JEDI-VCL Tutorials

Content

1 Plugin tutorials

This chapter contains a number of tutorials. Most tutorials just expand on the basic tutorial. For more background information you should read the chapter regarding writing plugins and plugin enabled applications in the JEDI-VCL online help (you'll find it in the Additional Information section of the help).

The tutorials are divided into two groups: plugin writing and writing host applications.

Plugin writing tutorials

Tutorial	Description
Plugin 001	Basic plugin tutorial. The tutorial will create a simple plugin with one command that shows a message when activated.

Host writing tutorials

Tutorial	Description
Host application 001	Basic host tutorial. The tutorial will create a simple host application that allows all package plugins in it's executables folder to be loaded, adding it's commands to the tools menu.

1.1 Tutorials: Plugin 001: Basic plugin tutorial

This tutoral will create a simple plugin with one command in it (a simple 'Hello world!') example.

We start this tutorial by creating a new plugin project. Go to File|New..., select the **Projects** tab, select the **JEDI Plugin Wizard** and press OK. In the dialog that appears, type in the name of the plugin, in this case we'll enter "Tutorial1". Below the edit box are two options: **Package plugin** and **DLL plugin** where package plugin is selected by default. We'll leave this setting for now.

Pressing OK will create a new project **PlgTutorial1.dpk** and a unit called **PluginTutorial1.pas**. The unit contains a data module named **Tutorial1**. When you select that module, the object inspector will show some additional properties and events.

First thing to do is to specify the ID of the plugin for easy identification by the host application. Set PluginID property to **JEDI.Tutorial1**. By including the **JEDI.** part, we make sure that this tutorial is unique in the host application, even if another company has written a plugin with the same name.

To make the plugin selection interface in the host application more usefull, we can now assign proper values to the following properties:

Property	Value
Author	Your name, your companies name or both e.g. **John Doe**
Copyright	The copyright notice for your plugin, e.g. **Copyright © 2003 Project JEDI**.
Description	A short description of the plugin, e.g. **My first plugin**.
PluginVersion	The version of your plugin, e.g. **1.0** or **1.0 RC1**.

Our tutorial plugin will provide one command that will just show a message box with the text 'Hello world!' (programmers are so creative when they write tutorials and examples ;)).

To create the command we'll doubleclick on the Commands property to show a familiar editor. Press **Insert** to create a new command. The object inspector will now show the properties and events for our command. The **property will be set to 'Command1' by default. We'll set the Caption property to 'Test1'.**

Next, doubleclick on the OnExecute event to generate a new event. In this event we'll show our most important message. The code will look like this:

```
procedure TTutorial1.Tutorial1Commands0Execute(Sender: TObject);
begin
  ShowMessage('Hello world!');
end;
```

Compile the package. A message will popup saying that the JVCL runtime package should be added to the requires node. If the host application specifies it uses the JVCL runtime package, you should add it, otherwise you can ignore the message and the warning that follows. See also the discussion about DLL vs. packages plugins in the JEDI-VCL online help (you'll find it in the Additional Information section of the help). If you plan on using the the test host as described in the basic host application tutorial, you can safely add the JVCL runtime package (this will automatically recompile the plugin).

At this point your first plugin is ready. Make sure the plugin .bpl file is placed in the same folder as your test host, and run the test host. A suitable test host can be build by following the basic host application tutorial. If all is well, you should now see a new menu item in your test host and when you click on it, you'll see our message.

1.2 Tutorials: Plugin hosts 001: Basic host application

This tutorial will create a simple application that accepts plugins found in it's executable's folder. The tutorial is the basis for most of the other tutorials. Every component dropped on the form will keep it's default name, unless stated otherwise.

We start this tutorial by creating a new application. Drop a TMainMenu on the form. The menu should have a **File** and a **Tools** menu. The file menu will have a two sub items: **Load plugins** and **Exit**. The tools menu will be left empty. It will be filled with the commands provided by the loaded plugins.

