## Contents

1 Introduction .................................................................................................................................................................................. 2
2 User Interface .................................................................................................................................................................................. 3
  2.1 Menus .................................................................................................................................................................................. 4
  2.2 Toolbars .................................................................................................................................................................................. 6
2.3 Design View .............................................................................................................................................................................. 7
  2.3.1 Headers .............................................................................................................................................................................. 9
  2.3.2 In-place-editing .............................................................................................................................................................. 11
  2.3.3 Keyboard Navigation ....................................................................................................................................................... 12
  2.3.4 Menu Designer ............................................................................................................................................................... 12
  2.3.5 Column/Row Groups ...................................................................................................................................................... 13
  2.3.6 Button Groups ............................................................................................................................................................... 14
  2.3.7 JTabbedPane ................................................................................................................................................................. 16
  2.3.8 Events .............................................................................................................................................................................. 17
2.4 Palette ................................................................................................................................................................................... 19
2.5 Structure View ........................................................................................................................................................................ 22
2.6 Properties View ...................................................................................................................................................................... 23
  2.6.1 Layout Manager Properties ........................................................................................................................................... 25
  2.6.2 Layout Constraints Properties ....................................................................................................................................... 25
  2.6.3 Client Properties ............................................................................................................................................................ 26
  2.6.4 Code Generation Properties ........................................................................................................................................... 26
  2.6.5 Property Editors ........................................................................................................................................................... 29
2.7 Bindings View .......................................................................................................................................................................... 39
2.8 Error Log View ...................................................................................................................................................................... 40
3 Localization ................................................................................................................................................................................ 41
4 Beans Binding (JSR 295) ............................................................................................................................................................ 47
5 Projects ........................................................................................................................................................................................ 52
6 Preferences .................................................................................................................................................................................. 54
7 IDE Integrations ........................................................................................................................................................................... 69
  7.1 Eclipse plug-in ...................................................................................................................................................................... 70
  7.2 IntelliJ IDEA plug-in ............................................................................................................................................................ 74
  7.3 IntelliJ IDEA plug-in ............................................................................................................................................................ 77
  7.4 JDeveloper plug-in ............................................................................................................................................................. 80
8 Layout Managers .......................................................................................................................................................................... 83
  8.1 BorderLayout ....................................................................................................................................................................... 85
  8.2 BoxLayout ............................................................................................................................................................................ 86
  8.3 CardLayout ........................................................................................................................................................................... 87
  8.4 FlowLayout ........................................................................................................................................................................... 88
  8.5 FormLayout (JGoodies) ....................................................................................................................................................... 89
  8.6 GridBagLayout .................................................................................................................................................................... 92
  8.7 GridLayout .......................................................................................................................................................................... 95
  8.8 GroupLayout (Free Design) ................................................................................................................................................... 96
  8.9 HorizontalLayout (SwingX) ............................................................................................................................................... 100
  8.10 IntelliJ IDEA GridLayout .................................................................................................................................................. 101
  8.11 MigLayout ....................................................................................................................................................................... 103
  8.12 null Layout ....................................................................................................................................................................... 108
  8.13 TableLayout .................................................................................................................................................................... 110
  8.14 VerticalLayout (SwingX) .................................................................................................................................................. 112
9 Java Code Generator ................................................................................................................................................................. 113
  9.1 Nested Classes .................................................................................................................................................................... 114
  9.2 Code Templates ................................................................................................................................................................. 116
10 Command Line Tool ................................................................................................................................................................. 117
11 Runtime Library ....................................................................................................................................................................... 121
12 JavaBeans ............................................................................................................................................................................... 123
13 Annotations .............................................................................................................................................................................. 126
14 JGoodies Forms & Looks ...................................................................................................................................................... 128
15 Examples & Redistributables .................................................................................................................................................. 129
1 Introduction

JFormDesigner is a professional GUI designer for Java Swing user interfaces. Its outstanding support for MigLayout, 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 JDeveloper. 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, JDeveloper plug-in or IDE plug-ins.

Key features

- Easy and intuitive to use, powerful and productive
- IDE plug-ins and stand-alone application
- MigLayout support
- 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 (发挥了) in the toolbar to make it 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.

![Menu Items](Image)

### File menu

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>New Project</td>
<td>Creates a new project.</td>
</tr>
<tr>
<td>Open Project</td>
<td>Opens an existing project.</td>
</tr>
<tr>
<td>Reopen Project</td>
<td>Displays a submenu of previously opened projects. Select a project to open it.</td>
</tr>
<tr>
<td>Project Properties</td>
<td>Displays the project properties.</td>
</tr>
<tr>
<td>Close Project</td>
<td>Closes the active project.</td>
</tr>
<tr>
<td>New Form</td>
<td>Creates a new form.</td>
</tr>
<tr>
<td>Open Form</td>
<td>Opens an existing form.</td>
</tr>
<tr>
<td>Reopen Form</td>
<td>Displays a submenu of previously opened forms. Select a form to open it.</td>
</tr>
<tr>
<td>Close</td>
<td>Closes the active form.</td>
</tr>
<tr>
<td>Close All</td>
<td>Closes all open forms.</td>
</tr>
<tr>
<td>Save</td>
<td>Saves the active form and generates the Java source code for the form (if Java Code Generation is enabled in the Preferences).</td>
</tr>
<tr>
<td>Save As</td>
<td>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).</td>
</tr>
<tr>
<td>Save All</td>
<td>Saves all open forms and generates the Java source code for the forms (if Java Code Generation is enabled in the Preferences).</td>
</tr>
<tr>
<td>Import</td>
<td>Imports NetBeans or IntelliJ IDEA form files and creates new JFormDesigner forms. Use File &gt; Save to save the new form in the same folder as the original form file. This also updates the .java file.</td>
</tr>
<tr>
<td>Exit</td>
<td>Exits JFormDesigner. Mac: this item is in the JFormDesigner application menu.</td>
</tr>
</tbody>
</table>

### Edit menu

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undo</td>
<td>Reverses your most recent editing action.</td>
</tr>
<tr>
<td>Redo</td>
<td>Re-applies the editing action that has most recently been reversed by the Undo action.</td>
</tr>
<tr>
<td>Cut</td>
<td>Cuts the selected components to the clipboard.</td>
</tr>
<tr>
<td>Copy</td>
<td>Copies the selected components to the clipboard.</td>
</tr>
<tr>
<td>Paste</td>
<td>Pastes the components in the clipboard to the selected container of the active form.</td>
</tr>
<tr>
<td>Rename</td>
<td>Renames the selected component.</td>
</tr>
<tr>
<td>Delete</td>
<td>Deletes the selected components.</td>
</tr>
</tbody>
</table>

### View menu

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Refresh Designer</td>
<td>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.</td>
</tr>
<tr>
<td>Classic Layout</td>
<td>Shows Properties view below Structure view.</td>
</tr>
<tr>
<td>Wide Layout</td>
<td>Shows Properties and Structure views side by side.</td>
</tr>
</tbody>
</table>
### Form menu

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Form</td>
<td>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.</td>
</tr>
<tr>
<td>Localize</td>
<td>Edit localization settings, resource bundle strings, create new locales or delete locales.</td>
</tr>
<tr>
<td>New Locale</td>
<td>Creates a new locale.</td>
</tr>
<tr>
<td>Delete Locale</td>
<td>Deletes an existing locale.</td>
</tr>
<tr>
<td>Externalize Strings</td>
<td>Moves strings to a resource bundle for localization. Use this command to start localizing existing forms.</td>
</tr>
<tr>
<td>Internalize Strings</td>
<td>Moves strings from a resource bundle into the form and remove the strings from the resource bundle.</td>
</tr>
<tr>
<td>Generate Java Code</td>
<td>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.</td>
</tr>
</tbody>
</table>

### Window menu

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activate Designer</td>
<td>Activates the Design view.</td>
</tr>
<tr>
<td>Activate Structure</td>
<td>Activates the Structure view.</td>
</tr>
<tr>
<td>Activate Properties</td>
<td>Activates the Properties view.</td>
</tr>
<tr>
<td>Activate Bindings</td>
<td>Activates the Bindings view. By default, the Bindings view is not visible.</td>
</tr>
<tr>
<td>Activate Error Log</td>
<td>Activates the Error Log view. By default, the Error Log view is not visible. It automatically appears if an error occurs.</td>
</tr>
<tr>
<td>Next Form</td>
<td>Activates the next form.</td>
</tr>
<tr>
<td>Previous Form</td>
<td>Activates the previous form.</td>
</tr>
<tr>
<td>Preferences</td>
<td>Opens the Preferences dialog. <strong>Mac</strong>: this item is in the JFormDesigner application menu.</td>
</tr>
</tbody>
</table>

### Help menu

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Help Contents</td>
<td>Displays help topics.</td>
</tr>
<tr>
<td>What's New</td>
<td>Displays what's new in the current release.</td>
</tr>
<tr>
<td>Tip of the Day</td>
<td>Displays a list of interesting productivity features.</td>
</tr>
<tr>
<td>Register</td>
<td>Activates your license.</td>
</tr>
<tr>
<td>License</td>
<td>Displays information about your license.</td>
</tr>
<tr>
<td>Check for Updates</td>
<td>Checks whether a newer version of JFormDesigner is available.</td>
</tr>
<tr>
<td>About</td>
<td>Displays information about JFormDesigner and the system properties. <strong>Mac</strong>: this item is in the JFormDesigner application menu.</td>
</tr>
</tbody>
</table>

### Context menus

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

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>New Project</td>
<td>Creates a new project.</td>
</tr>
<tr>
<td>Open Project</td>
<td>Opens an existing project.</td>
</tr>
<tr>
<td>Project Properties</td>
<td>Displays the project properties.</td>
</tr>
<tr>
<td>New Form</td>
<td>Creates a new form.</td>
</tr>
<tr>
<td>Open Form</td>
<td>Opens an existing form.</td>
</tr>
<tr>
<td>Save</td>
<td>Saves the active form and generates the Java source code for the form (if Java Code Generation is enabled in the Preferences).</td>
</tr>
<tr>
<td>Save All</td>
<td>Saves all open forms and generates the Java source code for the forms (if Java Code Generation is enabled in the Preferences).</td>
</tr>
<tr>
<td>Undo</td>
<td>Reverses your most recent editing action.</td>
</tr>
<tr>
<td>Redo</td>
<td>Re-applies the editing action that has most recently been reversed by the Undo action.</td>
</tr>
<tr>
<td>Cut</td>
<td>Cuts the selected components to the clipboard.</td>
</tr>
<tr>
<td>Copy</td>
<td>Copies the selected components to the clipboard.</td>
</tr>
<tr>
<td>Paste</td>
<td>Pastes the components in the clipboard to the selected container of the active form.</td>
</tr>
<tr>
<td>Delete</td>
<td>Deletes the selected components.</td>
</tr>
<tr>
<td>Test Form</td>
<td>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.</td>
</tr>
<tr>
<td>Refresh Designer</td>
<td>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.</td>
</tr>
<tr>
<td>Localize</td>
<td>Edit localization settings, resource bundle strings, create new locales or delete locales.</td>
</tr>
<tr>
<td>Show Bindings View</td>
<td>Shows the Bindings view.</td>
</tr>
<tr>
<td>Generate Java Code</td>
<td>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.</td>
</tr>
<tr>
<td>Help Contents</td>
<td>Displays help topics.</td>
</tr>
<tr>
<td>Classic Layout</td>
<td>Shows Properties view below Structure view.</td>
</tr>
<tr>
<td>Wide Layout</td>
<td>Shows Properties and Structure views side by side.</td>
</tr>
</tbody>
</table>
2.3 Design View

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

![Design View Example](image)

**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.

![Marquee Selection Example](image)

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

### Drag feedback

JFormDesigner provides four types of drag feedback.

