How to Use Pop-Up Windows in an Web Dynpro for Java Application

Applies to:

Web Dynpro for Java 7.11. For more information, visit the Web Dynpro Java homepage.

Summary

This tutorial presents the features and the API for creating pop-up windows. With release 7.11 the Web Dynpro pop-up window functionality has been enhanced and a complete new API has been introduced in the window controller for implementing pop-up windows.

Author: Web Dynpro Java Team

Company: SAP AG

Created on: 29 June 2010

Table of Contents

Introduction

In this tutorial you will learn about the new pop-up window features and the corresponding API introduced with Web Dynpro release 7.11.

Prerequisites

Systems, Installed Applications, and Authorizations

You need the NetWeaver Developer Studio (Version 7.11 or later) to compile and deploy the tutorial application. The application server used should have the same version as the NWDS or a newer version.

The tutorial application is available as a development component (DC). You need to import the software component HM-WDUIDMKTCNT, which contains the DC `tc/wd/tut/win/popup`. The exact steps are described in a separate document.

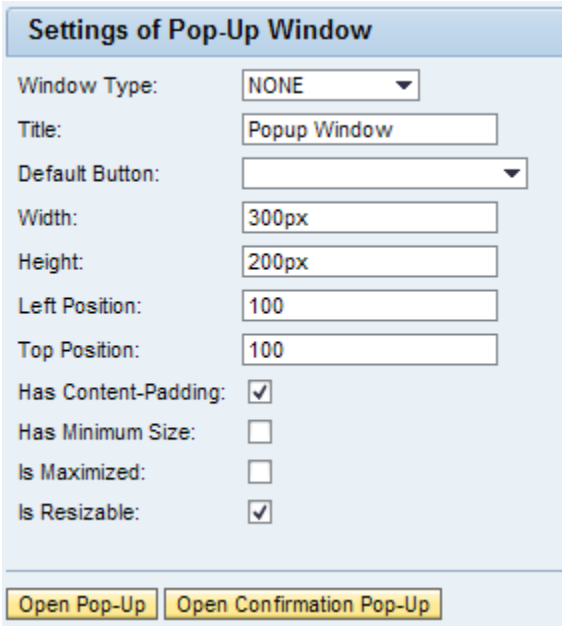
Objectives

After working through this tutorial, you should

- Know the new features of pop-up windows introduced with release 7.11
- Be able to use the new API to implement these features
- Understand the new `wdDoModifyView`() hook method of the window controller

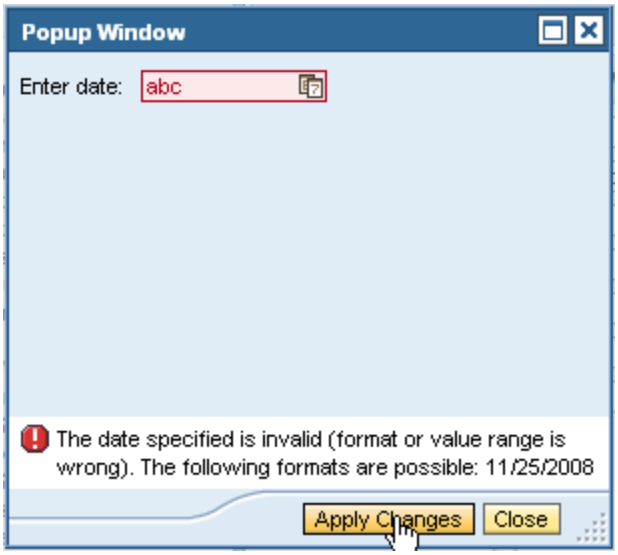
The Tutorial Application

The tutorial application shows a settings area on top where you can configure all pop-up window properties and two buttons for opening pop-up windows using the two available APIs.



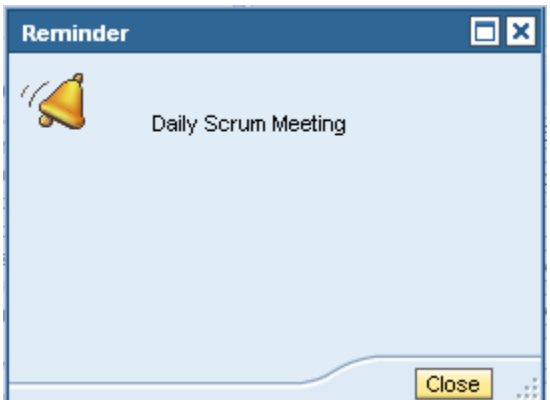
When the “Open Pop-Up” button is pressed, a modal pop-up window appears that is configured with the current settings.

The window has two buttons below the wave-line. The “Apply Changes” button has a validating action assigned. Enter some invalid date into the input field and press the button:



A validation error message appears at the bottom of the content area. The “Close” button in contrast has a non-validating action assigned such that the window can be closed even if the validation fails. It is generally a good idea to allow closing pop-up windows even if there are pending validation errors.

