

Using TFrame to create more advanced TJSEditDialogs

This tutorial provides instructions on how to create a more advanced dialog but still taking advantage of the other properties and methods available to TJSEditDialog controls.

Contents

Using TFrame to create more advanced TJSEditDialogs.....	1
Getting Started.....	2
Methods in the interface	5
Implementing the Interface	6
Getting TJSEditDialog to display the frame.....	9
Running for the first time.....	11
Dialog Validation	11
Cancelling the dialog	12
Getting values from the dialog	13
Adding a custom Apply button	16
Controlling the state of the Apply button.....	21
Completing the dialog.....	23
Conclusion.....	25

Getting Started

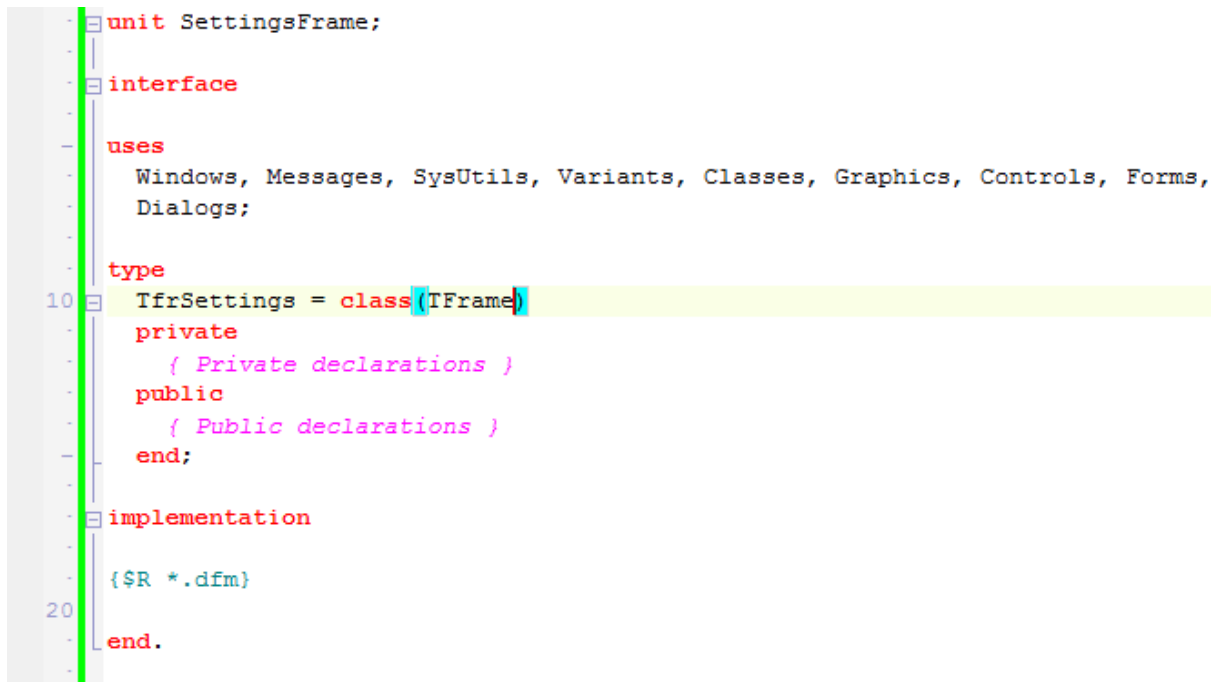
The dialog we are going to create is going to load and save some settings to an ini file. So the dialog will contain two radio buttons, an edit control, a drop down list and a check list box.

1. Create a new project. Save the project as Tutorial_TFrameDialogControl. Save the default form as MainForm.
2. To this project, add a TFrame. Rename the frame as TfrSettings and save the frame as SettingsFrame.
3. Add a TJSEditDialog control to MainForm.
4. Rename the TJSEditDialog to jsedSettings.

Now we need to make changes to the Frame control so that when it is displayed on a dialog, the dialog will be able to interact with it automatically. This is done by implementing the IJSEditDialogControl interface. To get advanced use from TJSEditDialog, implementing this interface is a good idea.

It is important to point out that it is not necessary to implement this interface to have a control used by TJSEditDialog, it just makes the whole process smoother and allows for more advanced features like allow for input data to be validated.

Switch to the SettingsFrame unit and make sure the code editor is visible. The code in the editor should look similar to the screen capture below.



```
unit SettingsFrame;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs;

type
  TfrSettings = class(TFrame)
  private
    { Private declarations }
  public
    { Public declarations }
  end;

implementation

{$R *.dfm}

end.
```

To implement the IJSEditControl interface we need to modify the class definition. Add the IJSEditControl reference after the descendant class. The screen capture below shows the location and syntax.

```
1 unit SettingsFrame;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs;
type
10 TfrSettings = class(TFrame, IJSEditDialogControl)
  private
    { Private declarations }
  public
    { Public declarations }
  end;
implementation
  {$R *.dfm}
20 end.
```

Notice how the IJSEditDialogControl interface is underlined as an error. This is because we need to add the unit to where the interface is declared. Add the JSEditDialogIntf unit to the uses class.

If you open the JSEditDialogIntf unit you can see the methods that are declared on the interface. This means you also must declare these methods in your frame class, even if you don't want to include any code in the method, it still must be handled.

Use code insight to display the list of all of the methods that belong to the interface that need to be implemented. Inside the class definition press **Ctrl+Space** to display the code insight popup. The screen capture below shows the list of methods (highlighted) for the interface.

```

unit SettingsFrame;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, JSEditDialogIntf;

type
  TfrSettings = class(TFrame, IJSEditDialogControl)
  private
    { Private declarations }
  public
    { Public declarations }
  end;

implementation

{$R}

end.

```

function	ActiveControl : TWinControl;
function	ActiveControlInError : TWinControl;
procedure	AdjustControlLayout (const AWidth: Integer);
function	GetRequiredWidth : Integer;
procedure	SetActiveControlInError (AControl: TWinControl);
procedure	SetDialog (JSDialog: TJSCustomEditDialog);
procedure	ValidateInput (out Message: string; const ModalResult: TModalResult): Boolean;
procedure	ActionChange (Sender: TObject; CheckDefaults: Boolean); override ;
procedure	AdjustClientRect (var Rect: TRect); override ;
procedure	AdjustSize ; override ;
procedure	AfterConstruction ; override ;
procedure	AlignControls (AControl: TControl; var ARect: TRect); override ;
procedure	Assign (Source: TPersistent); override ;
procedure	AssignTo (Dest: TPersistent); override ;
function	AutoScrollEnabled : Boolean; override ;

Select the methods from the list (multi-select works) and press enter. This will add those methods to your class ready for implementation. Use class completion (Ctrl+Shift+C) to complete the classes and generate the implementation code section.

Once you have completed the class you need to add a reference to the JSEditDialog unit since one of the interface methods accepts a TJSCustomEditDialog parameter. Also add a field to hold a TJSCustomEditDialog reference.

The modified code is displayed below.

```
1 unit SettingsFrame;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, JSEditDialogIntf, JSEditDialog;

type
10 TfrSettings = class(TFrame, IJSEditDialogControl)
  private
    FJSDialog: TJSCustomEditDialog;
  public
    function ActiveControl: TWinControl;
    function ActiveControlInError: TWinControl;
    procedure AdjustControlLayout(const AWidth: Integer);
    function GetRequiredWidth: Integer;
    procedure SetActiveControlInError(AControl: TWinControl);
    procedure SetDialog(JSDialog: TJSCustomEditDialog);
20    function ValidateInput(out Message: string;
      const ModalResult: TModalResult): Boolean;
  end;
```

Methods in the interface

Now it is time to describe each of the interface methods and how they should be used for our settings dialog.