![Drag Feedback Examples](image)
The gray figure shows the outline of the dragged components. It always follows the mouse location. The green figure indicates the drop location, the yellow figure indicates a new column/row and red figures indicate occupied areas.

**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 before the right-clicked column/row. To add a column/row after the last one, right-click on the area behind the last column/row.

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

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

Delete columns/rows

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

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

Split columns/rows

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

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

Move columns/rows

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

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

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

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

Header symbols

Following symbols are used in the headers:

<table>
<thead>
<tr>
<th>Column Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>↩</td>
</tr>
<tr>
<td>↤</td>
</tr>
<tr>
<td>→</td>
</tr>
<tr>
<td>⇨</td>
</tr>
<tr>
<td>⇧</td>
</tr>
<tr>
<td>⇩</td>
</tr>
<tr>
<td>⇣</td>
</tr>
<tr>
<td>▷</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Row Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>↑</td>
</tr>
<tr>
<td>↣</td>
</tr>
<tr>
<td>↓</td>
</tr>
<tr>
<td>⇧</td>
</tr>
<tr>
<td>⇩</td>
</tr>
<tr>
<td>⇣</td>
</tr>
<tr>
<td>⇢</td>
</tr>
<tr>
<td>▷</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up</td>
<td>move the selection up</td>
</tr>
<tr>
<td>Down</td>
<td>move the selection down</td>
</tr>
<tr>
<td>Left</td>
<td>move the selection left</td>
</tr>
<tr>
<td>Right</td>
<td>move the selection right</td>
</tr>
<tr>
<td>Home</td>
<td>select the first component</td>
</tr>
<tr>
<td>End</td>
<td>select the last component</td>
</tr>
</tbody>
</table>

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

You can assign a 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.
2.3.5 Column/Row Groups

Column and row groups (MigLayout and FormLayout only) 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 [0 and 2] and columns [1 and 3] 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 (FormLayout only).

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 MigLayout or FormLayout Column/row properties dialog.
2.3.6 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.

2.3.7 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 MigLayout, it's recommended to use Layout Insets. For JGoodies Forms use TABBED_DIALOG_BORDER. Otherwise use an EmptyBorder.

2.3.8 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.

**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 and 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

<table>
<thead>
<tr>
<th>Filter Menu Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Filter</td>
<td>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.</td>
</tr>
<tr>
<td>Show Interfaces</td>
<td>Includes interfaces in the list of matching classes.</td>
</tr>
</tbody>
</table>
**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/8/docs/technotes/guides/jar/jar.html#Per-Entry_Attributes](http://docs.oracle.com/javase/8/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:

- 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.
- 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 an invalid reference.

Toolbar commands

- Expand All
- Collapse All

Expand all nodes in the structure tree.

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 flag 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. Use space, comma or semicolon as separator for multiple property names.

![Properties filter example](image)

Common properties and categories

<table>
<thead>
<tr>
<th>Property/Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>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.</td>
</tr>
<tr>
<td>Class</td>
<td>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).</td>
</tr>
<tr>
<td>Button Group</td>
<td>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).</td>
</tr>
<tr>
<td>Layout Manager</td>
<td>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.</td>
</tr>
<tr>
<td>Layout Constraints</td>
<td>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.</td>
</tr>
<tr>
<td>Bindings</td>
<td>Bindings of the component.</td>
</tr>
<tr>
<td>Events</td>
<td>Events of the component.</td>
</tr>
<tr>
<td>Client Properties</td>
<td>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.</td>
</tr>
<tr>
<td>Code Generation</td>
<td>Code Generation properties of the component.</td>
</tr>
</tbody>
</table>

"(form)" properties

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

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Form file format</td>
<td>The format used to persist the form. See also &quot;Form file format&quot; option in General preferences.</td>
</tr>
<tr>
<td>Set Component Names</td>
<td>If true, invokes java.awt.Component.setName() on all components of the form.</td>
</tr>
</tbody>
</table>
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 of a container that has a MigLayout.

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 (MigLayout, 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 MigLayout.