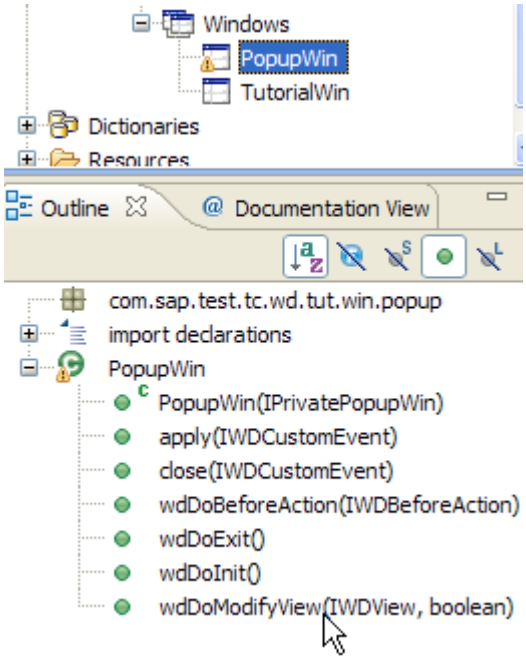
The “Open Confirmation Pop-Up” button opens a pop-up window with a reminder. The assigned action uses the simple API for creating confirmation pop-ups.



Pop-Up Window Features

Buttons below the wave-line

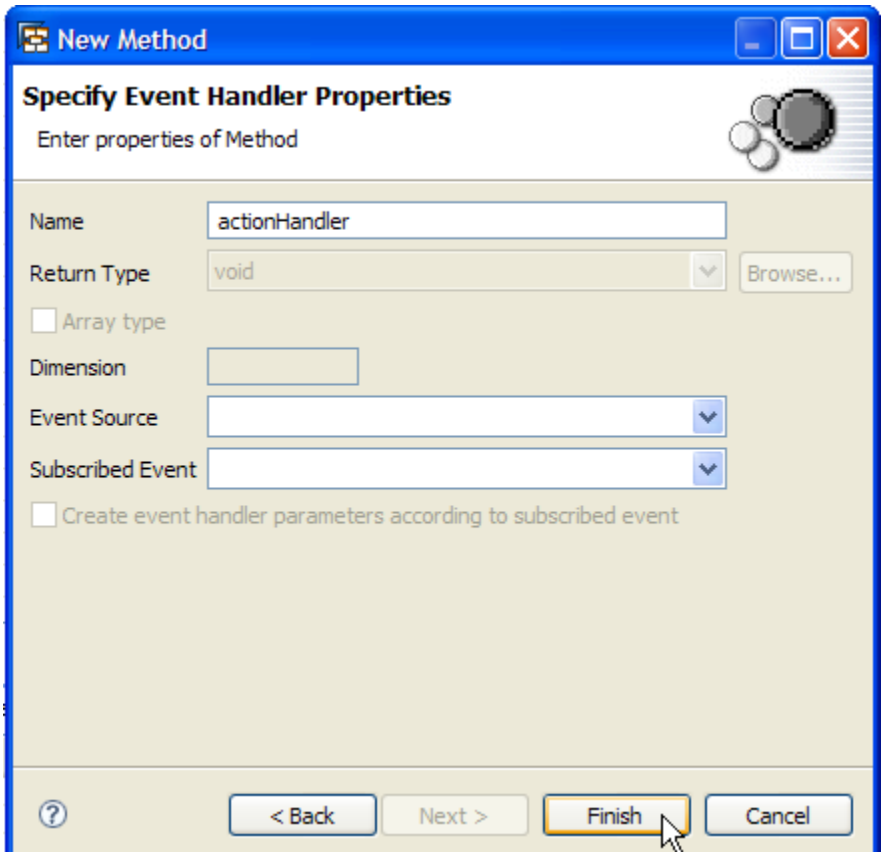
Buttons can now be placed below the so-called wave-line. These buttons are part of the pop-up window itself and not, as in the past, of some view embedded inside the pop-up window. To implement this feature, you have to use the new hook method `wdDoModifyView`() of the window controller for the pop-up window:



The pop-up window is represented by a new view element `IWDWindowViewElement` that can be accessed as the root of the view parameter of method `wdDoModifyView`():

```
// Get the view element that represents the pop-up window
IWDWindowViewElement win = (IWDWindowViewElement) view.getRootElement();
```

The buttons below the wave-line are created using the standard UI element API. The assigned actions also have to be created programmatically. Only the action handler methods may be created using the IDE: create a new method of type “Event Handler” and just assign it a name. The “Event Source” and “Subscribed Event” fields can be left empty.



