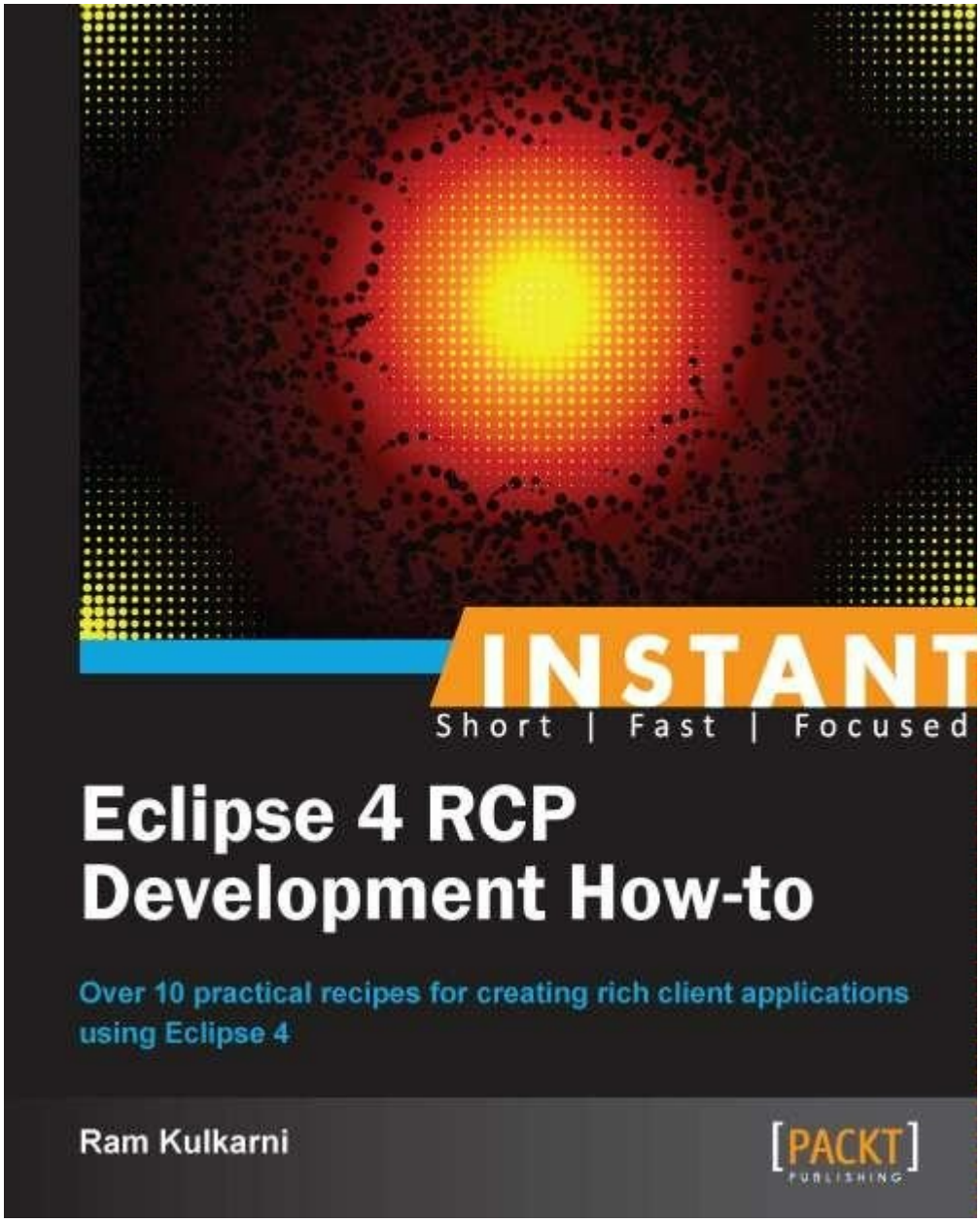

INSTANT
Short | Fast | Focused
Eclipse 4 RCP Development How-to
Over 10 practical recipes for creating rich client applications using Eclipse 4
Ram Kulkarni
[PACKT]
PUBLISHING

Table of Contents

Instant Eclipse 4 RCP Development How-to
Credits
About the Author
About the Reviewer
www.PacktPub.com
    Support files, eBooks, discount offers and more
        Why Subscribe?
        Free Access for Packt account holders
Preface
    What this book covers
    What you need for this book
    Who this book is for
    Conventions
    Reader feedback
    Customer support
        Downloading the example code
        Errata
        Piracy
        Questions
1. Instant Eclipse 4 RCP Development How-to
    Setting up a development environment (Simple)
        Getting ready
        How to do it...
        How it works...
        There's more...
            Eclipse concepts
    Creating a skeleton E4 application (Simple)
        How to do it...
        How it works...
        There's more...
    Adding menu and toolbar items (Intermediate)
        How to do it...
        How it works...
        There's more...
            Using direct menu items
            Compiler warnings for E4 APIs
    Adding views (Intermediate)
        Getting ready
        How to do it...
        How it works...
        There's more...
            Setting container data of parts
            Setting layout data
    Injecting your own objects (Advanced)

- Getting ready
- How to do it...
- How it works...
- There's more...

Creating a pop-up menu (Intermediate)
- How to do it...
- How it works...
- There's more...
  - Adding mouse listener to the TableViewer class
  - Creating core expression
  - Setting the core expression for Menu Item

Creating custom events and handlers (Intermediate)
- Getting ready
- How to do it...
- How it works...
- There's more...
  - Handling selection change events in the TableViewer class
  - Saving snippet data

Adding a keyboard shortcut (Simple)
- How to do it...
- How it works...

Creating custom objects using DI (Intermediate)
- How to do it...
- How it works...
- There's more…
  - Set initial size of SearchDialog
  - Perform a search operation on OK
  - Implement getter methods for SearchDialog members

Creating views dynamically (Advanced)
- Getting ready
- How to do it...
- How it works...
- There's more...
  - Displaying search results

Styling an application using CSS (Simple)
- How to do it...
- How it works...
- There's more...
  - Finishing the application

Customizing and exporting the application (Simple)
- How to do it...
- How it works...

Instant Eclipse 4 RCP Development How-to

Instant Eclipse 4 RCP Development How-to

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: May 2013

Production Reference: 1160513

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78216-952-9

www.packtpub.com

Credits

Author

Ram Kulkarni

Reviewer

Sreenivas Ramaswamy

Acquisition Editor

Kartikey Pandey

Lead Technical Editor

Harsha Bharwani

Technical Editors

Kaustubh S. Mayekar

Sharvari Baet

Project Coordinator

Siddhant Shetty

Proofreaders

Aaron Nash

Paul Hindle

Production Coordinator

Shantanu Zagade

Cover Work

Shantanu Zagade

Cover Image

Conidon Miranda

About the Author

Ram Kulkarni has more than 20 years of experience in developing software. He has architected and developed enterprise web applications, client-server and desktop applications, application servers, IDEs, and mobile applications. He works at Adobe and has implemented many features of the ColdFusion server and ColdFusion Builder. He blogs at ramkulkarni.com.

I am grateful to Ananth P.N., Director of Engineering at Adobe Systems India Pvt. Ltd, for encouraging me to take up the task of writing this book and being a mentor.

I would like to thank Mohammad Rizvi, Harsha Bharwani, and Siddhant Shetty from Packt Publishing for giving me an opportunity to write this book and for helping me in deciding the content and format.

Finally, I would like to thank my parents, my wife Vandana, and son Akash for their love and support.

About the Reviewer

Sreenivas Ramaswamy has nearly 20 years of experience in the IT industry, and has worked on C, C++, and Java for many years. He has a Bachelor's degree in Electronics and Communication. He has worked on 3D graphics products such as Studio and Maya in his earlier years while working at Tata Elxsi. He has been working at Adobe Systems (Bangalore) for the past 8 years. To begin with, he worked on many of the UI components of the Flex framework for the Flash Platform. In recent years, he has been developing many features in FlashBuilder, an Eclipse-based IDE for the Flex framework.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to our book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at <service@packtpub.com> for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



http://PacktLib.PacktPub.com

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Preface

Eclipse has been a popular integrated development environment (IDE) for Java developers for many years now. The extensible framework of Eclipse also enables you to create IDEs for different languages. Many IDEs have already been developed on top of Eclipse, for example, for languages such as PHP, Ruby, and so on.

In addition to developing IDEs, Eclipse can also be used to create traditional desktop applications, which are also known as rich client applications. Prior to Eclipse 4, the APIs for developing RCP applications using Eclipse were quite difficult to use. Eclipse 4 has introduced many new concepts and APIs that make development of RCP application a lot easier.

This book explains how to develop RCP applications using Eclipse 4. It takes more of a hands-on approach by taking an example of use-case, and explains in detail how to implement an RCP application for that use-case. In the process, it explains new APIs in Eclipse 4 and covers typical tasks that you need to perform when developing an RCP application. It covers an entire workflow, from setting up the development environment to packaging the application.

What this book covers

Setting up development environment (Simple), will explain the software required to develop the sample RCP application and locations from where you can download it. It also explains how to configure Eclipse 4 IDE for RCP development.

Creating skeleton E4 application (Simple), will help us to create a skeleton RCP application using the Eclipse 4 RCP application wizard. We will also understand different files created by the wizard. We will build our sample application by modifying this skeleton application.

Adding menu and toolbar item (Intermediate), will help us to create custom menus and a toolbar item using the Eclipse 4 application model. We will learn about the concept of dependency injection (DI) and how Eclipse 4 framework uses it to call menu and toolbar handlers.

Adding views (Intermediate), will teach us how to create static views. We will split the main application window into two views. We will create implementation classes for these views. We will also understand different types of containers in the Eclipse 4 application model.

Injecting your own objects (Advanced), will walk us through the dependency injection (DI) framework of Eclipse 4, which makes many system objects

available to our applications. However, there are times when you would want to inject your own objects in the framework to make them available to different objects in the application. This recipe explains how to do this.

Creating a pop-up menu (Intermediate), will guide us to create a pop-up menu and display it dynamically with a mouse click. We will also learn how to use core expressions to show/hide items in the pop-up menu.

Creating custom events and handlers (Intermediate), will discuss the custom events that are useful to communicate messages between different parts of the application. In this recipe, we will learn to create and handle custom events. We will update one view when data in the other view is modified or added.

Adding a keyboard shortcut (Simple), will provide guidelines on creating a keyboard shortcut for some of the custom actions we implemented in the application.

Creating custom objects using DI (Intermediate), will walk us through the instantiation of objects using the DI framework, which is useful because many framework and custom objects become easily available to the new object.

Creating view dynamically (Advanced), will teach us how to create and display views dynamically.

Styling application using CSS (Simple), will discuss how to style user interface controls using CSS. We will set attributes such as background, font size, color, and so on using CSS.

Customizing and exporting the application (Simple), will teach us how to set splash image and icons for the application. We will then learn how to package and export the application for single or multiple target platforms.

What you need for this book

You will need JDK 1.7 or later and Eclipse 4 to run example code in this book.

Who this book is for

This book is for anyone who is looking to create cross-platform rich client applications. The Eclipse platform is built with Java, so knowledge of core Java is essential. The focus of this book is to understand new APIs and concepts of the Eclipse 4 platform. Prior knowledge of basic concepts of the Eclipse framework (plugin, extension, perspective, workspace, and so on), **Standard Widget Toolkit** (**SWT**), and JFace would benefit in understanding the examples in this book. However, the book provides links to these topics wherever needed.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "Type `Eclipse E4 Tools` in the name field."