ActiveControl

This allows you to specify which control on the frame should be active when first displayed.

ActiveControlInError

This allows you to specify which control was in error when the ValidateInput method was called. If multiple controls are in error, you should return the first control in the User Interface in this method.

AdjustControlLayout

This method allows you to make dynamic changes to the form once the width of the dialog is known. This is because while you return the required dialog width in the GetRequiredWidth method, there are guidelines as to how wide the dialog should be by default, so your dialog may be wider than you are expecting.

GetRequiredWidth

Specify the required width of the dialog. See the AdjustControlLayout method for more information about this method.

SetActiveControlInError

This allows you to specify which control is in error during validation and then this control should be returned when the ActiveControlInError is called.

SetDialog

Set dialog allows you to have a reference to TJSEditDialog instance that is displaying your frame. This allows you to access the dialog directly if required while the dialog is visible.

ValidateInput

This method is called when the user has performed an action that is closing the dialog. This might mean they have clicked the OK button, but it can also mean they have clicked the Close button on the title bar.

Use this method to make sure that all details entered on your dialog are correct. The method is passed the pending modal result value of the dialog so you know if you need to validate the dialog or not. For example, when the ValidateInput method is called and the ModalResult parameter is mrCancel, then this means the dialog is being cancelled and you shouldn't need to actually validate any input.

If something in the dialog doesn't validate, the method should return false. You can also pass back a specific error message in the Message parameter.

Implementing the Interface

Now we will start to implement these items in our frame class. First add a new field to the class to hold a reference to the active control when in error. Call the TWinControl field FActiveControlInError and set it when the SetActiveControlInError method is called.

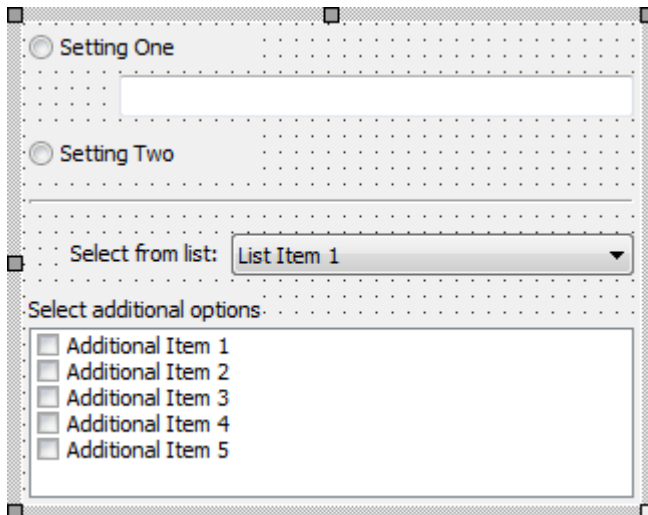
```
procedure TfrSettings.SetActiveControlInError(AControl: TWinControl);  
begin  
    FActiveControlInError := AControl;  
end;
```

Now we can also implement the ActiveControlControlInError method by just returning the FActiveControlInError field.

Before implementing any more methods it is time to create the user interface for our dialog.

As mentioned previously the dialog will control two radio button, one edit control, a drop down control and a check list box. Add these controls to your form along with a few labels and your user interface should look something like the following screen capture.

This is how the interface looks at design time. I've added a couple of items into each of the lists and set the index of the drop down box to be 0 (so displaying the first item in the list).



Now that the interface has been decided we can implement the rest of the methods. The following is a list of the Component Names and Types so that the implementation code makes some sense.

```
TfrSettings = class(TFrame, IJSEditDialogControl)
  rbSettingOne: TRadioButton;
  rbSettingTwo: TRadioButton;
  eSettingOne: TEdit;
  cboxListItems: TComboBox;
  Bevel1: TBevel;
  Label1: TLabel;
  Label2: TLabel;
  clbAdditionalOptions: TCheckListBox;
```

ActiveControl

Since the active control method should return the first control in the user interface, make it return the first radio button.

```
function TfrSettings.ActiveControl: TWinControl;
begin
  Result := rbSettingOne;
end;
```

AdjustControlLayout

When creating the user interface I set the Anchors (the anchors property was introduced for all TControl descendants a few Delphi versions ago) of the controls so any layout changes that the dialog needs to make should not mess with frame layout.

NOTE: It isn't often that the dialog will need to change the width of your dialog.

GetRequiredWidth

I created the dialog to be 308 pixels wide. No particular reason for that, but that is what should be returned by this method. Just a note about this value, you do not need to scale this value up if the end user is running large fonts, this will be done for you automatically.

SetDialog

Just set the FJSDialog reference to the JSDialog parameter.

ValidateInput

Just to show some features of the dialog, we will make up some rules for completing the dialog correctly.

First radio button is selected.

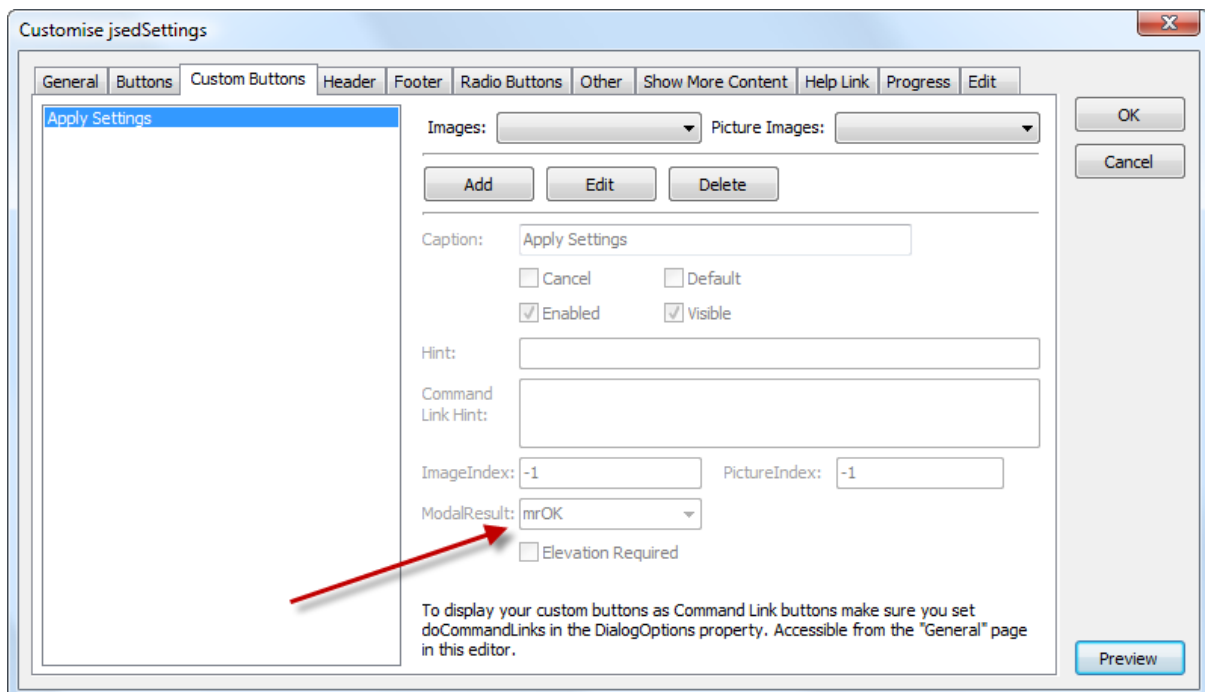
1. The edit box cannot be empty.
2. The edit box must start with a capital A.