For each event handler created with the method editor, the IDE will create a constant `WD_EVENT_HANDLER_<methodName>` for accessing the method in the API. We pass these constants to our helper method accordingly.

In the `wdDoModifyView`() hook method you can then create actions that use these event handler methods. Our tutorial application provides the following helper method to create such an action:

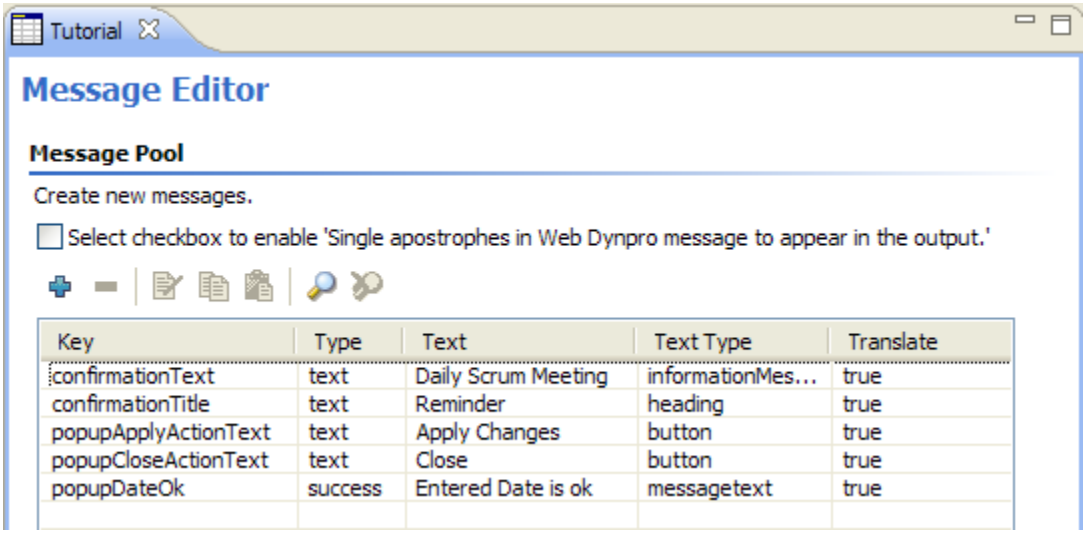
```
  /**
   * Helper method for creating an action with a localized text from the message
bundle.
   *
   * @param actionName the action name
   * @param eventHandlerId the event handler ID (constant is generated by IDE)
   * @param key the text key in the component's message bundle (constant is generated
by IDE)
   * @param validating if the action is validating
   */
  private IWDAction createAction(String actionName, IWDEventHandlerId eventHandlerId,
IWDText key, boolean validating)
  {
    IWDAction action = wdThis.wdGetAPI().createAction(actionName, eventHandlerId,
validating);
    String localizedText = wdComponentAPI.getTextAccessor().getText(key);
    action.setText(localizedText);
    return action;
  }
```

The code to create the “Apply Changes” and “Close” buttons with the corresponding actions looks as follows:

```
  public void wdDoModifyView(com.sap.tc.webdynpro.progmodel.api.IWDView view, boolean
firstTime)
  {
    //@@begin wdDoModifyView
    if (!firstTime)
      return;

    // Create a validating action and a corresponding button
    IWDAction applyAction = createAction
    (
      "Apply",
      wdThis.WD_EVENTHANDLER_APPLY,
      IMessageTutorial.POPUP_APPLY_ACTION_TEXT,
      true
    );
    IWDButton applyButton = view.createElement(IWDButton.class, "ApplyButton");
    applyButton.setOnAction(applyAction);
    applyButton.setText(applyAction.getText());
    // Create a non-validating action for closing the pop-up
    IWDAction closeAction = createAction
    (
      "Close",
      wdThis.WD_EVENTHANDLER_CLOSE,
      IMessageTutorial.POPUP_CLOSE_ACTION_TEXT,
      false
    );
```

(Settings the button text in addition to the action text is not needed because the button takes the action text automatically if it has no text on its own. We need this in this case only to provide the drop-down list entries for choosing the default button).

The localized texts for the actions have been added beforehand to the message pool of the component. The IDE will in turn create constants `IMessage<ComponentName>.MESSAGE_KEY` for accessing the message pool entries.



Finally, the buttons have to be added to the window view element such that they will appear under the wave-line:

```
        // Add buttons below the wave-line
        win.addButton(applyButton);
        win.addButton(closeButton);
```

The Close-Icon

The “Close” icon in the top-right window corner will appear if an action is assigned to the `onClose` event of the window:

```
        // This will render the close icon in the right-upper corner
        win.setOnClose(closeAction);
```

The close-action should be non-validating. Note that the icon is not displayed if the close-action is disabled.

