

JIDE Dialogs Developer Guide

Contents

| | |
|---|-----------|
| PURPOSE OF THIS DOCUMENT | 2 |
| WHAT IS JIDE DIALOGS | 2 |
| PACKAGES | 2 |
| STANDARD DIALOG | 3 |
| BANNER PANEL | 4 |
| BUTTON PANEL | 5 |
| BUTTON WIDTH..... | 5 |
| PLATFORM DIFFERENCE ON BUTTON ORDER | 6 |
| BUTTON TYPES AND ORDERS..... | 6 |
| UIDEFAULT IN LOOK AND FEEL..... | 7 |
| PAGE | 9 |
| MULTIPLE-PAGE DIALOG..... | 10 |
| AS A DEVELOPER, HOW DO I USE IT | 12 |
| WIZARD DIALOG | 12 |
| EACH AREA OF A WIZARD..... | 12 |
| AS A DEVELOPER, HOW DO I USE WIZARD COMPONENT | 13 |
| CREATE A WIZARD PAGE | 14 |
| ABSTRACTWIZARDPAGE | 14 |
| DEFAULTWIZARDPAGE..... | 15 |
| PRE-BUILT WIZARD PAGES..... | 16 |
| WIZARD BUTTONS..... | 16 |
| CHANGE BUTTON BEHAVIOR | 16 |
| PAGE VALIDATION | 16 |
| CHANGE WIZARD STYLE | 17 |
| TIPS OF THE DAY DIALOG..... | 18 |
| INTERNATIONALIZATION AND LOCALIZATION | 19 |
| REFERENCES..... | 20 |

Purpose of This Document

Welcome to the *JIDE Dialogs*, the product to make the creation of dialogs easier in Swing. As the name indicated, this product will focus on all kinds of dialogs. It will introduce a simple yet useful dialog template. On top of this template, several commonly used dialogs are built – such as dialogs to display user options and preference, wizards, and tips of today dialog.

This document is for developers who want to develop applications using *JIDE Dialogs*.

What is JIDE Dialogs

Dialog is a place for the user and the application to exchange information, just like a dialogue between them. Dialog window is considered as secondary windows, just like dockable windows provided by *JIDE Docking Framework*. However dockable windows are modeless. Dialog windows can be either modal or modeless.

The history of dialog can trace back as far as the first graphics user interface. Dialog is really an arbitrary collection of GUI controls. However as people understand dialog better and better, certain conventions are formed. Examples of these conventions are: There are always OK and Cancel buttons on modal dialog and Close button on modeless dialog; where are buttons placed and what are the order; how to layout controls in the dialog in a logic flow; how to handle the case when there are too many controls to fit in one screen; etc. People even divided all kinds of dialogs into different categories and each category has its own conventions. Alan Cooper introduces four categories in his book *About Face2* [FACE2]; they are Property dialog boxes, Function dialog boxes, Process dialog boxes and Bulletin dialog boxes. While Bulletin and Process category are relatively more specific, Function and Property category are still very vague. *JIDE Dialogs* will make it easy to follow the rules by introducing some helper components and classes.

Packages

The table below lists the packages in the *JIDE Dialogs*.

| Packages | Description |
|--------------------------|---|
| com.jidesoft.wizard | WizardDialog component. It allows you to create wizard compatible with Wizard 97 standard by Microsoft or JavaLookAndFeel standard wizard by Sun. |
| com.jidesoft.dialog | Basic components for dialog. It allows you to create dialog page, buttons and layout multiple dialog pages. |
| com.jidesoft.tipsoftoday | Tips of the Day Dialog (was moved from JIDE Components) |

Standard Dialog

StandardDialog extends *JDialog*. In addition to *JDialog*, it can handle a couple of things that all dialogs must handle anyway, such as layout, escape and enter key, initial focused component etc.

We certainly can be creative when designing a dialog. Just because UI designers are creative, that's how we see more and more new controls. However sometimes we'd better follow the convention. For example, I've seen a dialog layout as below with OK and Cancel on top.