A block of code is set as follows:

```
package codesnippetapp.handlers;

import org.eclipse.jface.dialogs.MessageDialog;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.e4.core.di.annotations.Execute;

public class FindCommandHandler {
  @Execute
  public void execute(Shell shell) {
    MessageDialog.openInformation(shell,
        "Find Command", "Find command executed");
  }
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
public void postConstruct(Composite parent) {
  snippetsList= new TableViewer (parent);
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Go to **File** | **New** | **Other**".

> ### Note
>
> Warnings or important notes appear in a box like this.

> ### Tip
>
> Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to <feedback@packtpub.com>, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting http://www.packtpub.com/submit-errata, selecting your book, clicking on the **erratasubmissionform** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from http://www.packtpub.com/support.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at <copyright@packtpub.com> with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at <questions@packtpub.com> if you are having a problem with any aspect of the book, and we will do our best to address it.

Chapter 1. Instant Eclipse 4 RCP Development How-to

Welcome to *Instant Eclipse 4 RCP Development How-to*. Eclipse is not only a great tool for creating IDEs but also for creating standalone, rich client applications. It provides an extensive framework for creating such applications and provides a rich set of **user interface** (**UI**) widgets. Eclipse Version 4 has introduced many new concepts and APIs, such as the application model, **dependency injection** (**DI**), and CSS styling. In this book, we will learn to build the **Rich Client Platform** (**RCP**) application using the new features of Eclipse 4.

An important part of learning any new technology is to build applications using that technology. We are going to follow the same approach. We will build a fully functional standalone application using the Eclipse 4 RCP framework, and in the process, we will explore and understand the new features of the Eclipse 4 RCP development.

The demo application that we are going to build in this book is called Code Snippets App. As the name suggests, it is a library of code snippets.

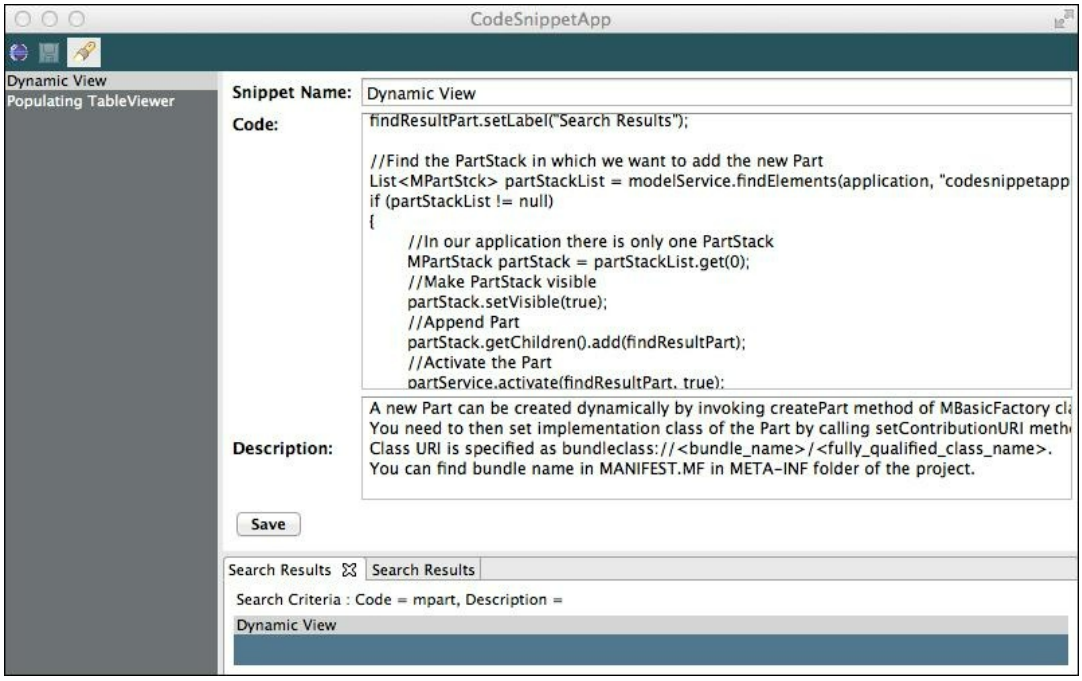
The application has two main views:

- The left-hand side view displays a list of snippets
- The right-hand side view displays the details of the snippet selected in the left-hand view

You can right-click on the left-hand view to display a pop-up menu. The menu has options to add and delete snippet.

Application has a menu bar that displays the **File** (**Open**, **Close**, **Quit**) and **Search** menu items. By going to **Search** | **Find**, a dialog box is displayed where search criteria can be entered. The search results are displayed in dynamic views at the bottom-right side of the application window.

As we develop this application, we will learn the following features of the Eclipse 4 RCP framework creating application model: using **Plain Old Java Object** (**POJO**) to create views, using DI framework of Eclipse 4, creating menus and toolbars, creating keyboard shortcuts, creating pop-up menus, using CSS to style widgets, and creating views dynamically. Finally, we will learn how to customize this application by adding a splash screen and icons. We will also see how to package the application for various platforms.



Setting up a development environment (Simple)

In this recipe, we will set up the development environment for creating Eclipse 4 RCP application.

Getting ready

Download the latest build of Eclipse from http://www.eclipse.org/downloads and unzip it. At the time of writing this book, the latest version of Eclipse was 4.2.1. On the Eclipse download page, you will see many packages available. Select `Eclipse for RCP and RAP Developers`. Make sure that you select the appropriate OS version. If your OS is 64 bit, you can download either a 32-bit or 64-bit version of Eclipse. If your OS is 32 bit, then you should download only a 32-bit Eclipse.

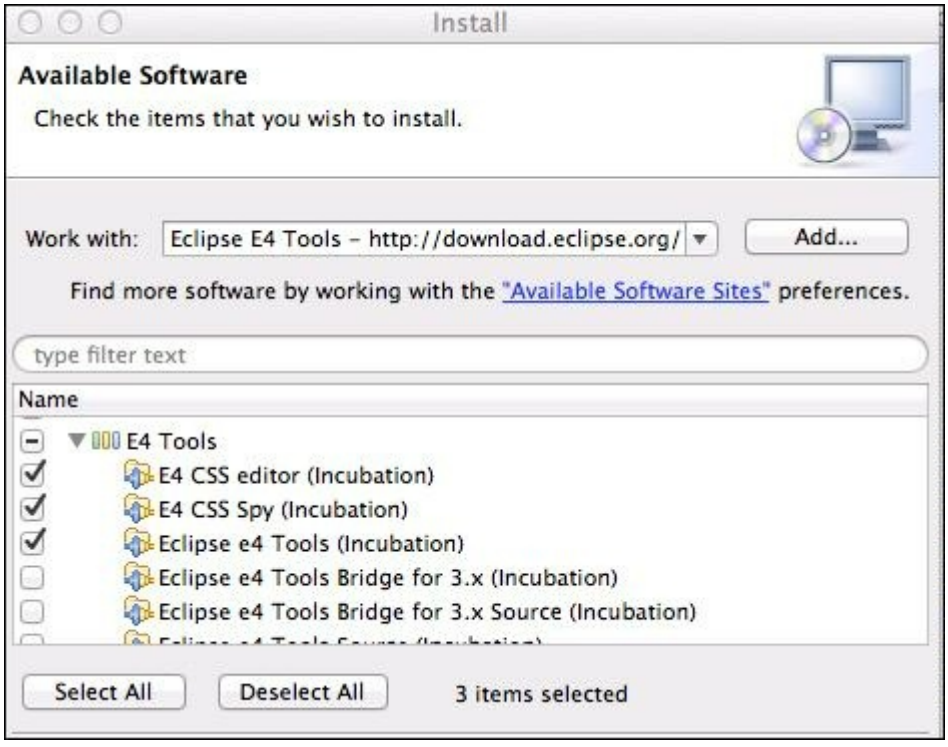
Download and install JDK from http://www.oracle.com/technetwork/java/javase/downloads/index.html if you haven't already. The demo application in this book is built with JDK 7, so you should preferably download JDK 7 or above if you want to follow code samples in this book.

Make sure that the Eclipse version you have downloaded is compatible with the JDK, that is, if you have downloaded 64-bit version of Eclipse, then the JDK must also be a 64-bit version.

Development of the RCP application is made easier by Eclipse E4 tools. The tools project is still in incubation, and hence, it is not part of the main Eclipse download. The following section describes how to set up appropriate environment for development.

How to do it...

1. Run Eclipse and go to **Help** | **Install New Software**.
2. Click on the **Add** button.
3. Type `Eclipse E4 Tools` in the name field. The name could be any text. However, providing a meaningful name makes it easier to find in the list of installed software.
4. Type `http://download.eclipse.org/e4/updates/0.12` in the **Location** field and click on **OK**.
5. In the **E4 Tools** group, select checkboxes as shown in the following screenshot:



6. Complete the wizard and restart Eclipse.
7. Verify that the tools are installed properly by going to **File** | **New** | **Other**. You should see **Eclipse 4** in the list of wizards.

How it works...

Though the plugins we installed in this section are not required for developing Eclipse 4 RCP applications, they make the process of development easy. The Eclipse e4 tools provide an easy-to-use visual

interface to edit application model. In the absence of this tool, you would have to edit an XML file to modify the application model and you will have to know the schema of the XML file.

E4 CSS Spy lets you inspect CSS properties of the UI components of your application when the application is running. It even allows you to modify CSS attributes at runtime. This would be useful if you want to try out different colors, fonts, backgrounds, and so on, on the running application.

E4 CSS editor provides features such as syntax coloring and code assist (intellisense) when editing CSS files in your project.

There's more...

If you want to target your application for multiple platforms, such as Windows, Mac, and Linux, then you should consider downloading Delta Pack for the Eclipse version you have downloaded. The Delta pack lets you export your RCP application to multiple platforms from the same development environment. For example, if your development setup is on Mac, you could export your application targeted for Windows/Linux/Mac platforms using the Delta pack.

Make sure that the Delta pack is for the same version of Eclipse that you are using for development. You can find the Delta pack for Eclipse Version 4.2 at http://download.eclipse.org/eclipse/downloads/drops4/R-4.2.1-201209141800/. Refer to http://download.eclipse.org/eclipse/downloads/ for downloading the Eclipse and Delta packs for other versions.

Eclipse concepts

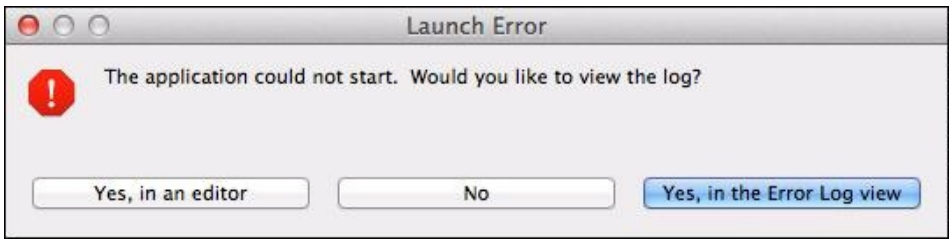
If you are new to Eclipse development, then you need to get yourself familiar with some of the basic concepts such as perspective, workspace, workbench, plugin, extension, extension point, and so on. These terms are frequently used in Eclipse APIs as well as documentation. Eclipse product help is an excellent source of information on these concepts. In Eclipse, go to **Help** | **Help Contents**. In the contents pane of the **Help** window, refer to **Workbench User Guide** | **Concepts and Plug-in Development Environment Guide** | **Concepts**.

Creating a skeleton E4 application (Simple)

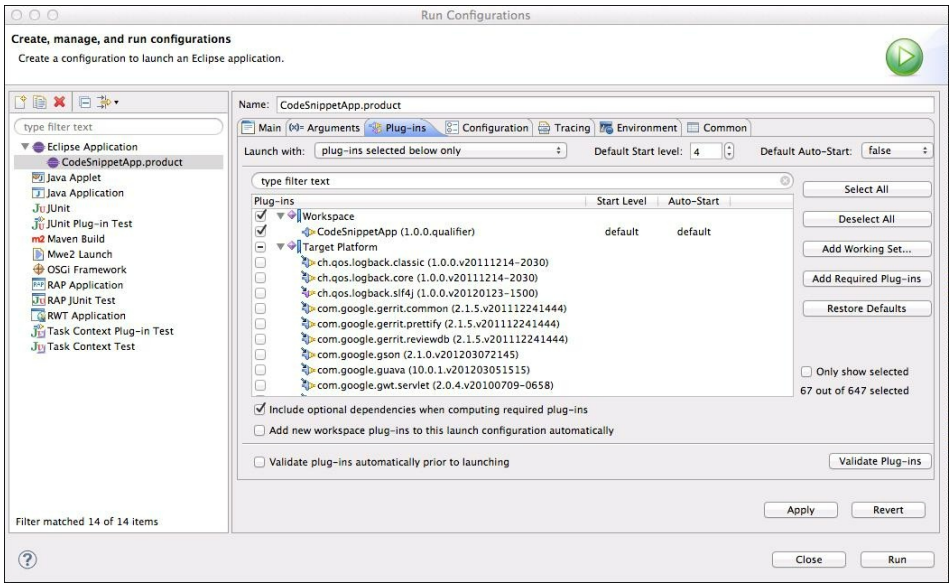
We will create a skeleton E4 RCP application in this task using the E4 tools wizard and look at the files generated by it. We will use this skeleton application to build our demo application.

How to do it...

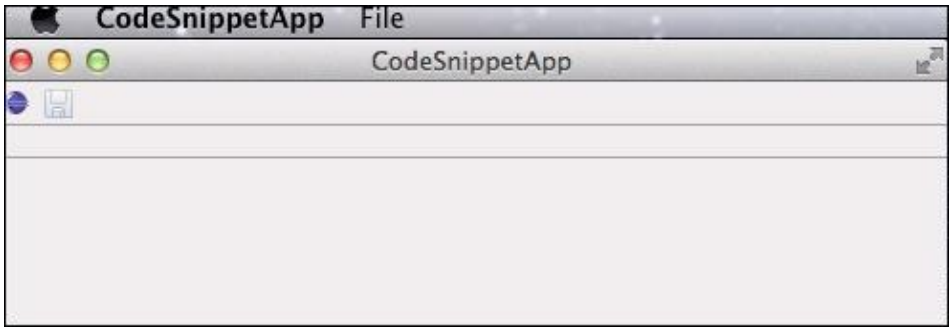
1. Go to **File** | **New** | **Other**.
2. Select the **Eclipse 4 Application Project** wizard under the **Eclipse 4** category.
3. In the **New Plug-in Project** wizard, enter the project name as `CodeSnippetApp` and click on **Next**.
4. Accept default values on the remaining pages of the wizard. When you complete the wizard, you will see the new project, **CodeSnippetApp**, in **Package Explorer**.
5. Before making any edits to the application, let's run it. Open `CodeSnippetApp.product` by double-clicking on it.
6. In the **Overview** tab, click on the **Launch an Eclipse** application link under the **Testing** group. You will most probably get the error shown in the following screenshot:



7. Click on the **No** button. This error occurred because some of the required plugins are missing from the **Run** configuration of the project. To fix this error, go to **Run** | **Run Configurations**.
8. Make sure that the **CodeSnippetApp.product** option is selected in the list on the left-hand side. Click on the **Plug-ins** tab. Click on the **Add Required Plug-ins** button.



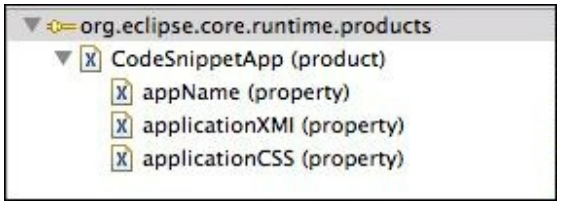
9. Click on the **Run** button. You should see **CodeSnippetApp** with a blank window, two menus, and toolbar items:



How it works...

The wizard creates a number of files. Refer to the following table:

File/Folder name	Description
`plugin.xml`	This contains the extension points that the plugin extends. The E4 application wizard adds one extension to the `org.eclipse.core.runtime.products` file. The information defined in this extension is used for branding the product. The product name appears in

File/Folder name	Description
	the title bar. Refer to the following properties: • `appName`: This property is used to create a folder when exporting the application for distribution • `applicationXMI`: This property contains the path to the application model file • `applicationCSS`: This property contains the path to the CSS file to be used to style some of the UI widgets of the application 
`CodeSnippetApp.product`	This contains the information for launching and branding the application.
`Build.properties`	These specify files and folders to be included in the source and binary builds of the application.
`Application.e4xmi`	This contains application model. You define views, menus, and toolbars of your application in this file. We will learn how to edit this file to create new views and menus later.
`META-INF/MANIFEST.MF`	This contains application metadata, for example, application name, version,

File/Folder name	Description
	and plugins (bundles) on which the application depends.
`icons`	This contains icons used in the application. The wizard creates two icon files, `sample.gif` and `save_edit.gif`, in this folder. Both the icons are used to create toolbar buttons.
`css`/`default.css`	This is the CSS file to style the look and feel of the application. We will learn how to use CSS in the E4 application later.
`src`/`codesnippetapp`/`Activator.java`	This is a class that handles some of the plugin lifecycle events such as start and stop. It can be used for initialization and cleanup of resources created in the application.
`src`/`codesnippetapp`/`handlers`	This folder contains classes for handling menu and toolbar actions, such as **Open**, **Save**, **Quit**, and **About**.

There's more...

We saw that when we try to run the application for the first time, we get an error saying **The application could not start. Would you like to view the log?**. If you choose to view the logfile, either in the editor or error log view, you will see following stack trace:

```
java.lang.RuntimeException: No application id has
been found.
at
org.eclipse.equinox.internal.app.EclipseAppContainer.startDefaultApp(Ecli
at
org.eclipse.equinox.internal.app.MainApplicationLauncher.run(MainApplicat
```