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:

```java
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.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nested Class Name</td>
<td>The name of the generated nested Java class. See <a href="#">Nested Classes</a> for details.</td>
</tr>
<tr>
<td>Variable Name</td>
<td>The variable name of the component used in the generated Java code. By default, it is equal to the component name.</td>
</tr>
<tr>
<td>Variable Modifiers</td>
<td>The modifiers of the variable generated for the component. Allowed modifiers: <code>public</code>, <code>default</code>, <code>protected</code>, <code>private</code>, <code>static</code> and <code>transient</code>. Default is <code>private</code>.</td>
</tr>
<tr>
<td>Use Local Variable</td>
<td>If true, the variable is declared as local in the initialization method. Otherwise at class level. Default is false.</td>
</tr>
<tr>
<td>Gen. Getter Method</td>
<td>If true, generate a public getter method for the component. Default is false.</td>
</tr>
<tr>
<td>Variable Annotations (Java 5)</td>
<td>Annotions of component variable (Java 5).</td>
</tr>
<tr>
<td>Type Parameters (Java 5)</td>
<td>Parameters of component type (Java 5). E.g. <code>MyTypedBean&lt;String&gt;</code>.</td>
</tr>
<tr>
<td>Custom Create</td>
<td>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().</td>
</tr>
<tr>
<td>Custom Creation Code</td>
<td>Custom code for creation of the component.</td>
</tr>
<tr>
<td>Pre-Creation Code</td>
<td>Code included before creation of the component.</td>
</tr>
<tr>
<td>Post-Creation Code</td>
<td>Code included after creation of the component.</td>
</tr>
<tr>
<td>Pre-Initialization Code</td>
<td>Code included before initialization of the component.</td>
</tr>
<tr>
<td>Post-Initialization Code</td>
<td>Code included after initialization of the component.</td>
</tr>
</tbody>
</table>
"(form)" properties

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

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generate Java Source Code</td>
<td>If true, generate Java source code for the form. Defaults to &quot;Generate Java source code&quot; option in the Java Code Generator preferences.</td>
</tr>
<tr>
<td>Default Variable Modifiers</td>
<td>The default modifiers of the variables generated for components. Allowed modifiers: public, default, protected, private, static and transient. Default is private.</td>
</tr>
<tr>
<td>Default Use Local Variable</td>
<td>If true, the component variables are declared as local in the initialization method. Otherwise at class level. Default is false.</td>
</tr>
<tr>
<td>Default Event Handler Modifiers</td>
<td>The default modifiers used when generating event handler methods. Allowed modifiers: public, default, protected, private, final and static. Default is private.</td>
</tr>
<tr>
<td>Member Variable Prefix</td>
<td>Prefix used for component member variables. E.g. &quot;m_&quot;</td>
</tr>
<tr>
<td>Use 'this' for member variables</td>
<td>If enabled, the code generator inserts 'this.' before all member variables. E.g. this.nameLabel.setText(&quot;Name: &quot;);</td>
</tr>
<tr>
<td>I18n Initialization Method</td>
<td>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.</td>
</tr>
<tr>
<td>I18n 'getBundle' Template</td>
<td>Template used by code generator for getting a resource bundle. Default is ResourceBundle.getBundle(&quot;${bundleName}&quot;);</td>
</tr>
<tr>
<td>I18n 'getString' Template</td>
<td>Template used by code generator for getting a string from a resource bundle. Default is ${bundle}.getString(&quot;${key}&quot;);</td>
</tr>
<tr>
<td>I18n 'translate' Template</td>
<td>Template used by code generator to translate a string into another locale (e.g. &quot;tr(${value})&quot;). E.g. for Gettext Commons.</td>
</tr>
<tr>
<td>I18n Key Constants Class</td>
<td>The name of a class that contains constants for resource keys.</td>
</tr>
<tr>
<td>Binding Initialization Method</td>
<td>If enabled, the code generator puts the code to create bindings into a method initComponentBindings().</td>
</tr>
<tr>
<td>MigLayout: API Constraints</td>
<td>If enabled, then MigLayout API is used to create constraints. Otherwise strings are used.</td>
</tr>
</tbody>
</table>
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:

- `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.

![ActionMap Editor](image)

Border (javax.swing)

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

![Border Editor](image)

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.

![Dimension Window](image)

**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.

![Font Window](image)

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

![Custom Font Editor](image)
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.

![Icon Custom Editor](image1)

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.

![InputMap Custom Editor](image2)
Insets (java.awt)

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

![Insets Editor](image)

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).

![KeyStroke Editor](image)

ListModel (javax.swing)

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

![ListModel Editor](image)
Object (java.lang)

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

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.

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.

The above table is editable. Select a cell and start typing. Use RETURN to commit, ESC to cancel and arrow keys to move selection.

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 (ıldığı) in the toolbar.

<table>
<thead>
<tr>
<th>Source</th>
<th>Target</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>bindingGroup</td>
<td></td>
<td></td>
</tr>
<tr>
<td>this - task.title</td>
<td>titleField - text</td>
<td></td>
</tr>
<tr>
<td>this - task.description</td>
<td>descriptionField - text</td>
<td></td>
</tr>
<tr>
<td>this - categories</td>
<td>categoryField - elements</td>
<td></td>
</tr>
<tr>
<td>this - task.category</td>
<td>categoryField - selectedItem</td>
<td></td>
</tr>
<tr>
<td>enablementBindingGroup</td>
<td></td>
<td></td>
</tr>
<tr>
<td>this - $(task != null)</td>
<td>titleField - editable</td>
<td></td>
</tr>
<tr>
<td>titleField - editable</td>
<td>titleLabel - enabled</td>
<td></td>
</tr>
<tr>
<td>descriptionField - editable</td>
<td>descriptionLabel - enabled</td>
<td></td>
</tr>
<tr>
<td>categoryField - enabled</td>
<td>categoryLabel - enabled</td>
<td></td>
</tr>
</tbody>
</table>

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)
- 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 thrown 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

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy Log</td>
<td>Copies all log records to the clipboard.</td>
</tr>
<tr>
<td>Clear Log</td>
<td>Clears the log.</td>
</tr>
<tr>
<td>Properties</td>
<td>Displays the properties of the selected log record in a dialog (see below).</td>
</tr>
<tr>
<td>Close</td>
<td>Closes the Error Log view.</td>
</tr>
</tbody>
</table>

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

![Error Properties](image)

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-edit** 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 1 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 2, you change it in the current locale. Small flags 3 in front of property values in the Properties view indicates that the string is localized (stored in a properties file).

![Image of JFormDesigner interface showing localization and in-place-editing features.](image-url)
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 (shown) 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 (⌘N) from the toolbar 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 (⌘D) from the toolbar 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** (F) 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 (⌘R)** 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 flag 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.

Maven Central Repository: groupid: org.jdesktop artifactId: beansbinding version: 1.2.1

The API documentation is available here: doc.formdev.com/beansbinding/.

The Bindings view  caractère un good overview of all bindings in the form. The Show Bindings View button  caractère makes this view visible. The Bindings property category  caractère in the Properties view shows the bindings of the selected component and you can add (+), edit (=) and remove (−) bindings. Small arrows  caractère indicate that the property is bound. Binding groups are also shown in the Structure view  caractère. The Binding palette category  caractère 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 (+/−) 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 (−) 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

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>The source object.</td>
</tr>
<tr>
<td>Source path</td>
<td>The path (or expression) that identifies the source property.</td>
</tr>
<tr>
<td>Detail path</td>
<td>The path (or expression) that determines what is displayed to the user in the target JList. (only if target is JList.elements)</td>
</tr>
<tr>
<td>Target</td>
<td>The target object.</td>
</tr>
<tr>
<td>Target path</td>
<td>The path (or expression) that identifies the target property.</td>
</tr>
<tr>
<td>Update strategy</td>
<td>Specifies how the properties are kept synchronized. Possible values: &quot;Always sync (read-write)&quot;, &quot;Only read from source (read-only)&quot; and &quot;Read once from source (read-once)&quot;.</td>
</tr>
<tr>
<td>Update source</td>
<td>Specifies when the source is updated from the target. Possible values: &quot;While typing&quot;, &quot;On focus lost&quot; and &quot;On focus lost or Enter key pressed&quot;. (only if target is JTextComponent.text)</td>
</tr>
<tr>
<td>Ignore adjusting</td>
<td>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))</td>
</tr>
</tbody>
</table>
**Advanced tab**

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

**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.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Editable</td>
<td>Specifies whether the table cells are editable or not.</td>
</tr>
<tr>
<td>Columns</td>
<td>The column bindings. The Source Path identifies the source property in &lt;E&gt;. The Column Name is shown in the JTable column header. Each column binding may have its own Converter, Validator and Alternative Values.</td>
</tr>
</tbody>
</table>
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**: `&&`, `||`, `&`, `|`, `not`, `!`
- **Relational**: `==`, `!=`, `<`, `>`, `<=`, `>=`, `le`, `ge`
- **Empty**: `empty`
- **Conditional**: `A ? B : C`

**EL expression examples:**

<table>
<thead>
<tr>
<th>EL expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>${task.title}</code></td>
<td>The <code>title</code> property of an object's <code>task</code> property.</td>
</tr>
<tr>
<td><code>${firstName} ${lastName}</code></td>
<td>Concatenation of <code>firstName</code> and <code>lastName</code> properties.</td>
</tr>
<tr>
<td><code>${mother.age &gt; 65}</code></td>
<td><code>true</code> if mother is older than 65, <code>false</code> otherwise.</td>
</tr>
<tr>
<td><code>${image.width * image.height}</code></td>
<td>Computes the number of pixels of an image.</td>
</tr>
<tr>
<td><code>${image.width * image.height * 4}</code></td>
<td>Computes the number of bytes of an 32 bit image.</td>
</tr>
</tbody>
</table>

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

- `and`, `or`, `not`, `div`, `mod`
- `eq`, `ne`, `lt`, `gt`, `le`, `ge`
- `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:

- MigLayout
- FormLayout (JGoodies)
- GridBagLayout
- 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 to 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, expand the "Other Settings" node and then select the "JFormDesigner" page.
- **JDeveloper plug-in**: The JFormDesigner preferences are fully integrated into the JDeveloper preferences dialog. Select Tools > Preferences from the menu to open it.

**Pages**

- General
- MigLayout
- 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
- 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 import/export 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.
- **JDeveloper 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.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Animate layout changes in Design view</td>
<td>If enabled, changes to the layout in the Design view are done animated.</td>
<td>On</td>
</tr>
<tr>
<td>Animation speed</td>
<td>The speed of the animation.</td>
<td>default</td>
</tr>
<tr>
<td>Buffer Design view in video memory</td>
<td>If enabled, parts of the Design view are buffered in the video memory of the graphics card to improve painting speed.</td>
<td>On</td>
</tr>
<tr>
<td>Undo history size</td>
<td>The maximum number of steps in the undo history of a form.</td>
<td>1000</td>
</tr>
<tr>
<td>Form file format</td>
<td>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 &quot;(form)&quot; node in the Structure view and change the &quot;Form file format&quot; property in the Properties view. Or use the JFormDesigner command-line tool to convert the format of many forms.</td>
<td>JFDML</td>
</tr>
</tbody>
</table>
**MigLayout**

On this page, you can specify MigLayout related options.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layout constraints</td>
<td>The layout constraints used for new forms/containers.</td>
<td>hidemode 3</td>
</tr>
<tr>
<td>Column constraints</td>
<td>The column constraints used for new columns.</td>
<td>fill</td>
</tr>
<tr>
<td>Row constraints</td>
<td>The row constraints used for new rows.</td>
<td></td>
</tr>
</tbody>
</table>

**FormLayout (JGoodies)**

On this page, you can specify FormLayout related options.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatically insert/remove gap columns/rows</td>
<td>If enabled, JFormDesigner automatically inserts/removes gap columns/rows.</td>
<td>On</td>
</tr>
<tr>
<td>JGoodies Forms version</td>
<td>Required JGoodies Forms version for the created forms.</td>
<td>auto-detect</td>
</tr>
<tr>
<td>Column/row templates for new columns/rows</td>
<td>Here you can specify the column and row templates that should be used when new columns or rows are inserted.</td>
<td></td>
</tr>
</tbody>
</table>
### Custom column/row templates

![Add Custom Column/Row Template](image)

#### Specify the custom column/row template information.

**Display name:** my line gap  
**Identifier:** mylinegap  
**Use for:**  
- columns  
- rows  
- both  
- gaps  

**Default alignment**  
- left  
- center  
- right  
- fill

**Size**  
- default  
- preferred  
- minimum  
- constant  
- minimum  
- maximum  

**Resize behavior**  
- none  
- grow

**Java code (optional)**  
- Column code:  
- Row code:

---

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Column</td>
<td>The column template used for new columns.</td>
<td>default</td>
</tr>
<tr>
<td>Column gap</td>
<td>The column template used for new gap columns.</td>
<td>label component gap</td>
</tr>
<tr>
<td>Row</td>
<td>The row template used for new rows.</td>
<td>default</td>
</tr>
<tr>
<td>Row gap</td>
<td>The row template used for new gap rows.</td>
<td>line gap</td>
</tr>
<tr>
<td>Custom column/row templates</td>
<td>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 &quot;LayoutMap Initialization Code&quot; link.</td>
<td></td>
</tr>
</tbody>
</table>

### Option  

<table>
<thead>
<tr>
<th>Description</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display name</td>
<td>The display name is used within JFormDesigner whenever the template is shown in combo boxes or popup menus.</td>
<td></td>
</tr>
<tr>
<td>Identifier</td>
<td>The (unique) identifier is stored in form files. Choose a short string. Only letters and digits are allowed.</td>
<td></td>
</tr>
<tr>
<td>Use for</td>
<td>Specifies whether the template should be used for columns, rows or both. Also specifies whether it represents a gap column/row.</td>
<td></td>
</tr>
<tr>
<td>Default alignment</td>
<td>The default alignment of the components within a column/row. Used if the value of the component constraint properties &quot;h align&quot; or &quot;v align&quot; are set to DEFAULT.</td>
<td></td>
</tr>
<tr>
<td>Size</td>
<td>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).</td>
<td></td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-------------------------</td>
<td>--------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>Resize behavior</td>
<td>The resize weight of the column/row.</td>
<td></td>
</tr>
<tr>
<td>Java code</td>
<td>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.</td>
<td></td>
</tr>
</tbody>
</table>

**GridBagLayout**

On this page, you can specify GridBagLayout related options.

![Preferences](image)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default properties for new columns/rows</td>
<td>Here you can specify the column and row properties that should be used when new columns or rows are inserted.</td>
</tr>
<tr>
<td>Column</td>
<td>The column properties used for new columns.</td>
</tr>
<tr>
<td>Row</td>
<td>The row properties used for new rows.</td>
</tr>
</tbody>
</table>

**null Layout**

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

![Preferences](image)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Snap to grid</td>
<td>If enabled, snap to the grid specified below when moving or resizing a component in null layout.</td>
</tr>
<tr>
<td>Grid X step</td>
<td>The horizontal step size of the grid.</td>
</tr>
<tr>
<td>Grid Y step</td>
<td>The vertical step size of the grid.</td>
</tr>
</tbody>
</table>
## Localization

On this page, you can specify localization related options.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rename resource keys when renaming components</td>
<td>If enabled, auto-rename resource keys when renaming components and the resource key contains the old component name.</td>
<td>On</td>
</tr>
<tr>
<td>Copy localized messages when copying components</td>
<td>If enabled, duplicate localized strings in all locales when copying components.</td>
<td>On</td>
</tr>
<tr>
<td>Delete localized messages when deleting components</td>
<td>If enabled, auto-delete localized strings, that were used by the deleted components, from all locales.</td>
<td>On</td>
</tr>
<tr>
<td>Delete localized messages when internalizing strings</td>
<td>If enabled, auto-delete localized strings, that were internalized, from all locales.</td>
<td>On</td>
</tr>
<tr>
<td>Delete messages only if key prefix is equal to form's key prefix</td>
<td>If enabled, messages will be auto-deleted only if their key prefix is equal to the key prefix of the form.</td>
<td>On</td>
</tr>
<tr>
<td>Insert new messages</td>
<td>Specifies where new messages will be inserted into properties files. &quot;next to similar keys&quot; inserts new messages next to other similar keys so that messages that belong together are automatically at the same location in the properties file. &quot;at the end of the properties file&quot; always appends new messages to the end of the properties file.</td>
<td>next to similar keys (ascending order)</td>
</tr>
<tr>
<td>Format used for generated keys</td>
<td>Format used when generating a resource key.</td>
<td>$(componentName)${sep}${propertyName}</td>
</tr>
<tr>
<td>Separator used for generated keys</td>
<td>Separator used when generating a resource key.</td>
<td>.</td>
</tr>
<tr>
<td>Template for properties files</td>
<td>Template used when creating new properties files.</td>
<td></td>
</tr>
<tr>
<td>Exclude properties from externalization</td>
<td>Specify properties that should be excluded from externalization. Useful when using auto-externalization to ensure that some string property is not externalized.</td>
<td></td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
<td>Default</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>Jar path</td>
<td>Full path name of the jar file that contains the look and feel classes. Use the <strong>Browse</strong> button to select a jar.</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Name of the look and feel used in the look and feel combo box in the <strong>Toolbar</strong>.</td>
<td></td>
</tr>
<tr>
<td>Class name</td>
<td>Class name of the look and feel class (derived from <code>javax.swing.LookAndFeel</code>).</td>
<td></td>
</tr>
<tr>
<td>License code</td>
<td>License code for the commercial <strong>Alloy Look and Feel</strong>.</td>
<td></td>
</tr>
</tbody>
</table>

**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 not supported.
Java Code Generator

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

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generate Java source code</td>
<td>If enabled, JFormDesigner generates Java source code when you save a form.</td>
<td>On</td>
</tr>
<tr>
<td>Source compatibility</td>
<td>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.).</td>
<td>Stand-alone: use JRE version IDE plug-ins: use project setting</td>
</tr>
<tr>
<td>Explicit imports</td>
<td>If enabled, the code generator adds explicit import statements (without <code>*</code>) for used classes.</td>
<td>Off</td>
</tr>
<tr>
<td>Container blocks</td>
<td>If enabled, the code generator puts the initialization code for each container into a block (enclosed in curly braces).</td>