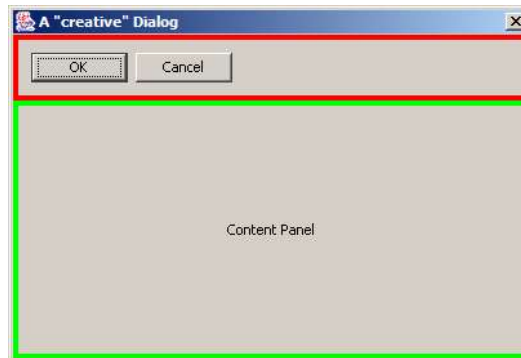
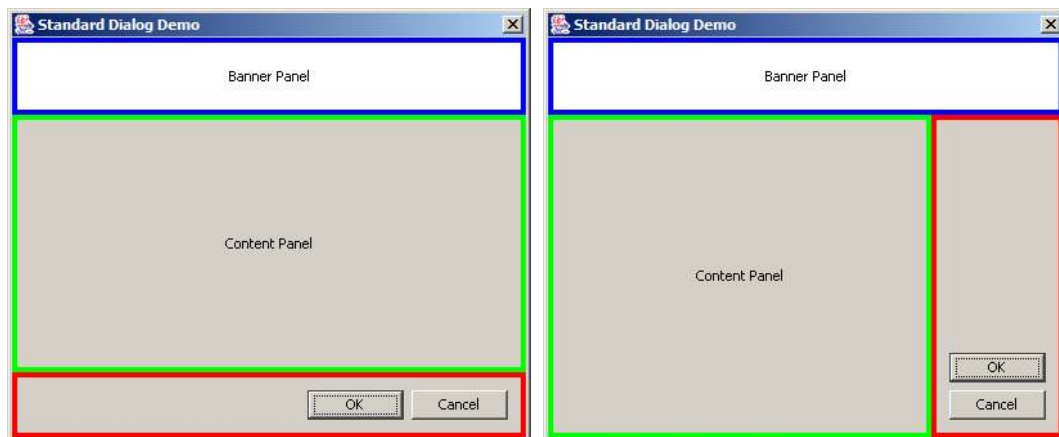


Figure 1 A dialog with buttons on top (bad design in a desktop application)

You might argue it's easy for user to reach OK and Cancel buttons. However in most culture, users get used to look from top to bottom and from left to right. User wants to see what's in dialog first before they click on OK or Cancel button. This dialog obviously breaks the flow.

The two screenshots below show the normal layout of a dialog. On top, you can put a banner panel. Button panel should be either on bottom or on right. Content panel is always in the center. These layouts match the logic flow when people read.



It might be tedious to layout those three panels every time creating a dialog. *StandardDialog* will layout automatically for you.

StandardDialog is an abstract class; you implement three methods. After you implemented these three methods, *StandardDialog* will put them at the right places.

```
abstract public JComponent createBannerPanel();  
abstract public JComponent createContentPanel();  
abstract public JPanel createButtonPanel();
```

Almost all UI guidelines require dialog to handle ESC key and ENTER key correctly. In modal dialog, ESC key should trigger the Cancel button and ENTER should trigger the default button. *StandardDialog* also make this easier by allowing you to set default action and cancel action.

Usually when a dialog is shown, a component in that dialog should have focus. By default, Swing doesn't set any component focus. It is not that straightforward if you try to do it yourself because you can set focus to a component only when a component is visible. With the help of *StandardDialog*, it's never being easier. All you need to do is during *createContentPanel()*, call *setInitFocusedComponent()* to set the initial focused component to whatever you want.

We promise that whenever we find some interesting or useful stuffs, we will continuously enhance *StandardDialog*. That's all about *StandardDialog* so far. Simple, right? Yes. Even though it's simple, when you code using simple *StandardDialog*, your code will become more organized and all your dialogs will look more consistent. Not only that, we also provide several components to make it creation of each methods easier.

Banner Panel

BannerPanel is very useful to display a title, a description and an icon. It can be used in dialog to show some help information or display a product logo in a nice way. You can also set background of *BannerPanel* using *Paint*.

This screenshot below shows three examples that banner panel can do.

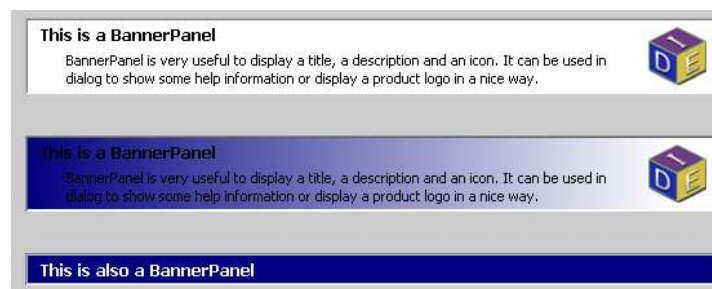


Figure 2 BannerPanel examples

Button Panel

We created *ButtonPanel* class in order to lay out buttons easily in any dialogs. It looks like a very easy thing to do, but when you really think about it, it turns out not so easy. There are two issues *ButtonPanel* try to solve – button width and button order.

Button Width

The problem arose when someone designed a panel like this. Notices the button widths are different.

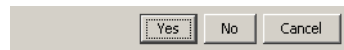


Figure 3 Yes, No and Cancel with different widths

I hope we all agree that this screenshot above doesn't look good. Not only it doesn't look good, but also the small size button is hard to click on. People realized that and argued that all buttons in the same button panel should have the same width. See below for the result. Most existing implementation of button panel did in this way.

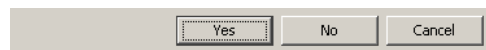


Figure 4 Yes, No and Cancel with the same width

With buttons at the same width, the panel certainly looks much better. However, when dealing with several buttons with text of one of them is much longer than other's, the problem comes up. See below for an example. This one is from GNOME design document [GHIG].