```
at
org.eclipse.core.runtime.internal.adaptor.EclipseAppLauncher.runApplicati
at
org.eclipse.core.runtime.internal.adaptor.EclipseAppLauncher.start(Eclips
…
```

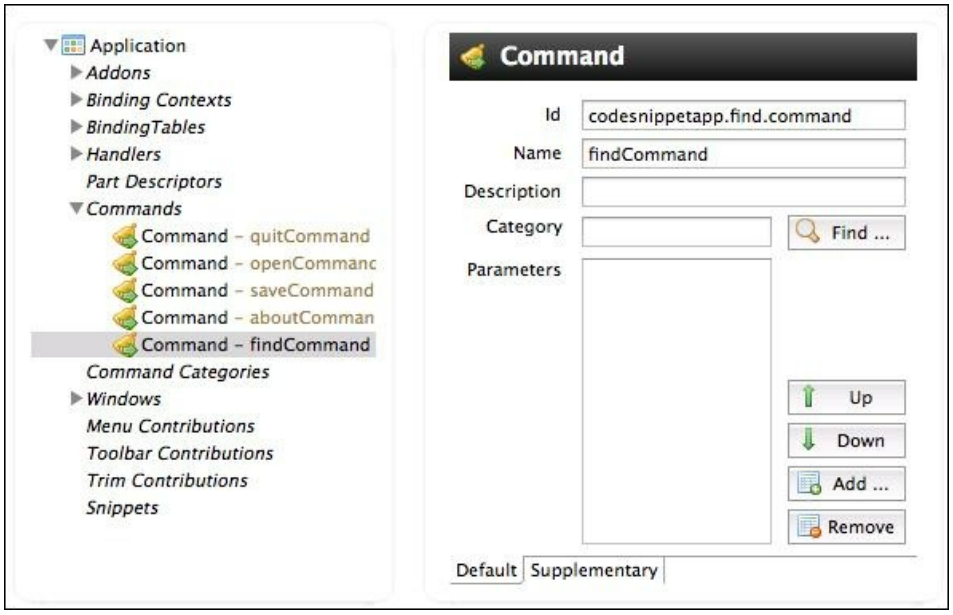
This error occurs commonly when you create a new application or modify application configuration by adding the new dependencies to it. The solution is the same as described in steps 8 and 9, where you open **Run Configuration** and add required plugins.

Adding menu and toolbar items (Intermediate)

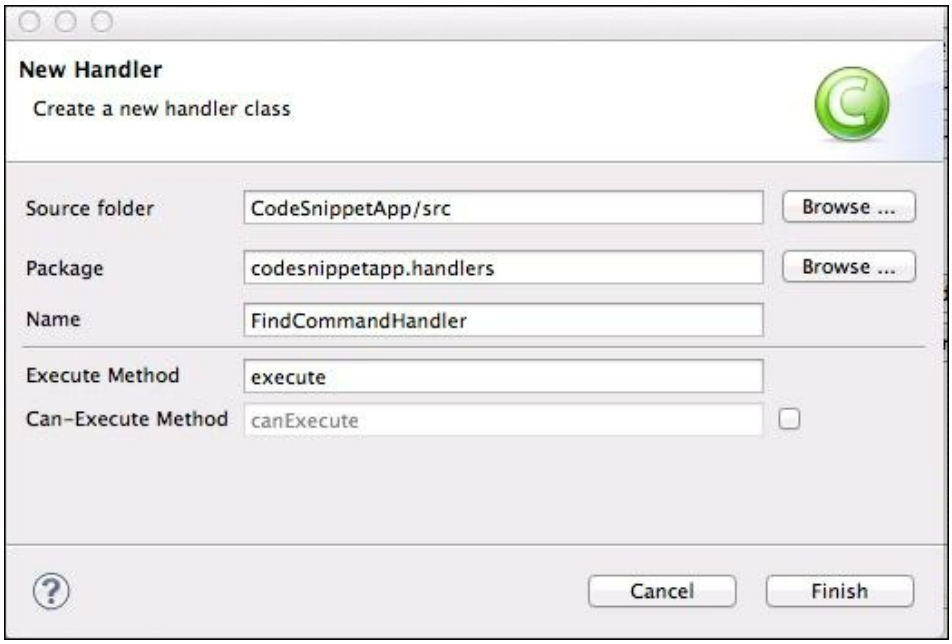
In this task, we will add the search menu in the main menu bar and also the find submenu. We will also add a shortcut button for the `Find` action in the toolbar. We will implement these features in the application we created in the previous recipe.

How to do it...

1. Open `Application.e4xml`, right-click on **Application**, and go to **Commands**.
2. Select **Add Child** | **Command**.
3. Change the ID of the new command to `codesnippetapp.find.command`.
4. Type `findCommand` in the **Name** textbox.



5. Right-click on **Application** and select the **Handlers** node, and then select **Add Child** | **Handler** from the pop-up menu.
6. Change ID of the handler to `codesnippetapp.find.handler`.
7. Click on the **Find** button next to the **Command** textbox. Select **findCommand** from the list and click on the **OK** button.
8. Click on the **Class URI** link. This will open a dialog box to create a new class for the handler.
9. Click on the **Browse** button to select **Package**. Then, select the `codesnippetapp.handlers` package.
10. Enter `FindCommandHandler` in the **Name** textbox and click on the **Finish** button.



11. The `FindCommandHandler` class is opened in the editor. We will not implement the `Find` functionality now, but we will just display a message in this handler. Add the code to display a message in the `execute` method. The code that you need to add is highlighted here:

```
package codesnippetapp.handlers;

import
org.eclipse.jface.dialogs.MessageDialog;
import org.eclipse.swt.widgets.Shell;
import
org.eclipse.e4.core.di.annotations.Execute;

public class FindCommandHandler {
  @Execute
  public void execute(Shell shell) {
    MessageDialog.openInformation(shell,
        "Find Command", "Find command
executed");
  }
}
```

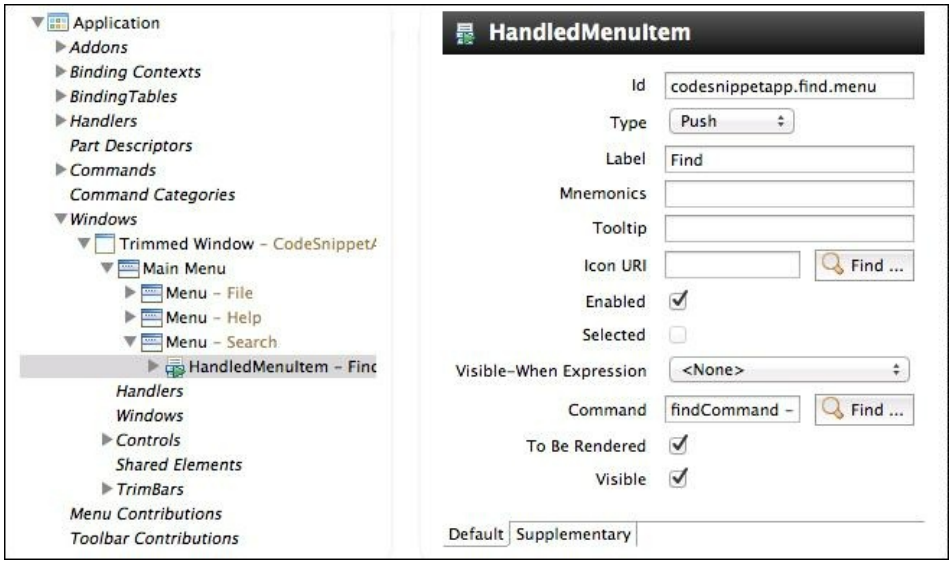
Tip

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book

elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

12. Go back to `Application.e4xmi`. Then, go to **Application** | **Windows** | **Trimmed Window** | |**Main Menu**.
13. Right-click on **Main Menu** and select **Add Child** | **Menu** from the pop-up menu.
14. Change ID of the menu to `codesnippetapp.search.menu`.
15. Type `Search` in the **Label** textbox.
16. Right-click on the newly added **Search** menu item.
17. Select **Add Child** | **HandledMenuItem** from the pop-up menu. Change ID of the menu item to `codesnippetapp.find.menu`.
18. Type `Find` in the **Label** textbox.
19. Click on the **Find** button next to the **Command** textbox. This will open a dialog box to select a command.
20. Select **findCommand** and click on the **OK** button.



21. Go to **Application** | **Windows** | **Trimmed Window** | **TrimBars** | **Window Trim** | **Toolbar** in the application model. Right-click on the **Toolbar** node.
22. Select **Add child** | **Handled ToolItem** from the pop-up menu.
23. Change ID of the new tool item to `codesnippetapp.find.toolbar`. Type `Find` in the **Label** textbox.
24. Click on the **Find** button next to **Command** textbox. This will open a dialog box to select from a list of commands.
25. Now, select **findCommand**.

Note

Optional: If you have an icon for the **Find** button, copy it in the `icons` folder. If you do not see the `icon` file in the `icons` folder that is in the `Package Explorer` folder (after adding the new icon), then right-click on the `icons` folder and select **Refresh**. Go back to the new toolbar item you added in the previous step and click on the **Find** button next to the **Icon URI** textbox. Type the icon name in the **Search** textbox and select the `icon` file.

If you do not add an icon for the toolbar button, then text that you have entered in the **Label** field will be displayed in the toolbar button.

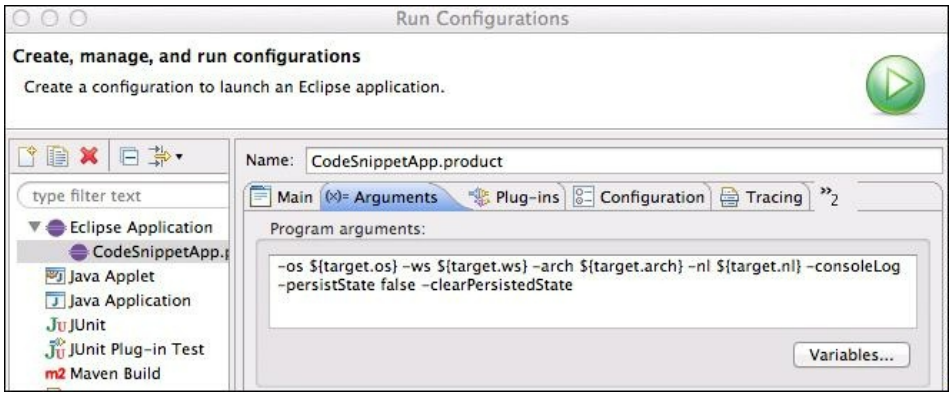
26. Run the application `()`.

Note

You may not see the menu and toolbar items that you just added. The reason is that Eclipse 4 caches the application model and reuses it. So, modifications to the application model are not reflected in the application.

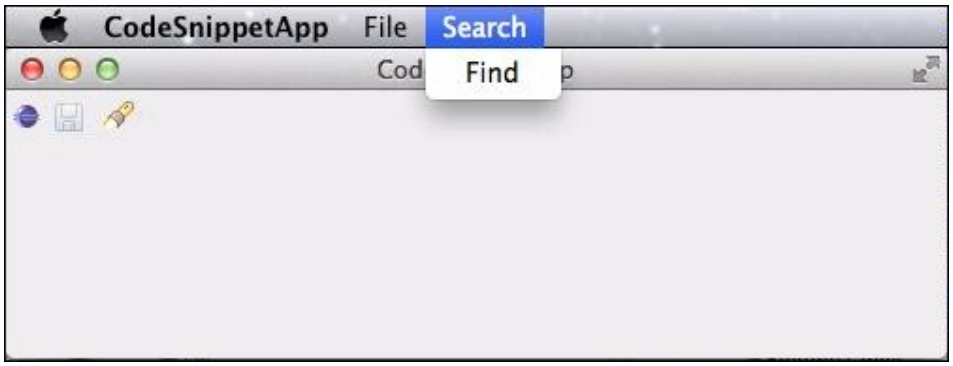
To fix this problem, you need to use the `-persistState false` and `-clearPersistedState` command line parameters. Go to **Run** | **Run Configurations** menu and click on `CodeSnippetApp.product` under the **Eclipse Application** group. Then, click on the **Arguments** tab and append `-persistState false -clearPersistedState` to `Program arguments`.

You can run the application either from the **Run** toolbar shortcut or from the `CodeSnippetApp.product` file.



> **Note**
>
> Another way to fix this problem is to clear the workspace every time you run the application. Go to **Run** | **Run Configurations**. Click on `CodeSnippetApp.product` under `Eclipse Application` group. In the **Main** tab, check the **Clear** checkbox.
>
> However, note that if you are saving any preferences or other settings in the workspace, then those will be lost when you clear the workspace.

27. Run the application again. You should see the **Find** option in the **Search** menu. Select the **Find** option and you'll see that the message **Find command executed** is displayed. The same message is displayed when you click on the **Find** toolbar button.



How it works...

The steps to add a menu and toolbar may seem like a lot, but they are really simple. First let's see what the **Application** model contains. Refer to the screenshot after step 4.

The **Application** model is where you declare menus, toolbars, views, and layout of the application. Though you may see many items under **Application**, in this task we are interested in **Commands**, **Handlers**, **Main Menu**, and **Toolbar** only. You will find **Main Menu** at **Application** | **Windows** | **Trimmed Window** | **Main Manu**, and **Toolbar** at **Application** | **Windows** | **Trimmed Window** | **TrimBars** | **Window Trim** | **Toolbar**. **Trimmed Window** in the application model refers to a window with trim bars such as toolbars and status bars, which is the main window of our application. Descriptions of the items are as follows:

- **Commands** are abstract actions, which can be attached to menu, toolbar, and keyboard shortcuts.

- **Handler** provides implementation to **Command**. You associate an implementation class in the handler for a command.
- **Main Menu** can have submenus or menu items. You can associate label and icons to menus and menu items. You can associate a command only with menu items. When you select a menu item, the handler for the command associated with the menu item is executed.
- **Toolbar** can have many tool items (they form the toolbar buttons). You can associate a command with a tool item. When you click on a toolbar item, the handler for the command associated with the tool item is executed.

One handler class can provide implementation to one or many menu and toolbar items.

Let's look at the code we have written in the handler class for the `Find` command. Refer to the code snippet given after the step 11. You would observe that the handler class is a POJO and does not implement/extend any interface/class to indicate to the Eclipse framework that this is a handler class for an action (or command) and which method to invoke to execute the action.

Unlike in the older versions, Eclipse 4 does not require you to define extensions or implement interfaces to define handler for menu or toolbar command. Eclipse 4 relies on Java annotations for this purpose. Note that the `execute` function in the `FindCommandHandler` class is annotated with `@Execute`. This annotation tells Eclipse 4 framework which function to execute in the handler class when the corresponding command is invoked either from a menu or a toolbar. The name of the handler method can be any valid name, as long as it is annotated with `@Execute`.

You might wonder how Eclipse passes required arguments to the handler method. In the older versions of Eclipse, an event object was passed in arguments to the handler method, which contained necessary information for executing the action. But because Eclipse 4 does not require you to implement any interface (which determines order and type of arguments), it also does not pass any argument to the `execute` method, unless the method asks for it. And the way to ask Eclipse 4 to pass required arguments is by using annotations and **Dependency Injection** (**DI**).

In simple terms, DI means that the caller of a method or class (when creating a new object from a class) does not pass (hardcoded) arguments to the method or constructor, but leaves it to the DI framework to find and set (inject) values at runtime. Using DI, you can avoid rigid design (for example, by not requiring implementation of interfaces) of your classes. This can help greatly when writing unit tests. Because in most cases objects created using DI would be POJO, they can be tested independently. DI is not a new concept and has been used in popular Java frameworks,

such as Spring (http://www.springsource.org/). Eclipse 4 uses DI extensively to set values of the method arguments and class members.

Coming back to the code in `FindCommandHandler`, the first argument of `MessageDialog.openInformation`, is a `Shell` object. This object is passed by the Eclipse 4 framework using DI, because the `execute` method is annotated with `@Execute`. In addition to marking the method as the handler method, the `@Execute` annotation implicitly tells the Eclipse 4 framework to inject any required arguments to the method.

There's more...

You can find more information about application model at http://wiki.eclipse.org/E4/UI/Modeled_UI. For more information on the Eclipse 4 annotations and DI, refer to http://wiki.eclipse.org/Eclipse4/RCP/Dependency_Injection. In the `execute` method of the `FindCommandHandler` class, we used DI to get a value of the `Shell` argument. `Shell` is a required argument in the `MessageDialog.openInformation` method. Another way to get the `Shell` instance is from current display object, for example: `MessageDialog.openInformation(Display.getCurrent().getActiveShell(), "Find Command", "Find command executed");`.

Using direct menu items

We created menu items and toolbar items using commands in this task. It made sense to use command because the same action was shared between menu and toolbar. However, you can create menu and toolbar items without the command also.

Right-click on the **Main Menu** option (refer to step 12) in `Application.e4xmi` and select **Add Child** | **DirectMenuItem**. You can then set the label, icon, and also handler class (class URI) for this menu item. Similarly, you can add **Direct ToolItem** to the toolbar.

We will create a **Direct Menu** item in our application in a recipe later when we implement a pop-up menu.

Compiler warnings for E4 APIs

In `FindCommandHandler.java`, you would see warnings for E4 APIs, for example, at import statements `org.eclipse.e4.core.di.annotations.Execute`. The warning message begins with **Discouraged access: …**.

The reason you see these warnings is that these APIs are still provisional and subject to changes in future. If you want to hide these warnings, you

can do so by going to **Preferences** | **Java** | **Compiler** | **Errors/Warnings**. Expand the **Deprecated and restricted API** group. Select the **Ignore** option for **Discouraged reference (access rules)**.

Adding views (Intermediate)

So far our application does not have any views to display data. In this task, we will add two views to the application and also see how to lay them out. We are going to add the following views:

- View on the left-hand side to display the list of snippets
- View on the right-hand side to display details of the selected snippet in the list

Note that in this application we will be creating UI using the classes from the SWT/JFace packages of Eclipse and not AWT/Swing classes. There could be classes in both UI frameworks with similar names, but make sure that you import packages SWT/JFace.

Getting ready

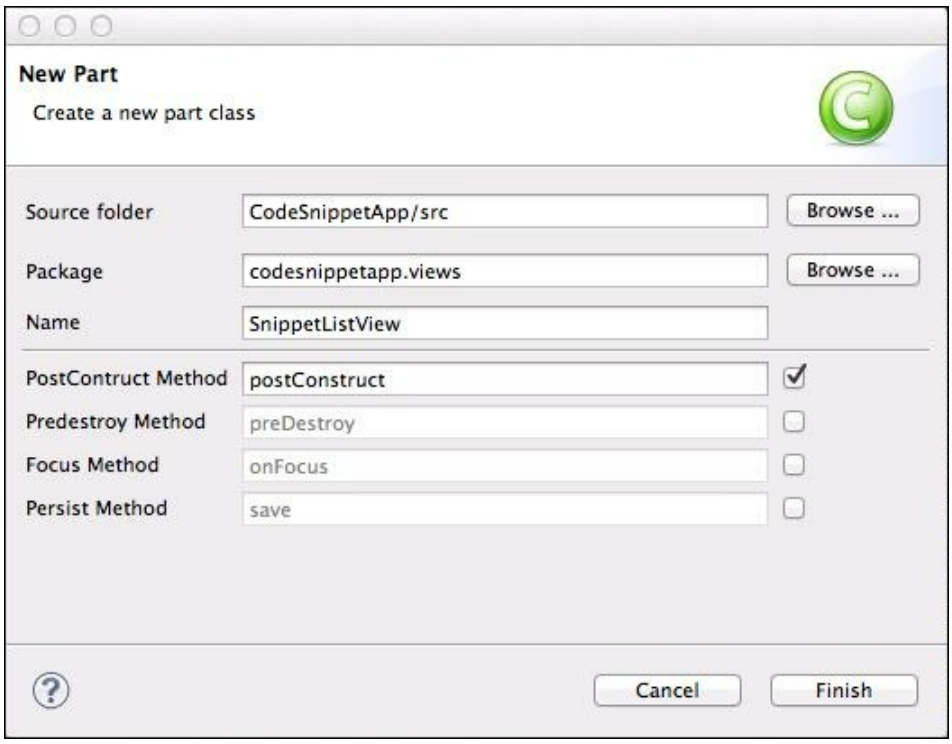
Create a new package `codesnippetapp.views` in the `src` folder in **Package Explorer**. We will create all the views of our application in this package.

How to do it...

1. Open `Application.e4xmi` and go to **Application** | **Windows** | **Trimmed Window** | **Controls** | **Perspective Stack** | **Perspective** | **Controls** | **PartSashContainer**.
2. Change orientation of **PartSashContainer** to **Horizontal**. This will split the main window vertically into two parts.
3. Right-click on the **PartStack** node (under **PartSashContainer**) and select **Remove**. We do not need **PartStack** in this task.
4. Right-click on **PartSashContainer** and select **Add Child** | **Part**. This will insert a new **Part** node under **PartSashContainer**. The part will contain the left-side view to display the list of code snippets.
5. Similarly, add one more part to the **PartSashContainer** node. This part will contain the right-side view to display details of the selected code snippet in the list.
6. Click on the first part and set its ID to `codesnippetapp.snippet.list` and label to **Code Snippets**. Set the ID and label of the second part to `codesnippetapp.snippet.details` and **Snippet Details** respectively:



7. Click on the **Code Snippets** part and the **Class URI** label. This will display a wizard to create a new part. Enter the information as shown in the following screenshot and click on the **Finish** button:



8. Similarly, create the `SnippetDetailsView` class for the **Snippet Details** part.
9. If you run the application at this point, you will not see the views we just created. That is because our views do not create any UI yet. So, first we will create a `TableViewer` class to display a list of snippets in the left-side view.
10. Open `SnippetDetailsView.java`. Modify the code in this file as follows (the code that you need to add is highlighted):