There must be two items checked in the "Select additional items" check list box.

Remember we only want to validate the dialog if the user has clicked on the OK button. So the dialogs ModalResult will be mrOK.

NOTE: If you use a Yes button, instead of OK you should test the ModalResult for mrYes. This applies to any other button values.

NOTE: If you are using custom buttons you should give them a proper ModalResult value that you can also use. An example of a custom button and setting the ModalResult value for this button is displayed below.



If you preview a dialog at design time, currently an edit box will show in its place. This is expected.

The validation code is displayed below.

```

function TfrSettings.ValidateInput(out Message: string;
  const ModalResult: TModalResult): Boolean;
var
  I: Integer;
  LChecked: Integer;
  LError: string;
begin
  Result := True;
  if ModalResult = mrOK then
  begin
    if rbSettingOne.Checked then
    begin
      if eSettingOne.Text = '' then
      begin
        Result := False;
        Message := 'You must enter a value for Setting One.';
        SetActiveControlInError(eSettingOne);
      end
      else if eSettingOne.Text[1] <> 'A' then
      begin
        Result := False;
        Message := 'The value for setting one must start with "A".';
        SetActiveControlInError(eSettingOne);
      end
    end;
    LChecked := 0;
    for I := 0 to clbAdditionalOptions.Items.Count - 1 do
    begin
      if clbAdditionalOptions.Checked[I] then
        Inc(LChecked);
    end;
    if (LChecked < 2) then
    begin
      LError := 'You must check at least two items in the additional options list.';
      if Result then
      begin
        SetActiveControlInError(clbAdditionalOptions);
        Message := LError;
        Result := False;
      end
      else
        Message := Message + #13#10 + LError;
    end;
  end;
end;
end;

```

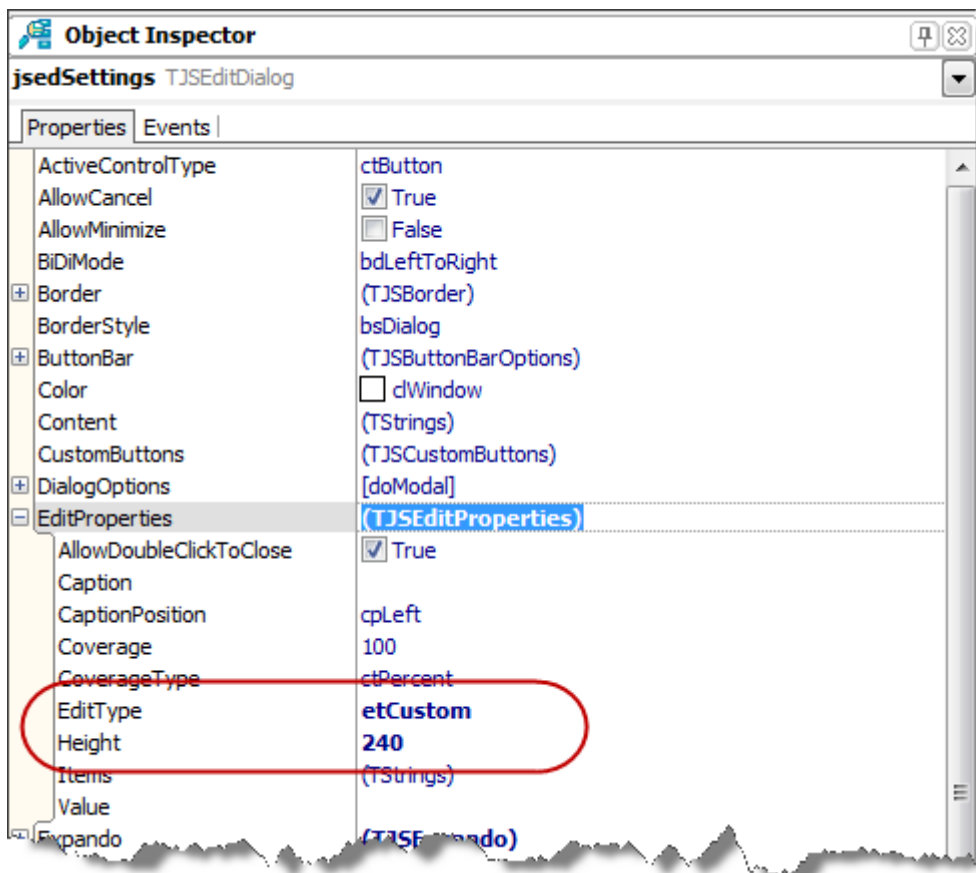
Now we have implemented the IJSDialogControl interface, we now need to tell our TJSEditDialog component to use our frame to display these additional features.

Getting TJSEditDialog to display the frame

Switch back to the MainForm unit select the TJSEditDialog component in the designer.

1. Expand the EditProperties property for the selected dialog component.
2. Change the EditType property from etEdit (the default) to etCustom.
3. Change the Height property to the height value of the frame. This is 240 for the frame I created.

A screen capture of the modified properties is below.



Now we need to tell the dialog component what custom edit class to use. This is done by handling the OnGetEditControlClass event.

Switch to the Events page of the object inspector and generate an OnGetEditControlClass event for your TJSEditDialog (by double clicking on it).

The event has three parameters:

1. Sender – references the TJSEditDialog component.
2. Index – the index of the control being referenced. This value is reserved for future use and it currently is always 0.
3. ControlClass – this var parameter is where you return the class name of control you want the dialog to create and use. So set the ControlClass parameter to TfrSettings.

```
procedure TfrmMain.jsedSettingsGetEditControlClass(Sender: TObject;
  const Index: Integer; var ControlClass: TControlClass);
begin
  ControlClass := TfrSettings;
end;
```

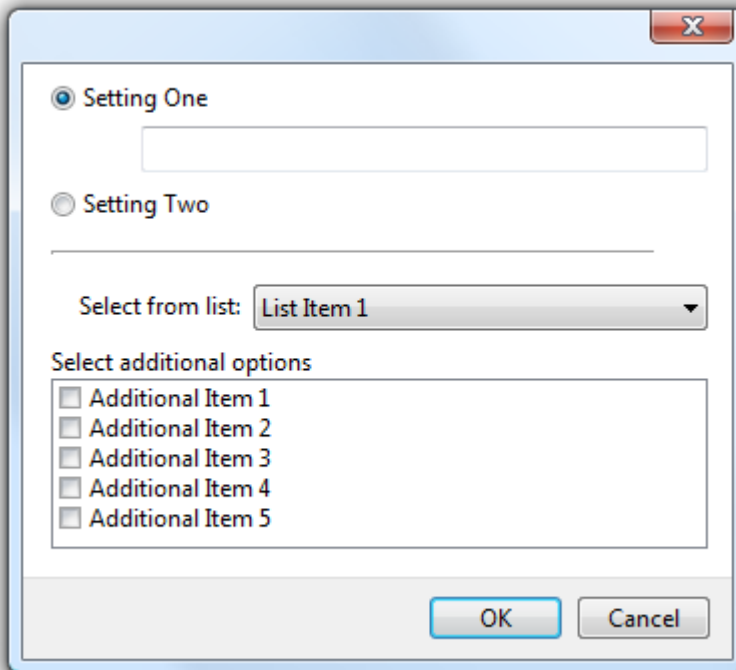
This is all we need to do to have the dialog show our frame. It is now time to run our application.