Next we drop a TJvPluginManager component on the form. We assume package plugins will be used, so we set the PluginKind property to **plgPackage**. You'll notice the Extension property changes to **bpl** automatically. We leave PluginFolder blank, which will make LoadPlugins load all packages found in the executable's folder.

Head over to the events of the TJvPluginManager and double click on the OnNewCommand event to generate a new handler. The handler will have the following implementation:

```
procedure TForm1.JvPluginManager1NewCommand(Sender: TObject; ACaption, AHint,
  AData: string; ABitmap: TBitmap; AEvent: TNotifyEvent);
var
  MI: TMenuItem;
begin
  MI := TMenuItem.Create(Self);
  try
    MI.Caption := ACaption;
    MI.Hint := AHint;
    MI.Bitmap := ABitmap;
    MI.OnClick := AEvent;
    Tools1.Add(MI);
  except
    MI.Free; // error occured, cleanup menu item
    raise;
  end;
end;
```

The code in this handler will add a new menu item to the tools menu for each command found in each loaded plugin.

Next we will implement the **Load plugins** menu item's OnClick event. The handler will get the following code:

```
procedure TForm1.Loadplugins1Click(Sender: TObject);
begin
  JvPluginManager1.LoadPlugins;
end;
```

This will load all plugins in the executables folder.

Implement the **Exit** menu item's OnClick event. The handler should just call the Close method of the form.

We'd prefer the test hosts to use all runtime packages so we will set the **Build with runtime package** option (go to **Project|Options** and select the **Package** tab), making sure the JVCL runtime package is listed in the edit below it (if it's not there, just add **;JVCL200R?0**, replacing the **?** with your Delphi version). The reason we do this is to make the plugin writers job a little easier (if we wouldn't add the JVCL package to the host, the plugin writer would get a warning every time the plugin is compiled).

At this point the basic host application is ready. Make sure you have package plugin(s) in the folder where this executable is written to, run the application and go to **File|Load plugins**. All plugins in the folder will be loaded. Go over to the **Tools** menu and all commands of all loaded plugins should be there. **Note** if the **Tools** menu remains empty, make sure the plugins are package plugins and their .bpl files are in the same folder as the one where the host application's executable was generated. If the menu still remains empty, check that the plugins have commands available (not all plugins need to provide commands).

To create a plugin for our test host, you can follow the basic plugin writing tutorial.

2 Scheduled events tutorials

This chapter contains a number of tutorials. Most tutorials just expand on the basic tutorial.

The following tutorials are available:

Tutorial	Description
ScheduledEvents 001	Basic scheduled events tutorial. The tutorial will create a simple application that generates two recurring events, adding the information to a RichEdit control.
ScheduledEvents 002	This tutorial will show how you can add and manage scheduled from code.

2.1 Tutorials: ScheduledEvents 001: Basic tutorial

This tutorial will create a simple application that generates two recurring events, adding the information to a RichEdit control.

To start the tutorial you should create a new application. Name the form **frmTestSched** save the unit as **TestSchedMainForm** and the project as **TestSched**.

Drop a **TRichEdit** on the form and set the **Align** property to **alClient**. Rename the component to **reLog**.

Next, drop a TJvScheduledEvents component on the form. Rename it to **seTester**. Now open the events editor by either right-clicking the component and choose Events editor or doubleclicking the Events property or by clicking the ellipsis button next to the Events property. In all cases you'll end up with an editor much like the one for TActionList.

Once you're in the Events editor you can press **Insert** to add a new event. The Object Inspector will show the properties of this new scheduled event. We leave the default name of Event1. There are not many properties and there's only 1 event. The most important property is the **Schedule** property as it will determine when the handler will be triggered.

Activate the schedule editor by either doubleclicking the Schedule property or clicking the ellipsis button to the right of the property. The schedule editor will popup. The left side is the editing area where you'll set the schedule. To the right of the screen is the testing area that allows you to check if the specified schedule will do what you want it to do.

