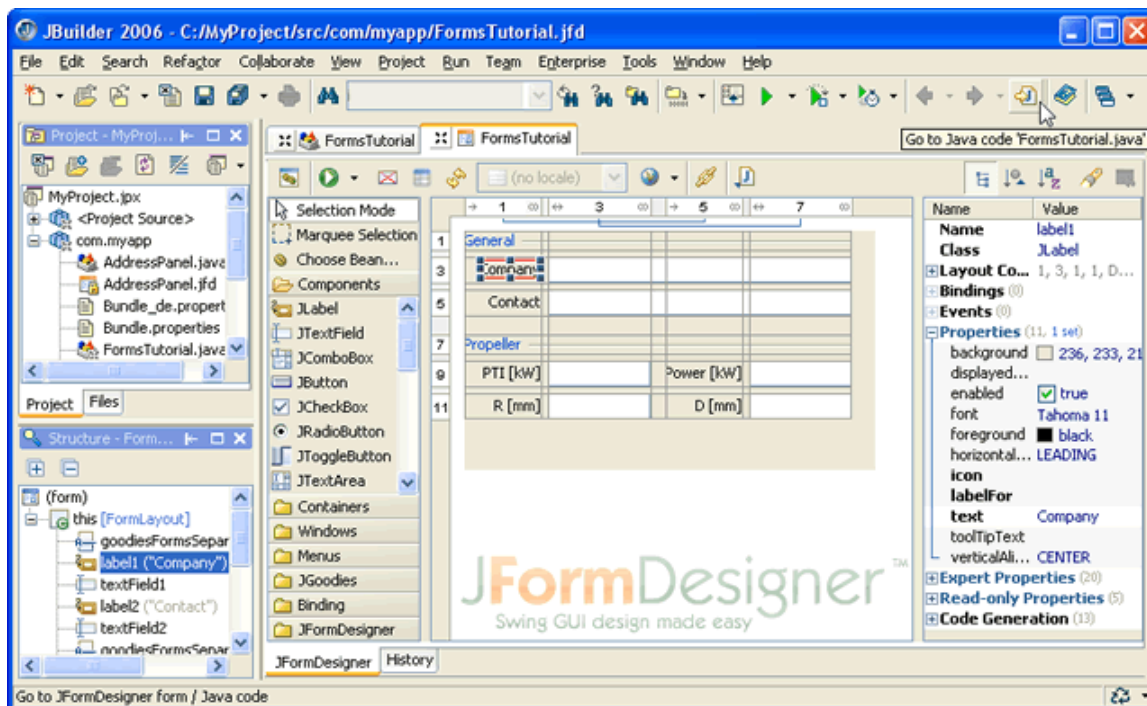
A JFormDesigner editor consists of:

- 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.
- Bindings View: Located below the Design view. This view is not visible by default. Click the **Show Bindings View** button ( ) in the toolbar to make is visible.
- 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**.
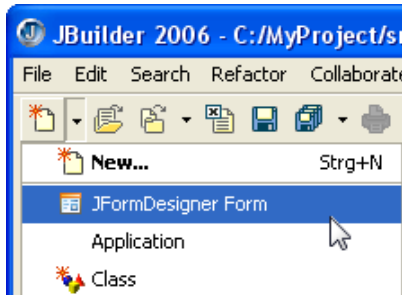
In the **New JFormDesigner Form** dialog, enter the form name (which is also used as class name), choose a superclass, a layout manager and set localization options.

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** ( ).
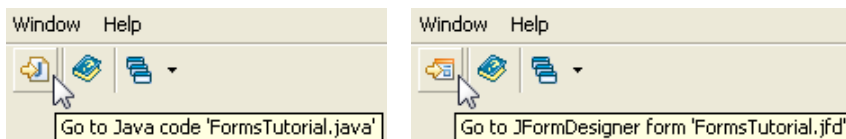
Code folding

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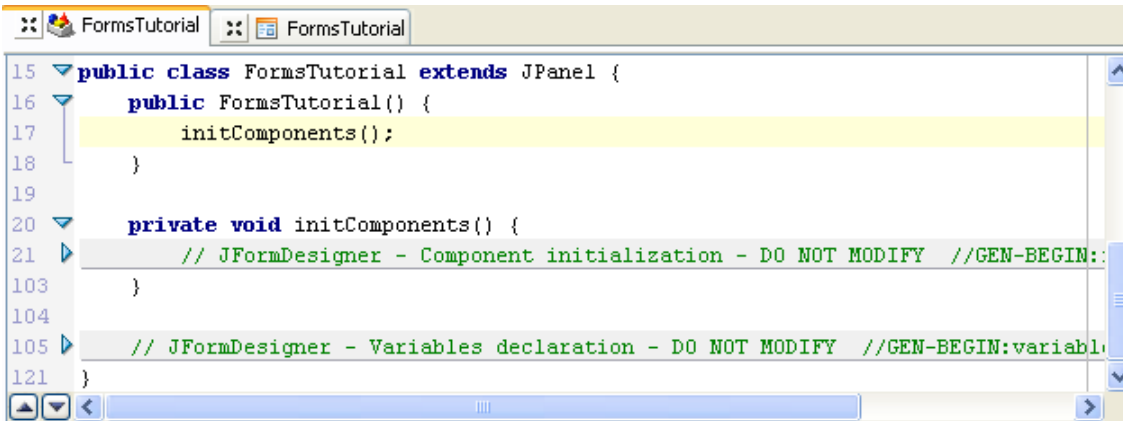
Convert JBuilder forms

You can convert existing JBuilder forms to JFormDesigner forms. Right-click on the Java file and select **Convert to JFormDesigner Form**.
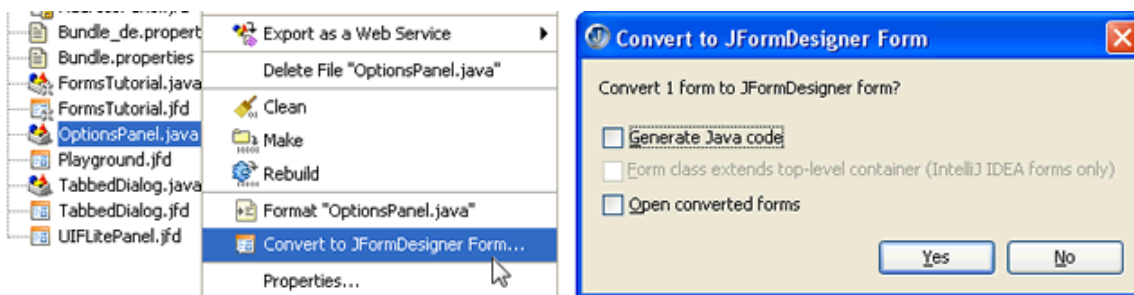
Note: JFormDesigner inserts its own generated GUI code into the existing Java class, but does not remove JBuilder's GUI code. You have to remove JBuilder's component variables and initialization code yourself.