Running for the first time

Place a button on the form and add the following code to display the dialog.

```
procedure TFormMain.Button1Click(Sender: TObject);  
begin  
    jsedSettings.Execute;  
end;
```

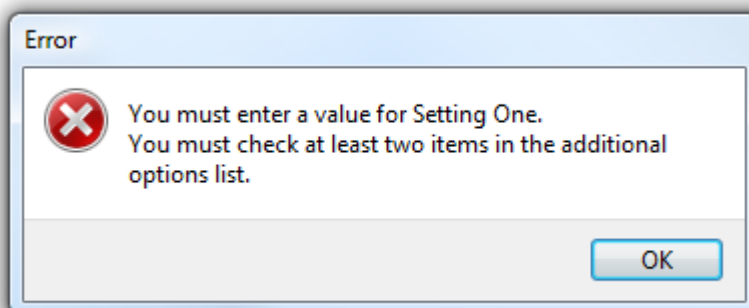
Running the application a clicking on the button should now display the dialog.



The dialog box has a title bar with a close button (X). It contains two radio buttons: "Setting One" (selected) and "Setting Two". Below "Setting One" is a text input field. Below "Setting Two" is a horizontal line. Below the line is a label "Select from list:" followed by a dropdown menu showing "List Item 1". Below the dropdown is a section titled "Select additional options" containing a list of five checkboxes: "Additional Item 1", "Additional Item 2", "Additional Item 3", "Additional Item 4", and "Additional Item 5". At the bottom are "OK" and "Cancel" buttons.

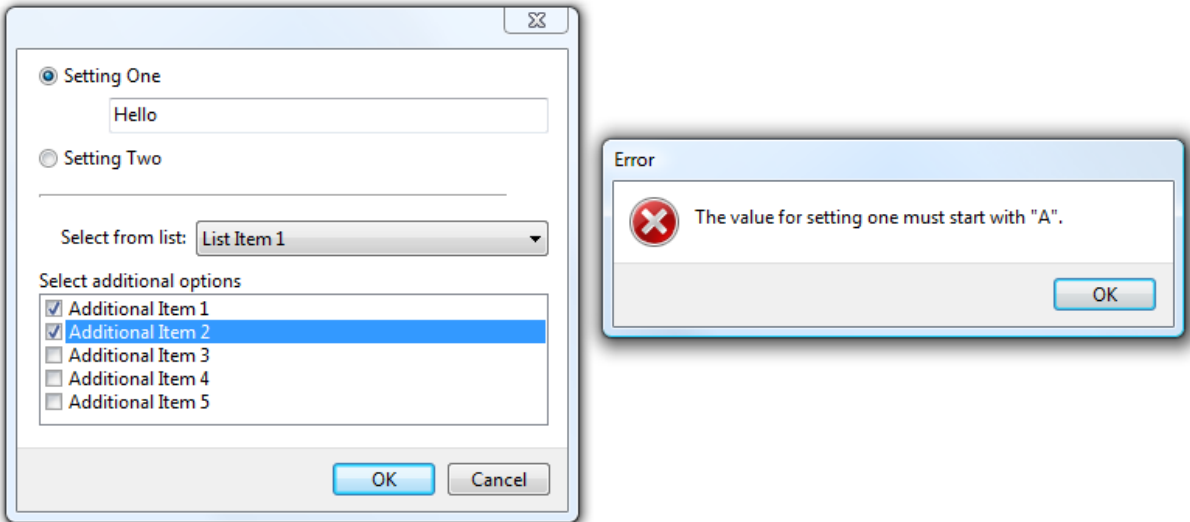
Dialog Validation

If you click OK the validation logic should kick in and display the error messages returned.

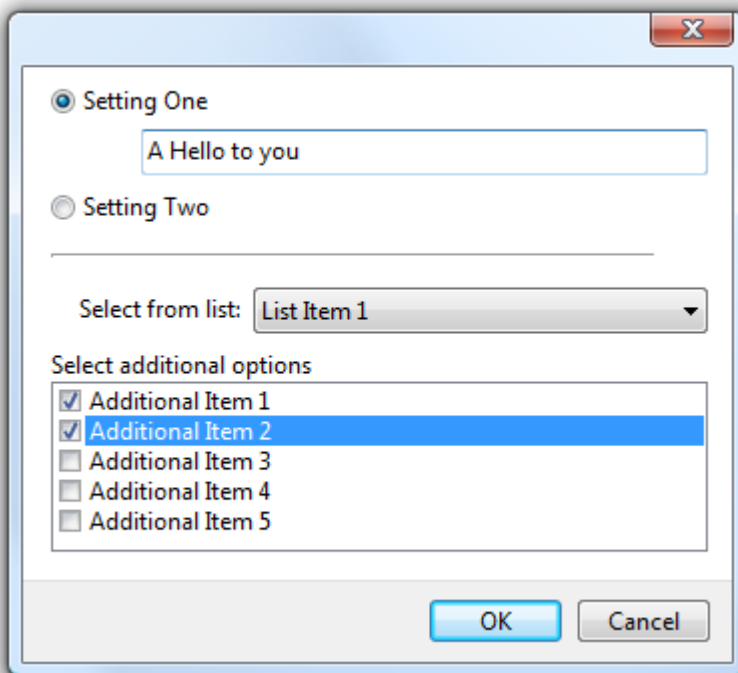


The error dialog box has a title bar with the word "Error". It contains a red circular icon with a white "X". To the right of the icon are two lines of text: "You must enter a value for Setting One." and "You must check at least two items in the additional options list." At the bottom right is an "OK" button.

If we populate the Setting One edit control and select two of the additional option items and press OK, we should see the other error message display.



Update the dialog to validate correctly should result in no errors being displayed. Modify the dialog as below and verify this occurs.



Cancelling the dialog

Clicking either the Cancel button or the X on the title bar should close the dialog without any errors being displayed.

So now we have our custom frame displaying in our dialog. How do we get back the values the user enters in the dialog so I can use them in my application? Good question, lets answer it.

Getting values from the dialog

So far we've created a custom frame with its own validation logic and had it display within our dialog. Now we need to know what settings the user has changed while the dialog is visible.

Switch to the MainForm unit and select the TJSEditDialog component and change to the event page of the object inspector.

To get access to the frame control on the dialog we need to handle the OnGetEditValue event. Double click on the event to generate it.

The event has four parameters:

1. Sender – references the TJSEditDialog component.
2. AControl – references the control in the dialog. This is a direct reference to our frame.
3. Index - the index of the control being referenced. This value is reserved for future use and it currently is always 0.
4. Value – this var parameter allows you to set the value for the dialogs EditValue property.

Now we will implement the method to grab details from the dialog and apply them to the main form. Before we do this, drop a TListBox control on the form.

The radio button controls will determine what the Caption of the form will be. If the Setting One value is checked, then the value for the edit box will be the forms caption, otherwise the caption will be set to "n/a".

We will populate the list box control with the checked items from the additional items list.

Depending on the index of the drop down control, we will change the color of the form.

Below is the code that implements the above actions.

```

procedure TfrmMain.jsedSettingsGetEditValue(Sender: TObject; AControl: TControl;
  const Index: Integer; var Value: string);
var
  LSettings: TfrSettings;
  I: Integer;
begin
  if TJSEditDialog(Sender).ModalResult = mrOK then
  begin
    LSettings := TfrSettings(AControl);
    if LSettings.rbSettingOne.Checked then
      Value := LSettings.eSettingOne.Text
    else
      Value := 'n/a';
    ListBox1.Items.Clear;
    for I := 0 to LSettings.clbAdditionalOptions.Items.Count - 1 do
    begin
      if LSettings.clbAdditionalOptions.Checked[I] then
        ListBox1.Items.Add(LSettings.clbAdditionalOptions.Items[I]);
    end;
    case LSettings.cboxListItems.ItemIndex of
      0: Color := clBtnFace;
      1: Color := clWhite;
      2: Color := clYellow;
      3: Color := clLime;
    end;
  end;
end;

```

We need to typecast the AControl parameter to our frame class so then we can access any properties or components on it.