<td>On</td>
</tr>
<tr>
<td>Comments</td>
<td>If enabled, the code generator puts a comment line above the initialization code for each component.</td>
<td>On</td>
</tr>
<tr>
<td>Set component names</td>
<td>If enabled, the code generator inserts <code>java.awt.Component.setName()</code> statements for all components of the form.</td>
<td>Off</td>
</tr>
<tr>
<td>Use Eclipse code formatter</td>
<td>If enabled, the Eclipse code formatter is used to format the generated code. (Eclipse plug-in only)</td>
<td>Off</td>
</tr>
<tr>
<td>Eclipse non-nls tags</td>
<td>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.</td>
<td>Off</td>
</tr>
<tr>
<td>IntelliJ IDEA non-nls tags</td>
<td>If enabled, the code generator appends non-nls comments to lines containing strings. These comments are used by IntelliJ IDEA to denote strings that should not be externalized.</td>
<td>Off</td>
</tr>
<tr>
<td>NetBeans no-i18n tags</td>
<td>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.</td>
<td>Off</td>
</tr>
</tbody>
</table>
### Option | Description | Default
--- | --- | ---
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 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`, `abstract`, `final` and `static` | 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.

![New Template](image)

### Layout Managers (Java Code Generator)

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

#### Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use API constraints in generated code</td>
<td>If enabled, then MigLayout API is used to create constraints. Otherwise strings are used.</td>
<td>Off</td>
</tr>
<tr>
<td>Use PanelBuilder in generated code</td>
<td>If enabled, the PanelBuilder class of JGoodies Forms is used for FormLayout.</td>
<td>Off</td>
</tr>
<tr>
<td>Make anonymous PanelBuilder JPanel transparent</td>
<td>If enabled, the JPanel of the PanelBuilder is made transparent.</td>
<td>Off</td>
</tr>
<tr>
<td>Use empty GridBagConstraints constructor</td>
<td>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.</td>
<td>Off</td>
</tr>
<tr>
<td>GroupLayout Generation Style</td>
<td>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 compatibility.</td>
<td>use source compatibility</td>
</tr>
</tbody>
</table>
### Localization (Java Code Generator)

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

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generate initComponentsI18n() method</td>
<td>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 &quot;(form)&quot; properties.</td>
<td>Off</td>
</tr>
<tr>
<td>'getBundle' template</td>
<td>Template used by code generator for getting a resource bundle.</td>
<td>ResourceBundle.getBundle(${bundleName})</td>
</tr>
<tr>
<td>'getString' template</td>
<td>Template used by code generator for getting a string from a resource bundle.</td>
<td>${bundle}.getString(${key})</td>
</tr>
</tbody>
</table>

### Binding (Java Code Generator)

On this page, you can specify code generation options for Beans Binding (JSR 295).

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generate initComponentBindings() method</td>
<td>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 &quot;(form)&quot; properties.</td>
<td>Off</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indent size</td>
<td>The number of spaces used for one indentation level.</td>
<td>4</td>
</tr>
<tr>
<td>Tab size</td>
<td>The number of spaces that represents one tabulation.</td>
<td>4</td>
</tr>
<tr>
<td>Use tab character</td>
<td>Specifies whether the tab character (\t) is used for indentation or only space characters.</td>
<td>On</td>
</tr>
<tr>
<td>Line separator</td>
<td>The line separator used for newly created .java and .properties files.</td>
<td>Platform default</td>
</tr>
<tr>
<td>Encoding</td>
<td>The character encoding used for reading and writing Java files.</td>
<td>Platform default</td>
</tr>
</tbody>
</table>

**Client Properties**

On this page, you 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.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>The key that identifies the client property.</td>
</tr>
<tr>
<td>Component class(es)</td>
<td>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.</td>
</tr>
<tr>
<td>Value type</td>
<td>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.</td>
</tr>
<tr>
<td>Predefined values</td>
<td>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.</td>
</tr>
<tr>
<td>Property editor class</td>
<td>Optional class name of a property editor that should be used when editing the client property in the Properties view.</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classpath for JavaBeans, which use native libraries</td>
<td>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.</td>
</tr>
<tr>
<td>Native Library Path</td>
<td>Folders used to search for native libraries.</td>
</tr>
</tbody>
</table>

BeanInfo Search Paths

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

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BeanInfo search path</td>
<td>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 <code>java.beans.Introspector</code> and <code>Introspector.setBeanInfoSearchPath()</code> for details.</td>
</tr>
<tr>
<td>Property editor search path</td>
<td>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 <code>java.beans.PropertyEditorManager</code> and <code>PropertyEditorManager.setEditorSearchPath()</code> for details.</td>
</tr>
</tbody>
</table>
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
- JDeveloper 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 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 (MigLayout, 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 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 it 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).

![New JFormDesigner Form dialog](image)

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.

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 and IntelliJ IDEA forms

You can convert existing NetBeans and IntelliJ IDEA 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 (MigLayout, JGoodies Forms, TableLayout, etc) to the project.
- 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.

![NetBeans JFormDesigner Editor](image)

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 it 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 and IntelliJ IDEA forms

You can convert existing NetBeans and IntelliJ IDEA 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 (MigLayout, 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.

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 it 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.

![Code folding example](image)

Convert IntelliJ IDEA and NetBeans forms

You can convert existing IntelliJ IDEA and NetBeans forms to JFormDesigner forms. Right-click on the form file (or any container) and select Convert to JFormDesigner Form.

![Convert form](image)

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, expand the "Other Settings" node and then select the "JFormDesigner" page. To set project specific settings, select the item named "JFormDesigner (Project)". 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 Plug-ins > JFormDesigner.
7.4 JDeveloper plug-in

This plug-in integrates JFormDesigner into Oracle JDeveloper.

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 JDeveloper. No need to switch between applications.
- Uses the source folders and classpath of the current JDeveloper 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).
- 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 (MigLayout, JGoodies Forms, TableLayout, etc) to the project.

User interface

The screenshot below shows the JDeveloper 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 it 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 JDeveloper's **New** command. In the category **Swing/AWT** choose **JFormDesigner Form** and click OK to proceed.

In the **Create 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 JDeveloper. Locate it in JDeveloper's Application Navigator view and double-click it.

### Go to Java code / Go to form

JFormDesigner adds a button to JDeveloper'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** (🔗).
Convert JDeveloper 12c, NetBeans and IntelliJ IDEA forms

You can convert existing JDeveloper 12c forms (which are actually NetBeans forms), NetBeans and IntelliJ IDEA 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 JDeveloper preferences dialog. Select **Tools > Preferences** from the menu to open it and then expand the node “JFormDesigner” in the tree. See **Preferences** for details.

Unsupported features

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

- Folding of generated GUI code in Java editor.
- Use look and feels in Design view.
- Project specific settings.
- 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
- MigLayout
- null Layout
- TableLayout
- VerticalLayout (SwingX)

How to choose a layout manager?

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

- MigLayout has most features (units, alignment, grouping, docking, flowing, in-cell flow and more).
- FormLayout has many features (dialog units, column/row alignment, column/row grouping), but may have problems if a component spans 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 four grid-based layout managers, but JFormDesigner hides its complexity and adds additional features like gaps. Use GridBagLayout if you cannot use MigLayout, 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 MigLayout, 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.

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. The API documentation is available here.

Layout manager properties

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

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>horizontal gap</td>
<td>The horizontal gap between components.</td>
<td>0</td>
</tr>
<tr>
<td>vertical gap</td>
<td>The vertical gap between components.</td>
<td>0</td>
</tr>
</tbody>
</table>

Layout constraints properties

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

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>constraints</td>
<td>Specifies where the component will be placed. Possible values: CENTER, NORTH, SOUTH, EAST, WEST, PAGE_START, PAGE_END, LINE_START and LINE_END.</td>
</tr>
</tbody>
</table>
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. The API documentation is available here.

Layout manager properties

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

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>axis</td>
<td>The axis to lay out components along. Possible values: X_AXIS, Y_AXIS, LINE_AXIS and PAGE_AXIS.</td>
</tr>
</tbody>
</table>
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. The API documentation is available [here](#).

**Layout manager properties**

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

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>horizontal gap</td>
<td>The horizontal gap at the left and right edges.</td>
<td>0</td>
</tr>
<tr>
<td>vertical gap</td>
<td>The vertical gap at the top and bottom edges.</td>
<td>0</td>
</tr>
</tbody>
</table>

**Layout constraints properties**

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

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Card Name</td>
<td>Identifier that can be used to make a card visible. See API documentation for CardLayout.show(Container, String).</td>
</tr>
</tbody>
</table>
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. The API documentation is available [here](#).

**Layout manager properties**

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

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>alignment</td>
<td>The alignment of the layout. Possible values: LEFT, RIGHT, CENTER, LEADING and TRAILING.</td>
<td>CENTER</td>
</tr>
<tr>
<td>horizontal gap</td>
<td>The horizontal gap between components and between the component and the border of the container.</td>
<td>5</td>
</tr>
<tr>
<td>vertical gap</td>
<td>The vertical gap between components and between the component and the border of the container.</td>
<td>5</td>
</tr>
<tr>
<td>align on baseline (Java 6)</td>
<td>Specifies whether components are vertically aligned along their baseline. Components that do not have a baseline are centered.</td>
<td>false</td>
</tr>
</tbody>
</table>
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 two additional libraries with your application. JFormDesigner includes jgoodies-forms.jar and jgoodies-common.jar in its redist folder. For more documentation and tutorials, visit www.jgoodies.com/freeware/libraries/forms/ or forms.java.net.

Maven Central Repository: groupid: com.jgoodies artifactId: jgoodies-forms version: 1.8.0

The API documentation is 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 libraries (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:

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>columnSpecs</td>
<td>Comma separated encoded column specifications. This property is for experts only. Use the column header instead of editing this property.</td>
</tr>
<tr>
<td>rowSpecs</td>
<td>Comma separated encoded row specifications. This property is for experts only. Use the row header instead of editing this property.</td>
</tr>
</tbody>
</table>
Column/row properties

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Column/Row</td>
<td>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.</td>
</tr>
<tr>
<td>Template</td>
<td>FormLayout provides several predefined templates for columns and rows. Here you can choose one.</td>
</tr>
<tr>
<td>Specification</td>
<td>The column/row specification. This is a string representation of the options below.</td>
</tr>
<tr>
<td>Default alignment</td>
<td>The default alignment of the components within a column/row. Used if the value of the component constraint properties &quot;h align&quot; or &quot;v align&quot; are set to DEFAULT.</td>
</tr>
<tr>
<td>Size</td>
<td>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).</td>
</tr>
<tr>
<td>Resize behavior</td>
<td>The resize weight of the column/row.</td>
</tr>
<tr>
<td>Grouping</td>
<td>See column/row grouping for details.</td>
</tr>
</tbody>
</table>

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:

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

Tip: The component context menu allows you to alter the alignment for multi-selections.
**Column/Row Templates**

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

### Column templates

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Gap</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>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.</td>
<td>no</td>
</tr>
<tr>
<td>preferred</td>
<td>Determines the column width by computing the maximum of all column component preferred widths.</td>
<td>no</td>
</tr>
<tr>
<td>minimum</td>
<td>Determines the column width by computing the maximum of all column component minimum widths.</td>
<td>no</td>
</tr>
<tr>
<td>related gap</td>
<td>A logical horizontal gap between two related components. For example the OK and Cancel buttons are considered related.</td>
<td>yes</td>
</tr>
<tr>
<td>unrelated gap</td>
<td>A logical horizontal gap between two unrelated components.</td>
<td>yes</td>
</tr>
<tr>
<td>label component gap</td>
<td>A logical horizontal gap between a label and an associated component.</td>
<td>yes</td>
</tr>
<tr>
<td>glue</td>
<td>Has an initial width of 0 pixels and grows. Useful to describe columns that fill the space between other columns.</td>
<td>yes</td>
</tr>
<tr>
<td>button</td>
<td>A logical horizontal column for a fixed size button.</td>
<td>no</td>
</tr>
<tr>
<td>growing button</td>
<td>A logical horizontal column for a growing button.</td>
<td>no</td>
</tr>
</tbody>
</table>

### Row templates

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Gap</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>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.</td>
<td>no</td>
</tr>
<tr>
<td>preferred</td>
<td>Determines the row height by computing the maximum of all row component preferred heights.</td>
<td>no</td>
</tr>
<tr>
<td>minimum</td>
<td>Determines the row height by computing the maximum of all row component minimum heights.</td>
<td>no</td>
</tr>
<tr>
<td>related gap</td>
<td>A logical vertical gap between two related components.</td>
<td>yes</td>
</tr>
<tr>
<td>unrelated gap</td>
<td>A logical vertical gap between two unrelated components.</td>
<td>yes</td>
</tr>
<tr>
<td>label component gap</td>
<td>A logical vertical gap between a label and an associated component. (requires JGoodies Forms 1.4 or later)</td>
<td>yes</td>
</tr>
<tr>
<td>narrow line gap</td>
<td>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.</td>
<td>yes</td>
</tr>
<tr>
<td>line gap</td>
<td>A logical vertical default gap between two rows. A little bit larger than the narrow line gap.</td>
<td>yes</td>
</tr>
<tr>
<td>paragraph gap</td>
<td>A logical vertical default gap between two paragraphs in the layout grid. This gap is larger than the default line gap.</td>
<td>yes</td>
</tr>
<tr>
<td>glue</td>
<td>Has an initial height of 0 pixels and grows. Useful to describe rows that fill the space between other rows.</td>
<td>yes</td>
</tr>
</tbody>
</table>
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. The API documentation is available here.

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`.

- **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.

- **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:

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>horizontal gap</td>
<td>The horizontal gap between components.</td>
<td>5</td>
</tr>
<tr>
<td>vertical gap</td>
<td>The vertical gap between components.</td>
<td>5</td>
</tr>
<tr>
<td>align left</td>
<td>If true, aligns the layout to the left side of the container. If false, then the layout is centered horizontally.</td>
<td>true</td>
</tr>
<tr>
<td>align top</td>
<td>If true, aligns the layout to the top side of the container. If false, then the layout is centered vertically.</td>
<td>true</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Column/Row</td>
<td>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.</td>
</tr>
<tr>
<td>Default alignment</td>
<td>The default alignment of the components within a column/row. Used if the value of the constraints properties &quot;h align&quot; or &quot;v align&quot; is DEFAULT.</td>
</tr>
<tr>
<td>Size</td>
<td>The minimum width of a column or height of a row.</td>
</tr>
<tr>
<td>Resize behavior</td>
<td>The resize weight of the column/row.</td>
</tr>
</tbody>
</table>

**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*:

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>grid x</td>
<td>Specifies the component's horizontal grid origin (column index).</td>
<td>0</td>
</tr>
<tr>
<td>grid y</td>
<td>Specifies the component's vertical grid origin (row index).</td>
<td>0</td>
</tr>
<tr>
<td>grid width</td>
<td>Specifies the component's horizontal grid extend (number of columns).</td>
<td>1</td>
</tr>
<tr>
<td>grid height</td>
<td>Specifies the component's vertical grid extend (number of rows).</td>
<td>1</td>
</tr>
<tr>
<td>h align</td>
<td>The horizontal alignment of the component within its cell. Possible values: DEFAULT, LEFT, CENTER, RIGHT and FILL.</td>
<td>DEFAULT</td>
</tr>
<tr>
<td>v align</td>
<td>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).</td>
<td>DEFAULT</td>
</tr>
<tr>
<td>weight x</td>
<td>Specifies how to distribute extra horizontal space.</td>
<td>0.0</td>
</tr>
<tr>
<td>weight y</td>
<td>Specifies how to distribute extra vertical space.</td>
<td>0.0</td>
</tr>
<tr>
<td>insets</td>
<td>Specifies the external padding of the component, the minimum amount of space between the component and the edges of its display area.</td>
<td>0,0,0</td>
</tr>
<tr>
<td>Property Name</td>
<td>Description</td>
<td>Default</td>
</tr>
<tr>
<td>---------------</td>
<td>------------------------------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>ipad x</td>
<td>Specifies the internal padding of the component, how much space to add to the minimum width of the component.</td>
<td>0</td>
</tr>
<tr>
<td>ipad y</td>
<td>Specifies the internal padding, that is, how much space to add to the minimum height of the component.</td>
<td>0</td>
</tr>
</tbody>
</table>

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. The API documentation is available here.

Layout manager properties

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

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>columns</td>
<td>The number of columns. Zero means any number of columns.</td>
<td></td>
</tr>
<tr>
<td>rows</td>
<td>The number of rows. Zero means any number of rows.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Note</strong>: 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.</td>
<td></td>
</tr>
<tr>
<td>horizontal gap</td>
<td>The horizontal gap between components.</td>
<td>0</td>
</tr>
<tr>
<td>vertical gap</td>
<td>The vertical gap between components.</td>
<td>0</td>
</tr>
</tbody>
</table>
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://netbeans.org/kb/60/java/quickstart-gui.html](http://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.

**Maven Central Repository:**