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

8   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
- GroupLayout (Free Design)
- HorizontalLayout (SwingX)
- IntelliJ IDEA GridLayout
- null Layout
- TableLayout
- VerticalLayout (SwingX)

How to choose a layout manager?

For "normal" forms use either one of the grid-based layout managers (FormLayout, TableLayout or GridBagLayout) or use "Free Design" (GroupLayout). Each layout manager 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 grid-based layout managers, but JFormDesigner hides its complexity and adds additional features like gaps. Use GridBagLayout if you cannot use FormLayout or TableLayout.
- GroupLayout (Free Design) allows you to lay out your forms by simply placing components where you want them. Visual guidelines suggest optimal spacing, alignment and resizing of components.

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.
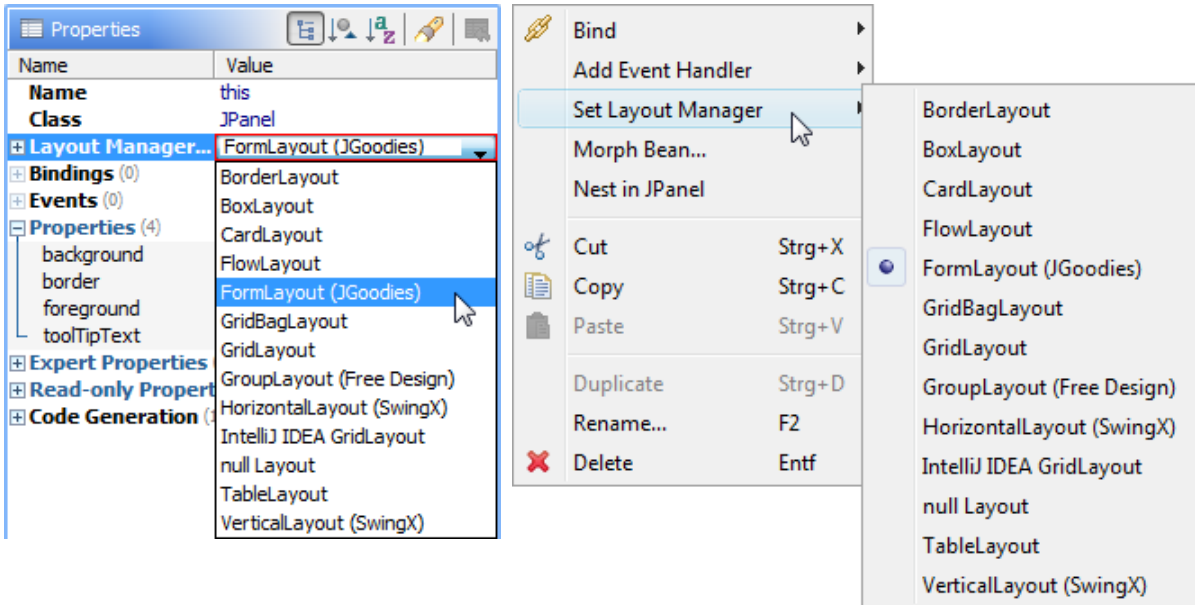
8.1 BorderLayout

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

North		
West	Center	East
South		

PageStart		
LineStart	Center	LineEnd
PageEnd		

PageStart		
LineEnd	Center	LineStart
PageEnd		
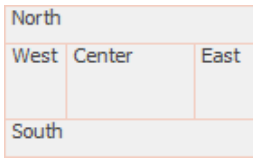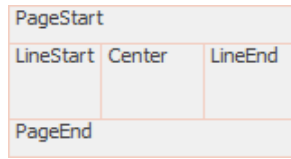
(absolute positioning)   (left-to-right relative positioning)   (right-to-left relative positioning)
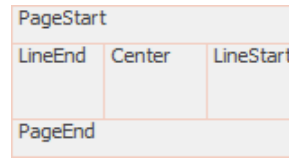
The components are laid out according to their preferred sizes. The north and south components may be stretched horizontally. The east and west components may be stretched vertically. The center component may be stretched horizontally and vertically to fill any space left over.

In addition to absolute positioning, BorderLayout supports relative positioning, which swaps west and east components if the component orientation of the container is set to right-to-left. To use relative positioning, first add a component to one of the four side areas and then change the layout constraints property of that component to PAGE_START, PAGE_END, LINE_START or LINE_END.

BorderLayout is part of the standard Java distribution.

Layout manager properties

A container with this layout manager has following layout manager properties:

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

Layout constraints properties

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

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

8.2   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 manager properties

A container with this layout manager has following layout manager properties:

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

8.3 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 card layout is the visible component when the container is first displayed.

CardLayout is part of the standard Java distribution.

Layout manager properties

A container with this layout manager has following layout manager properties:

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

Layout constraints properties

A component contained in a container with this layout manager has following layout 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)`.

8.4   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 manager properties

A container with this layout manager has following layout manager properties:

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

8.5   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.
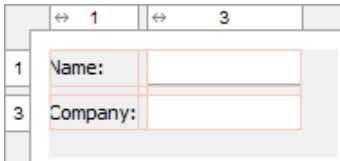
Unlike other grid-based layout managers, FormLayout uses 1-based column/row indices. And it uses "real" columns/rows as gaps. Therefore the unusual column/row numbers in the above screenshot. Using gap columns/rows has the advantage that you can give gaps different sizes.