Before we can run our application if you look at the code above you can see that we test the dialogs ModalResult value before making any changes – this is because the event is called regardless of the dialogs result. It is not possible for the dialog to know what a developer may set as an affirmative result.

Also we set the Value parameter, this means we need to actually use the EditValue property to reflect this change on the form. We could have easily just set the forms caption from within this method if we wanted.

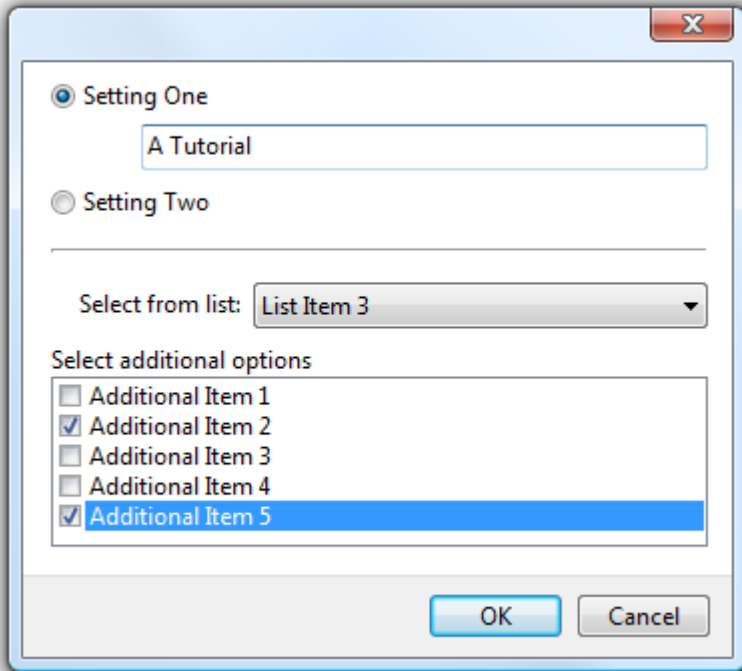
To set the caption using the EditValue property we need to change the code that displays the dialog as below.

```

procedure TfrmMain.Button1Click(Sender: TObject);
begin
  if jsedSettings.Execute = mrOK then
  begin
    Caption := jsedSettings.EditValue;
  end;
end;

```

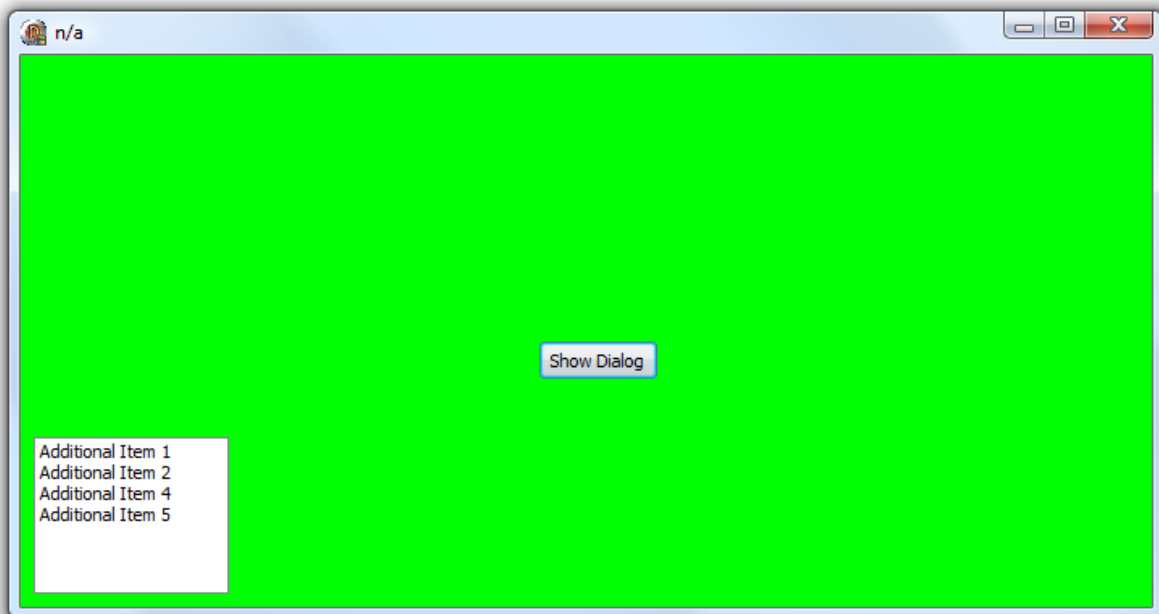
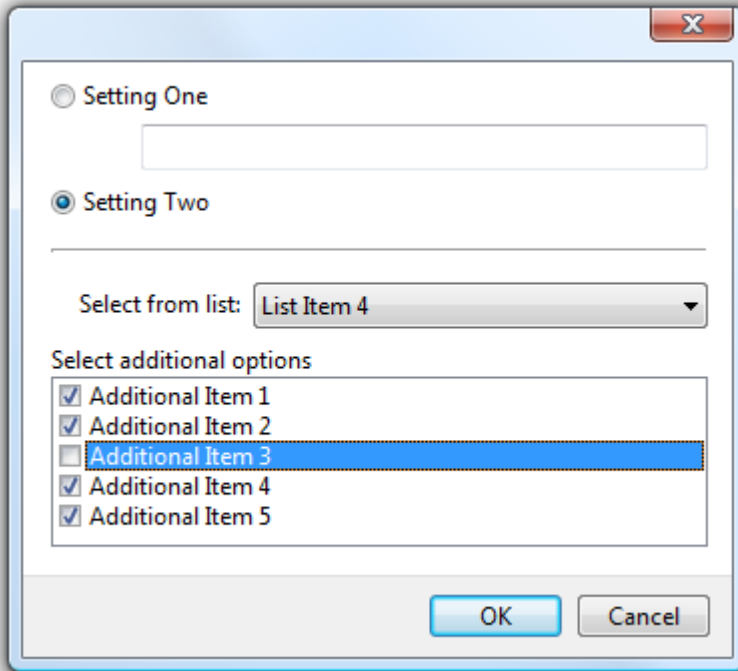
It is now time to run the application again.



Clicking OK will then display the changes we want to the main form. The screen capture below shows these changes.



Displaying the dialog again and setting different values such as the following are also reflected correctly when the OK button is pressed.