Bindable Window Properties

All the properties for controlling the pop-up window appearance can be bound to the window controller context. The following code shows how this can be done. Of course you need to bind only those properties to the context that need to be controlled via data binding. For all other properties you can just use the setters to assign fixed values.

```
        // Allow controlling the pop-up via data binding
        bindProperties(win);
      /**
       * Binds all properties of the window to the context attributes in the settings
    node.
       * This allows controlling the window properties via data-binding.
       */
      private void bindProperties(IWDWindowViewElement win)
      {
        IWDNodeInfo settings = wdContext.nodePopupSettings().getNodeInfo();

    win.bindDefaultButtonId(settings.getAttribute(IPopupSettingsElement.DEFAULT_BUTTON_ID
    ));

    win.bindHasContentPadding(settings.getAttribute(IPopupSettingsElement.HAS_CONTENT_PAD
    DING));

    win.bindHasMinimumSize(settings.getAttribute(IPopupSettingsElement.HAS_MINIMUM_SIZE))
    ;
        win.bindHeight(settings.getAttribute(IPopupSettingsElement.HEIGHT));
        win.bindLeft(settings.getAttribute(IPopupSettingsElement.LEFT));
        win.bindMaximized(settings.getAttribute(IPopupSettingsElement.MAXIMIZED));
        win.bindResizable(settings.getAttribute(IPopupSettingsElement.RESIZABLE));
        win.bindTitle(settings.getAttribute(IPopupSettingsElement.TITLE));
        win.bindTop(settings.getAttribute(IPopupSettingsElement.TOP));
        win.bindWidth(settings.getAttribute(IPopupSettingsElement.WIDTH));
        win.bindWindowType(settings.getAttribute(IPopupSettingsElement.WINDOW_TYPE));
      }
```

In our example we use a context node named “PopupSettings” that contains all attributes to which the window properties are bound. It is good programming practice to use the generated constants `IPopupSettings.<ATTRIBUTE>` instead of string literals. This will make your code aware of possible renaming of these attributes.

Opening and Closing the Pop-Up Window

In our example we have mapped the pop-up window context and the settings view context to the component context. Thus the edited settings are automatically applied when the pop-up window is reopened.

The pop-up window instance is created on-demand and reused afterwards. You don't have to destroy and recreate the pop-up window every time to get the changed settings.

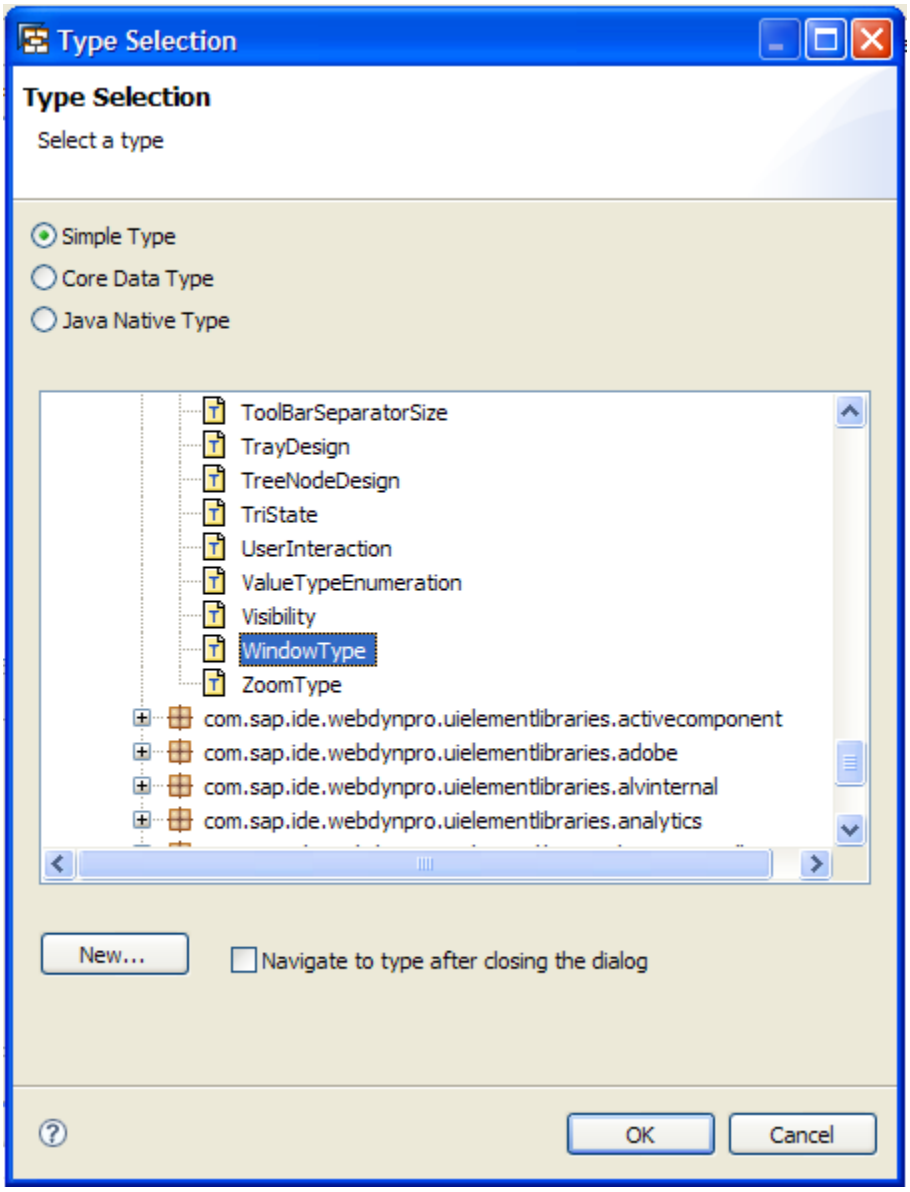
We store the pop-up window instance in a private property of the component controller and allow access only via the methods `openPopup`() and `closePopup`() of the component controller.

```
  public void openPopup( )  {
    //@@begin openPopup()
    if (popup == null)
    {
      IWDWindowInfo windowInfo =
wdComponentAPI.getComponentInfo().findInWindows("PopupWin");
      popup = wdComponentAPI.getWindowManager().createModalWindow(windowInfo);
    }
    popup.show();
    //@@end
  }
  public void closePopup( )  {
    //@@begin closePopup()
    popup.hide();
    //@@end
  }
  //@@begin others
  private IWDWindow popup;
  //@@end
```