On the left side of the editor you'll see the start date and time (currently set to the date and time this scheduled event was created), with below that the schedule type and a large empty area. The empty area will be populated with additional settings when you set the schedule type to something else than One-shot. The part to the right of the schedule type options will show the settings appropiate to that specific recurring type, while the part below it will show the setting to specify the daily frequency (when will events be generated) and the range of the recurring schedule (when will the schedule end).

Our first event will be a recurring job, firing every day and every 15 seconds of that day. First we set the schedule type to **Daily**. Immediately the right side will show the settings for a daily job. We have two options: **Every weekday** (runs every monday to friday) and **Every *x* day(s)** (runs every *x* day(s), where 1 means every day, 2 will skip 1 day, etcetera). We want the job to run every day, so we select the second option. We'll keep the default value of **1** in the edit box.

In the area below (**Daily frequency**) we have again two options: **Occurs once at** (the event will be fired only once on the requested day(s) at the specified time) and **Occurs every *interval* between *start time* and *end time*** (the event will be fired between the specified times with the specified interval). We want the job to run every 15 seconds so we select the second option. Type **15** in the edit box and select **Seconds** from the drop down box. Leave the start time set to **00:00:00** and the end time set to **23:59:59**.

Now it's time to test the schedule. Press the **Run** button in the testing area. The box below will fill with the dates and times the scheduled event would fire. The button will change to **Stop** to allow you to end the generating process and investigate the list more closely. Just press the **Stop** button when you feel you have enough trigger times to assert the schedule is correct. The list contains the following information (from left to right):

Item	Description
Trigger counter	Contains the event number. Every event generated (or missed depending if the **Count missed events** is specified).
(Day count)	Contains the number of days on which events are generated (or missed if the **Count missed events** is specified).
Date@Time	The date and time at which the event will be generated.

The test will start with events from the current date and time (at the moment you pressed the **Run** button) and missed events are counted (which is why the first number will not be 1, unless you changed the starting date and time to a date and time in the future). Unchecking the **Start with current date** will start the testing process from the start date and time given to the left (the first number in the list will be 1). Leaving the **Start with current date** checked and unchecking the **Count missed events** will start the testing process from the current date and time but will not count any events that should have occured before that time (the first number in the list will always be 1).

Now that we have verified the schedule, we can press OK to apply the schedule. You'll be back at the Object Inspector, showing the properties of the scheduled event. Head over to the events page and doubleclick the **OnExecute** event to generate a new handler. Implement the handler like this:

```
procedure TfrmTestSched.seTesterEvents0Execute(
  Sender: TJvEventCollectionItem; const IsSnoozeEvent: Boolean);
var
```

```
  OldCol: TColor;
begin
  OldCol := reLog.SelAttributes.Color;
  reLog.SelAttributes.Color := clRed;
  reLog.Lines.Add('[Event1 has fired!]');
  reLog.SelAttributes.Color := OldCol;
end;
```

This will add the text **[Event1 has fired!]** in red.

Now we'll add the second event. This event will again fire each day, but this time every 750 milliseconds. If you can still see the Events editor, select it. If you can't see it, press **Alt+0** or go to **View|Window List...** and select the **Editing seTester.Events** window.

In the events editor, press **Insert** to add the second event. Edit the schedule as described for the first event. This time you should set the daily frequency interval to 750 milliseconds. When you're done with the schedule, head over to the events page and doubleclick the **OnExecute** event to generate a new handler. Implement the handler like this:

```
procedure TfrmTestSched.seTesterEvents1Execute(
  Sender: TJvEventCollectionItem; const IsSnoozeEvent: Boolean);
var
  OldCol: TColor;
begin
  OldCol := reLog.SelAttributes.Color;
  reLog.SelAttributes.Color := clGreen;
  reLog.Lines.Add('[Event2 has fired!]');
  reLog.SelAttributes.Color := OldCol;
end;
```

This will add the text **[Event2 has fired!]** in green.