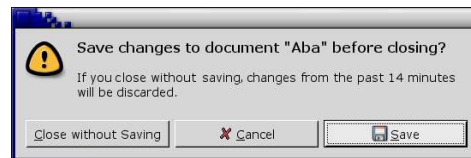


Figure 5 Same button width

“Close without Saving” is much longer than the other two buttons. Since all buttons should have the same width, the “Cancel” and “Save” are forced to have the same width even though it's not really necessary. You can see the screen gets really crowd and will soon run out of spaces. It might get worse after localization if “Close with Saving” is even longer in certain some languages.

Mac OS X takes a different approach to handle this. See below. [AHIG]

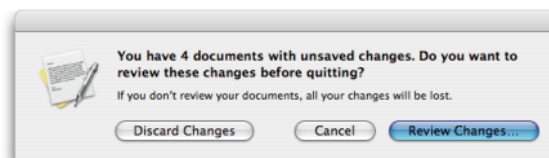


Figure 6 Different button widths on Mac OS X

Even though buttons have different width, this one looks better than the GNOME one.

It seems there are some contradictions here, isn't it? **It's not necessary that all buttons should have the same width. However all buttons should have the same minimum width.** In fact, this convention is followed on several OS. On Windows, the minimum button width is 75 pixels. On Mac OS X, it is 69 pixels. If the preferred width of button is less than the minimum, minimum width should be used.

To implement this requirement in *ButtonPanel*, we added *setSizeConstraint()* method. If you pass in *ButtonPanel.SAME_SIZE*, all buttons will have the same width. If you pass in *ButtonPanel.NO_LESS_THAN*, the button width will be no less than the minimum width. The actual minimum width is different with different LookAndFeel.

Please note, *ButtonPanel* allows you to layout button horizontally and vertically. The *setSizeConstraint()* method only has effect when the buttons are laid out horizontally. If buttons are laid out vertically, the *setSizeConstraint()* will be ignored and *ButtonPanel.SAME_SIZE* is always used. Knowing this drawback of vertical button panel, we suggest you use horizontal button panel as possible as you can. On Mac OS X, it is very rare to see a vertical button panel. On Windows, vertical button panels are used in some dialogs design but are much fewer than horizontal ones.

Platform Difference on Button Order

The second problem we try to solve is the platform difference. To be more specific, how to layout the buttons in the right order on different platform?

On Windows, the OK button comes first, then Cancel. However on Mac OS, the Cancel button comes first, then OK button. People have different opinion on it. You would think that you could choose one of the two ways and follow it as a guide line in your application. But you can't. Since Java application is cross platform, no matter which way you follow, it's wrong on other platform. If you choose the Windows way and button order will look strange on Mac OS. If you choose the Mac way, Windows user will get confused because all other Windows applications' OK button is the first one. So this gives us a hint – should the button order be part of LookAndFeel which can be changed on fly on different LookAndFeels? The answer is yes. *ButtonPanel* will do it for you.

Button Types and Orders

In order to solve the button order problem, we divide buttons into four categories based on the purpose of the button – Affirmative buttons, Cancel buttons, Help buttons and all other buttons. Please refer to GNOME Human User Interface Guidelines [GHIG] for details. Except we call other button rather than alternative buttons, we pretty much follow their naming convention of those categories so that people can understand it easily.

Affirmative buttons are buttons like OK or Yes, which represent an affirmative action to the dialog. Cancel buttons usually cancel out from the dialog. It is usually Cancel or Close. Help button is a button which provide help information. See below for several examples of button types.

To make it simple, we use one character to represent each type – they are A, C, O, and H – representing Affirmative, Cancel, Other and Help respectively. The order will be a permutation of those four letters.



Figure 7 Button Order on Windows

Taking the screenshot above as an example, if the button panel is left-alignment, the order is “ACO”. This is a typical order of buttons on Windows.



Figure 8 Button Order on Mac OS X

On Mac OS X, the order is “CA” with right alignment and “HO” on the opposite side. So in this case, the order “CA” and the opposite order is “HO”.

UIDefault in Look And Feel

Below is a list of UIDefault keys and values on different L&F.

Windows LookAndFeel

```
"ButtonPanel.order", "ACO",
"ButtonPanel.oppositeOrder", "H",
"ButtonPanel.buttonGap", new Integer(6),
"ButtonPanel.groupGap", new Integer(6),
"ButtonPanel.defaultButtonWidth", new Integer(75),
```

Java LookAndFeel

```
"ButtonPanel.order", "ACO",
"ButtonPanel.oppositeOrder", "H",
"ButtonPanel.buttonGap", new Integer(5),
"ButtonPanel.groupGap", new Integer(5),
"ButtonPanel.defaultButtonWidth", new Integer(57),
```

Mac AquaLookAndFeel

```
"ButtonPanel.order", "CA",  
"ButtonPanel.oppositeOrder", "HO",  
"ButtonPanel.buttonGap", new Integer(6),  
"ButtonPanel.groupGap", new Integer(12),  
"ButtonPanel.defaultButtonWidth", new Integer(69),
```

So if you want to use *ButtonPanel*, you just need to add buttons to it and specify the category while adding. The *ButtonPanel* will use values from *UIDefault* to layout the button correctly.