```
groupId: org.swinglabs
artifactId: swing-layout
version: 1.0.3
```

The API documentation is available [here](http://netbeans.org/kb/60/java/quickstart-gui.html).

**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.

![Baseline Alignment Example](image)

**Edge** alignments (top, bottom, left and right) are possible relationships between adjacent components.

![Edge Alignment Examples](image)

**Indentation** alignment is a special alignment relationship in which one component is located below another and offset slightly to the right.

![Indentation Alignment Example](image)

**Anchoring indicators**

Anchoring indicators appear when components have snapped into position. They illustrate the alignment and relationship among components.

![Anchoring Indicators](image)

Anchors connecting components to their container or to adjacent components are represented by small semi-circular indicators with dashed lines.

**Visualization of gaps**

The gaps between components are visualized as light gray rectangles. Fixed size gaps are solid and resizable gaps are shown with springs inside. Adjacent gaps are shown when a component is selected. All gaps between all components are shown if a container is selected.

![Gap Visualization Example](image)

To disable visualization of gaps, right-click on a GroupLayout container and deselect **Show Gaps**.
Commands

The designer context menu provides following GroupLayout specific commands:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Align in column/row</td>
<td>Aligns the selected components left/right/top/bottom/center in column/row.</td>
</tr>
<tr>
<td>Align</td>
<td>Aligns the selected components left/right/top/bottom.</td>
</tr>
<tr>
<td>Anchor</td>
<td>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.</td>
</tr>
<tr>
<td>Horizontal Auto Resizing</td>
<td>Makes the selected components resize horizontally at runtime if the container size changes.</td>
</tr>
<tr>
<td>Vertical Auto Resizing</td>
<td>Makes the selected components resize vertically at runtime if the container size changes.</td>
</tr>
<tr>
<td>Same Width</td>
<td>Makes the selected components all the same width. If one of the selected components is already in a group of &quot;Same Width&quot; 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.</td>
</tr>
<tr>
<td>Same Height</td>
<td>Makes the selected components all the same height. See &quot;Save Width&quot; command for more details.</td>
</tr>
<tr>
<td>Set to Default Size</td>
<td>Makes the selected components have its default size.</td>
</tr>
<tr>
<td>Edit Layout Space</td>
<td>Changes the gaps around the selected component.</td>
</tr>
<tr>
<td>Show Gaps</td>
<td>Shows/hides the gaps around the selected components.</td>
</tr>
<tr>
<td>Duplicate</td>
<td>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.</td>
</tr>
</tbody>
</table>

Layout manager properties

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

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>honors visibility</td>
<td>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.</td>
<td>true</td>
</tr>
</tbody>
</table>

Layout constraints properties

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

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>horizontal size</td>
<td>Specifies the component’s horizontal size in pixel or Default. If set to Default, the component’s preferred width is used.</td>
<td>Default</td>
</tr>
<tr>
<td>vertical size</td>
<td>Specifies the component’s vertical size in pixel or Default. If set to Default, the component’s preferred height is used.</td>
<td>Default</td>
</tr>
<tr>
<td>horizontal resizable</td>
<td>Specifies whether the component is horizontal resizable.</td>
<td>false</td>
</tr>
<tr>
<td>Property Name</td>
<td>Description</td>
<td>Default</td>
</tr>
<tr>
<td>----------------------</td>
<td>--------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>vertical resizable</td>
<td>Specifies whether the component is vertical resizable.</td>
<td>false</td>
</tr>
<tr>
<td>top gap</td>
<td>Specifies size of the top gap.</td>
<td></td>
</tr>
<tr>
<td>left gap</td>
<td>Specifies size of the left gap.</td>
<td></td>
</tr>
<tr>
<td>bottom gap</td>
<td>Specifies size of the bottom gap.</td>
<td></td>
</tr>
<tr>
<td>right gap</td>
<td>Specifies size of the right gap.</td>
<td></td>
</tr>
<tr>
<td>top gap resizable</td>
<td>Specifies whether the top gap is vertical resizable.</td>
<td>false</td>
</tr>
<tr>
<td>left gap resizable</td>
<td>Specifies whether the left gap is horizontal resizable.</td>
<td>false</td>
</tr>
<tr>
<td>bottom gap resizable</td>
<td>Specifies whether the bottom gap is vertical resizable.</td>
<td>false</td>
</tr>
<tr>
<td>right gap resizable</td>
<td>Specifies whether the right gap is horizontal resizable.</td>
<td>false</td>
</tr>
</tbody>
</table>
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.

Layout manager properties

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

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>gap</td>
<td>The horizontal gap between components.</td>
<td>0</td>
</tr>
</tbody>
</table>
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.

![GridLayout Example](image)

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` in its `redist` folder. For more documentation and tutorials, visit [www.jetbrains.com/idea/](http://www.jetbrains.com/idea/).

**IDE plug-ins:** If you use IntelliJ IDEA GridLayout the first time, the JFormDesigner IDE plug-in asks 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**:

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

### Layout constraints properties

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

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>grid x</td>
<td>Specifies the component's horizontal grid origin (column index).</td>
<td>0</td>
</tr>
<tr>
<td>grid y</td>
<td>Specifies the component's vertical grid origin (row index).</td>
<td>0</td>
</tr>
<tr>
<td>grid width</td>
<td>Specifies the component's horizontal grid extend (number of columns).</td>
<td>1</td>
</tr>
<tr>
<td>grid height</td>
<td>Specifies the component's vertical grid extend (number of rows).</td>
<td>1</td>
</tr>
<tr>
<td>fill</td>
<td>Specifies how the component fills its cell. Possible values: None, Horizontal, Vertical and Both.</td>
<td>None</td>
</tr>
<tr>
<td>anchor</td>
<td>Specifies how the component is aligned within its cell. Possible values: Center, North,</td>
<td>Center</td>
</tr>
<tr>
<td>Property Name</td>
<td>Description</td>
<td>Default</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>indent</td>
<td>The indent of the component within its cell. In pixel multiplied by 10.</td>
<td>0</td>
</tr>
<tr>
<td>align grid with parent</td>
<td>If true, align the grid of nested containers, which use IntelliJ IDEA GridLayout, with the grid of this container.</td>
<td>false</td>
</tr>
<tr>
<td>horizontal size policy</td>
<td>Specifies how the component affects horizontal resizing behavior. Possible values: Fixed, Can Shrink, Can Grow, Want Grow and combinations.</td>
<td>Can Shrink and Can Grow</td>
</tr>
<tr>
<td>minimum size</td>
<td>The minimum size of the component.</td>
<td>-1, -1</td>
</tr>
<tr>
<td>preferred size</td>
<td>The preferred size of the component.</td>
<td>-1, -1</td>
</tr>
<tr>
<td>maximum size</td>
<td>The maximum size of the component.</td>
<td>-1, -1</td>
</tr>
</tbody>
</table>
8.11 MigLayout

MigLayout is a superbly versatile and powerful layout manager. It is grid-based, but also supports docking and grouping.

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

Compared to other layout managers, MigLayout 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: LogicalPixel, Pixel, Point, Millimeter, Centimeter, Inch, Percent and ScreenPercent. Especially LogicalPixel units are very useful to create layouts that scale with the screen resolution.
- Gaps between columns, rows and components.
- Flexible Growing and Shrinking.
- Column/row grouping.
- In-cell Flow allows putting more than one component into a single grid cell.
- Docking Components to the edges of the container.
- Button Bars and Button Order.
- Override minimum, preferred and maximum component sizes.
- Visual Bounds improves/fixes layout (especially on macOS).
- Baseline support.

MigLayout is open source and not part of the standard Java distribution. You must ship two additional libraries with your application. JFormDesigner includes miglayout-swing.jar and miglayout-core.jar in its redist folder. For more documentation and tutorials, visit miglayout.com or github.com/mikaelgrev/miglayout.

Maven Central Repository: **groupId**: com.miglayout **artifactId**: miglayout-swing **version**: 5.0

The API documentation is available here: doc.formdev.com/miglayout-swing/ and doc.formdev.com/miglayout-core/.

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

By default, all MigLayout containers have insets around the grid. This is similar to setting an `EmptyBorder` on the container. You can change the insets in the Layout manager properties.

Default insets (panel):

Zero insets:

If you prefer zero insets by default, you can change the default layout constraints in the MigLayout preferences.

In-cell Flow

MigLayout allows you to place more than one component into a single grid cell. This is very useful for radio button groups and avoids nested containers.

Docking Components

MigLayout supports docking components to its edges (similar to BorderLayout). You can dock more than one component to one edge. The center is laid out with a grid.

Order: north, west, south, east

Order: east, south, west, north

The docked components are laid out based on the component order. Earlier components get more space as you can see in the above screenshots. Use drag and drop in the Structure view to change order of docked components.

To dock a component, first place it somewhere in the grid, then right-click on the component and select one of the Dock items from the context menu.
Visual Bounds

Some component bounds are larger than their visual bounds (especially on macOS), which gives too large gaps on macOS but optimal gaps on other platforms. MigLayout solves this by considering visual padding when computing component sizes.

Visual Padding on:  

![Visual Padding on](image)

Visual Padding off:  

![Visual Padding off](image)

Layout manager properties

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

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
<th>White Paper</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layout Constraints</td>
<td>Comma separated list of layout constraints. This is a string representation of the options below.</td>
<td>Layout Constraints</td>
<td></td>
</tr>
<tr>
<td>Insets</td>
<td>Specifies the insets for the container. Use this instead of an EmptyBorder.</td>
<td>insets panel</td>
<td></td>
</tr>
<tr>
<td>Gaps</td>
<td>Specifies the default gaps between the columns/rows.</td>
<td>gap related</td>
<td></td>
</tr>
<tr>
<td>h align</td>
<td>The horizontal alignment of the layout within its container. Possible values: Default, Left, Center, Right, Leading and Trailing.</td>
<td>alignx Default</td>
<td></td>
</tr>
<tr>
<td>v align</td>
<td>The vertical alignment of the layout within its container. Possible values: Default, Top, Center and Bottom.</td>
<td>aligny Default</td>
<td></td>
</tr>
<tr>
<td>Fill</td>
<td>Specifies whether columns and/or rows should claim all available space in the container. Possible values: (none), X, Y and Both.</td>
<td>fill (none)</td>
<td></td>
</tr>
<tr>
<td>Hide Mode</td>
<td>Specifies how the layout manager handles invisible components.</td>
<td>hidemode 0</td>
<td></td>
</tr>
<tr>
<td>Flow Y</td>
<td>If true, multiple components in a single cell are lay out vertically.</td>
<td>flowy false</td>
<td></td>
</tr>
<tr>
<td>right-to-left</td>
<td>If true, the columns are added from right-to-left.</td>
<td>righttoleft false</td>
<td></td>
</tr>
<tr>
<td>bottom-to-top</td>
<td>If true, the rows are added from bottom-to-top.</td>
<td>bottomtotop false</td>
<td></td>
</tr>
<tr>
<td>Visual Padding</td>
<td>If true, padding of visual bounds is considered when computing component sizes.</td>
<td>novisualpadding true</td>
<td></td>
</tr>
<tr>
<td>Column Constraints</td>
<td>Constraints of all columns of the container. This property is for experts only. Use the column header instead of editing this property.</td>
<td>Column Constraints</td>
<td></td>
</tr>
<tr>
<td>Row Constraints</td>
<td>Constraints of all rows of the container. This property is for experts only. Use the row header instead of editing this property.</td>
<td>Row Constraints</td>
<td></td>
</tr>
</tbody>
</table>
Column/row properties

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>White Paper</th>
</tr>
</thead>
<tbody>
<tr>
<td>Column/Row</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>Constraints</td>
<td>The column/row constraints. This is a string representation of the options below.</td>
<td>Column/Row Constraints</td>
</tr>
<tr>
<td>Gap before/after</td>
<td>The gaps before and after the column/row.</td>
<td>BoundSize</td>
</tr>
<tr>
<td>Default alignment</td>
<td>The default alignment of the components within a column/row. Used if the value of the component constraint properties &quot;h align&quot; or &quot;v align&quot; are set to Default.</td>
<td>align, fill</td>
</tr>
<tr>
<td>Size</td>
<td>The width of a column or height of a row. You can specify preferred, minimum and a maximum sizes.</td>
<td>UnitValue, BoundSize</td>
</tr>
<tr>
<td>Resize behavior</td>
<td>The grow/shrink weight and priority of the column/row.</td>
<td>grow, growprio, shrink, shrinkprio</td>
</tr>
<tr>
<td>Grouping</td>
<td>See column/row grouping for details.</td>
<td>sizegroup</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
<th>White Paper</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layout Constraints</td>
<td>Comma separated component constraints.</td>
<td>Component Constraints</td>
<td></td>
</tr>
<tr>
<td>Grid Bounds</td>
<td>The computed grid cell bounds (read-only).</td>
<td>cell</td>
<td>0,0</td>
</tr>
<tr>
<td>Cell</td>
<td>The component's grid cell origin (column and row indices).</td>
<td>span</td>
<td>1,1</td>
</tr>
<tr>
<td>Span</td>
<td>The component's grid cell extend (number of columns and rows).</td>
<td>alignx</td>
<td>Default</td>
</tr>
<tr>
<td>h align</td>
<td>The horizontal alignment of the component within its cell. Possible values: Default, Left, Center, Right, Fill, Leading, Trailing and Label.</td>
<td>aligny</td>
<td>Default</td>
</tr>
<tr>
<td>v align</td>
<td>The vertical alignment of the component within its cell. Possible values: Default, Top, Center, Bottom, Fill and Baseline.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Width</td>
<td>Overrides the component's minimum, preferred and maximum widths.</td>
<td>width, wmin, wmax</td>
<td></td>
</tr>
<tr>
<td>Height</td>
<td>Overrides the component's minimum, preferred and maximum heights.</td>
<td>height, hmin, hmax</td>
<td></td>
</tr>
<tr>
<td>Gaps</td>
<td>The gaps between the component and the cell edges. Increases cell size.</td>
<td>gap</td>
<td>0,0,0,0</td>
</tr>
<tr>
<td>Padding</td>
<td>The padding between the component and the cell edges. Does not increase cell size.</td>
<td>pad</td>
<td>0,0,0,0</td>
</tr>
<tr>
<td>Property Name</td>
<td>Description</td>
<td>White Paper</td>
<td>Default</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>Dock</td>
<td>Dock the component at an edge or the center of the container. Possible values: (none), North, South, West, East and Center.</td>
<td>dock</td>
<td>(none)</td>
</tr>
<tr>
<td>Tag</td>
<td>Tag used for platform dependent button ordering. Possible values: (none), ok, cancel, help, help2, yes, no, apply, next, back, finish, left, right and other.</td>
<td>tag</td>
<td>(none)</td>
</tr>
</tbody>
</table>

**Tip:** The component context menu allows you to alter some constraints for multi-selections.
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 and 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 \textbf{Shift} key while moving or resizing components.

Keyboard

You can move selected components with \textbf{Ctrl}+\textbf{ArrowKey} and change size with \textbf{Shift}+\textbf{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.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Align Left</td>
<td>Line up the left edges of the selected components with the left edge of the first selected component.</td>
</tr>
<tr>
<td>Align Center</td>
<td>Horizontally line up the centers of the selected components with the center of the first selected component.</td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Align Right</td>
<td>Line up the right edges of the selected components with the right edge of the first selected component.</td>
</tr>
<tr>
<td>Align Top</td>
<td>Line up the top edges of the selected components with the top edge of the first selected component.</td>
</tr>
<tr>
<td>Align Middle</td>
<td>Vertically line up the centers of the selected components with the center of the first selected component.</td>
</tr>
<tr>
<td>Align Bottom</td>
<td>Line up the bottom edges of the selected components with the bottom edge of the first selected component.</td>
</tr>
<tr>
<td>Same Width</td>
<td>Make the selected components all the same width as the first selected component.</td>
</tr>
<tr>
<td>Same Height</td>
<td>Make the selected components all the same height as the first selected component.</td>
</tr>
<tr>
<td>Make Horizontal Space Equal</td>
<td>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.</td>
</tr>
<tr>
<td>Make Vertical Space Equal</td>
<td>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.</td>
</tr>
</tbody>
</table>

**Layout manager properties**

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

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto-size</td>
<td>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.</td>
<td>true</td>
</tr>
</tbody>
</table>

**Layout constraints properties**

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

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>The x coordinate of the component relative to the left corner of the container.</td>
<td>0</td>
</tr>
<tr>
<td>y</td>
<td>The y coordinate of the component relative to the upper corner of the container.</td>
<td>0</td>
</tr>
<tr>
<td>width</td>
<td>The width of the component in pixel or Preferred. If set to Preferred, the component’s preferred width is used.</td>
<td>Preferred</td>
</tr>
<tr>
<td>height</td>
<td>The height of the component in pixel or Preferred. If set to Preferred, the component’s preferred width is used.</td>
<td>Preferred</td>
</tr>
</tbody>
</table>
8.13 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 in its redist folder. For more documentation and tutorials, visit www.cleartought.info/sun/products/jfc/tsc/articles/tablelayout/ or tablelayout.java.net.

Maven Central Repository: groupId: tablelayout artifactId: TableLayout version: 20050920

The API documentation is 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:

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>horizontal gap</td>
<td>The horizontal gap between components.</td>
<td>5</td>
</tr>
<tr>
<td>vertical gap</td>
<td>The vertical gap between components.</td>
<td>5</td>
</tr>
</tbody>
</table>
Column/row properties

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Column / Row</td>
<td>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.</td>
</tr>
<tr>
<td>Default alignment</td>
<td>The default alignment of the components within a column /row. Used if the value of the constraints properties &quot;h align&quot; or &quot;v align&quot; is DEFAULT.</td>
</tr>
<tr>
<td>Size</td>
<td>Specifies how TableLayout computes the width/height of a column/row.</td>
</tr>
</tbody>
</table>

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:

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

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.
8.14 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.

Layout manager properties

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

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>gap</td>
<td>The vertical gap between components.</td>
<td>0</td>
</tr>
</tbody>
</table>
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.:

- \texttt{C:\MyProject\src\com\myproject\WelcomeDialog.jfd} (form file)
- \texttt{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 \texttt{package} statements. For the above example, you should add \texttt{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.:

\begin{verbatim}
// JFormDesigner - ... //GEN-BEGIN: initComponents
// JFormDesigner - ... //GEN-END: initComponents
\end{verbatim}

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

<table>
<thead>
<tr>
<th>Keyword name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>initComponents</td>
<td>Used for code that instantiates and initializes the components of the form.</td>
</tr>
<tr>
<td>variables</td>
<td>Used for code that declares the class level variables for components.</td>
</tr>
<tr>
<td>initI18n</td>
<td>Used for code that initializes localized component properties if option &quot;Generate initComponentsI18n() method&quot; is enabled in the Localization (Java Code Generator) preferences or &quot;(form)&quot; properties.</td>
</tr>
<tr>
<td>initBindings</td>
<td>Used for code that initializes bindings if option &quot;Generate initComponentBindings() method&quot; is enabled in the Localization (Java Code Generator) preferences or &quot;(form)&quot; properties.</td>
</tr>
</tbody>
</table>
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 -----------------------------------------------------
        Tab1Panel 
        private class 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); 
```
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.