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 `jgoodies-forms.jar`, `jgoodies-forms-javadoc.zip` and `jgoodies-forms-src.zip` in its `redist` folder. For more documentation and tutorials, visit www.jgoodies.com or forms.java.net.
>
> The API documentation is also available here: doc.formdev.com/jgoodies-forms/.
>
> **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 manager properties

A container with this layout manager has following layout manager 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 header 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.

Field	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.

Layout constraints properties

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

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

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

8.5.1   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
label component gap	A logical vertical gap between a label and an associated component. (requires JGoodies Forms 1.4 or later)	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

8.5.2   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.
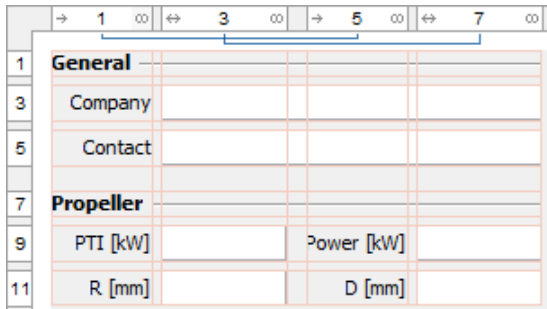
In the above example, columns [1 and 5] and columns [3 and 7] have the same width.

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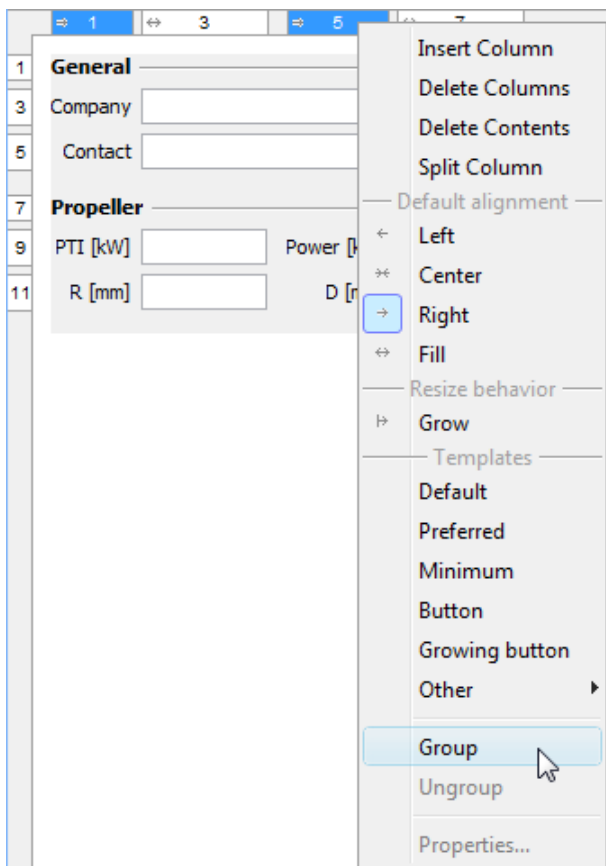
Note that selected gap columns/rows will be ignored when grouping.

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 single 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.

8.6 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.
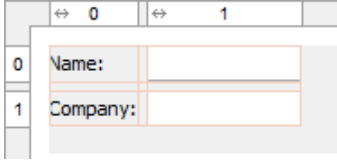
Use the column and row headers to insert or delete columns/rows and change column/row properties.

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.

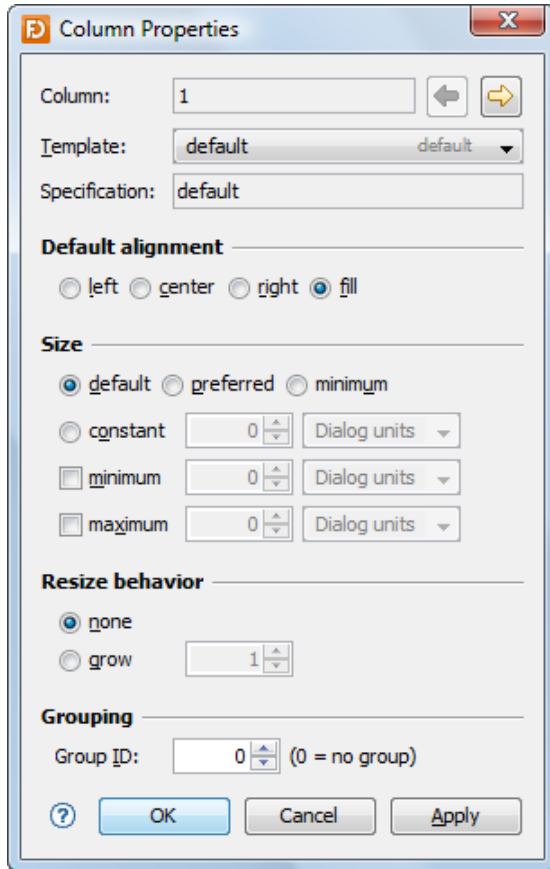## Layout manager properties

A container with this layout manager has following layout manager properties:

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

Field	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.

Layout constraints properties

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

Property Name	Description	Default
grid x	Specifies the component's horizontal grid origin (column index).	0
grid y	Specifies the component's vertical grid origin (row index).	0
grid width	Specifies the component's horizontal grid extend (number of columns).	1
grid height	Specifies the component's vertical grid extend (number of rows).	1
h align	The horizontal alignment of the component within its cell. Possible values: DEFAULT, LEFT, CENTER, RIGHT and FILL.	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
weight x	Specifies how to distribute extra horizontal space.	0.0

Property Name	Description	Default
weight y	Specifies how to distribute extra vertical space.	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.	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.	0
ipad y	Specifies the internal padding, that is, how much space to add to the minimum height of the component.	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.