You can also change those values for a particular button panel instance. Those methods are available to you. It will overwrite the value from *UIDefaults*.

```
setButtonOrder(String order)  
setOppositeButtonOrder(String order)  
setSizeConstraint(SAME_SIZE / NO_LESS_THAN)  
setGroupGap(int gap)  
setButtonGap(int gap)
```

Example of *ButtonPanel* is in “examples/W4. *ButtonPanel*”.

Page

What is Page? Let's use book page as an example. Imagining you have a book, you flip through each page and read them. What if I tell you there is nothing on that page until you flip to it? This question sounds a little stupid when you first hear it. However how can you prove that I am wrong?

This goes back to a common technique that used in user interface design – lazy loading. In a multiple-page dialog such as a tabbed pane dialog or a wizard dialog, there could be many pages. Creating all of them at the beginning will take a long time. If you delay the creation of those pages until that page is set visible, it will save time.

When we searched on the web, we found there is an article on JavaWorld talking about the same thing we are trying to do here [JWPAGE]. Using the same concept, we create a class called *AbstractPage*. When you extends this class, instead of initializing the panel in constructor, you need put the code in *lazyInitialize()* method to take advantage of lazy loading.

AbstractPage is very useful when using with CardLayout or tabbed pane which has several panels. Delaying the construction means it will start up fast. Sometimes, delay means never.

AbstractPage overwrites several methods of JComponent to make sure *lazyInitialize()* method will be called. If subclasses choose to override any of the following methods, it is their responsibility to ensure their overridden methods call the parent's method first. The methods are:

- public void paint (Graphics)
- public void paintComponents(Graphics)
- public void paintAll (Graphics)
- public void repaint ()
- public void repaint (long)
- public void repaint (int, int, int, int)
- public void repaint (long, int, int, int, int)
- public void update (Graphics)

By default, if any of the methods is called, *lazyInitialize()* will be called to populate the panel. After it populates the panel, it will remember and will not populate the panel again. User can *setInvokeCondition()* to customize when the panel is populated. For example, you can change the invoke condition to INVOKE_ON_PAINT. If so, *lazyInitialize()* will be called only when *paint()* or *paintAll()* or *paintComponents()* methods are called. You can even set the invoke condition to INVOKE_ON_NONE. If so, you will be responsible to call *lazyInitialize()* since none of those methods mentioned above will call *lazyInitialize()*.

AbstractPage also support *PageListener*. *PageListener* can fire *PageEvent* like *PAGE_OPENED*, *PAGE_CLOSING*, and *PAGE_CLOSED*. Again, same as in *DocumentEvent* in *DocumentPane*, you can stop closing process by call *setAllowClosing()*.

Multiple-Page Dialog

There are cases that there are just too many controls that there is no way to fit them in one screen. To solve this problem, you can divide it into several dialogs. However if you want keep them in one dialog, multiple-page dialog will be needed. Options or Preference dialog is such as example. As applications get bigger and bigger, options dialog also gets more and more complex. You certainly don't want to have many options dialogs. The most commonly used technique used to do it is to use tabbed pane. It's OK if you have few pages in the tabbed pane. However when there are too many pages, the wrapped tabs make the dialog hard to use because those tabs are moving around when selection changes. See the screenshot of Word options dialog below. You can see how crowd it is.