<table>
<thead>
<tr>
<th>Template name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>File header</td>
<td>Used when creating new Java files. Contains a header comment and a package statement.</td>
</tr>
<tr>
<td>Class</td>
<td>Used when generating a new (nested) class. Contains a class declaration, a constructor, a component initialization method and variable declarations.</td>
</tr>
<tr>
<td>Empty Class</td>
<td>Used when generating a new empty class. This can happen, if all form components are contained in nested classes.</td>
</tr>
<tr>
<td>Event Handler Body</td>
<td>Used for event handler method bodies.</td>
</tr>
<tr>
<td>Component Initialization</td>
<td>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.</td>
</tr>
<tr>
<td>Component I18n Initialization</td>
<td>Used for code that initializes localized component properties if option &quot;Generate initComponentsI18n() method&quot; is enabled in the Localization (Java Code Generator) preferences or &quot;(form)&quot; properties.</td>
</tr>
<tr>
<td>Component Binding Initialization</td>
<td>Used for code that initializes bindings if option &quot;Generate initComponentBindings() method&quot; is enabled in the Localization (Java Code Generator) preferences or &quot;(form)&quot; properties.</td>
</tr>
<tr>
<td>Variables Declaration</td>
<td>Replaces the variable $(variables_declaration) used in other templates.</td>
</tr>
<tr>
<td>java.awt.Dialog</td>
<td>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.</td>
</tr>
<tr>
<td>java.awt.Frame</td>
<td>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.</td>
</tr>
<tr>
<td>java.awt.Window</td>
<td>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.</td>
</tr>
<tr>
<td>javax.swing.AbstractAction</td>
<td>Used for nested action classes.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Description</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(date)</td>
<td>Current date.</td>
<td>global</td>
</tr>
<tr>
<td>$(user)</td>
<td>User name.</td>
<td>global</td>
</tr>
<tr>
<td>$(package_declaration)</td>
<td>package statement. If the form is not saved under one of the source folders specified in the Project dialog, the variable is empty (no package statement will be generated).</td>
<td>file header</td>
</tr>
<tr>
<td>$(class_name)</td>
<td>Name of the (nested) class.</td>
<td>class</td>
</tr>
<tr>
<td>$(component_initialization)</td>
<td>See template “Component initialization”.</td>
<td>class</td>
</tr>
<tr>
<td>$(constructor_modifiers)</td>
<td>Modifiers of the constructor. Based on the class modifiers.</td>
<td>class</td>
</tr>
<tr>
<td>$(extends_declaration)</td>
<td>The extends declaration of the class; empty if the class has no superclass.</td>
<td>class</td>
</tr>
<tr>
<td>$(modifiers)</td>
<td>Modifiers of the (nested) class. You can specify the default modifiers in the Preferences dialog.</td>
<td>class</td>
</tr>
<tr>
<td>$(variables_declaration)</td>
<td>See template &quot;Variables declaration&quot;.</td>
<td>class</td>
</tr>
</tbody>
</table>
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 it's 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's useful to re-generate the Java code of JFormDesigner forms. E.g. if you want to upgrade to JGoodies FormLayout 1.2 (or later), 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 layout manager**: Allows you to convert all usages of one layout manager to another one. Useful for migrating forms to a modern powerful layout manager (e.g. MigLayout).