Adding a custom Apply button

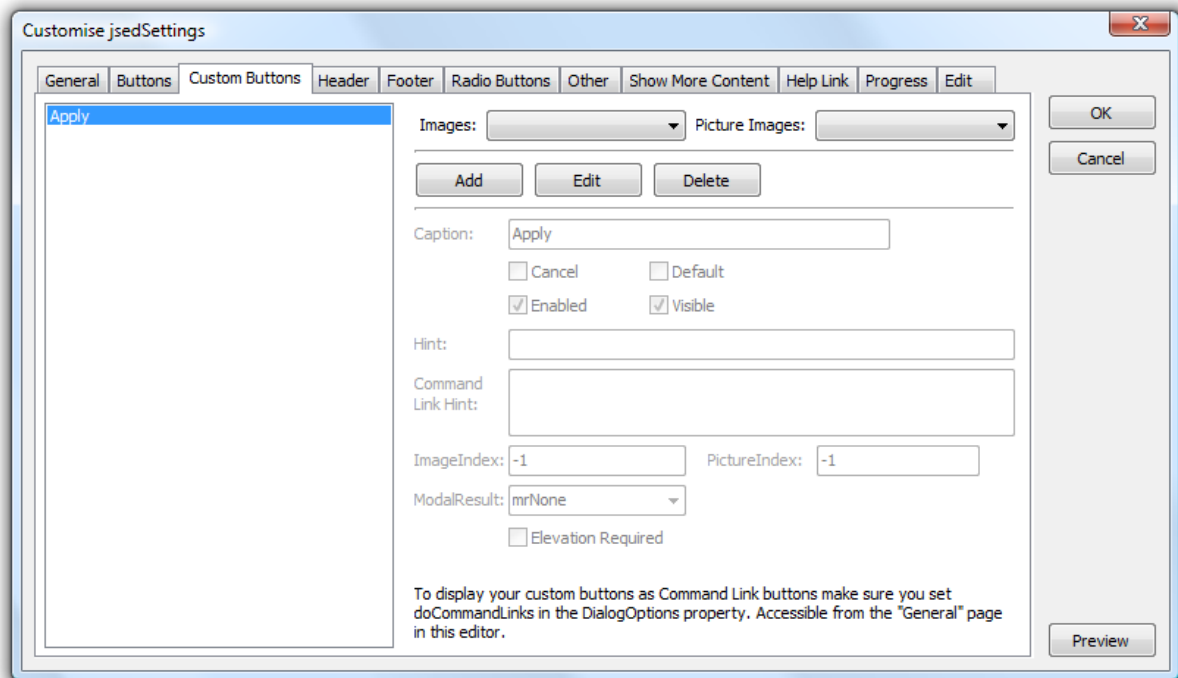
There is another way to access the edit control instance of the dialog while it is running. This is by using the EditControls property. This is an array property but like the Index parameter of the edit specific events already mentioned this is reserved for future purposes. This means that you can pass any value to the EditControls property since the Index value is not validated.

Let's use this property by adding an Apply button to the dialog. So when the user clicks the Apply button the changes are made to the form automatically.

To do this we need to add a custom button to the dialog.

1. Select the dialog in the designer and double click it to display the component editor.
2. Select the Custom Buttons tab.
3. Click on the Add button to add a new custom button to the dialog.
4. Enter in the buttons caption (Apply).
5. Click on the Yes button to save the new custom button.

The updated component editor.

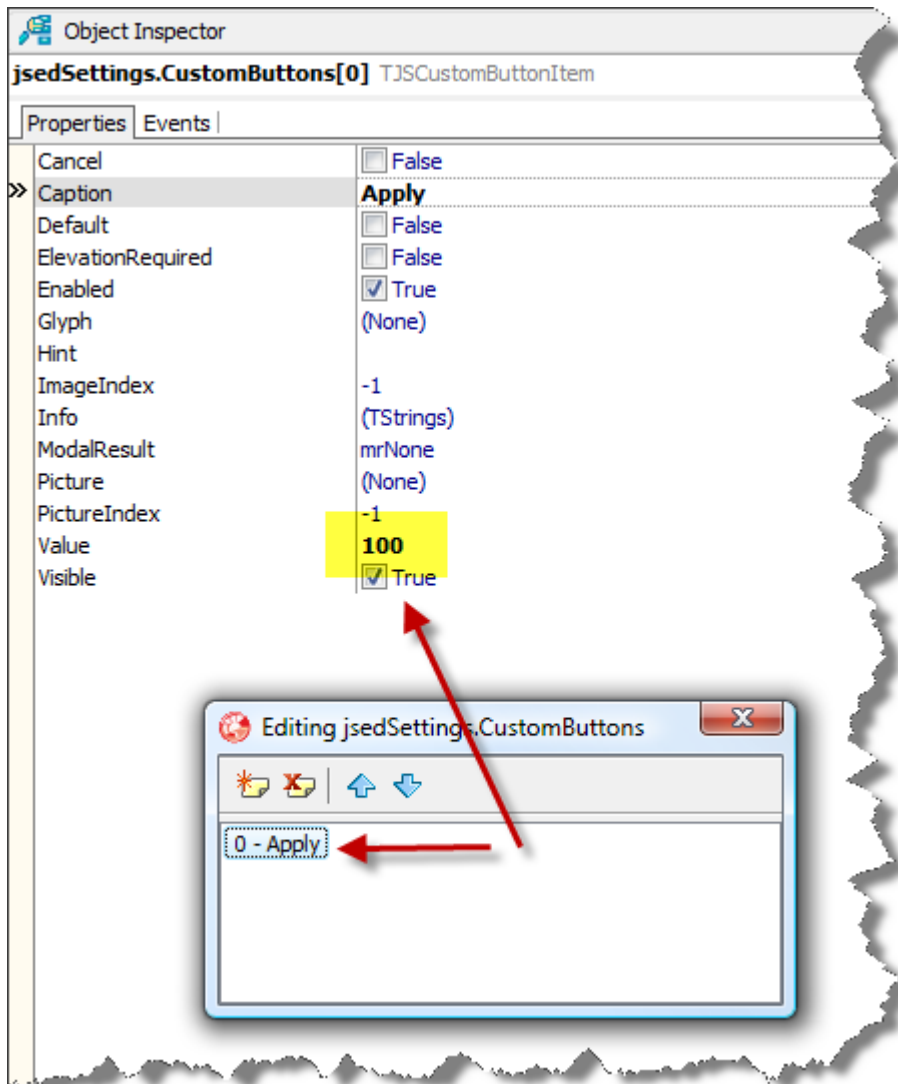


Close the component editor by clicking the OK button.

To illustrate an important point, run the application and click on the new Apply button.

What happens... well clicking on the Apply button closed the dialog! This is because each button will automatically close the dialog unless you tell the dialog that it shouldn't. Let's implement this, again by handling a specific event.

First let's look at the custom button in the object inspector. To do this double click on the CustomButtons property value for the dialog and select the Apply button from the list. What we want is the value assigned to the Value property. You cannot change this property manually. The screen capture below shows the selected custom button and the properties associated with it.



The first button will always have a value of 100. Each additional button will increment this value by 1. The order is how they appear in the list of Custom Buttons. There is a constant defined in the JSDialog unit that you should use to represent the 100 value. This constant is called `BASE_CUSTOMBUTTON`.

To respond to the user clicking on the Apply button, we need to handle the `OnControlClick` event.

Double click on the event to generate it.

The event has five parameters:

1. Sender – references the TJSEditDialog component.
2. ControlType – specifies the internal control type of the control that generated the event. For our purposes we want to respond to the event when the ControlType is `ctCommandLink`. Custom buttons are defined as `CommandLink` controls even if they are just being displayed as a normal button.
3. Control – references the instance of the control that has generated the event.
4. Value – the value of the event. This is where you can filter out the exact control you want. This value responds to the Value property set for Custom Buttons. So we want to Apply the

dialog changes when the control type is `ctCommandLink` and value is `BASE_CUSTOMBUTTON`.