8.7   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 manager properties

A container with this layout manager has following layout manager properties:

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

8.8   GroupLayout (Free Design)

The goal of the group layout manager is to make it easy to create professional cross platform layouts. It is designed for GUI builders, such as JFormDesigner, to use the "Free Design" paradigm. You can lay out your forms by simply placing components where you want them. Visual guidelines suggest optimal spacing, alignment and resizing of components.
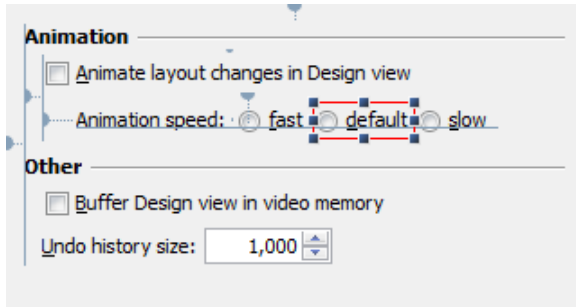
GroupLayout has been developed by the NetBeans team and is also used by the NetBeans GUI Builder (formerly Project Matisse). They provide a comprehensive tutorial on designing GUIs using GroupLayout, which is also suitable for JFormDesigner: http://www.netbeans.org/kb/60/java/quickstart-gui.html

> GroupLayout is part of the standard Java distribution since Java 6. If you need to run your application also on Java 5 or 1.4, you can use the open-source Swing Layout Extension library, which is compatible to the Java 6 GroupLayout, but uses different package names. Change the option "GroupLayout Generation Style" in the Layout Managers (Java Code Generator) preferences if necessary.
>
> The API documentation is available here: doc.formdev.com/grouplayout/.
>
> **IDE plug-ins**: If you use GroupLayout from the Swing Layout Extension library 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.

Alignment guidelines

Alignment guidelines appear only when adding or moving components. They indicate the preferred positions to which components snap when releasing the mouse button.

Insets are the preferred spacings between components and their container.
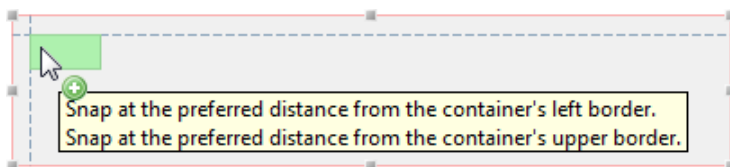
Offsets are the preferred spacings between adjacent components.
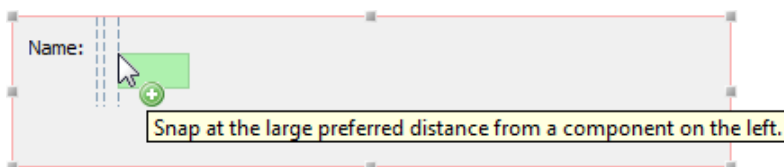
Baseline alignment is the preferred relationship between adjacent components that display text.
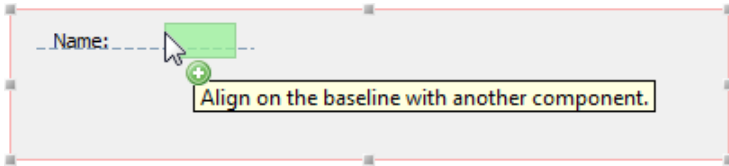
Edge alignments (top, bottom, left and right) are possible relationships between adjacent components.
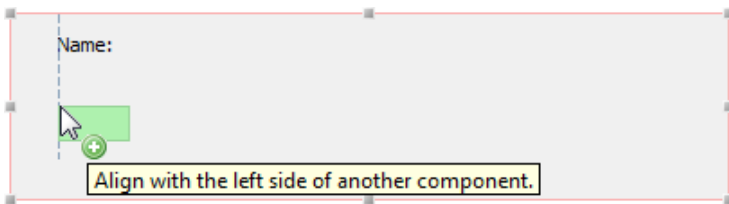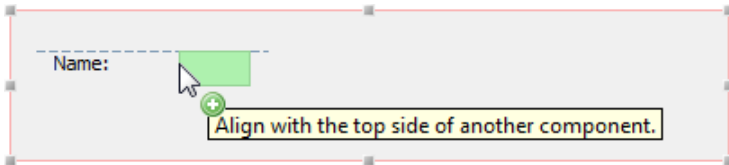
Indentation alignment is a special alignment relationship in which one component is located below another and offset slightly to the right.
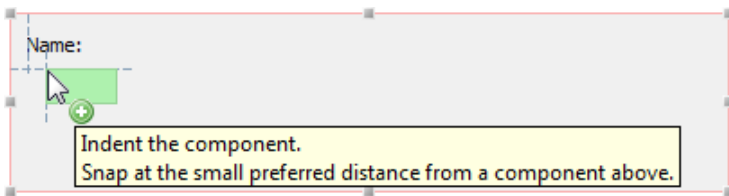
Anchoring indicators

Anchoring indicators appear when components have snapped into position. They illustrate the alignment and relationship among components.
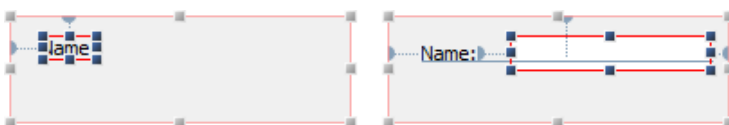
Anchors connecting components to their container or to adjacent components are represented by small semi-circular indicators with dashed lines.

Commands

The designer context menu provides following GroupLayout specific commands:

| Command	Description
Align in column/row	Aligns the selected components left/right/top/bottom/center in column/row.
Align	Aligns the selected components left/right/top/bottom.
Anchor	Changes the anchoring of the selected components. A component is usually horizontally anchored left/right and vertically anchored top/bottom. Anchoring connects a component to a container edge or a neighborhood component edge.
Horizontal Auto Resizing	Makes the selected components resize horizontally at runtime if the container size changes.
Vertical Auto Resizing	Makes the selected components resize vertically at runtime if the container size changes.
Same Width	Makes the selected components all the same width. If one of the selected components is already in a group of "Same Width" components, the other components are added to the existing group. To remove components from a group, select them and then execute this command. Grouped components are marked with a small indicator.
Same Height	Makes the selected components all the same height. See "Save Width" command for more details.
Set Default Size	Makes the selected components have its default size.
Space Around Component	Changes the empty space around the selected component.
Duplicate	Duplicates the selected components and places the new components below the original components. Use **Ctrl+Left**, **Ctrl+Right**, **Ctrl+Up** or **Ctrl+Down** keys to place the duplicated components left, right, above or below the original components.