- **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 transferred 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-layout|--convert-jfd]
[--dry-run] [--verbose|-v] [--recursive|-r]
<command-specific-options>
[<project-path>/MyProject.jfdproj]
[<folder> or <path>/MyForm1.jfd]
[...]```

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-classpath &lt;jfd-install&gt;/lib/JFormDesigner.jar</td>
<td>Specifies the JAR that contains the command-line tool. This is a standard argument of the Java application launcher.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>com.jformdesigner.application.CommandLineMain</td>
<td>The class name of the command-line tool.</td>
</tr>
<tr>
<td>--generate</td>
<td>Generate Java code for the given forms or folders.</td>
</tr>
<tr>
<td>--i18n-externalize</td>
<td>Externalize strings in the given forms or folders. This requires that you've specified Source Folders in the used project.</td>
</tr>
<tr>
<td>--convert-layout</td>
<td>Convert one layout manager to another one.</td>
</tr>
<tr>
<td>--convert-jfd</td>
<td>Convert the given .jfd files to another format.</td>
</tr>
<tr>
<td>--dry-run</td>
<td>Execute the given command, but do not save modifications. Only shows what would happen. This option enables --verbose.</td>
</tr>
<tr>
<td>--verbose or -v</td>
<td>Prints file names of processed .jfd and .java files to the console.</td>
</tr>
<tr>
<td>--recursive or -r</td>
<td>Recursively process folders.</td>
</tr>
<tr>
<td>--bundle-name=&lt;bundleName&gt;</td>
<td>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 &quot;(package).Bundle&quot;, which creates Bundle.properties in same package as the form. This option is ignored when processing already localized forms.</td>
</tr>
<tr>
<td>--key-prefix=&lt;keyPrefix&gt;</td>
<td>Only used for --i18n-externalize. The prefix for generated key. You can use variable (basename) (basename of form). Default is &quot;(basename)&quot;. This option is ignored when processing already localized forms.</td>
</tr>
<tr>
<td>--auto-externalize=&lt;true</td>
<td>false&gt;</td>
</tr>
<tr>
<td>--old-layout=&lt;layoutClassName&gt;</td>
<td>Only used for --convert-layout. The full qualified class name of the layout manager that will be converted to another layout manager.</td>
</tr>
<tr>
<td>--new-layout=&lt;layoutClassName&gt;</td>
<td>Only used for --convert-layout. The full qualified class name of the target layout manager.</td>
</tr>
<tr>
<td>--lookAndFeel=&lt;lookAndFeelClassName&gt;</td>
<td>Only used for --convert-layout. The full qualified class name of a look and feel that will be used for layout manager conversion. This is useful if the old layout manager uses units that depend on the look and feel (e.g. FormLayout dialog units). Default is the system look and feel.</td>
</tr>
<tr>
<td>--format=&lt;JFDML</td>
<td>XML&gt;</td>
</tr>
<tr>
<td>--encoding=&lt;encoding&gt;</td>
<td>Only used for --convert-jfd. The encoding used to store JFDML content. See java.nio.charset.Charset for supported encodings. Defaults is &quot;UTF-8&quot;.</td>
</tr>
<tr>
<td>--header-comment=&lt;headerComment&gt;</td>
<td>Only used for --convert-jfd. A comment that is stored in the header of the converted .jfd files. May contain &quot;\n&quot;, which is converted to real newline character.</td>
</tr>
<tr>
<td>&lt;project-path&gt;/MyProject.jfdproj</td>
<td>Optional JFormDesigner stand-alone edition project used to extend the classpath and to specify other preferences. Useful when using custom components.</td>
</tr>
<tr>
<td>&lt;folder&gt; or &lt;path&gt;/MyForm1.jfd […]</td>
<td>List of folders or .jfd files. If a folder is specified, all .jfd files in the folder are processed.</td>
</tr>
</tbody>
</table>