Compile and run the application. Every .75 seconds the text **[Event2 has fired!]** will be added to the rich edit and every 15 seconds the text **[Event1 has fired!]** will be added. While the application run you are free to resize the form or even type in some text. The messages will be added at the appropiate times.

2.2 Tutorials: ScheduledEvents 002: Adding and managing schedules from code

This tutorial will show how you can add and manage schedules from code. For this tutorial we are going to extend the application created by the basic tutorial.

Select the main form and create an OnShow event for it. In this event we will add a new schedule in code. The schedule will be a recurring schedule on a weekly basis, ocurring every monday, wednesday and friday, every 15 minutes between 9:00 am and 5:00 pm.

First we declare a local variable to hold the new scheduled event (the full implementation is repeated at the end of the tutorial):

```
procedure TfrmTestSched.FormShow(Sender: TObject);
var
  SE: TJvEventCollectionItem;
begin
end;
```

Adding new events is done by calling Add on the Events property of the component:

```
  SE := seTester.Events.Add;
```

The SE variable will now hold the new scheduled event. It will be in the uninitialized state until the Start method is called. We will first setup the schedule.

The first thing to do is specify the schedule type:

```
  SE.Schedule.RecurringType := srkWeekly;
```

We've now set the schedule to a weekly recurring job. Now we will specify the days on which the event should be triggered:

```
  (SE.Schedule as IJclWeeklySchedule).DaysOfWeek := [swdMonday, swdWednesday, swdFriday];
```

The schedule will now fire every monday, wednesday and friday but only at midnight. To correct this, we change the properties of the daily frequency:

```
  with (SE.Schedule as IJclScheduleDayFrequency) do
  begin
    Start := 9 * 60 * 60 * 1000; // 9:00 am expressed in milliseconds
    End := 17 * 60 * 60 * 1000; // 5:00 pm expressed in milliseconds
    Interval := 15 * 60 * 1000; // 15 minutes expressed in milliseconds
  end;
```

The schedule is now completed. Next we assign a event handler to the OnExecute event. First we copy the handler of one of the scheduled events from the base tutorial and rename it to **seTesterEvents2Execute** (don't forget to copy both the declaration as well as the implementation. The implementations should be changed to:

```
procedure TfrmTestSched.seTesterEvents2Execute(
  Sender: TJvEventCollectionItem; const IsSnoozeEvent: Boolean);
var
  OldCol: TColor;
begin
  OldCol := reLog.SelAttributes.Color;
  reLog.SelAttributes.Color := clNavy;
  reLog.Lines.Add('[Event3 has fired!]');
  reLog.SelAttributes.Color := OldCol;
end;
```

This will add the text **[Event3 has fired!]** in navy blue. Now assign the handler to event in the OnShow event of the form and activate the scheduled event:

```
  SE.OnExecute := seTesterEvents2Execute;
  SE.Start;
```

The complete OnShow event implementation is thus as follows:

```
procedure TfrmTestSched.FormShow(Sender: TObject);
var
  SE: TJvEventCollectionItem;
begin
  SE := seTester.Events.Add;
  (SE.Schedule as IJclWeeklySchedule).DaysOfWeek := [swdMonday, swdWednesday, swdFriday];
  with (SE.Schedule as IJclScheduleDayFrequency) do
  begin
    Start := 9 * 60 * 60 * 1000; // 9:00 am expressed in milliseconds
    End := 17 * 60 * 60 * 1000; // 5:00 pm expressed in milliseconds
```

```
      Interval := 15 * 60 * 1000; // 15 minutes expressed in milliseconds
    end;
    SE.OnExecute := seTesterEvents2Execute;
    SE.Start;
  end;
```

If you compile and run this application, the usual messages about the two schedules created in the base tutorial should appear at their normal intervals. In addition, if this application is run on monday, wednesday or friday between 9:00 am and 5:00 pm, an additional message will appear every 15 minutes. On all other days or times outside of the schedule, this message will not appear.

Index

P

S

T