```
package codesnippetapp.views;
import javax.annotation.PostConstruct;
import javax.inject.Inject;

import
org.eclipse.jface.viewers.TableViewer;
import org.eclipse.swt.widgets.Composite;

public class SnippetListView {

  TableViewer snippetsList;
```

```
    @Inject
    public SnippetListView() {
    }

    @PostConstruct
    public void postConstruct(Composite
parent) {
      snippetsList= new TableViewer (parent);
    }
}
```

Add member variable `snippetsList` of type `TableViewer`.

Add a `parent` argument to the `postConstruct` method.

In the `postConstruct` method, create an instance of `TableViewer` and assign it to the `snippetsList` variable.

You can automatically insert required import statements in the file by using keyboard shortcut *Shift* + *command*/*Ctrl* + *O*.

> **Note**
>
> Note that the `postConstruct` method is annotated with `@PostConstruct`. This tells Eclipse 4 framework that this method is to be called after the part is injected in the DI framework. This method is typically used for creating UI for parts and for any other initialization. The method name need not be `postConstruct`. As long as it is annotated with `@postConstruct`, the name of this method can be any valid method name.
>
> The `@PostConstruct` annotation also tells the DI framework to inject arguments of the method. So, the framework would inject the `parent` argument that we added to the `postConstruct` method in this step.

11. We will now create the snippet details view. Open `SnippetDetailsView.java` and follow these steps:

    1. Add three private members to the `SnippetDetailsView` class:

```
private Text snippetText,
codeText, descText;
```

2. Add a private member for the save button:

```
private Button saveButton;
```

3. Add a parent argument to the `postConstruct` method:

```
public void
postConstruct(Composite parent)
```

4. Create a composite in the `postConstruct` method to hold all other UI controls:

```
Composite snippetDetailsComposite
= new Composite(parent, SWT.None);
```

5. Set `GridLayout` with two columns:

```
nippetDetailsComposite.setLayout(new
GridLayout(2, false));
```

6. Create a label and textbox for the snippet name field:

```
Label snippetLabel = new
Label(snippetDetailsComposite,
SWT.None);

snippetLabel.setText("Snippet
Name:");
snippetText = new
Text(snippetDetailsComposite,
SWT.BORDER);
```

7. Create a label and textbox for the snippet code:

```
Label codeLabel = new
Label(snippetDetailsComposite,
SWT.None);

codeLabel.setText("Code:");

codeText = new
Text(snippetDetailsComposite,SWT.MULTI
| SWT.BORDER | SWT.V_SCROLL |
SWT.H_SCROLL);


Create label and text box for
snippet description:
```

```
Label descLabel = new
Label(snippetDetailsComposite,
SWT.None);
descLabel.setText("Description:");
descText = new
Text(snippetDetailsComposite,
SWT.MULTI | SWT.BORDER |
SWT.V_SCROLL | SWT.H_SCROLL);
```

8. Create a save button:

```
saveButton = new
Button(snippetDetailsComposite,
SWT.BORDER);
saveButton.setText("Save");
```

12. Run the application.



13. As you can see, the two views that we created are displayed. However, the textboxes in the **Snippet Details** view are not sized properly. We will fix this later.

How it works...

The views are added as parts in **Application** | **Windows** | **Controls** of the application model (`Application.e4xmi`). You can either add parts directly under **Controls** or you can create **Perspective** and add **Parts** in the perspective.

Perspective can be thought of as a layout of views. The **Perspective** layout determines which views to display and where to place them in the parent container. You can define more than one perspective, but only one is active at a time.

The **Perspectives** layout are added under the perspective **Stack** node. **Parts** are added under the **Controls** node of the **Perspective** layout.

The two view containers are provided in Eclipse 4 to arrange parts (views). The first one is **PartSashContainer**. It displays all the parts added to it either vertically or horizontally, which is determined by its **Orientation** attribute. **PartSashContainer** creates parts in split windows and they can be resized by moving the splitter bar. The second type of container is **PartStack**. It displays parts in a tabbed window. Only one part is visible at a time and you switch between parts by clicking on the tabs.

In our application we needed two views to be displayed at the same time, next to one another. Therefore, we used **PartSashContainer**.

Part acts as a blank canvas on which you can add UI controls. To do this, you need to associate a class with the part. You do so by specifying **Class URI** of the part.

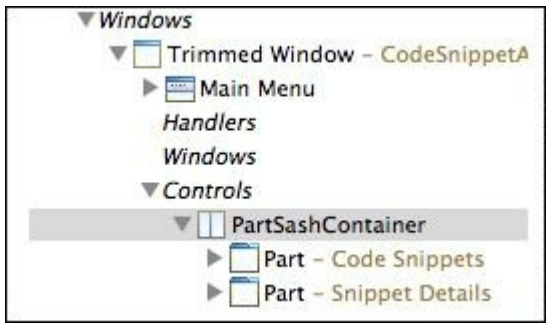
The two types of URI can be specified in the application model: **Class URI** and **Resource URI**. **Class URI** is used for specifying path of the Java classes and **Resource URI** is used for specifying the path of resources such as images or icons and so on.

The format of **Class URI** is `bundleclass://[Bundle-SymbolicName]/[package].[classname]`. For example, **Class URI** of the **Code Snippets** part is `bundleclass://CodeSnippetApp/codesnippetapp.views.SnippetListView`. The symbolic name of the bundle (plugin) is specified in `MANIFEST.MF`, in the `META-INF` folder of the project. The format of the resource URI is `platform:/plugin/[Bundle-SynbolicName]/[path]/[file]`, for example, `platform:/plugin/CodeSnippetApp/icons/find.gif`.

There's more...

We have used **PerspectiveStack** and **Perspective** in our application model in step 2. This allows us to create different layouts of views in the main window. For example, in the perspective that we have created, the main window is split vertically to display two views. We could create another perspective where we split the main window horizontally. We can then provide an option in the application to switch the perspectives.

However, if you do not plan to create different layouts in your application, you can put all your parts directly under **Application** | **Windows** | **Controls**:



Setting container data of parts

PartSashContainer allocates equal spaces to all the views contained in it. Therefore, you see that the main window is split vertically at the center, but we would want to size views differently; for example, we want the **Code Snippets** part to take 20 percent of width of the main window and the **Snippet Details** part to take 80 percent. You can do this by setting the **Container Data** attribute of **Parts** in the application model. Set **Container Data** of the **Code Snippet** part to **2** and that of the **Snippet Details** part to **8**. Run the application after these changes. You will see that the **Code Snippets** view is smaller than the **Snippet Details** view.

Setting layout data

The input textboxes in our applications are not properly sized. We would want textboxes for the code and description to be bigger and we want all the textboxes to be wider, preferably occupying the remaining width of the view.

This can be done by setting the layout data of each text control. Refer to `Understanding Layouts in SWT` (http://www.eclipse.org/articles/article.php?file=Article-Understanding-Layouts/index.html) for descriptions of layouts in SWT.

We are going to skip the description of the code for setting layout data of UI controls in our application. However, it is implemented in the code accompanying this recipe.

Refer to the `postConstruct` method of the `SnippetDetailsView` class to see how we have resized textboxes using `GridData`.

Injecting your own objects (Advanced)

So far we have seen how to use objects injected by the Eclipse 4 DI framework in **Parts** and **Handlers**. For example, the `Composite` (parent) object was injected by the framework in the `postConstruct` methods of `SnippetDetailView` and `SnippetListView`. The `Composite` object was not created by us but by the framework. However, there are situations when we would want to inject objects created by us in the framework, so that they can be shared with other objects in the application.

One such scenario in our application is sharing snippet data such as name, code, and description. We would like to save snippets created in this application on the disk, so that they can be loaded again. We could choose to save them in a flat file or in a database table. Whichever way we save it, once they are loaded in the application, this data is required by the **Code Snippets** view (to list them) and by the **Snippets Details** view (to show details of the selected snippet). So, we will inject snippets in the DI framework, which will then be available to all the views and handlers in our application.

But before that, we need to create classes to hold snippet data and a list of snippets. So, we will start with that.

Getting ready

Create a package `codesnippetapp.data` in the `src` folder. Create a class `SnippetData` in this package. Refer to the accompanying code this recipe for the implementation of this class. It is a simple class with three members (for snippet name, code, and description) and the `toString` method that returns the name of the snippet.

Create another class `SnippetRepository` in the `codesnippetapp.data` package. This class contains the array of snippets and physical path of the repository on the disk:

```
package codesnippetapp.data;

import java.util.ArrayList;

public class SnippetRepository {
  public ArrayList<SnippetData> snippets = new
ArrayList<>();
  public String repositoryPath;
}
```

How to do it...

1. Double-click and open `Activator.java` from **Package Explorer**. You can find this class in the `src`/`codeassistapp` folder. The `Activator` class is loaded initially when the plugin is loaded and it can be used to handle plugin lifecycle events such as start and stop. Add the following code in the `start` method of the `Activator` class:

```
IEclipseContext ctx =
E4Workbench.getServiceContext();
SnippetRepository repository = new
SnippetRepository();
ctx.set(SnippetRepository.class,
repository);
```

   Organize imports (*Shift* + *command*/*Ctrl* + *O*) if you need to.

   We first obtain service context from a static method in the `E4Workbench` class. Then we create an instance of the `SnippetRepository` class (which holds array of snippets) and set it in the context class with the key as the class itself.

2. `SnippetListView` needs list of snippets to display in the `TableViewer` class. Open the `SnippetListView` class. The `postConstruct` method of this class takes one argument of type `Composite`. Add another argument of type `IEclipseContext`:

```
@PostConstruct
  public void postConstruct(Composite
parent, IEclipseContext ctx)
```

   The Eclipse 4 DI framework would inject both these arguments.

3. Get `SnippetRepository` from the context. Remember that we injected this object in the step 1:

   SnippetRepository repository = ctx.get(SnippetRepository.class)

4. The next step is to populate the `TableViewer` class with snippets in the repository. For that we will have to set content provider of the `TableViewer` class:

```
snippetsList.setContentProvider(new
IStructuredContentProvider() {

  @Override
      public void inputChanged(Viewer
viewer, Object oldInput,
        Object newInput) {
```

```
            //do nothing
        }

          @Override
          public void dispose() {
            //do nothing
          }

          @Override
          public Object[] getElements(Object
inputElement) {
            if (inputElement instanceof
SnippetRepository)
            {
return ((SnippetRepository)inputElement).
snippets.toArray();
        }

        return new Object[]{};
      }
});
snippetsList.setInput(repository);
```

The code first sets the content provider of `snippetsList` by creating a new (inline) instance of `IStructuredContentProvider`. The `getElements` method of this class is called when initial input is set. In the `getElements` method, we return an array of snippets in the `SnippetRepository` object.

Then, we set the initial input of the `snippetsList` type with the repository (`SnippetsRepository`) object that we obtained from the Eclipse context.

When you run the application, you may not see any change, because we have not added any snippets to the repository.

How it works...

`IEclipseContext` is an important interface participating in Eclipse 4DI. The context is a map containing the key-value pairs. Eclipse 4 runtime maintains stack of `IEclipseContext`, the top most being the current context at that point.

When Eclipse 4 runtime needs to inject an object, it first searches in the current active `IEclipseContext`. If the object is not found, it searches in its parent context, till it reaches the root context. The key in the set method of `IEclipseContext` can be either a string or a class. We have set `SnippetRepository` in the root context using its class as a key. The root

context can be accessed using a static method, `getServiceContext`, of the `E4Workbench` class.

In the `postConstruct` method of `SnippetListView`, we got reference to the `SnippetRepository` instance by calling the `get` method on the `IEclipseContext` object, which itself is injected by the framework. We then used this instance of the `SnippetRepository` instance to populate `TableViewer` in the class `SnippetListView`.

There's more...

You might wonder which framework objects are available using DI.

We have already seen `Composite`, `Shell`, and `IEclipseContext` being injected in the code. All the elements declared in the application model are accessible through DI, for example, **Application**, **Parts**, **Perspectives**, **Menus**, **Toolbars**, and so on. Some of these elements are accessible directly, for example, `MApplication`, and others are accessible through different services. In addition to this, **Preferences** and some of the OSGi services are also available using DI.

The parts declared in the application model are not directly accessible (except active **Part**) using `IEclipseContext`. You first need to get a reference to `EPartService` and then use that to find the part with the given ID; for example, to obtain reference of the part that contains `SnippetDetailsView`, which has ID `codesnippetapp.snippet.details`, you could write the following code:

```
@Inject
public void someMethod(IEclipseContext ctx)
{
EPartService partServie =
ctx.get(EPartService.class);
MPart detailsViewPart =
partService.findPart("codesnippetapp.snippet.details");
}
```

You can also ask the framework to inject a named object; for example, to inject an active part, you would declare a method as follows:

```
@Inject
public void someMethod
(@Named(IServiceConstants.ACTIVE_PART) MPart
activePart)
```

You can find keys for named parameters in the `org.eclipse.e4.ui.services.IServiceConstants` interface. Some of the

other constants defined in this interface that can be used in the named parameter are `ACTIVE_SELECTION`, `ACTIVE_CONTEXTS`, and `ACTIVE_SHELL`.

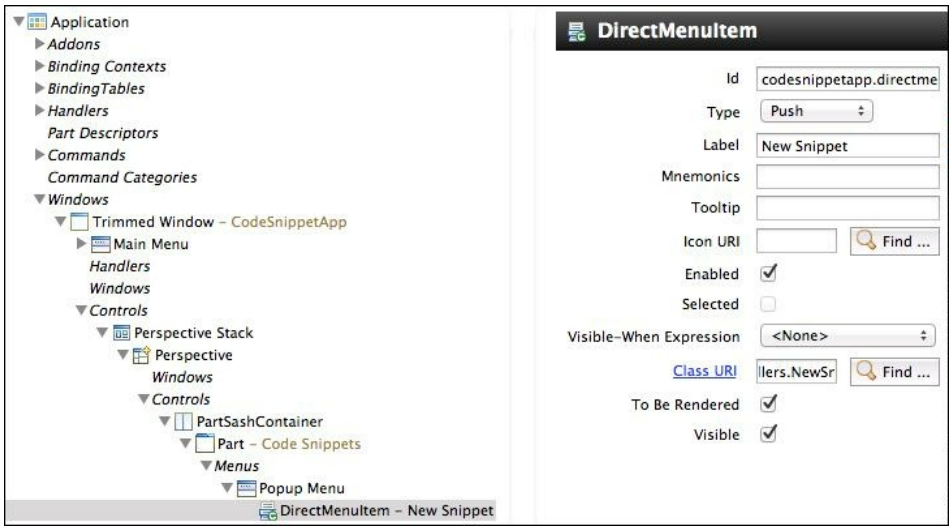
Refer to http://wiki.eclipse.org/Eclipse4/RCP/EAS/List_of_All_Provided_Services for the list of objects and services accessible from different contexts.

Creating a pop-up menu (Intermediate)

In this task, we will create a pop-up menu that will be displayed when you right-click in the snippets list. If no snippet is selected at a location where you right-click, then we will display a pop-up menu with a single option to add a snippet. If there is a snippet at the location, then we would display a menu that has options to delete the snippet and add a snippet.

How to do it...

1. Open the application model (`Application.e4xmi`) and go to **Application** | **Windows** | **Trimmed Window** | **Controls** | **Perspective Stack** | **Perspective** | **Controls** | **PartSashContainer** | **Part** (**Code Snippets**).
2. Expand the **Code Snippet** part and right-click on the **Menus** node.
3. Select **Add child** | **Popup Menu**. Set the ID of the pop-up menu to `codesnippetapp.snippetlist.popupmenu`.
4. Right-click on the newly added pop-up menu and select **Add Child** | **DirectMenuItem**.
5. Set **Label** of the menu item as **New Snippet**.
6. Click on **Class URI** link. This opens the **New Handler** wizard.
7. Click on the **Browse** button next to the **Package** textbox and select `codesnippetapp.handlers` from the list of packages displayed.
8. Set **Name** as **NewSnippetMenuHandler** and click on the **Finish** button. The new class file is opened in the editor. Go back to the application model. Refer to the following screenshot:



9. Right-click on the **Popup Menu** node and add another pop-up menu item with the `Delete` label and the `DeleteSnippetMenuHandler` class.
10. Now we need to register this pop-up menu with the `TableViewer` class in the **Code Snippets** part. Open the class `SnippetListView` (you can find this class in the `codesnippetapp.views` package). We will have to register the pop-up menu using **Menu Service**.
11. Add the `EMenuService` argument to the `postConstruct` method:

```
@PostConstruct
public void postConstruct(Composite
parent, IEclipseContext ctx, EMenuService
menuService)
```

12. Append the following code to the `postConstruct` method:

```
menuService.registerContextMenu(snippetsList.getTable(),
"codesnippetapp.snippetlist.popupmenu");
```

13. Run the application. Right-click in the `TableViewer` class on the left-hand side. You should see a pop-up menu with two options: **New Snippet** and **Delete**.

How it works...

To add a pop-up menu, you first need to create a menu in the application model for a part in which you want to display the menu. In this recipe, we added a menu to the **Code Snippets** part.

Then, you add menu items. In this recipe, we added two `DirectMenuItem`. For the main menu bar in the task of adding menu and toolbar buttons, we added `HandledMenuItem`, because we wanted to share the handler for the menu between toolbar button and menu item. However, in this case, we need only one implementation of options in the pop-up menu, so we created `DirectMenuItem`. But, if you want to add keyboard shortcuts for the menu options, then you may want to create `HandledMenuItem` instead of `DirectMenuItem`.

For each menu item, you set a class URI that is a handler class for the menu item.

The next step is to register this pop-up menu with a UI control. In our application, we want to associate this menu with the `TableViewer` class that displays a list of snippets. To register a menu with any UI control, you need to get an instance of `EMenuService`. We obtained this instance in the `postConstruct` method of `SnippetListView` using DI—we added the `EMenuService` argument to the `postConstruct` method.

Then, we used `registerContextMenu` of `EMenuService` to associate the pop-up menu with the `TableViewer` class. `registerContextMenu` takes instances of the UI control and menu ID as arguments.

There's more...

The **Delete** option in our pop-up menu makes sense only when you click on any snippet. So, when you right-click on an area of `TreeViewer` that does not have any snippet at that location, the **Delete** option should not be displayed, only the **New Snippet** option.

This can be done using core expressions. You can find more information about core expressions at http://wiki.eclipse.org/Platform_Expression_Framework and http://wiki.eclipse.org/Command_Core_Expressions.

We will use a core expression to decide if the **Delete** menu option should be displayed. We will add a mouse listener to the `TableViewer` class. If the mouse was clicked on a Snippet, then we will add `SnippetData` to `IEclipseContext` with the `snippet_at_mouse_click` key. If there is no snippet at the location, then we will remove this key from `IEclipseContext`.

Then, we will add a core expression to check if the `snippet_at_mouse_click` variable is of type `codesnippetapp.data.SnippetData`. We will then associate this core expression with the **Delete** menu item in the application model.

Adding mouse listener to the TableViewer class

1. Create a static field in the `SnippetListView` class.

```
private static String
SNIPPET_AT_MOUSE_CLICK =
"snippet_at_mouse_click";
```

2. Make the `ctx` argument of the `postConstruct` method, `final`.
3. Append the following code in the `postConstruct` method:

```
//Add mouse listener to check if there is
a snippet at mouse click

 snippetsList.getTable().addMouseListener(new
MouseAdapter() {
      @Override
      public void mouseDown(MouseEvent e)
      {
        if (e.button == 1) //Ignore if
left mouse button
          return;
```

```
        //Get snippet at the location of
mouse click
        TableItem itemAtClick =
snippetsList.getTable().getItem(new
   Point(e.x, e.y));
        if (itemAtClick != null)
        {
          //Add selected snippet to the
context
          ctx.set(SNIPPET_AT_MOUSE_CLICK,
itemAtClick.getData());
        }
        else
        {
          //No snippet at the mouse click.
Remove the variable

 ctx.remove(SNIPPET_AT_MOUSE_CLICK);
        }
      }
    });
```

Creating core expression

Carry out to the following steps:

1. Open `plugin.xml` and go to the **Dependencies** tab.
2. Add `org.eclipse.core.expression` as a required plugin.
3. Go to the **Extensions** tab. Add the `org.eclipse.core.expressions.definitions` extension. This will add a new definition. Change the ID of the definition to `CodeSnippetApp.delete.snippet.expression`.
4. Right-click on the definition and select **New** |**With**. Change the name of the variable to `snippet_at_mouse_click`. This is the same variable name we set in the `SnippetListView` class.
5. Right-click on the **With** node, and select **New** | **instanceof** option. Set the value to `codesnippetapp.data.SnippetData`.

   This core expression will be `true` when the type of (`instanceof`) the `snippet_at_mouse_click` variable is `codesnippetapp.data.SnippetData`.

6. Click on `plugin.xml` and verify that the core expression definition is as follows:

```
   <extension

 point="org.eclipse.core.expressions.definitions">
      <definition

 id="CodeSnippetApp.delete.snippet.expression">
```

```
            <with

variable="snippet_at_mouse_click">
                <instanceof

value="codesnippetapp.data.SnippetData">
                </instanceof>
            </with>
        </definition>
    </extension>
```

Setting the core expression for Menu Item

1. Open the application model (`Application.e4xmi`) and go to **DirectMenuItem** for the **Delete** pop-up menu.
2. Right-click on the menu item and select **Add child** | **VisibleWhen Core Expression**. This will add a **Core Expression** child node.
3. Click on the **Core Expression** node and then on the **Find** button next to the **Expression Id** textbox and select `CodeSnippetApp.delete.snippet.expression` from the list. This is the ID of the core expression definition we added in `plugin.xml`.

Run the application. When you right-click on the **Snippets List** view, which does not have any snippet at this point, you should see only the **New Snippet** menu option.

Creating custom events and handlers (Intermediate)

In this task, we will see how to generate custom events and event handlers for communicating messages between different objects in the application. We will implement menu handler for adding a new snippet in the **Snippets List** view, which will add a new snippet to the repository and then publish an `onNewSnippet` event. We will implement an event handler in the **Snippet Details** view to display the snippet name.

We will perform most of the communication between different views of our application using this publish-subscribe model. When snippet selection changes in the **Snippets List** view, we will publish an event and make the **Snippet Details** view respond to this event by updating its content. We will use the `IEventBroker` service of Eclipse 4 to publish and subscribe to events.

Getting ready

Create a class `CodeSnippetAppConstants` in the `codesnippetapp` package. We will use this class to hold all constant strings, for example, custom event names in our application.

Add a static `String` field in this class:

```
package codesnippetapp;

public class CodeSnippetAppConstants {
  public static final String NEW_SNIPPET_EVENT =
"onAddNewSnippet";
}
```

How to do it...

1. Open the class `NewSnippetMenuHandler` (from package `codesnippetapp.handlers`) in the editor.
2. Add an argument of type `IEventBroker` to the `execute` method in this class.
3. Add the following code in the `execute` method:

```
eventBroker.send(CodeSnippetAppConstants.NEW_SNIPPET_EVENT,
null);
```

4. We will use annotation `@UIEventTopic` to handle this event. This annotation is defined in the `org.eclipse.e4.ui.di` plugin. Open

`plugin.xml`, go to the **Dependencies** tab and add dependency for the `org.eclipse.e4.ui.di` plugin.

5. Open the `SnippetListView` class (from the `codesnippetapp.views` package) and add the following static member to keep count of new snippets created. We will use this counter to create a name for a new snippet:

```
private static int newSnippetCounter = 1;
```

6. Add an event handler method for the `onAddNewSnippet` event in the `SnippetListView` class:

```
@Inject @Optional
public void onAddNewSnippet
(@UIEventTopic(CodeSnippetAppConstants.NEW_SNIPPET_EVENT)
  Object data, SnippetRepository
repository)
{
SnippetData newSnippet = new
SnippetData("Untitled" +
  (newSnippetCounter++));
repository.snippets.add(newSnippet);
snippetsList.refresh();
snippetsList.setSelection(new
StructuredSelection(newSnippet));

}
```

7. Run the application. Right-click on the **Snippets List** view and select the **New Snippet** menu option. Observe that a new snippet with name `Untitled1` is added in the list and selection is set to the newly added snippet. As you add more snippets, the new snippet counter increments, for example, `Untitled2`, `Untitled3`, and so on.

How it works...

`IEventBroker` is one of the services provided by Eclipse 4. Using this service, you can publish a custom event and subscribe to events too. There are two methods available in the `IEventBroker` service to publish an event:

- `send`: This method publishes an event synchronously
- `post`: This method publishes an event asynchronously

In our application, we have used the `send` method to publish the event (see step 3).

There are two ways to subscribe for events. The first one is getting instance of the `IEventBroker` service and calling the `subscribe` method. At the

minimum, the `subscribe` method takes topic name and event handler class as arguments.

The second way to subscribe for events is by using annotations. You can use the `@UIEventTopic` annotation (for a `method` argument) to subscribe to a topic. We have used this method to subscribe to the `onAddNewSnippet` event in this task (see step 6).

When the **New Snippet** menu option is selected in our application, the `execute` method of the `NewSnippetMenuHandler` class is called. In this method, all we are doing is publishing an event (null event data because we do not need to pass any data for this event) to the `onAddNewSnippet` topic (see step 3).

The `onAddNewSnippet` method of `SnippetListView` is called when an event is published to the `onAddNewSnippet` topic. Note that the method itself is annotated with `@Inject` and `@Optional`. The optional annotation is necessary, otherwise the Eclipse 4 runtime will throw `InjectionException` at the time of rendering the view. If you find that your event handler method is not called at all when the event is published, one of the reasons could be that none of the optional arguments could be resolved or found in the context. For example, if `SnippetRepository` were not available (injected) in the context, then the `onAddNewSnippet` method would not be called at all.

In the `onAddNewSnippet` method, we first create an untitled `SnippetData` object and add it to the repository. Then we refresh the `TableViewer` class to display the newly added snippet in the view. We then set selection of the `TableViewer` class to the new snippet (refer to step 6).

There's more...

If you want to test the core expression, we wrote in the previous task for the **Delete** menu option, then right-click on a snippet and verify that the **Delete** menu option appears in the pop-up menu.

Handling selection change events in the TableViewer class

We want the **Snippet Details** view to be updated when the selection changes in the **Snippets List** view. We can do this using `IEventBroker`. First, we will add a selection change listener for the `TableViewer` class in the `postConstruct` method of `SnippetListView`. In the event handler, we will publish a custom event `onSnippetSelectionChange` and pass selected snippet as an event data. Then, in the `SnippetDetailsView` class, we will add an event handler for the `onSnippetSelectionChange` event and populate the fields with the selected snippet data.

Refer to the `postConstruct` method of the `SnippetListView` and `snippetSelectionChanged` methods of `SnippetDetails` in the accompanying code for this recipe for the details of this implementation.

Run the application after these changes and add a few snippets. Observe that the **Snippet Details** view is updated whenever selection changes in the **Snippets List** view.

Saving snippet data

We still haven't implemented the `Save` functionality for snippets in our application. When we modify data in the **Snippet Details** view and click on the **Save** button, we need to save modifications and update the **Snippet List** view with the new snippet name if it is changed.

To implement the `Save` functionality, we will again use `IEventBroker` to publish and handle the `onSnippetSaved` event. We will implement a selection listener for the **Save** button in the `postConstruct` method of `SnippetDetailsView`. In this handler, we will update currently displayed snippet and publish the `onSnippetSaved` event using `IEventBroker`. We will then handle this event in `SnippetListView` and refresh the `TableViewer` class.

Refer to the `postConstruct` and `saveSnippetData` methods in the `SnippetDetailsView` and `onSnippetSaved` methods of `SnippetListView` in the accompanying code for this recipe for the details of this implementation.

Run the application after you do the preceding changes, add a few snippets, modify snippet name in the **Snippet Details** view, click on the **Save** button, and observe that the snippet name is updated in the **Snippets List** view.

Adding a keyboard shortcut (Simple)

In this task, we will add a keyboard shortcut for the `Find` operation. We are going to assign the shortcut keys `Ctrl`/`command` + `F` for the `Find` action.

How to do it...

1. Open the application model (`Application.e4xmi`) and go to **Application** | **Binding Tables** | **Binding Table**.
2. Right-click on the **Binding Table** node and select **Add child** | **Keybinding**.
3. Type `M1+F` in the **Sequence** field.
4. Click on the **Browse** button next to the **Command** textbox and select **findCommand**.
5. Save the file and run the application. Press *Ctrl*/*command* + *F* and you'll see a message displayed from the `FindCommandHandler` class.

How it works...

The keyboard shortcuts for a command can be defined in the application model. Some of the keys are OS-specific; for example, in Windows you typically use *Ctrl* + *F* for the `Find` operation, whereas in Mac OSX the shortcut for the same is *command* + *F*.

To handle such differences across different operating systems, Eclipse has defined meta keys. The actual mapping of these meta keys with physical keys depends on the OS in which Eclipse is running. The following are the most commonly used meta keys:

- **M1**: This is the *Ctrl* or *command* key
- **M2**: This is the *Shift* key
- **M3**: This the *Alt* key
- **M4**: This is controlled in Macintosh and undefined in other OS

Creating custom objects using DI (Intermediate)

In an earlier recipe, we saw how to inject custom objects in the DI framework of Eclipse 4. In this recipe, we will see how to create/instantiate custom objects using DI. The advantage of creating objects using DI is that all the objects in the Eclipse context and services become easily accessible (because they are injected transparently by the framework) to the object that is created.

We have not yet implemented the `Find` functionality in our application. In this task, we will create a custom dialog box and instantiate it using DI. The search dialog box will have input fields for search criteria for the code and description.

How to do it...

1. Create a new class, `SearchDialog`, in the `codesnippetapp.views` package, extending `org.eclipse.jface.dialogs.Dialog`.
2. Add the following member variables in the `SearchDialog` class:

```
  private Text codeSearchTxt,
descSearchText;
  private SnippetRepository repository;
  private java.util.List<SnippetData>
searchResult = null;
  private String
codeCriteria ,descCriteria;
```

3. Add a constructor:

```
  @Inject
  public SearchDialog(Shell parentShell,
SnippetRepository repository) {
    super(parentShell);
    this.repository = repository;
}
```

4. We will create UI controls of this dialog box later. For now, the earlier code will display an empty dialog box.
5. We will create and open `SearchDialog` from the `executeFind` method of `FindCommandHandler` (recall that we created the `Find` menu and the `FindCommandHandler` class in the task of adding menu and toolbar button).
6. Open the `FindCommandHandler` class and remove the code to display a message in the `executeFind` method.
7. Add an argument of type `IEclipseContext` to the `executeFind` method:

```
@Execute
public void executeFind
(@Named(IServiceConstants.ACTIVE_SHELL)
  Shell shell, IEclipseContext ctx)
```

8. Remove the code to display message and add the following code in the `executeFind` method:

```
  SearchDialog dlg =
ContextInjectionFactory.make(SearchDialog.class,
ctx);

  if (dlg.open() != Dialog.OK)
    return;
```

9. Run the application and press *Ctrl*/*command* + *F*. Observe that an empty dialog box is displayed.

How it works...

In step 3 we created a constructor for `SearchDialog` that takes two arguments, `Shell` and `IEclipseContext`. We also annotated the constructor with `@Inject`, which tells Eclipse 4 to inject these arguments.

If we instantiate `SearchDialog` using the `new` operator, then dependency injection framework is bypassed, and the automatic injection of arguments would not happen. In such cases, we will have to explicitly pass arguments.

To create objects using the DI framework, we need to call the `make` method of `ContextInjectionFactory`. The `make` method takes two arguments: a class to instantiate and context object to be used for obtaining objects for dynamic injection. In step 6 we used the `ContextInjectionFactory.make` method to create an instance of `SearchDialog`. The Eclipse 4 DI framework injects the arguments required by the constructor of `SearchDialog`.

There's more…

We need to add input fields in the `SearchDialog` class where the user can enter search criteria for the snippet code and description. We also need to set dialog title and its initial size.

In order to create the `SearchDialog` input fields for creating the UI of a dialog, you need to override the `createDialogArea` method. In this method, we will create two label fields and two text boxes (to enter code criteria and description criteria). To set the title of the dialog box, we will call the `setText` method on the `Dialog Shell` object.