Layout manager properties

A container with this layout manager has following layout manager properties:

Property Name	Description	Default
honors visibility	Specifies whether component visibility is considered when positioning and sizing components. If true, non-visible components are not treated as part of the layout. If false, components are positioned and sized regardless of visibility.	true

Layout constraints properties

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

Property Name	Description	Default
horizontal size	Specifies the component's horizontal size in pixel or Default. If set to Default, the component's preferred width is used.	Default
vertical size	Specifies the component's vertical size in pixel or Default. If set to Default, the component's preferred height is used.	Default

Property Name	Description	Default
horizontal resizable	Specifies whether the component is horizontal resizable.	false
vertical resizable	Specifies whether the component is vertical resizable.	false
top space	Specifies the top empty space.	
left space	Specifies the left empty space.	
bottom space	Specifies the bottom empty space.	
right space	Specifies the right empty space.	
top space resizable	Specifies whether the top empty space is vertical resizable.	false
left space resizable	Specifies whether the left empty space is horizontal resizable.	false
bottom space resizable	Specifies whether the bottom empty space is vertical resizable.	false
right space resizable	Specifies whether the right empty space is horizontal resizable.	false

8.9   HorizontalLayout (SwingX)

The horizontal layout manager places components horizontally. The components are stretched vertically to the height of the container. The components will not wrap as in FlowLayout.

HorizontalLayout is part of the SwingX open source project and **not** part of the standard Java distribution. You must ship an additional library with your application. The JFormDesigner distribution does not include the SwingX library. For downloads, documentation and tutorials, visit swingx.java.net (or www.swinglabs.org).

Layout manager properties

A container with this layout manager has following layout manager properties:

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

8.10   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.
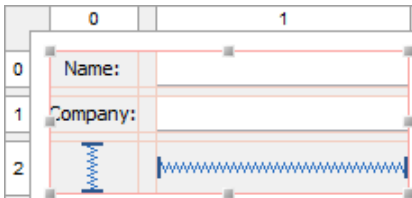
Use the column and row headers to insert or delete columns/rows and change column/row properties. Use horizontal and vertical spacers, which are available in the Palette, to define space between components.

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 manager properties

A container with this layout manager has following layout manager properties:

Property Name	Description	Default
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.	-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.	-1
same size horizontally	If true, all columns get the same width.	false
same size vertically	If true, all rows get the same height.	false
margin	Size of the margin between the containers border and its contents.	0,0,0,0

Layout constraints properties

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

Property Name	Description	Default
grid x	Specifies the component's horizontal grid origin (column index).	0
grid y	Specifies the component's vertical grid origin (row index).	0
grid width	Specifies the component's horizontal grid extend (number of columns).	1

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

8.11   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.
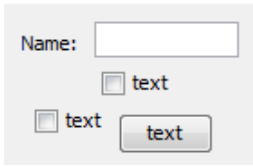
It is useful for making quick prototypes. But it is not recommended for production because it is not portable. The fixed locations an sizes do not change with the environment (e.g. different fonts on various platforms).

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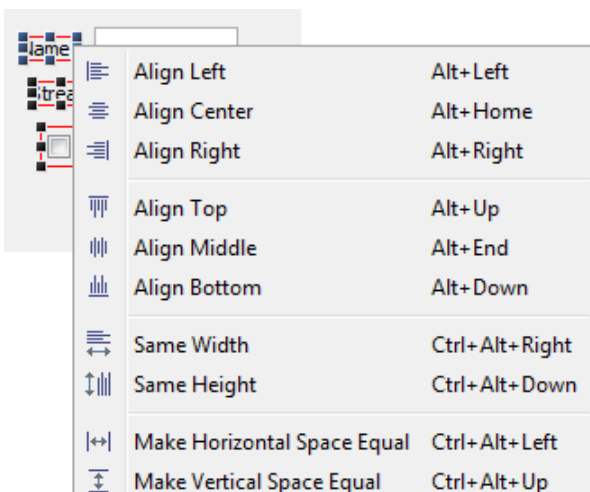
The dark blue handles in the above screenshot indicate the first selected component.

Command	Description
⊫ Align Left	Line up the left edges of the selected components with the left edge of the first selected component.

	Command	Description
≡	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 manager properties

A container with this layout manager has following layout manager properties:

Property Name	Description	Default
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.	true

Layout constraints properties

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

Property Name	Description	Default
x	The x coordinate of the component relative to the left corner of the container.	0
y	The y coordinate of the component relative to the upper corner of the container.	0
width	The width of the component in pixel or Preferred. If set to Preferred, the component's preferred width is used.	Preferred
height	The height of the component in pixel or Preferred. If set to Preferred, the component's preferred width is used.	Preferred

8.12   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.
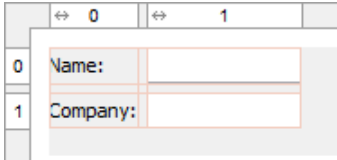
Use the column and row headers to insert or delete columns/rows and change column/row properties.

> 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.java.net.
>
> The API documentation is also available here: doc.formdev.com/tablelayout/.
>
> **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 manager properties

A container with this layout manager has following layout manager properties:

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

Column/row properties

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

Field	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.

Layout constraints properties

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

Property Name	Description	Default
grid x	Specifies the component's horizontal grid origin (column index).	0
grid y	Specifies the component's vertical grid origin (row index).	0
grid width	Specifies the component's horizontal grid extend (number of columns).	1
grid height	Specifies the component's vertical grid extend (number of rows).	1
h align	The horizontal alignment of the component within its cell. Possible values: DEFAULT, LEFT, CENTER, RIGHT and FILL.	DEFAULT
v align	The vertical alignment of the component within its cell. Possible values: DEFAULT, TOP, CENTER, BOTTOM and FILL.	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.

8.13   VerticalLayout (SwingX)

The vertical layout manager places components vertically. The components are stretched horizontally to the width of the container.
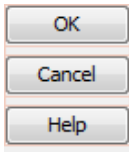
VerticalLayout is part of the SwingX open source project and **not** part of the standard Java distribution. You must ship an additional library with your application. The JFormDesigner distribution does not include the SwingX library. For downloads, documentation and tutorials, visit swingx.java.net (or www.swinglabs.org).