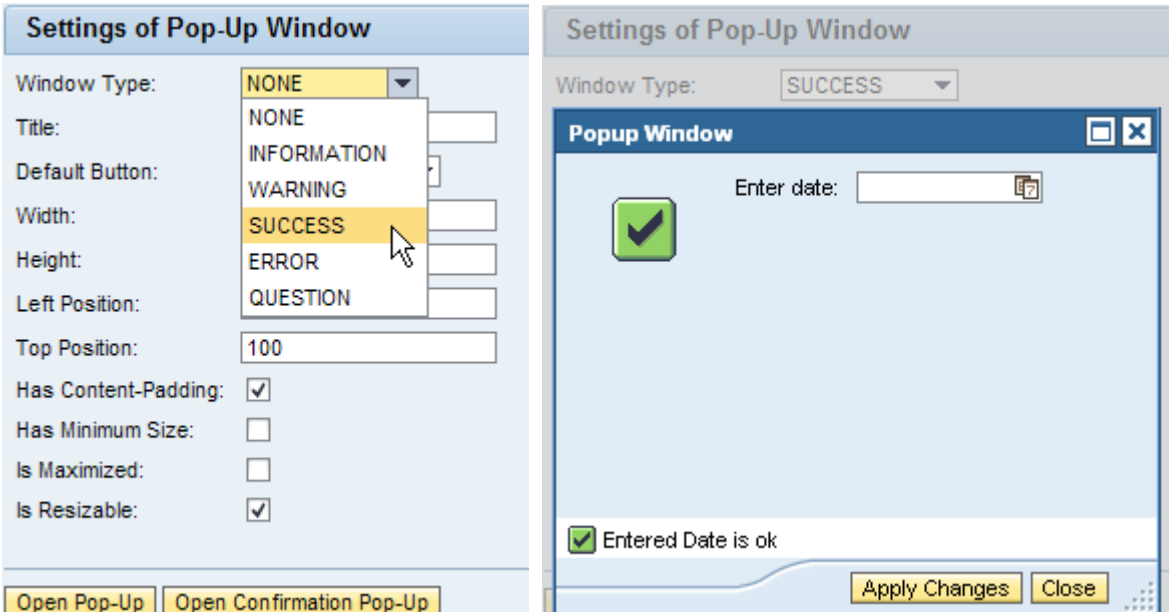
This makes it unnecessary to keep the pop-up instance in some context attribute that has to be mapped from the view controller that opens the pop-up and the view controller or window controller that closes the popup. The controllers just call the component controller methods shown above. You could also put these methods into some custom-controller if needed.