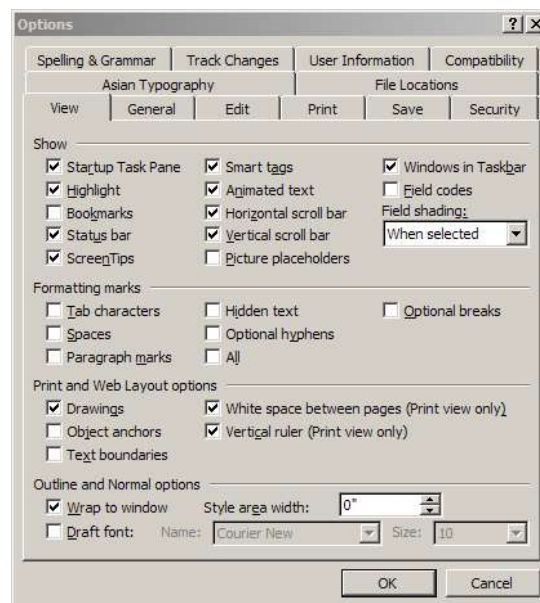


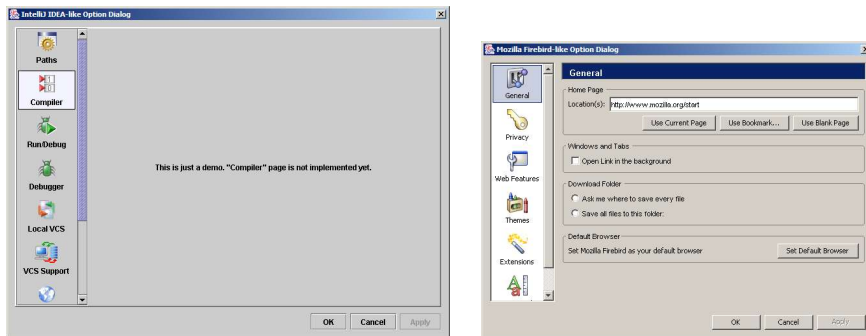
Figure 9 Options Dialog of Microsoft Word

UI designers introduced different ways to deal with this issue. In some applications, a list of buttons is used to replace the tabbed pane such as in IntelliJ IDEA. In Visual Studio .NET for example, a tree is used. Using a tree, you can not only have many pages, but also can have hierarchy among those pages.

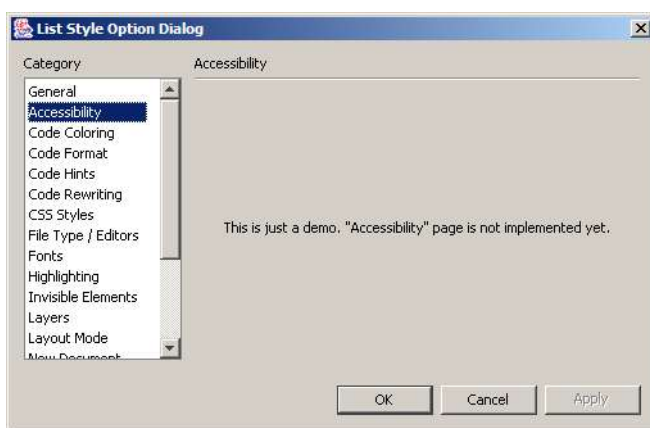
In summary, here are different styles of a Multiple-Page Dialog:

TAB_STYLE: as in Microsoft Office. You should only use it when those tabs can fit in one row. If they cannot be fit in one row, try to use other styles. So the Microsoft Office is not a good example of this style. It's best with 2 to 5 pages.

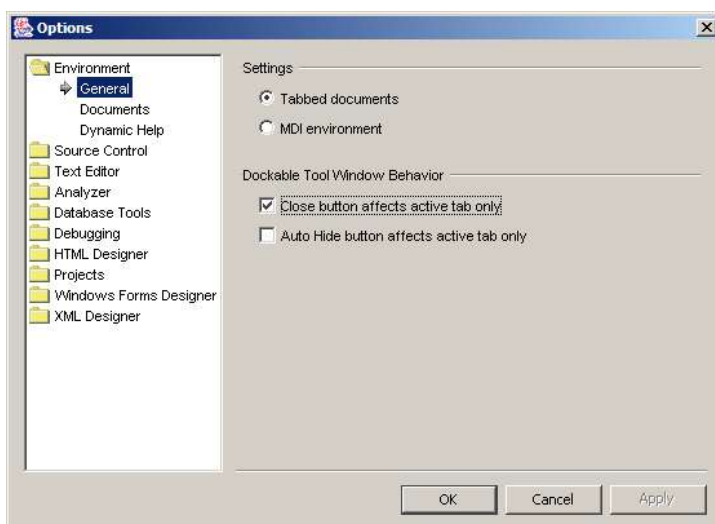
ICON_STYLE: as in IntelliJ IDEA or Mozilla Firebird. You can certainly have more pages than TAB_STYLE as they are arranged in a vertical list. It's best with 4 to 10 pages.



LIST_STYLE: This is just a simplified version of ICON_STYLE when you got a lot of page to show and/or don't want to create icon for each page. It's best with 8 to 20+ pages.



TREE_STYLE: as in Visual Studio .NET. You don't really want to use this style unless you got a lot of pages and those pages can be organized in hierarchy. It's best with 10 and more pages with logic hierarchy.



As a developer, how do I use it

All you need to do create a *PageList* which is a list of *AbstractDialogPage*.

AbstractDialogPage extends *AbstractPage*. However it has title, subtitle, and an icon and parent page. The parent page is only used in *TREE_STYLE*.

AbstractDialogPage also support a listener called *ButtonListener*. A common use case for dialog is that when something is changed, the buttons need to be disabled or enabled. That's what the *ButtonListener* for. *ButtonPanel* implements *ButtonListener* so that you can add it to any *AbstractDialogPage*. If you want to disable a button named *APPLY*, for example, just call this method in *AbstractDialogPage*.

```
fireButtonEvent(ButtonEvent.DISABLE_BUTTON, ButtonNames.APPLY);
```

If *ButtonPanel* does have a button called *APPLY*, it will be disabled. If not, the event is ignored. This is just one way to make *ButtonPanel* and *Page* loosely coupled.