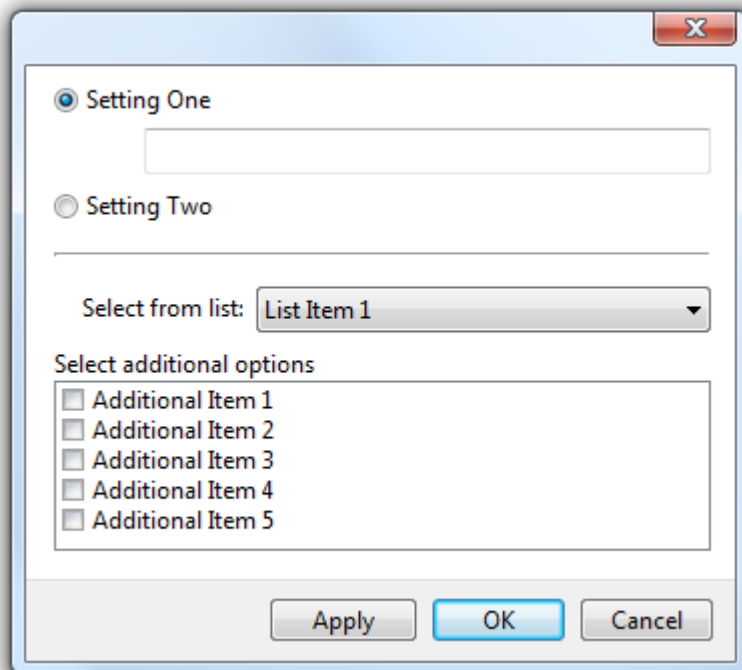
NOTE: When the control type is `ctButton` (for normal buttons) the value parameter is set to the `ModalResult` value.

5. **Handled** – this var parameter allows you to tell the dialog that you have handled all the functionality for this control so don't perform any default processing. This basically tells buttons not to close the dialog when pressed.

Knowing this information, we can implement the code below to stop the Apply button from closing the dialog when clicking it.

```
procedure TfrmMain.jsedSettingsControlClick(Sender: TObject;
  ControlType: TControlType; Control: TControl; const Value: Integer;
  var Handled: Boolean);
begin
  if ControlType = ctCommandlink then
  begin
    if Value = BASE_CUSTOMBUTTON then
      Handled := True;
    end;
  end;
end;
```

Running the application now will confirm the expected behaviour.



So now we can prevent the dialog from closing we now want to apply the same logic to the main form when the user clicks either the OK or Apply buttons on the dialog. First we should extract the core logic from the `OnGetEditValue` event so we can share it between the methods.

Since the EditValue property is readonly, we can only set it within the OnGetValue event. Armed with this the extracted method follows.

```
function TfrmMain.UpdateFormSettings(AControl: TControl): string;
var
  LSettings: TfrSettings;
  I: Integer;
begin
  LSettings := TfrSettings(AControl);
  if LSettings.rbSettingOne.Checked then
    Result := LSettings.eSettingOne.Text
  else
    Result := 'n/a';
  ListBox1.Items.Clear;
  for I := 0 to LSettings.clbAdditionalOptions.Items.Count - 1 do
  begin
    if LSettings.clbAdditionalOptions.Checked[I] then
      ListBox1.Items.Add(LSettings.clbAdditionalOptions.Items[I]);
  end;
  case LSettings.cboxListItems.ItemIndex of
    0: Color := clBtnFace;
    1: Color := clWhite;
    2: Color := clYellow;
    3: Color := clLime;
  end;
end;
```

This means that the OnGetEditValue method has changed to call this new method and it is below.

```
procedure TfrmMain.jsedSettingsGetEditValue(Sender: TObject; AControl: TControl;
const Index: Integer; var Value: string);
begin
  if TJSEditDialog(Sender).ModalResult = mrOK then
  begin
    Value := UpdateFormSettings(AControl);
  end;
end;
```


Call the UpdateFormSettings method on the OnControlClick handler. We use the EditControl property of the dialog to get the actual frame control instance. Also the validation logic is not called automatically, so we should also call that before making any changes.

The modified code is below. Note the use of the **EditControls** property to access the frame instance. This allows you to access anything on the frame you want.

```

procedure TfrmMain.jsedSettingsControlClick(Sender: TObject;
ControlType: TControlType; Control: TControl; const Value: Integer;
var Handled: Boolean);
var
LControl: TControl;
LMessage: string;
begin
if ControlType = ctCommandlink then
begin
if Value = BASE_CUSTOMBUTTON then
begin
LControl := TJSEditDialog(Sender).EditControls[0];
if TfrSettings(LControl).ValidateInput(LMessage, mrOK) then
Caption := UpdateFormSettings(LControl)
else if Length(LMessage) > 0 then
ShowErrorMessage(LMessage);
Handled := True;
end;
end;
end;

```



This will now allow us to apply settings from within the dialog without having to close the dialog first.

Controlling the state of the Apply button

Sometimes you might want a particular button disabled depending on a selection from your frame control. Since the frame has a reference to the dialog, it is possible to call the methods that control that state of buttons on the dialog.

To demonstrate this ability we will remove the need to call the `ValidateInput` directly on the frame when the user presses the Apply button. Instead the dialog's Apply button will only be enabled, when the contents in the dialog pass validation.

To control the enabled state of a button use the `EnableButton` method that is introduced in the `TJSCustomDialog` class. Remember we have access to the actual dialog instance using the `FJSDialog` field we created earlier so we can call dialog specific methods.

This method accepts the following parameters:

1. `aValue` - the value of the button you want to change. If your button is a custom button then you should use the same steps discussed in the "Adding an Apply button" section to get the value. For the Apply button it is the value of `BASE_CUSTOMBUTTON` (100). If you wanted to control the enabled state of a common button then you should pass the `ModalResult` value for that button as the button value parameter.
2. `aEnabled` – the value you want to change the `Enabled` property of the button to.

Since the validation logic is needed in a number of places we will create a new method for it.

```

procedure TfrSettings.CheckValidation;
var
  LDialogValid: Boolean;
  LMessage: string;
begin
  LDialogValid := ValidateInput(LMessage, mrOK);
  // Since we are silently validating the dialog, we won't display the
  // validation messages to the user.
  FJSDialog.EnableButton(BASE_CUSTOMBUTTON, LDialogValid);
end;

```

Note the comment about ignoring the returned message when validation fails. There is no reason why you couldn't output this message on to a label on the dialog for the user to see.

Depending on the controls on your frame will decide where you want to place the call to the CheckValidation method.

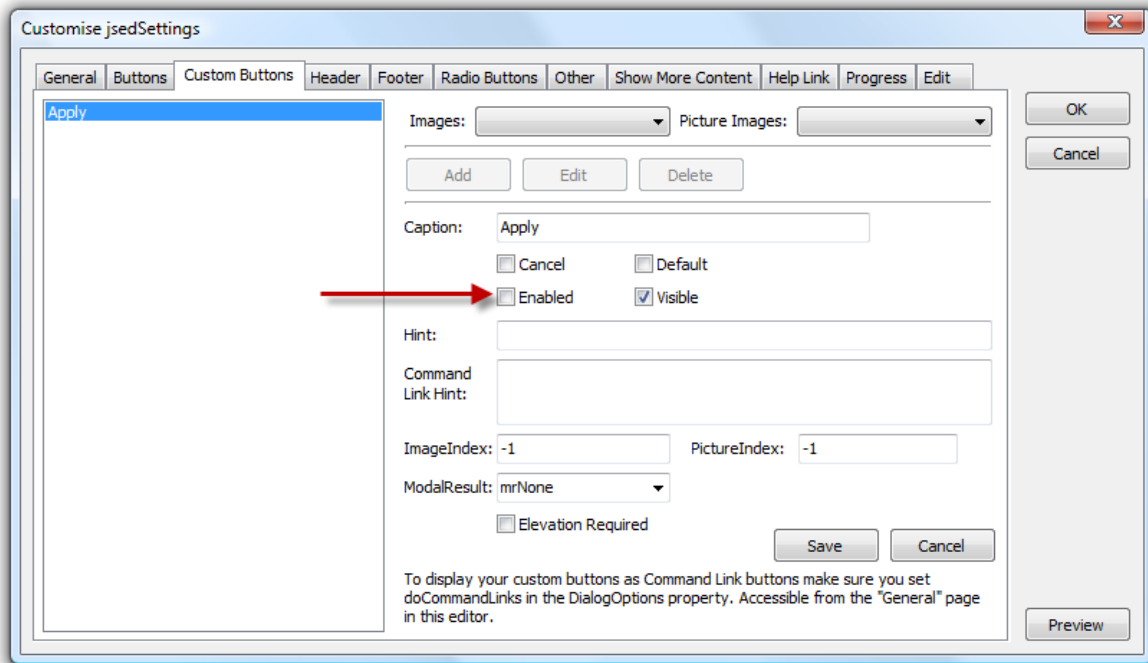
For the settings dialog, I've placed the call to the CheckValidation method on the following events:

- Radio button OnClick events.
- The edits OnChange event.
- The check list boxes OnClickCheck event.

Since the radio buttons OnClick event gets called when the frame is streamed, there is no need to manually call the CheckValidation method. This may not be the case for your dialogs, so you can either call the CheckValidation method by overriding the Loaded method of your frame or setting the custom button to be disabled by default when the dialog is shown.

To set the button as disabled, follow these steps:

1. Select the TJSEditDialog on your form.
2. Double click the component or right click and select the Customize... command.
3. Select the Custom Buttons tab.
4. Select the Apply button in the list of available custom buttons (our dialog only has one).
5. Click on the Edit button so you can modify the selected custom button.
6. Uncheck the Enabled check box.
7. Click the Save button to save changes.



You can also access custom buttons directly using the Object Inspector.

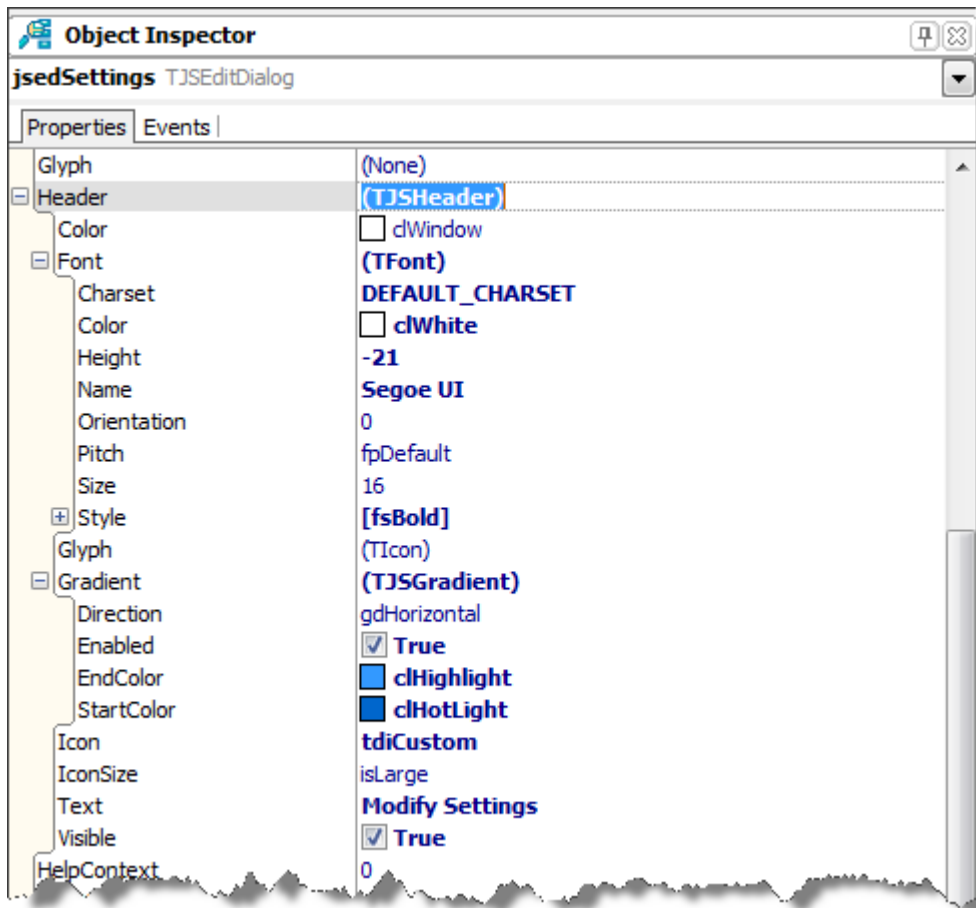
NOTE: There is also an `EnabledRadioButton` that works the same way except on the Radio Buttons displayed in a dialog.

Completing the dialog

Since the `TJSEditDialog` descends from `TJSDialog`, it has access to all of the functionality that `TJSDialog` has.

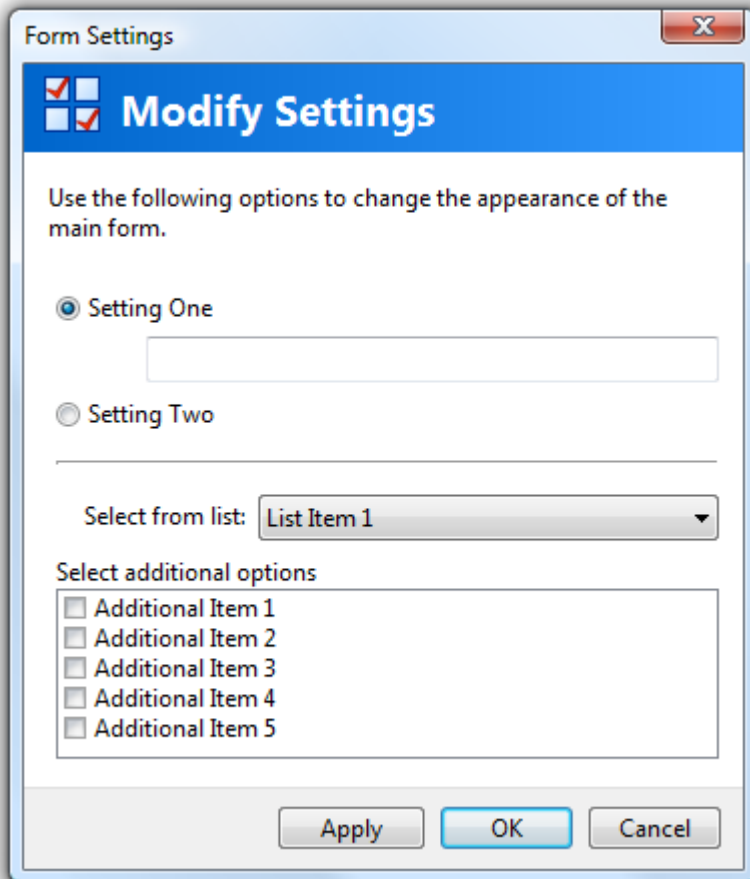
To jazz up the dialog let's give the dialog a gradient header, an image (using the `Header.Glyph` property) and some content text.

Below is a screen capture that displays the changes made to the `Header` property of the `TJSEditDialog` component.



The Title property was set to *Form Settings* and the content text was changed to *Use the following options to change the appearance of the main form.*

The end result of the changes.



Conclusion

Hopefully this has given you enough information to use some of the more advanced features of JSDialog Pack.