Pop-Up Window Types

A pop-up window can have one of the following types: NONE, INFORMATION, WARNING, SUCCESS, ERROR and QUESTION. In the runtime API these types are represented by the enumeration `WDWindowType`. To control the window type via data-binding, you have to bind the `windowType` property to a context attribute of the corresponding dictionary type `com.sap.ide.webdynpro.uielementdefinitions.WindowType`:



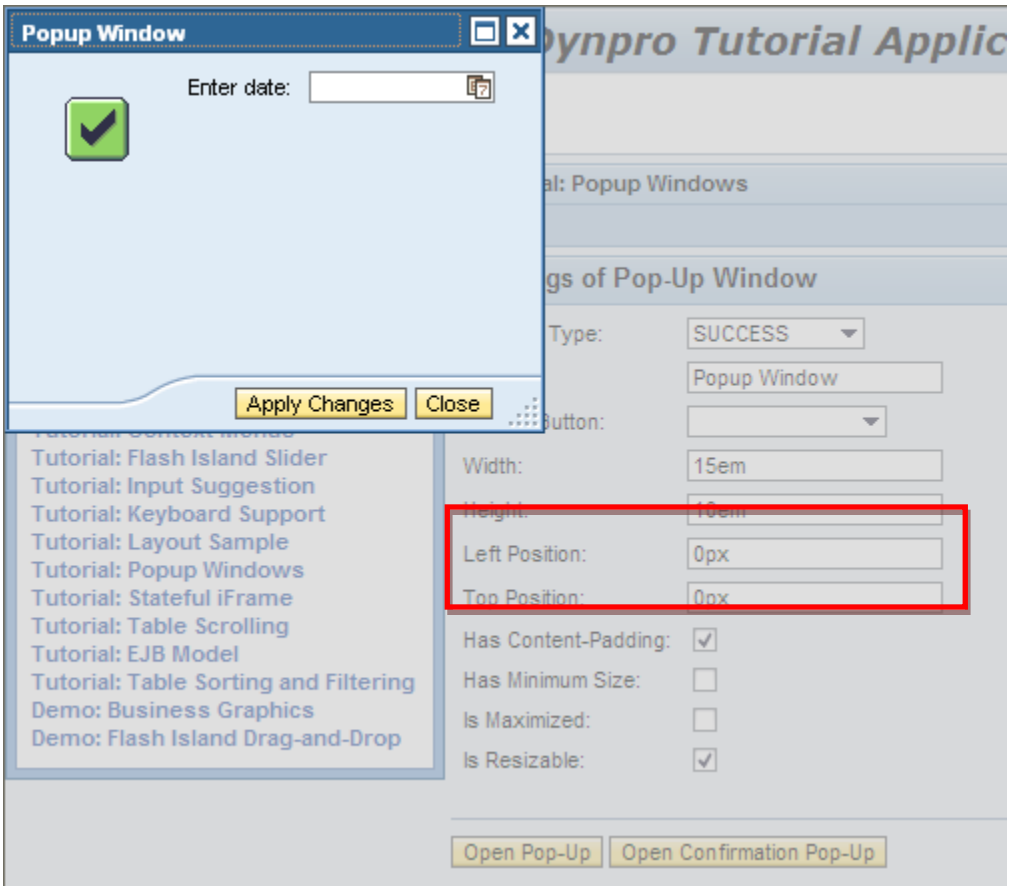
Depending on the used type a different icon appears in the left-upper corner the pop-up window:



You should not try to replace the standard message manager functionality with pop-up windows. Pop-up windows for signaling error or warning messages are inferior to inline messages from a usability point of view.

Controlling the Pop-Up Window Size and Position

The window size and position may be controlled by binding the properties `left`, `right`, `width` and `height` to context attributes. The context attributes can store a CSS value like “100px” or “10em”. The left and right position is measured in relation to the parent window of the pop-up. That means if top and left positions are set to zero, the top-left corner of the pop-up window is positioned at the top-left corner of its parent window:



Other Properties

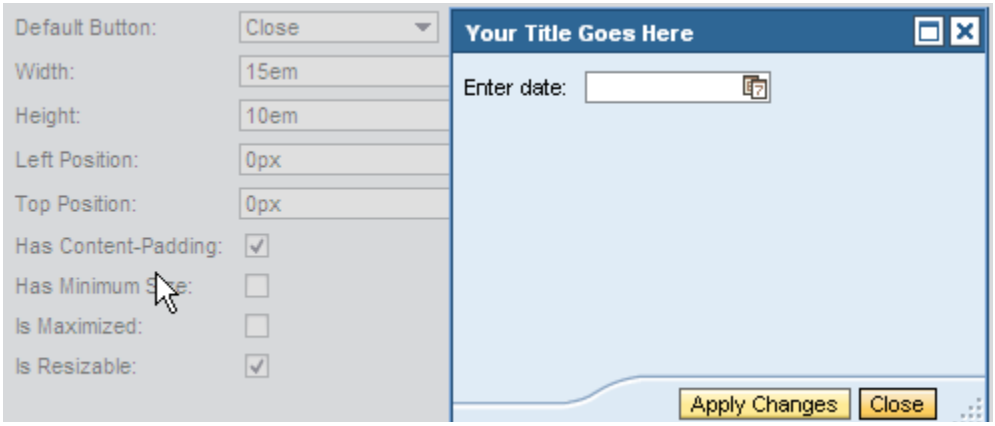
Window Title

The window title can be controlled by setting the `title` property to some fixed value or by binding it to some context attribute of type string. The application has to take care for the translation, for example by reading the translated title text from the component’s message pool.