Example of *ButtonPanel* is in "examples/W2. OptionsDialog".

Wizard Dialog

Wizard is a well-known user interface that is ideal to guide user through for complex and unfamiliar tasks. A typical usage of it is project wizard - which asks user a couple questions and generate source code of a project automatically for user.

There are several wizard standards. The most famous two are Microsoft Wizard 97 standard [WIZ97] and Java L&F Wizard standard [JAWAWIZ]. Please see references for details. I strongly suggest you read those standards before designing any wizards. Those documents are very well written. They are also the specs for our wizard component. You can find links to them at the reference section of this developer guide.

Each area of a Wizard

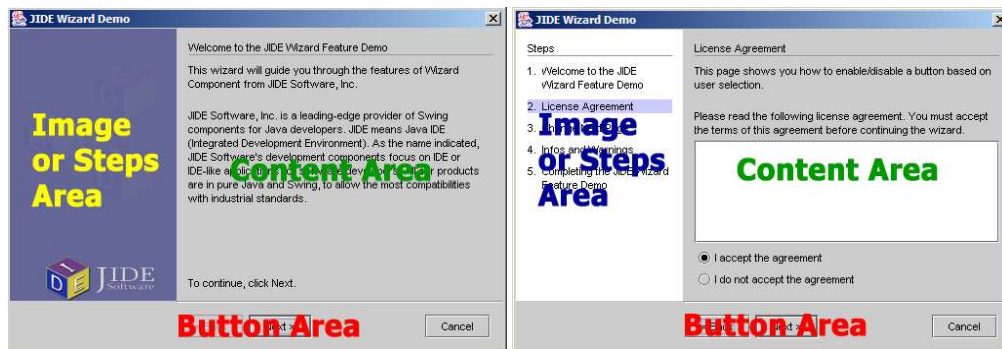
A wizard page can be divided into four areas. They are content area, image or steps area, banner area and button area. See below for several screenshots of various pages and areas on each page.

Windows 97 Standard Wizard



In the case of Wizard 97 standard, welcome page and completion page have image area; interior page have banner area. Since the banner area and image area are both for decoration and description purpose, it doesn't look very good and make sense if a page has both.

Java LookAndFeel Standard Wizard



In Java LookAndFeel wizard standard, it always has image/steps area but never uses a separate banner area. However it has a title and description on top of the content area of each page which shows the same information as in banner area of Wizard 97 standard.

As a developer, how do I use wizard component

Same as *MultiplePageDialog*, the model of wizard is also *PageList*. It's a list of *AbstractWizardPage*. After you create those pages, add them to *PageList*, then call *setPageList()*. See below for an example.

```
WizardSample wizard = new WizardSample("JIDE Wizard Demo");

PageList model = new PageList();

AbstractWizardPage page1 = new WelcomePage("Welcome to the JIDE Wizard Feature Demo",
    "This wizard will guide you through the features of Wizard Component from JIDE Software, Inc.");
```

```
AbstractWizardPage page2 = new LicensePage("License Agreement",
    "This page shows you how to enable/disable a button based on user selection.");

AbstractWizardPage page3 = new InfoWarningPage("Infos and Warnings",
    "This page shows you how to show info message and warning message.");

AbstractWizardPage page4 = new ChangeNextPagePage("Change Next Page",
    "This page shows you how to change next page based on user selection.");

AbstractWizardPage page5 = new CompletionPage("Completing the JIDE Wizard Feature
Demo",
    "You have successfully run through the important features of Wizard Component from
JIDE Software, Inc.");

model.append(page1);
model.append(page2);
model.append(page4);
model.append(page3);
model.append(page5);

wizard.setPageList(model);

wizard.pack();
wizard.setResizable(false); // for wizard, it's better to make it not resizable.
wizard.setVisible(true);
```

Create a Wizard Page

WizardDialog will take care of banner, button and image/steps area. The only area developer need to create is the content area. The content area is represented by wizard page. To make the creation easier, we create two classes for wizard page.

AbstractWizardPage

AbstractWizardPage is the base class for wizard page. It extends *AbstractDialogPage*. In addition, it has two more abstract methods.

```
abstract public JComponent createWizardContent();
abstract public void setupWizardButtons();
```

AbstractWizardPage also has several methods which you can overwrite to get certain features.

```
boolean showBannerPane()
int getLeftPanelItems()
Image getGraphic()
java.util.List getSteps()
int getSelectedStepIndex()
```

I'd like to mention *getLeftPanelItems()* especially. *getLeftPanelItems()* will return a bitwise OR of the following values - LEFTPANE_NONE, LEFTPANE_GRAPHIC, LEFTPANE_STEPS, LEFTPANE_HELP and LEFTPANE_CUSTOM. However not all values combination are support right now. Valid combinations are

LEFTPANE_NONE: display nothing in left pane.