```
getShell().setText("Find Snippets");
```

Refer to the `createDialogArea` method of the `SearchDialog` class in the accompanying code for this recipe for the details of this implementation.

Set initial size of SearchDialog

We will set initial size of the `SearchDialog` box to 400 pixels wide and 150 pixels tall. Override the `getInitialSize` method in the `SearchDialog` class.

Perform a search operation on OK

We want to perform a search operation when the **OK** button of the dialog box is pressed. We will override the `okPressed` method in `SearchDialog`. We will iterate over snippets in the repository and perform a simple text match. If a match is found, we will save it in the class member `searchResult`. We will also override the `cancelPressed` method and set `searchResult` to `null`.

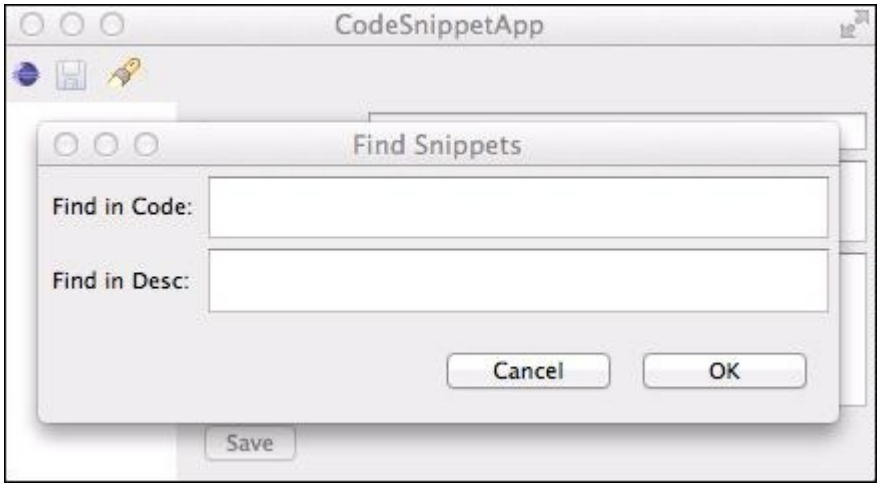
Refer to the `okPressed` and `cancelPressed` methods of the `SearchDialog` class in the accompanying code for this recipe for the details of this implementation.

Implement getter methods for SearchDialog members

We will add getter methods for member variables for search result and search strings, so that they can be accessed from outside the `SearchDialog` class.

Refer to `getSearchResult`, `getCodeSearchCriteria`, and `getDescriptionSearchCriteria` methods of `SearchDialog`.

Run the application and press *Ctrl*/*command* + *F*. You should see search dialog with two input fields. Press **OK**. If you had not entered any search string, then appropriate message should be displayed. If you had entered at least one search string, then the dialog box will close after performing search operation. You will not see search results yet because we have not implemented the code to display search results.


CodeSnippetApp
Find Snippets
Find in Code:
Find in Desc:
Cancel
OK
Save

Creating views dynamically (Advanced)

We have implemented two views in our application so far: **Snippets List** and **Snippet Details**. Both these views were statically declared in the application model. However, it is possible to create views dynamically and add them to the application model. We will see how to do that in this recipe.

We implemented `SearchDialog` in the previous task. In this task, we will create the `SearchResults` view dynamically and display search results. We will add this view to the bottom-right side (below the **Snippet Details** view) of the main application window.

Getting ready

Before we create a dynamic view, we need to create a placeholder for it in the application model. Currently **Perspective** in the application model has one **PartSashContainer**, which contains two parts: **Code Snippets** and **Snippet Details**.

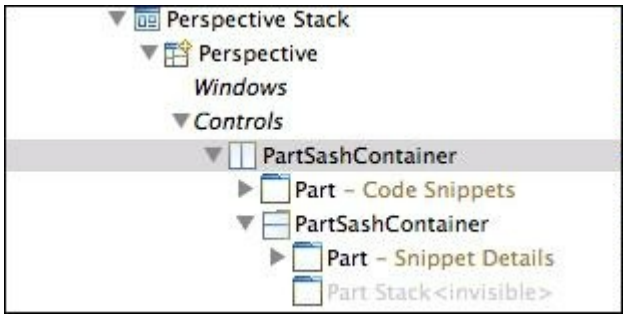
Now we need to split the right-hand side (space currently taken by the **Snippet Details** part) of the application into two parts: the top will have the **Snippet Details** part and the bottom will have the **Search Results** view that we are going to create in this task. We know that whenever we need to split the window, we need to use **PartSashContainer**. So, we will replace the **Snippet Details** part in the application model with **PartSashCointainer** and make the **Snippet Details** part the first child of the new **PartSashContainer** instance.

We want to display one search result view per search operation, which means that there could be multiple search result views open at a time, and we want to arrange them in a tabbed view. We know that we can use **PartStack** for tabbed views. So, we will add a **PartStack** node to the newly added **PartSashContainer** instance. When the search operation in the `SearchDialog` class is successfully executed, we will add a `SearchResults` view dynamically to the **PartStack** instance.

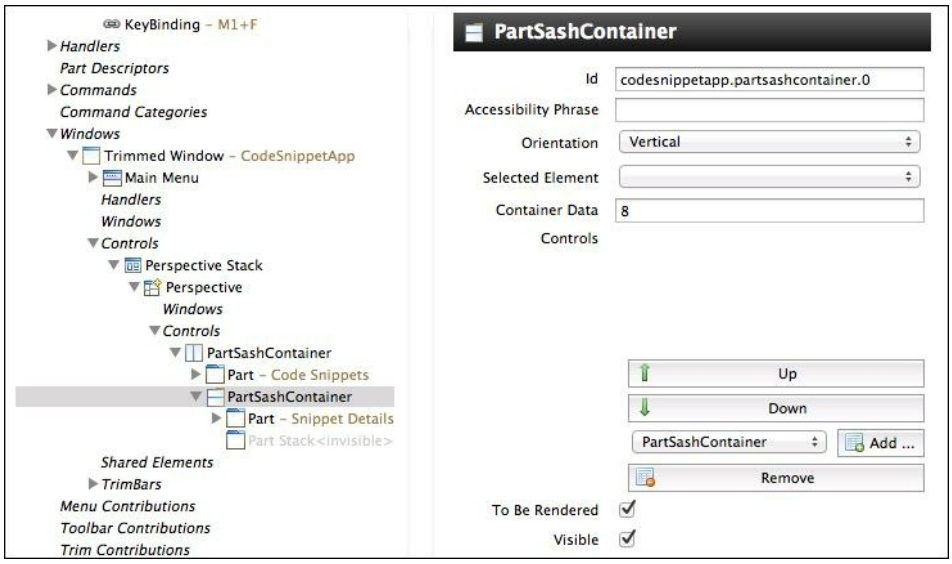
So, we will have to rearrange parts in our application model:

1. Open the application model (`Application.e4xmi`) and go to **Application** | **Windows** | **Trimmed Window** | **Controls** | **Perspective Stack** | **Perspective** | **Controls** | **PartSashContainer**.
2. Right-click on the **PartSashContainer** node and select **Add** | **PartSashContiner**. This will add a new **PartSashContainer** node.

3. Click on the **Snippet Details** part and holding the mouse button down, drag-and-drop it onto the newly created **PartSashContainer** node. This will make the **Snippet Details** part a child of a new **PartSashContainer** node.
4. Right-click on the new **PartSashContainer** node and select **Add child** | **PartStack**. This will add a new **PartStack** node. We want to make this part stack visible only when we need to display the **Search Result** view. So, initially we want to hide this **PartStack** node. Uncheck the **Visible** checkbox of the new **PartStack** node. Now your perspective should look as follows:



5. If you run the application at this point, you will see that the **Snippet Details** view takes up all the space in the main application window, and the **Code Snippets** view is not displayed at all. This is because we had set relative sizes of the **Code Snippets** and **Snippet Details** parts in the **Container Data** field of both the parts: the **Code Snippets** part's **Container Data** was set as **2** and that of the **Snippet Details** was set to **8**. However, we replaced the **Snippet Details** part in the parent **PartSashContainer** with another **ParentSashContainer**. So, now we need to set **Container Data** of the new **PartSashContainer** node to **8**.



6. Within the new **PartSashContainer** node (that contains the **Snippet Details** part and a **PartStack** node), we want the **Snippet Details** part to take 80 percent of the space and **PartStack** (which is a container for the **Search Results** views) to take 20 percent. The **Container Data** view of the **Snippet Details** part is already set to **8**. Set the **Container Data** view of the **PartStack** node to **2**.

If you run the application now, you will not see any difference, because **PartStack** is invisible.

How to do it...

1. We will first create a new class to display search results. Select **File** | **New** | **Other**. From the wizard, select **Eclipse 4** | **Classes** | **New Part Class**. This will open **New Part Wizard**.
2. Click on the **Browse** button next to the **Package** textbox and select the `codesnippetsapp.views` package.
3. Set name as `SearchResultsView`.
4. Check the **PostConstruct Method** checkbox and uncheck all others.
5. Click on **Finish**. This will create the `SearchResultsView` class and open it in the editor.
6. We will create search results view after `SearchDialog` is closed by clicking on the **OK** button. Open the `FindCommandHandler` class. To create a view dynamically, we will need instances of `EPartService`, `EModelService`, and `MApplication`, and all of them can be injected by Eclipse 4. Add these arguments to the `executeFind` method:

```
  @Execute
public void executeFind
(@Named(IServiceConstants.ACTIVE_SHELL)
  Shell shell, IEclipseContext ctx,
EPartService partService,
    EModelService modelService,
MApplication application)
```

7. Before we create search results view, check if we have any result data available to display. Add the following code after opening the dialog box in the `executeFind` function:

```
//Get search results from SearchDialog
List<SnippetData> searchResult =
dlg.getSearchResult();

//If there are no results to show, display
a message and return
if (searchResult == null ||
searchResult.size() == 0)
```

```
{
MessageDialog.openInformation(shell, "No
match found", "No match found for the
search criteria");
return;
}
```

8. Create a new Eclipse context. This context will be used by the search results view to resolve dependencies:

```
IEclipseContext newPartContext =
ctx.createChild("search_result_context");
```

9. Create a new part:

```
MPart findResultPart =
MBasicFactory.INSTANCE.createPart();
```

Note that `MBasicFactory` is in the `org.eclipse.e4.ui.model.application.ui.basic` package.

10. Set the implementation class (`SearchResultsView`) for the part:

```
findResultPart.setContributionURI("bundleclass://CodeSnippetApp/c
```

11. Set the Eclipse context of the part. Use the Eclipse context we created in step 8:

```
findResultPart.setContext(newPartContext);
```

12. We want to be able to close results view, so mark it as closable. Also, set a label text for the part:

```
findResultPart.setCloseable(true);
findResultPart.setLabel("Search Results");
```

13. The next step is to add the part to the application model. We want this part to be a child of `PartStack` that we had created in the previous section. We can get reference to that `PartStack` using its ID. We will use `EModelService` to find the `PartStack` instance:

```
  List<MPartStack> partStackList =
    modelService.findElements(application,
      "codesnippetapp.partstack.0",
MPartStack.class, null);
```

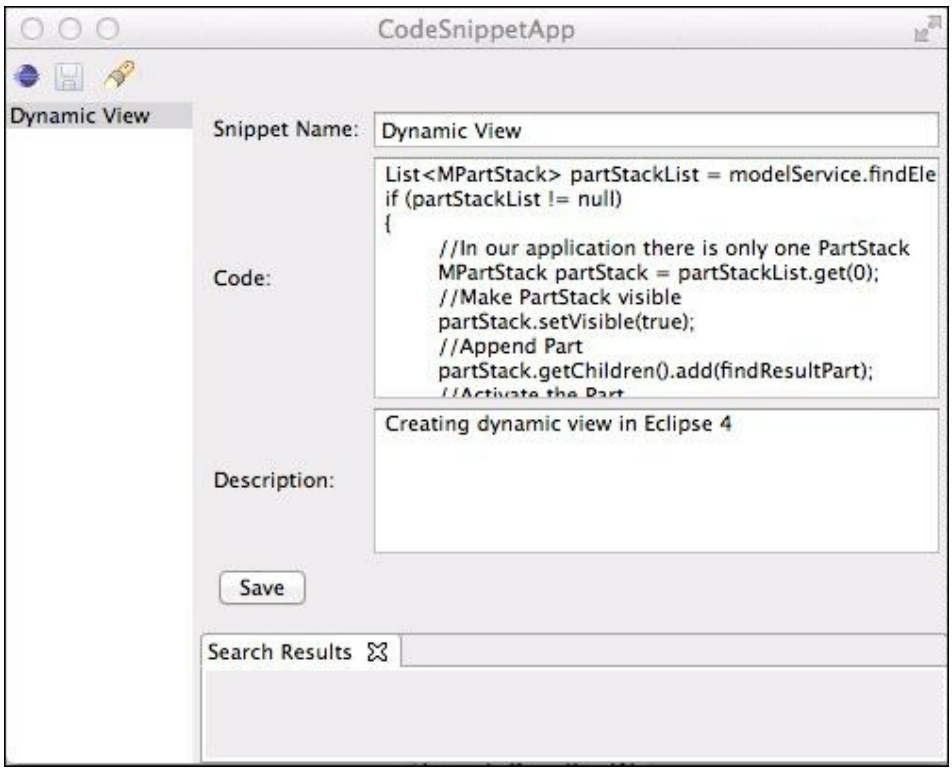
Note that application model editor created ID `codesnippetapp.partstack.0` for the `PartStack` instance when we added that `PartStack`. Since we have only one `PartStack` instance the `partStackList` instance in the preceding code will contain only one `MPartStack` instance.

14. Add the new part (for search result view) as a child of `PartStack`:

```
    if (partStackList != null)
    {
      //In our application there is only
one PartStack
      MPartStack partStack =
partStackList.get(0);
      //Make PartStack visible
      partStack.setVisible(true);
      //Append Part

 partStack.getChildren().add(findResultPart);
      //Activate the Part
      partService.activate(findResultPart,
true);
    }
```

15. Run the application, add a new snippet, enter the snippet code, and save the snippet (by clicking on the **Save** button). Press *Ctrl*/*command* + *F* and enter search string in the **Code** input box. Press **OK**. If the search string you entered exists in the snippet code, then you should see an empty **Search Results** view displayed at the bottom:



How it works...

A new part can be created dynamically by invoking the `createPart` method of the `MBasicFactory` class. You need to then set implementation of a class

of the part by calling the `setContributionURI` method. The class URI is specified as `bundleclass://<bundle_name>/<fully_qualified_class_name>`. You can find the bundle name in `MANIFEST.MF` in the `META-INF` folder of the project.

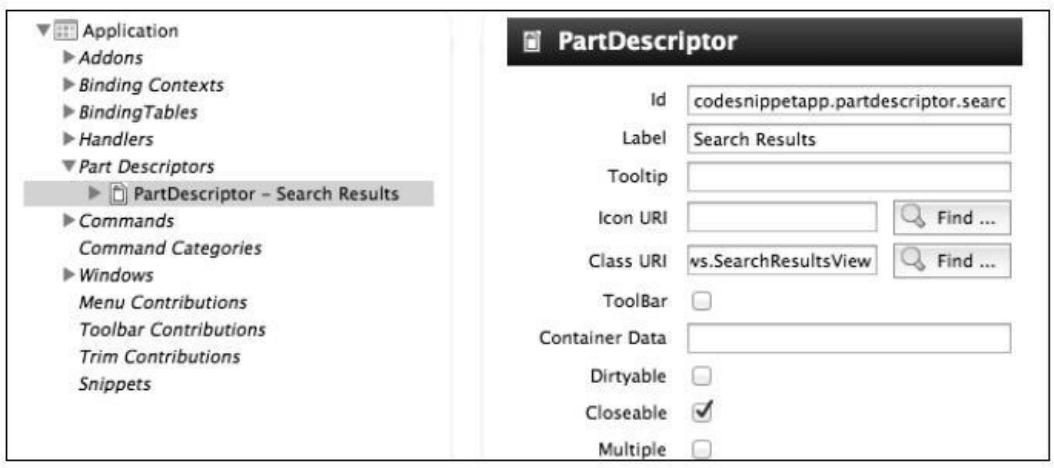
Setting the Eclipse context of the new part is optional. If you do not set the context, the Eclipse 4 runtime will use the active context when the part is rendered. In this task we created a new Eclipse context and set it on the new part, because later we will add objects to the context, which we want to make available to the new part only.

After the part is created, you need to add it to one of the containers in the application model, for example, in `Perspective`, `PartSashContainer`, or `PartStack`. In our application we added `SearchResultView` to a `PartStack` instance.

To get reference of the container in the application model, you can use the `findElements` method of `EModelService` (see step 12). You then need to get list of children in the container and append the new part to the list. Finally, you need to activate the new part by calling the `activate` method of `EPartService`. The first argument to the `activate` method is `MPart` that is to be activated.

There's more...

We created a new part by invoking the `createPart` method of `MBasicFactory` and then we set attributes of the part. An alternative way to create a part dynamically is to first define a part descriptor in the application model and then invoke the `createPart` method of `EPartService`. The `createPart` method takes the part ID and returns an instance of `MPart`:



In the `executeFind` method of `FindCommandHandler`, you could create the part as follows:

```
//Create a new Part
MPart findResultPart =
  partService.createPart(
"codesnippetapp.partdescriptor.search.results");

findResultPart.setContext(newPartContext);
```

There is no need to set a contribution URI, label, and closable in the code because these settings are already defined in the part descriptor.

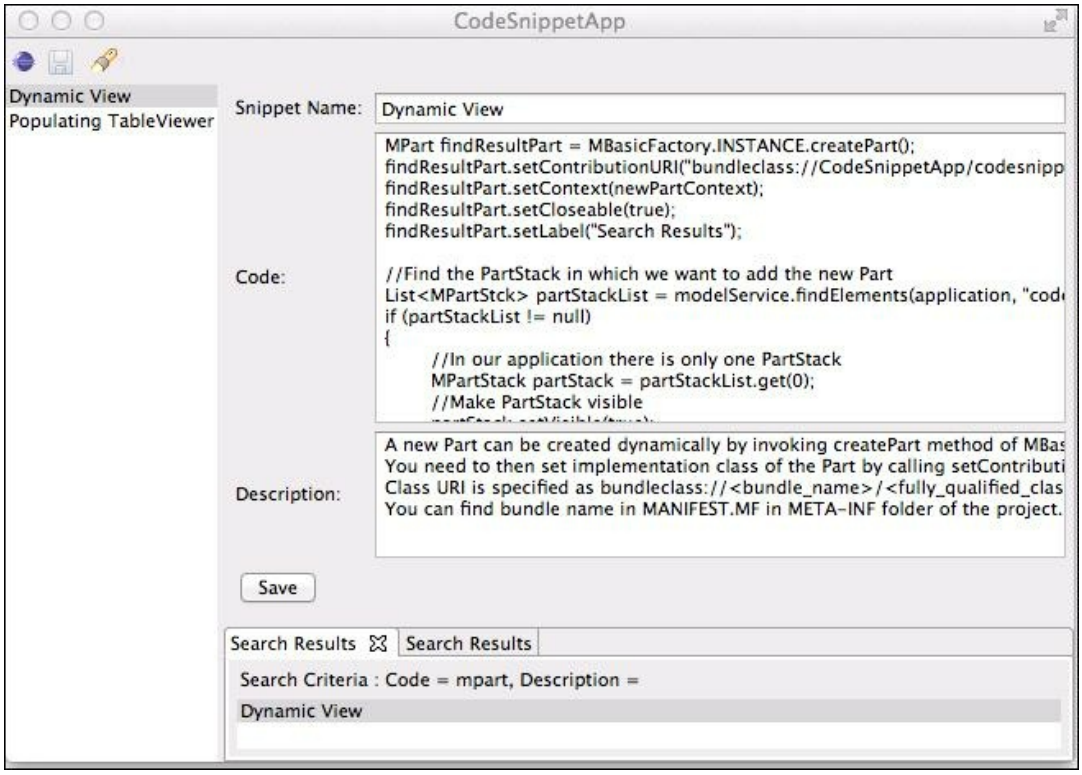
Displaying search results

We will skip the code and description to display search results here, because we have already seen how to display content in the `TableViewer` class and process selection changes. If you are interested in knowing how search results are displayed, then take a look at the `SearchResultsView` and `FindCommandHandler` classes in the accompanying source code for this recipe. The following additional functionality is implemented:

- Display search criteria and search results in the view. We share this information by setting them in the context of new search result part. Refer to the `executeFind` method of `FindCommandHandler`. The search criteria are displayed in a label and the research results are displayed in a `TableViewer` class. Refer to the `postConstruct` method of `SearchResultsView`.
- When selection changes in the `SearchResultView` class, we highlight the same snippet in the `SnippetListView` instance and update information in the `SnippetDetailsView` instance. Refer to the `postConstruct` method of the `SearchResultsView` and `searchResultSelectionChanged` methods of `SnippetListView`.
- When all search result views are closed, hide the `PartStack` instance that contained result views. We do this by keeping count of number of views created and destroyed. Refer to the `postConstruct` and `preDestroy` methods of `SearchResultsView`.
- Implement the **Delete** menu option in the `SnippetListView` instance. The `execute` method in the `DeleteSnippetMenuHandler` class publishes `onSnippetDelete` when the **Delete** menu option is selected. The `onSnippetDelete` method of `SnippetListView` handles this event and removes the selected snippet from the repository and publishes the `beforeSnippetDelete` event. An inline event handler in the `postConstruct` method of `SearchResultsView` handles the `beforeSnippetDelete` event and updates search results list. To implement this functionality, we have used the `IEventBroker.subscribe` method to subscribe to the `beforeSnippetDelete` event. This method requires you to provide implementation of `org.osgi.service.event.EventHandler`, so you need to add

dependency of plugin `org.eclipse.osgi.services` in your plugin's `MANIFEST.MF`.

Here is the screenshot of application after implementing the previous features:



Styling an application using CSS (Simple)

Eclipse 4 provides an easy way to style the UI widgets using CSS. It supports CSS 2.0 specification. You can set attributes such as font color, background color, font size, and so on, of UI widgets.

In this task we will change color and font attributes of some of the UI controls, such as toolbar, text boxes, and so on.

How to do it...

1. Open `default.css` from the `css` folder under the project.
2. Set the background color of all parts to white:

```
.MPart {
background-color:white;
}
```

3. Set the font of textboxes to `12pt`:

```
Text {
font-size:12pt;
}
```

4. Set the font weight of the button text to `bold`:

```
Button {
font-weight:bold;
}
```

5. Set the background of the toolbar:

```
.MTrimBar {
background-color:#27535C;
}
```

6. We will set labels in `SnippetDetailsView` to `bold` and set font size to `12pt`. We will first set the `CSS` class name for the part that contains `SnippetDetailsView`:

    1. Open the application model and go to **Application** | **Windows** | **Trimmed Window** | **Controls** | **Perspective Stack** | **Perspective** | **Controls** | **PartSashContainer** | **PartSashContainer** | **Part (Snippet Details)**.
    2. Click on the **Supplementary** tab. Type `detailsView` in the **Tags** textbox and click on the **Add** button next to it. You should see **detailsView** appear in the list box below.
    3. Go back to `default.css` and add the following styles:

```
.detailsView  Label {
font-size:12pt;
font-weight:bold;
}
```

7. Lastly, we will set the background color of `TableViewer` in `SnippetListView` and set its font color to `white`:

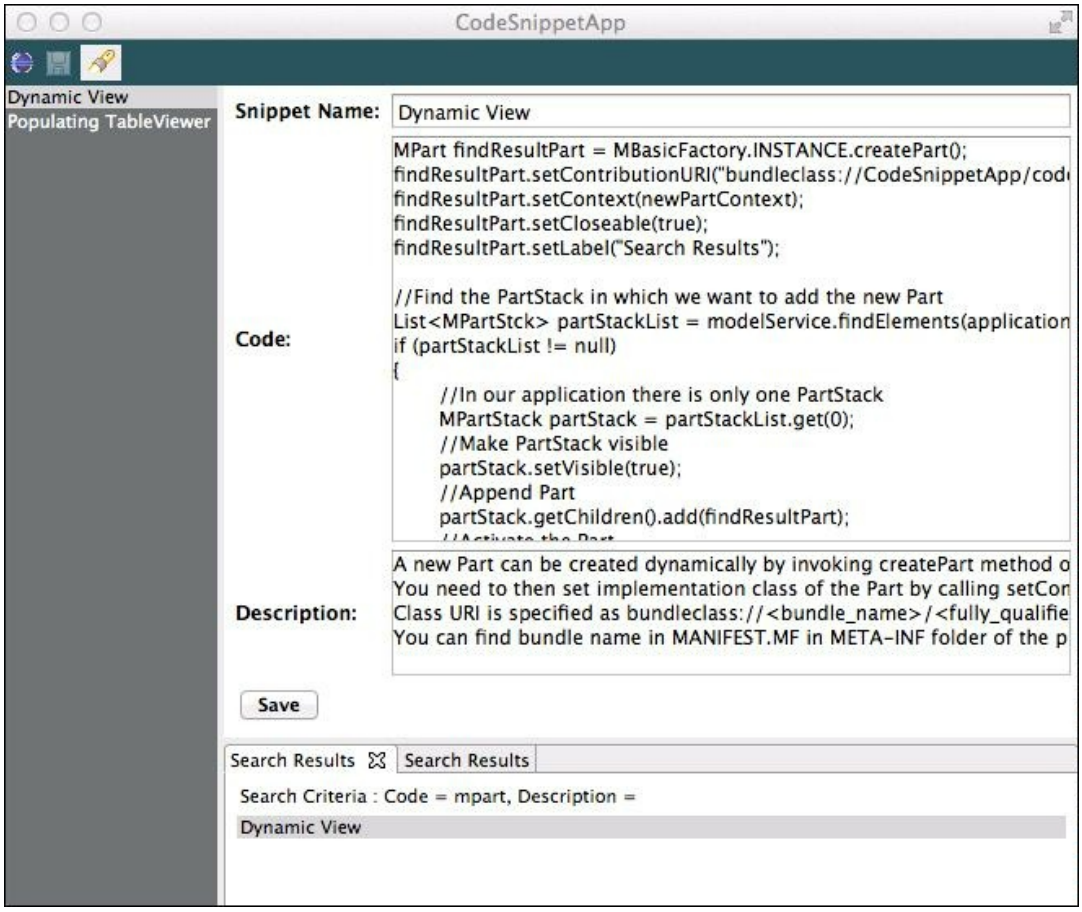
    1. Open the `SnippetListView` class. In the `postContstruct` method, after creating a new instance of `TableViewer` and assigning it to the `snippetsList` variable, add the following code:

```
snippetsList.getTable().setData("org.eclipse.e4.ui.css.id"
"snippetsList");
```

    2. Go back to `default.css` and add the following style:

```
#snippetsList {
background-color:#6C7273;
color:white;
}
```

Run the application after performing the preceding changes. Observe that the previous style changes are applied on the UI controls:



How it works...

The CSS file to be used for styling the application is specified in the `plugin.xml` format, in the `applicationCSS` property of the `org.eclipse.core.runtime.products` extension point. The Eclipse 4 application wizard created this property for us and set its value to `default.css`.

You can specify the following selectors in the `CSS` file:

- The class names of UI Widgets in SWT, for example, `Text`, `Button`, `Label`, and so on. (see step 3 and 4). Just as HTML element names, such as body, table span, and so on, you can refer SWT controls also by names.
- The class names of application model elements, for example, `MPart`, `MTrimBar`, `MPartStack`, and so on. You need to select them as the `CSS` class names (see step 2 and 5).
- Tag names set for elements of application model (see step 6). They become class names in CSS.
- Unique CSS id of UI widgets. You can specify CSS IDs of UI controls by calling the `setData` method of the widget class. Use `org.eclipse.e4.ui.css.id` as key to set CSS ID (see step 7).

- The `CSS` class name set for UI widgets. Use the `setData` method of widget with the `org.eclipse.e4.ui.css.CssClassName` key to set the `CSS` class name.

There's more...

Refer to http://wiki.eclipse.org/Eclipse4/RCP/CSS for more information on CSS support in Eclipse 4. Refer to http://wiki.eclipse.org/E4/CSS/SWT_Mapping to understand mapping between SWT properties and corresponding CSS properties.

Finishing the application

We have not implemented the `save` and `open` repository options for our application. Since these features are not specific to new Eclipse 4 features, we are going to skip the code and description of these features. However, the complete application with these features is available in the accompanying code.

The `save` option is implemented using simple serialization of the `SnippetData` list to a file. We made the `SnippetData` class to implement the serializable interface. The `open` operation desterilizes the data and loads in the application.

Refer to the save and open methods of the `SnippetRepository` class for details. We have also modified the `execute` methods in `OpenHandler`, `QuitHandler`, and `SaveHandler`.

Customizing and exporting the application (Simple)

We have completed the functionality of our demo Eclipse 4 RCP application and now it is time to add icons and splash screen. We will also see how to export this application to multiple platforms.

How to do it...

1. Adding the splash screen:
    1. Create a bitmap image for the splash screen and name it `splash.bmp`. Copy it in the root of the project folder.
    2. Open the product configuration file, `CodesnippetApp.product`. Go to the **Splash** tab. Click on **Browse** button next to the plugin location and select `CodeSnippetApp` from the list.
2. Setting the initial application window size:
    1. Open the application model (`Application.e4xmi`).
    2. Go to **Application** | **Windows** | **Trimmed Window**.
    3. Set width to **800** and height to **500** in the **Bounds** textbox.
3. Setting the application icons:
    1. Create application icons (PNG) images of different sizes, for example, 16x16, 32x32, 64x64, and 128x128 (sizes are in pixels). For Windows, you need to create the BMP files.
    2. Open the `CodeSnippetApp.product` file. Go to the **Launching** tab.
    3. In the **Program Launcher** group you will find options to specify the icon files for different OS. For Linux, you need to create a single XPM file containing all images. For Mac OSX, you need to create a single ICNS file containing all images. For Solaris and Windows, you can specify separate image files. For the purpose of this demo application, we create BMP files for Windows and ICNS file for Mac OSX.
4. Creating a `Launcher` filename:

1. In the **Launching** tab of the `CodeSnippetApp.product` file editor, enter `CodeSnippetsApp` in the **Launcher Name** textbox.

5. Enable the option to export the application to multiple platforms (these steps are required only if you want to export your application to platforms other than the one you are developing this application on):

    1. Open Eclipse preferences and go to **Plug-in Development** | **Target Platform**. Click on the **Add** button in the **Target Platform** page.
    2. Select the option **Current Target: Copy settings from the current target platform**.
    3. Click on the **Add** button on the **Target Content** page. Select the **Directory** option and click on the **Next** button.
    4. Select the location of Delta pack for Eclipse that you downloaded in the *Setting up development environment (Simple)* recipe (make sure that you select the `eclipse` folder of the Delta pack). Click on the **Next** button and finish the wizard.
    5. At the end of the wizard, you should see a new target platform added to the list. Make sure that this target is checked. Click on **OK** and close preferences.

6. Exporting the application:

    1. We need to add the `icons` folder to the binary build before we export the application, else t `icons` files will not be part of the exported application.
    2. Open `build.properties` (from the root of the project). The file will be opened in the plugin editor and the **Build** tab will be active. Alternatively, you can open `plugin.xml` or `MANIFEST.MF` and go to the **Build** tab.
    3. In the **Binary Build** list, check the `icons` folder and `splash.bmp` and save the file.
    4. Open product configuration file, `CodeSnippetApp.product`. Make sure that the **Overview** tab is active. Click on the **Eclipse Product export wizard** link in the **Exporting** section.
    5. In the **Product Export** wizard, set **Root directory** as `CodeSnippetApp`. Set **Destination Directory** and uncheck **Generate metadata repository**. If you had set the Delta pack in the target platform, you would see the **Export for multiple platforms** option. Select this option if you want to export the application to multiple OS.

6. If you had selected the option to export to multiple platforms, then the **Next** button would be enabled. Click on **Next** and select platforms to which you want to export the application.
7. Click on **Finish**.
8. Run the exported application. The application might throw an error if some of the dependencies are not resolved. If this happens, open `CodeSnippetApp.product`, go to the **Dependencies** tab, and click on the **Add Required Plug-ins** button. Save the file and export the application again. Note that that the folder where you export the application should not contain previously exported application. Delete any previously exported application before exporting again.

How it works...

To add a splash screen to your application, you need to create a BMP image of name `splash.bmp`. This file has to be in the root of the plug-in project. Eclipse only lets you select the plugin name to load the splash screen from (in the **Splash** tab of product configuration editor) and not the filename and location.

As described in step 2, you can specify initial size and location of the application window in the application model.

In step 3 we set application icons. The format of icons file is different for different OS. In this application, we have set icon files up to 64x64 pixels. However, on the high-resolution displays, you may want to set icons files up to 512x512 pixels.

If you do not specify the launcher filename (in step 4), Eclipse creates one with the default name `eclipse` (for example, in Windows, it is `eclipse.exe`, and in Mac OSX, it is `eclipse.app`). We have set the launcher name `CodeSnippetsApp`.

Step 5 for configuring delta pack is optional. Delta pack for Eclipse contains all the files (executable and libraries) required for creating the launch program for different platforms.

In the last step, we actually exported the application using the **Product Export** wizard. The wizard builds the application, creates a launcher file, and copies all the files required to run the application to the output folder. You can copy exported folder to any other machine of the same target platform, and should be able to run the application. You can also create an installer using exported files.