Window Default Button

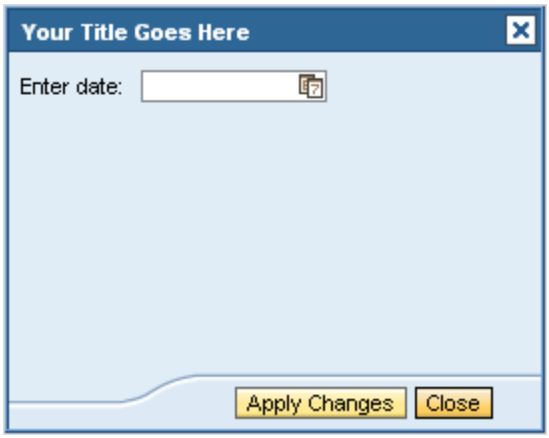
For the buttons under the wave-line you can specify a default button by setting the `defaultButtonId` property to the identifier of the button. You can also use data-binding for changing the default button dynamically.

The default button appears in a slightly darker color than the other buttons and is triggered if the “Enter” key is pressed while the pop-up window is open.



Window Resizing

By default every pop-up window can be resized interactively. By setting the `resizable` property to the value `FALSE` resizing can be disabled and the resizing grip in the lower right corner disappears:



Content Padding

You can switch-off content padding inside the window by setting the `hasContentPadding` to the value `FALSE`.

Minimum Size

If `hasMinimumSize` is set to `true`, the window takes initially a minimum size (defined by the rendering framework) and does not take exactly the size of its content (which might be too small).

Maximized State

The window can be opened in maximized state by setting the `maximized` property to the value `TRUE`.

Creating Confirmation Windows

For creating confirmation windows the interface `IWDWindowManager` provides the following methods:

```
public interface IWDWindowManager
{
  /**
   * Method createConfirmationWindow - creates a new action based confirmation
dialog. Please take attention,
   * that it is not possible in the moment to use both: EventHandlers and Actions in
the same dialog.
   * @param confirmationText - the text which is displayed inside of the dialog
   * @param action - the action which is fired, if the representing button is pressed
in the dialog.
   * @param label - the label of the representing button
   * @return - the instance of the dialog
   */
  public IWDConfirmationDialog createConfirmationWindow(String confirmationText,
IWDAction action, String label);
  /**
   * Method createConfirmationWindow - creates a new event handler based confirmation
dialog. Please take attention,
   * that it is not possible in the moment to use both: EventHandlers and Actions in
the same dialog.
   * @param confirmationText - the text which is displayed inside of the dialog
   * @param eventHandlerId - the id of the event handler, which is called, if the
representing button is pressed in the dialog.
   * @param label - the label of the representing button
   * @return - the instance of the dialog
   */
  public IWDConfirmationDialog createConfirmationWindow(String confirmationText,
IWDEventHandlerId eventHandlerId, String label);
```

The first method expects an action that will be assigned to the button which is created inside the confirmation window by the framework. The second method creates the action implicitly and uses the event handler defined by the given event handler ID.

After having created the confirmation window, you can add additional buttons using the different `addChoice`() methods of the `IWDConfirmationDialog` API:

```
public interface IWDConfirmationDialog
{

  /**
   * Method addChoice - adds a choice (represented by a action button) to the
confirmation dialog. Each choice references an event handler, which is called, if the
button is pressed.
   * @param eventHandler - is called, if the button-representation is pressed
   * @param label - the label of the button representation
   */
  public void addChoice(IWDEventHandlerId eventHandlerId, String label);
  /**
   * Method addChoice - adds a choice (represented by a action button) to the
confirmation dialog. Each choice references an event handler, which is called, if the
button is pressed.
   * @param eventHandler - is called, if the button-representation is pressed
   * @param label - the label of the button representation
   * @param enabled - the flag, if the button representation is enabled
```

```
     */
    public void addChoice(IWDEventHandlerId eventHandlerId, String label, boolean
  enabled);
    /**
     * Method addChoice - adds a choice (represented by a action button) to the
  confirmation dialog. Each choice references an action, which is called, if the button
  is pressed.
     * @param action - is fired, if the button-representation is pressed
     * @param label - the label of the button representation
     */
    public void addChoice(IWDAction action, String label);
    /**
     * Method addChoice - adds a choice (represented by a action button) to the
  confirmation dialog. Each choice references an action, which is called, if the button
  is pressed.
     * @param action - is fired, if the button-representation is pressed
     */
    public void addChoice(IWDAction action);
    /**
     * Method setOnClose - this method assigns an event handler to the close icon of
  the confirmation dialog.
     * @param eventHandler - is called, if the button-representation is pressed
     * @param enabled - the flag, if the button representation is enabled
     */
    public void setOnClose(IWDEventHandlerId eventHandlerId, boolean enabled);
    /**
     * Method setOnClose - this method assigns an event handler to the close icon of
  the confirmation dialog.
     * @param action - is fired, if the button-representation is pressed
     */
    public void setOnClose(IWDAction action);

    /**
     * Method setIcon - this method sets an icon to the confirmation dialog. Only
  absolute URLs are expected @see
  com.sap.tc.webdynpro.services.sal.url.api.IWDURLGenerator for details.
     * @param icon - the absolute URL for the icon
     */
    public void setIcon(String icon);

    /**
     * makes the confirmation pop-up visible, the layout is generated by information
  (icon, text, actions etc) given at the moment
     * this method is called.
     * @see com.sap.tc.webdynpro.services.session.api.IWDWindow#show()
     */
    public void show();
  }
```