LEFTPANE_GRAPHICS: display a graphic in left pane. You can set default graphic for the whole wizard using *setDefaultGraphic()*. Or you can overwrite *getGraphic()* of each page to return whatever graphic you want.

LEFTPANE_STEPS: display a list of steps. By default, it will be a list of titles of all pages. You can overwrite *getSteps()* and *getSelectedStepIndex()* to create your customized list of steps.

LEFTPANE_HELP: display help information. It should contain the content sensitive help of the focused component in the wizard page. Call *setHelpText()* to set the new help text when focus changes

LEFTPANE_CUSTOM: display any component on left pane. Subclass should overwrite *getCustomLeftPane()* to return the left pane.

LEFTPANE_STEPS | LEFTPANE_HELP: display both steps and help in tabbed pane.

Please note, for Wizard 97 standard, left pane is only used to display graphic. If you decide to use Wizard 97 standard, do not use LEFTPANE_STEPS and LEFTPANE_HELP.

DefaultWizardPage

AbstractWizardPage allows you to create any components and put in content pane. However most components in wizard content are laid out vertically and have a fixed gap in between. *DefaultWizardPage* can be useful in this case. In *DefaultWizardPage*, all you need to do is to overwrite *initContentPane()* and call a bunch of add methods.

```
public void addText(String text): add a multiple line text
public void addText(String text, Font font): add a multiple line text with specified font
public void addTitle(String text): add a multiple line title (Wizard 97 Standard only)
public void addWarning(String text): add a multiple line warning text. It will show a warning icon before the text
```

```
public void addInfo(String text): add a multiple line important text but not yet warning. It will
show an info icon before the text
public void addIconText(Icon icon, String text): add a multiple line text with any icon
public void addComponent(JComponent component): add any component
public void addSpace(): add space. All components added after addSpace will align to bottom.
```

Pre-built Wizard Pages

To further make creation of wizard page easier, we also make several wizard page templates. For example *WelcomeWizardPage* can be used to display welcome information and *CompletionWizardPage* can be used to display summary information.

Wizard Buttons

By default *WizardDialog* will create four buttons – Back, Next, Finish and Cancel. They are usually good enough. In case you need to add more buttons or change action of existing buttons, you can overwrite the *createButtonPanel()* method. There are getter and setter to access those four buttons.

Change Button Behavior

Button state can be changed in each page. *ButtonListener* is used to archive this.

AbstractWizardPage has defined an abstract method called *setupWizardButtons()*. Each page should overwrite this method to change the button state.

Here is an example of *setupWizardButtons()* for completion page. In completion page, finish button should be visible and next button should be set invisible.

```
public void setupWizardButtons () {
    fireButtonEvent(ButtonEvent.ENABLE_BUTTON, BACK);
    fireButtonEvent(ButtonEvent.HIDE_BUTTON, NEXT);
    fireButtonEvent(ButtonEvent.SHOW_BUTTON, FINISH);
    fireButtonEvent(ButtonEvent.CHANGE_BUTTON_FOCUS, FINISH);
}
```

Page Validation

In wizard, there are often requirements to validate each page before going to next page. The ideal place to do this is in *PageListener*. When a page is about to close, it will fire *PAGE_CLOSING* event. In the event, you can check the source of the event object. If the *PAGE_CLOSING* is caused by Next button, the *event.getSource()* will be that Next button. In this

case, the name of the button should be `ButtonNames.NEXT`. If you also need to do something when Back button is pressed, just check for button name which should be `ButtonNames.BACK`. If you override to create your own Next or Back buttons, you just need to make sure you set the button names to those predefined names in `ButtonNames` interface.

Change Wizard Style

Right now we support two styles – `WIZARD97_STYLE` and `JAVA_STYLE` defined in *WizardStyle*. You can call *WizardStyle.setStyle()* to change the style. Since it's a static method on *WizardStyle*, this style you set you affect all subsequence wizard you will create. So basically you should decide what style to use and set the style at the beginning of your `main()`. We don't support to change style on fly. There is really no point to support changing on fly any way. No matter which style you choose, you should use the same style across your application to be consistent.

Here are some noticeable differences between the two styles.

Left Pane: Java style provides more flexibility in left pane. It can display steps, help, or graphics. In Wizard 97 style, it can only be graphics or nothing.

Banner: Wizard 97 standard uses Banner in each interior page to show title and subtitle. Java style doesn't use it at all but show title and subtitle on top of right pane of each page.

Buttons: All buttons on bottom are aligned to left in Wizard 97. Java style aligned NEXT and BACK button to the right.

You can see those differences are not trivial. That's why we suggest you choose a style and just support that one in your application.