Layout manager properties

A container with this layout manager has following layout manager properties:

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

9   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 if option "Generate initComponentsI18n() method" is enabled in the Localization (Java Code Generator) preferences or "(form)" properties.
initBindings	Used for code that initializes bindings if option "Generate initComponentBindings() method" is enabled in the Localization (Java Code Generator) preferences or "(form)" properties.

9.1 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 in the Code Generation category 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.
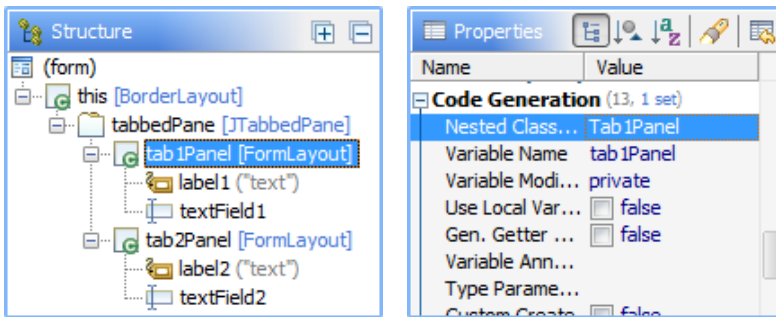
Components having a nested class are marked with a ⊕ overlay symbol in the Structure view.

Example source code:

```java
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

    //---- nested class Tab1Panel ---------------------------------------------------

    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
}

//---- nested class Tab2Panel ---------------------------------------------------

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 in the Code Generation category, 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.

9.2 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 *${component_initialization}* used in other templates. Contains a method named `initComponents`. Invoke this method from your code to instantiate the components of your form. Feel free to change the method name if you don't like it.
Component I18n Initialization	Used for code that initializes localized component properties if option "Generate initComponentsI18n() method" is enabled in the Localization (Java Code Generator) preferences or "(form)" properties.
Component Binding Initialization	Used for code that initializes bindings if option "Generate initComponentBindings() method" is enabled in the Localization (Java Code Generator) preferences or "(form)" properties.
Variables Declaration	Replaces the variable *${variables_declaration}* used in other templates.
java.awt.Dialog	Used 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.
java.awt.Frame	Used for classes derived from `java.awt.Frame`. Equal to the "Class" template, but necessary because `java.awt.Frame` extends `java.awt.Window`, which has its own template and a constructor that is not compatible with `java.awt.Frame`.
java.awt.Window	Used for classes derived from `java.awt.Window`. Compared to the "Class" template, this has a special constructor, which are necessary for `java.awt.Window` 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

10   Command Line Tool

The command-line tool allows you to run some commands (e.g. Java code generation) on many forms.

Available commands

- **Java Code Generation**: Usually its not necessary to run the Java code generator from command-line because the Java code is automatically generated and updated while editing a form in JFormDesigner. However in rare cases it is useful to re-generate the Java code of JFormDesigner forms. E.g. if you want upgrade to JGoodies FormLayout 1.2, which introduced a new much shorter syntax for encoded column and row specifications.
- **Externalize strings**: If you have to localize many existing non-localized forms, then this command does the job very quickly.
- **Convert .jfd file format**: Since version 5.1, JFormDesigner supports the compact, easy-to-merge and fast-to-load persistence format JFDML. This command allows you to convert all your .jfd files from XML to JFDML and benefit from the new format.

Requirements

You need an installation of the JFormDesigner stand-alone edition. If you usually use one of the IDE plug-ins, then simply download the stand-alone edition and install it.

Preferences

To specify preferences for the command-line tool, you should create a stand-alone edition project, enable and set project specific settings and pass the project .jfdproj file to the command-line tool.

If you usually use the JFormDesigner stand-alone edition and already have a .jfdproj file, then you can use it for the command-line tool. Otherwise start the JFormDesigner stand-alone edition and create a new project.

If you don't use a project, then the command-line tool uses the preferences store of the stand-alone edition. If you use one of the IDE plug-ins of JFormDesigner, you have to start the stand-alone edition and set the necessary preferences. To transfer JFormDesigner preferences from an IDE to the stand-alone edition, you can use the **Import** and **Export** buttons in the Preferences dialogs. Make sure that the Code Style preferences are correct because they are not transfered from the IDE.

Command Line Syntax

Launch the command-line tool as follows, where [ ] means optional arguments and arguments in *italics* must be provided by you.

```
java -classpath <jfd-install>/lib/JFormDesigner.jar
    com.jformdesigner.application.CommandLineMain
    [--generate|--i18n-externalize|--convert-jfd]
    [--dry-run] [--verbose|-v] [--recursive|-r]
    [<command-specific-options>]
    [<project-path>/MyProject.jfdproj]
    <folder> or <path>/MyForm1.jfd [...]
```

Option	Description
-classpath *<jfd-install>*/lib/JFormDesigner.jar	Specifies the JAR that contains the command-line tool. This is a standard argument of the Java application launcher.
com.jformdesigner.application.CommandLineMain	The class name of the command-line tool.

Option	Description
--generate	Generate Java code for the given forms or folders.
--i18n-externalize	Externalize strings in the given forms or folders. This requires that you've specified Source Folders in the used project.
--convert-jfd	Convert the given .jfd files to another format.
--dry-run	Execute the given command, but do not save modifications. Only shows what would happen. This option enables --verbose.
--verbose or -v	Prints file names of processed .jfd and .java files to the console.
--recursive or -r	Recursively process folders.
--bundle-name=*<bundleName>*	Only used for --i18n-externalize. The resource bundle name used to store strings. You can use variables {package} (package name of form) and {basename} (basename of form). Default is "{package}.Bundle", which creates Bundle.properties in same package as the form. This option is ignored when processing already localized forms.
--key-prefix=*<keyPrefix>*	Only used for --i18n-externalize. The prefix for generated key. You can use variable {basename} (basename of form). Default is "{basename}". This option is ignored when processing already localized forms.
--auto-externalize=*<true|false>*	Only used for --i18n-externalize. Set the auto-externalize option in the processed forms. Default is true.
--format=<JFDML|XML>	Only used for --convert-jfd. The target format into which the .jfd files will be converted. Default is "JFDML".
--encoding=<encoding>	Only used for --convert-jfd. The encoding used to store JFDML content. See java.nio.charset.Charset for supported encodings. Defaults is "UTF-8".
--header-comment=<headerComment>	Only used for --convert-jfd. A comment that is stored in the header of the converted .jfd files. May contain "\n", which is converted to real newline character.
<project-path>/MyProject.jfdproj	Optional JFormDesigner stand-alone edition project used to extend the classpath and to specify other preferences. Useful when using custom components.
<folder> or *<path>/MyForm1.jfd* [...]	List of folders or .jfd files. If a folder is specified, all .jfd files in the folder are processed.

The options and parameters are processed in the order they are passed. An option is always used for subsequent parameters, but not for preceding ones. E.g. "src1 --recursive src2" processes src2 recursively, but not src1. It is also possible to specify options or projects more than once. E.g. "project1.jfdproj src1 project2.jfdproj src2" uses project1.jfdproj for src1 and project2.jfdproj for src2.

Using custom components

If you're using custom components (JavaBeans) in your forms, it is necessary to tell the command-line tool the classpath of your components, because e.g the code generator needs to load the classes of custom components. There are two options to specify the classpath for your custom components:

- JFormDesigner stand-alone edition project: The JARs and folders specified on the Classpath page in the project settings are used by the command-line tool. This is the preferred way is you use the stand-alone edition.

- Classpath of Java application launcher: Simply add your JARs to the -classpath option of the Java application launcher. This is the preferred way if you use Ant (see below) or one of the IDE plug-ins (which don't use JFormDesigner project files).

Examples

Generate code for a single form:

```
cd C:\MyProject
java -classpath C:\ProgramFiles\JFormDesigner\lib\JFormDesigner.jar
    com.jformdesigner.application.CommandLineMain
    --generate src/com/myproject/MyForm1.jfd
```

Generate code for all forms in a project that use custom components:

```
cd C:\MyProject
java -classpath C:\ProgramFiles\JFormDesigner\lib\JFormDesigner.jar;classes;swingx.jar
    com.jformdesigner.application.CommandLineMain
    --generate --recursive src
```

Externalize strings in all forms of the *src* folder and use one bundle file per form and no key prefix:

```
cd C:\MyProject
java -classpath C:\ProgramFiles\JFormDesigner\lib\JFormDesigner.jar
    com.jformdesigner.application.CommandLineMain
    --i18n-externalize --recursive
    --bundle-name={package}.{basename} --key-prefix=
    MyProject.jfdproj src
```

Ant

Although we don't provide a special task for Ant, it is easy to invoke the JFormDesigner command-line tool from an Ant script. The <classpath> element makes it easy to specify JARs and folders of custom components.

```
<property name="command_line_html__jfd-install-dir" value="C:/Program Files/JFormDesigner"/>

<java classname="command_line_html__com.jformdesigner.application.CommandLineMain"
      fork="true" failonerror="true" logError="true">
   <classpath>
      <pathelement location="${jfd-install-dir}/lib/JFormDesigner.jar"/>
      <pathelement location="myLibrary.jar"/>
   </classpath>
   <arg value="--generate"/>
   <arg value="--recursive"/>
   <arg value="src"/>
</java>
```

11   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 or in the Project settings 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` and the documentation is in `jfd-loader-javadoc.zip`.
>
> The API documentation is also available here: doc.formdev.com/jfd-loader/.

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.

```java
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();
    }
}
```

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

12 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:

- is public and not abstract
- has a public "null" constructor (without parameters)
- has properties defined by public 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.

You can also use BeanInfo Annotations to specify these attributes without the pain of implementing BeanInfo classes.

For examples using BeanInfo Annotations, example implementations of BeanInfo classes and PropertyEditors, take a look at the examples.

BeanDescriptor Attributes

Following attributes are supported in BeanDescriptor:

Attribute Name	Description
isContainer	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	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	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.

Attribute Name	Description
	`beanDesc.setValue("layoutManager", BorderLayout.class);`

persistenceDelegate	Specifies an instance of a class, which extends `java.beans.PersistenceDelegate`, that can be used to persist an instance of the bean. E.g.
	
	`beanDesc.setValue("persistenceDelegate",`
	`    new MyBeanPersistenceDelegate());`

PropertyDescriptor Attributes

Following attributes are supported in PropertyDescriptor:

Attribute Name	Description
category	Specifies the property category to which the property belongs. JFormDesigner adds the specified category to the Properties view. The value must be a `String`. `propDesc.setValue("category", "My Properties");`

enumerationValues	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", JSlider.HORIZONTAL, "JSlider.HORIZONTAL",`
	`    "vertical",   JSlider.VERTICAL,   "JSlider.VERTICAL" });`

extraPersistenceDelegates	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 the property value. Use this attribute to specify persistence delegates for classes that are referenced by the 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	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	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	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.

Attribute Name	Description
	```propDesc.setValue("notNull", Boolean.TRUE);```
notRestoreDefault	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);```
persistenceDelegate	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	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	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	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 automatically reloads classes of custom JavaBeans when changed. So you can change the source code of used custom JavaBeans, compile them in your IDE and use them in JFormDesigner immediately without restarting.

You can also manually reload classes:

- **Stand-alone**: Select **View > Refresh Designer** from the menu or press **F5**.
- **IDE plug-ins**: Click the **Refresh Designer** 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.

Unsupported standard components

- all AWT components

13   Annotations

The @BeanInfo and @PropertyDesc annotations make it very easy to specifying BeanInfo information directly in the custom component. Its no longer necessary to implement extra BeanInfo classes. This drastically reduces time and code needed to create BeanInfo information.

> When using the JFormDesigner annotations, you have to add the library jfd-annotations.jar (from redist folder) to the build path of your project (necessary for the Java compiler). The documentation is in jfd-annotations-javadoc.zip. It is **not** necessary to distribute jfd-annotations.jar with your application.
>
> The API documentation is also available here: doc.formdev.com/jfd-annotations/

@BeanInfo

This annotation can be used to specify additional information for constructing a BeanInfo class and its BeanDescriptor.