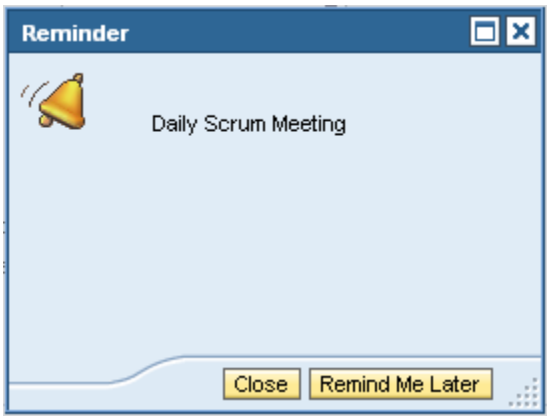
Similar to the `IWDWindowViewElement` API you can assign a close-action and set an icon that appears in the left-upper area of the window.

In our tutorial application the action handler for the “Open Confirmation Pop-Up” button uses the described API like this:

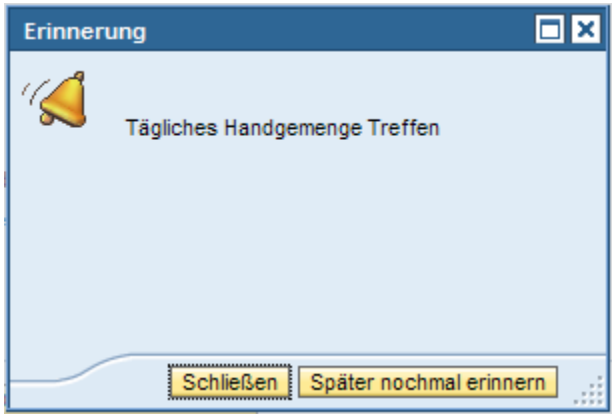
```
  public void
onActionOpenConfirmationPopUp(com.sap.tc.webdynpro.progmodel.api.IWDCustomEvent
wdEvent )
  {
    //@@begin onActionOpenConfirmationPopUp(ServerEvent)
    IWDTextAccessor ta = wdComponentAPI.getTextAccessor();
    IWDConfirmationDialog confirmation =
wdComponentAPI.getWindowManager().createConfirmationWindow
    (
      ta.getText(IMessageTutorial.CONFIRMATION_TEXT),
      wdThis.wdGetCloseConfirmationPopUpAction(),
      ta.getText(IMessageTutorial.POPUP_CLOSE_ACTION_TEXT)
    );
    confirmation.addChoice
    (
      wdThis.wdGetRemindAgainAction(),
      ta.getText(IMessageTutorial.CONFIRMATION_BTN_REMIND_AGAIN_TEXT)
    );
    confirmation.setOnClose(wdThis.wdGetCloseConfirmationPopUpAction());
    confirmation.setTitle(ta.getText(IMessageTutorial.CONFIRMATION_TITLE));

confirmation.setIcon(WDResourceFactory.getSystemResource(WDIconLarge.Reminder).toStri
ng());
    confirmation.setWindowPosition(100, 100);
    confirmation.setWindowSize(200, 150);
    confirmation.show();
    //@@end
  }
```

The confirmation window instance is automatically destroyed after closing the window. To get the localization right, you should access all texts using the `IWDTextAccessor` API of the component.

The confirmation dialog will look as follows depending on the current session locale:



English



German

Restrictions

IDE support for the new pop-up window features is still missing.

The described features are only guaranteed to work in browser versions that are officially supported by the given Web Dynpro release

More Information

Wei-Guo Peng: New Features of Web Dynpro Popup-Window – SAP NetWeaver CE 7.11

SAP Developer Network SDN http://sdn.sap.com

Text Symbols

Symbol	Usage
	Note
	Recommendation
	Warning
	See also
→	Arrow for navigation paths

Copyright