Example of `ButtonPanel` is in "examples/W1. Wizard".

Tips of the Day Dialog

A lot of applications use Tips of the Day dialog to get a new user up and running quickly. The content of Tips of the Day is little different from the online help document, but most users prefer to read Tips of the Day because they are concise and they only take a short amount of time per day. To allow more applications to easily create their own Tips of the Day, we created the TipOfTheDayDialog component.

To use this component, you will interact with both the TipOfTheDayDialog dialog and the TipOfTheDaySource information source interface. You can implement the TipOfTheDaySource by just providing two method implementations – getNextTip() and getPreviousTip(). Once you have a TipOfTheDaySource you can pass this to a TipOfTheDayDialog to display the tip of the day. You can create any type of TipOfTheDaySource and still use the same TipOfTheDayDialog to display it. We also provide a ResourceBundleTipOfTheDaySource which implements TipOfTheDaySource. As the name indicates, it reads tips from a ResourceBundle.

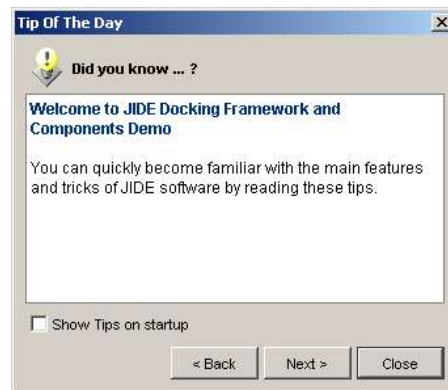


Figure 10 Tip of the Day

~~It's~~The *TipOfTheDayDialog* is very easy to use ~~it.~~ First of all, you need to create a properties file with each tip in a separate line and with zero-based tip numbers. ~~See,~~ as shown below:-

```
0=<b>Welcome to JIDE Docking Framework and Components Demo</b><p>You can quickly ...
1=All the <b>tabs</b> can be drag-n-dropped. Just press mouse button on a tab, hold and ...
.....
```

You can use html in the tips to get a better-looking result. You can also use css to customize the display. Let's say your properties file is tips.properties and your css file is tips.css. Place them at the top level of your class path. You can then write code like following, to display a tip of the day dialog.

```
ResourceBundleTipOfTheDaySource tipOfTheDaySource
```

```

        = new ResourceBundleTipOfTheDaySource(ResourceBundle.getBundle("tips"));
tipOfTheDaySource.setCurrentTipIndex(-1);
URL styleSheet = TipOfTheDayDialog.class.getResource("/tips.css");
TipOfTheDayDialog dialog
    = new TipOfTheDayDialog(_frame, tipOfTheDaySource,
        new AbstractAction("Show Tips on startup"){
            public void actionPerformed(ActionEvent e) {
                if(e.getSource() instanceof JCheckBox) {
                    JCheckBox checkBox = (JCheckBox) e.getSource();
                    setPrefBooleanValue("tip", checkBox.isSelected());
                }
            }
        }, styleSheet);
dialog.setShowTooltip(getPrefBooleanValue("tip", true));
dialog.setDefaultCloseOperation(JDialog.DISPOSE_ON_CLOSE);
dialog.pack();
dialog.setLocation(200, 200);
dialog.show();

```

Example of ButtonPanel is in “examples/ W3. TipsOfTheDay”.

Internationalization and Localization

We have fully considered i18n and l10n. All strings used by *JIDE Dialogs* have been put into properties files under each package. It is buttons.properties under com.jidesoft.dialog. Some users contributed localized version of those files and we put those files inside jide-properties.jar. If you want to support languages other than those we provided, just extract this properties file, translated to the language you want, add the correct postfix and then jar them back into jide-properties jar. You are welcome to send the translated properties file back to us if you want to share it.

References

[WIZ97] *Wizard 97 Standard at MSDN*. Microsoft Corporation.

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wizard/sdkwizv4_7awn.asp

[JAVAWIZ] *Java Look and Feel Design Guidelines – Advanced Topics - Wizards*. Sun Microsystems, Inc. Also available at <http://java.sun.com/products/ilf/at/book/Wizards.html>

[AHIG] *Apple Human Interface Guideline*. Apple Computer, Inc.

<http://developer.apple.com/documentation/UserExperience/Conceptual/OSXHIGuidelines/XHIGIntro/index.html>

[GHIG] *GNOME Human Interface Guidelines*. Calum Benson, Adam Elman, Seth Nickell, colin z Robertson. Available at <http://developer.gnome.org/projects/gup/hig/1.0/windows.html>

[FACE2] Alan Cooper and Robert M. Reimann *About Face 2: The Essentials of Interaction Design* John Wiley & Sons; 2nd edition (March 17, 2003) ISBN: 0764526413

[JWPAGE] Mark Roulo. *Java Tip 90: Accelerate your GUIs*. Available at JavaWorld at http://www.javaworld.com/javatips/jw-javatip90_p.html