Example for specifying a description, an icon, property display names and flags, and property categories:

```
@BeanInfo(
    description="My Bean",
    icon="MyBean.gif",
    properties={
        @PropertyDesc(name="magnitude", displayName="magnitude (in %)", preferred=true)
        @PropertyDesc(name="enabled", expert=true)
    },
    categories={
        @Category(name="Sizes", properties={"preferredSize", "minimumSize", "maximumSize"}),
        @Category(name="Colors", properties={"background", "foreground"}),
    }
)
public class MyBean extends JCompoment { ... }
```

Example for a container component that has a content pane:

```
@BeanInfo(isContainer=true, containerDelegate="getContentPane")
public class MyPanel extends JPanel { ... }
```

@PropertyDesc

This annotation can be used to specify additional information for constructing a PropertyDescriptor.

This annotation may be used in a @BeanInfo annotation (see @BeanInfo.properties()) or may be attached to property getter or setter methods. If the getter method of a property is annotated, then the setter method of the same property is not checked for this annotation.

Important: This annotation requires that the @BeanInfo annotation is specified for the bean class. Otherwise this annotation is ignored when specified at methods.

Example for attaching this annotation to a property getter method:

```
@PropertyDesc(displayName="magnitude (in %)", preferred=true)
public int getMagnitude() {
    return magnitude;
}
```

Example for specifying this annotation in a `@BeanInfo` annotation:

```
@BeanInfo(
    properties={
        @PropertyDesc(name="magnitude", displayName="magnitude (in %)", preferred=true)
    }
)
public class MyBean extends JCompoment { ... }
```

@DesignCreate

This annotation can be used to mark a static method that should be invoked by JFormDesigner to create instances of the bean, which are then used in the JFormDesigner Design view. The annotated method must be static, must not have parameters and must return the instance of created bean.

Example for using this annotation to initialize the bean with test data for the Design view:

```
public class MyBean extends JCompoment {
    @DesignCreate
    private static MyBean designCreate() {
        MyBean myBean = new MyBean();
        myBean.setData( new SomeDummyDataForDesigning() );
        return myBean;
    }
    public MyBean() {
        // ...
    }
}
```

14   JGoodies Forms & Looks

JFormDesigner supports and uses software provided by JGoodies Karsten Lentzsch.

The **JGoodies Forms** framework 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`.

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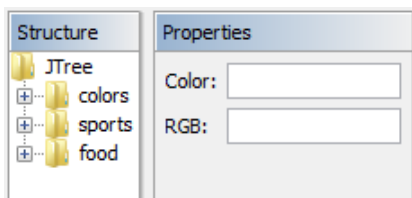
`SimpleInternalFrame` is an Eclipse like frame. `UIFSplitPane` is a subclass of `JSplitPane` that hides the divider border. Use `UIFSplitPane` if you want to put two `SimpleInternalFrame`s into a split pane. See example examples/UIFLitePanel.jfd.

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.

The API documentation is also available here: doc.formdev.com/jgoodies-uif-lite/.

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.

15   Examples & Redistributables

A JFormDesigner installation includes example source code and redistributable files. Because JFormDesigner is available in several editions and each IDE plug-in has its own requirements regarding plug-in directory structure and installation location, the installation location of the examples and redistributables depends on the JFormDesigner edition. The tables below list the locations for each JFormDesigner edition.

Examples

The `examples` folder (or `examples.zip` archive) contains example source code and forms. See `examples/README.html` for details.

Edition	Location
Stand-alone	`<jformdesigner-install>/examples/` *Mac OS X*: `<JFormDesigner.app>/examples/` (right-click on JFormDesigner application and select "Show Package Contents" from the context menu to see contents of `<JFormDesigner.app>`)
Eclipse plug-in	`<eclipse-install>/features/com.jformdesigner_x.x.x/examples.zip` or `<eclipse-install>/dropins/JFormDesigner-x.x-eclipse/features/com.jformdesigner_x.x.x/examples.zip`

NetBeans plug-in	`<netbeans-install>/jformdesigner/examples.zip`
	
	Mac OS X:
	`<NetBeans.app>/Contents/Resources/NetBeans/jformdesigner/examples.zip`
	(right-click on NetBeans application and select "Show Package Contents" from the context menu to see contents of <NetBeans.app>)

IntelliJ IDEA plug-in	`<user-home>/.IntelliJIdeaXX/config/plugins/JFormDesigner/examples.zip` or `<intellij-idea-install>/plugins/JFormDesigner/examples.zip` *Mac OS X*: `<user-home>/Library/Application Support/IntelliJIdeaXX/JFormDesigner/examples.zip`
JBuilder plug-in	`<jbuilder-install>/lib/ext/JFormDesigner/examples.zip`

Redistributables

The `redist` folder contains the JFormDesigner Annotations Library, the JFormDesigner Runtime Library and 3rd party open source files (layout manager, beans binding, etc). See `redist/README.html` for information about licenses.

Edition	Location
Stand-alone	`<jformdesigner-install>/redist/` *Mac OS X*: `<JFormDesigner.app>/redist/` (right-click on JFormDesigner application and select "Show Package Contents" from the context menu to see contents of `<JFormDesigner.app>`)
Eclipse plug-in	`<eclipse-install>/plugins/com.jformdesigner.redist_x.x.x/` or `<eclipse-install>/dropins/JFormDesigner-x.x-eclipse/plugins/com.jformdesigner.redist_x.x.x/`
NetBeans plug-in	`<netbeans-install>/jformdesigner/redist/` *Mac OS X*: `<NetBeans.app>/Contents/Resources/NetBeans/jformdesigner/redist/` (right-click on NetBeans application and select "Show Package Contents" from the context menu to see contents of <NetBeans.app>)

Edition	Location
IntelliJ IDEA plug-in	`<user-home>/.IntelliJIdeaXX/config/plugins/JFormDesigner/redist/` or `<intellij-idea-install>/plugins/JFormDesigner/redist/` *Mac OS X*: `<user-home>/Library/Application Support/IntelliJIdeaXX/JFormDesigner/redist/`
JBuilder plug-in	`<jbuilder-install>/lib/ext/JFormDesigner/redist/`