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 if 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).

Examples

**Generate code for a single form:**

```bash
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:**

```bash
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:**

```bash
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
```

**Convert all usages for FormLayout to MigLayout in all forms of the src folder:**

```bash
cd C:\MyProject
java -classpath C:\ProgramFiles\JFormDesigner\lib\JFormDesigner.jar
com.jformdesigner.application.CommandLineMain
  --convert-layout
  --old-layout=com.jgoodies.forms.layout.FormLayout
  --new-layout=net.miginfocom.swing.MigLayout
  --lookandfeel=com.sun.java.swing.plaf.windows.WindowsLookAndFeel
  --recursive
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.

```xml
<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 .jfd 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/](http://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);
            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();
        }
    }
}
```

private void checkBoxActionPerformed(ActionEvent e) {
    JOptionPane.showMessageDialog(dialog, "check box clicked");
}

private void okButtonActionPerformed() {
    dialog.dispose();
}
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`:

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>isContainer</code></td>
<td>Specifies whether a component is a container or not. A container can have child components. The value must be a <code>Boolean</code>. Default is false. E.g.</td>
</tr>
<tr>
<td></td>
<td><code>beanDesc.setValue(&quot;isContainer&quot;, Boolean.TRUE);</code></td>
</tr>
<tr>
<td><code>containerDelegate</code></td>
<td>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. <code>JFrame.getContentPane()</code> is an example for such a method. The value must be a <code>String</code> and specifies the name of a method that takes no arguments and returns a <code>java.awt.Container</code>. E.g.</td>
</tr>
<tr>
<td></td>
<td><code>beanDesc.setValue(&quot;containerDelegate&quot;, &quot;getContentPane&quot;);</code></td>
</tr>
<tr>
<td><code>layoutManager</code></td>
<td>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 <code>Class</code>. E.g.</td>
</tr>
<tr>
<td></td>
<td><code>beanDesc.setValue(&quot;layoutManager&quot;, LayoutManager.class);</code></td>
</tr>
</tbody>
</table>
### Attribute Name | Description
---|---
beanDesc.setValue("layoutManager", BorderLayout.class); | 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`:

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
</table>
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", JSwingConstants.HORIZONTAL, "JSlider.HORIZONTAL", "vertical", JSwingConstants.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");

**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.

propDesc.setValue("notNull", Boolean.TRUE);

**notRestoreDefault**

Specifies that a property value can not restored to the default in the Properties view. If true, the
### Attribute Name | Description
--- | ---
| **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());`
| **preferredBinding** | Specifies that a property is a preferred binding property. If true, the property is added to the **Bind** submenu (right-click on component) and highlighted in bold in the **Binding** dialog. The value must be a `Boolean`. Default is false. E.g. 
`propDesc.setValue("preferredBinding", Boolean.TRUE);`
| **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:

```java
@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 JComponent {...}
```

Example for a container component that has a content pane:

```java
@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:

```java
@PropertyDesc(displayName="magnitude (in %)", preferred=true)
public int getMagnitude() {
    return magnitude;
}
```
Example for specifying this annotation in a `@BeanInfo` annotation:

```java
@BeanInfo
    @PropertyDesc(name="magnitude", displayName="magnitude (in %)", preferred=true)
}

public class MyBean extends JComponent {
    ...
}
```

@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:

```java
public class MyBean extends JComponent {
    @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 FormSpecs (was FormFactory).

JGoodies Looks look and feels are built-in so that you can preview your forms using those popular look and feels.

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.
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.zip` archive contains example source code and forms. See included `README.html` for details.

<table>
<thead>
<tr>
<th>Edition</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stand-alone</td>
<td><code>&lt;jformdesigner-install&gt;/examples.zip</code></td>
</tr>
<tr>
<td>macOS</td>
<td><code>&lt;JFormDesigner.app&gt;/examples.zip</code> (right-click on JFormDesigner application and select &quot;Show Package Contents&quot; from the context menu to see contents of <code>&lt;JFormDesigner.app&gt;</code>)</td>
</tr>
<tr>
<td>Eclipse plug-in</td>
<td><code>&lt;eclipse-install&gt;/features/com.jformdesigner_x.x.x/examples.zip</code> or <code>&lt;eclipse-install&gt;/dropins/JFormDesigner-x.x-eclipse/features/com.jformdesigner_x.x.x/examples.zip</code></td>
</tr>
<tr>
<td>NetBeans plug-in</td>
<td><code>&lt;netbeans-install&gt;/jformdesigner/examples.zip</code></td>
</tr>
<tr>
<td>macOS</td>
<td><code>&lt;NetBeans.app&gt;/Contents/Resources/NetBeans/jformdesigner/examples.zip</code> (right-click on NetBeans application and select &quot;Show Package Contents&quot; from the context menu to see contents of <code>&lt;NetBeans.app&gt;</code>)</td>
</tr>
<tr>
<td>IntelliJ IDEA plug-in</td>
<td><code>&lt;user-home&gt;/../.IdeaIC&lt;version&gt;/config/plugins/JFormDesigner/examples.zip</code> or <code>&lt;intellij-idea-install&gt;/plugins/JFormDesigner/examples.zip</code></td>
</tr>
<tr>
<td>macOS</td>
<td><code>&lt;user-home&gt;/Library/Application Support/IdeaIC&lt;version&gt;/JFormDesigner/examples.zip</code></td>
</tr>
<tr>
<td>JDeveloper plug-in</td>
<td><code>&lt;jdeveloper-install&gt;/jdev/extensions/com.jformdesigner/examples.zip</code></td>
</tr>
</tbody>
</table>

### 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.

<table>
<thead>
<tr>
<th>Edition</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stand-alone</td>
<td><code>&lt;jformdesigner-install&gt;/redist/</code></td>
</tr>
<tr>
<td>macOS</td>
<td><code>&lt;JFormDesigner.app&gt;/redist/</code> (right-click on JFormDesigner application and select &quot;Show Package Contents&quot; from the context menu to see contents of <code>&lt;JFormDesigner.app&gt;</code>)</td>
</tr>
<tr>
<td>Eclipse plug-in</td>
<td><code>&lt;eclipse-install&gt;/plugins/com.jformdesigner.redist_x.x.x/</code> or <code>&lt;eclipse-install&gt;/dropins/JFormDesigner-x.x-eclipse/plugins/com.jformdesigner.redist_x.x.x/</code></td>
</tr>
<tr>
<td>NetBeans plug-in</td>
<td><code>&lt;netbeans-install&gt;/jformdesigner/redist/</code></td>
</tr>
<tr>
<td>macOS</td>
<td><code>&lt;NetBeans.app&gt;/Contents/Resources/NetBeans/jformdesigner/redist/</code> (right-click on NetBeans application and select &quot;Show Package Contents&quot; from the context menu to see contents of <code>&lt;NetBeans.app&gt;</code>)</td>
</tr>
<tr>
<td>IntelliJ IDEA plug-in</td>
<td><code>&lt;user-home&gt;/../.IdeaIC&lt;version&gt;/config/plugins/JFormDesigner/redist/</code> or <code>&lt;intellij-idea-install&gt;/plugins/JFormDesigner/redist/</code></td>
</tr>
<tr>
<td>macOS</td>
<td><code>&lt;user-home&gt;/Library/Application Support/IdeaIC&lt;version&gt;/JFormDesigner/redist/</code></td>
</tr>
<tr>
<td>JDeveloper plug-in</td>
<td><code>&lt;jdeveloper-install&gt;/jdev/extensions/com.jformdesigner/redist/</code></td>
</tr>
</tbody>
</table>