
Building cheat sheets in Eclipse

How to provide interactive tutorials for your Eclipse product

Skill Level: Intermediate

[Philipp Tiedt \(philipp_tiedt@de.ibm.com\)](mailto:philipp_tiedt@de.ibm.com)
Software Engineer
IBM

13 Dec 2005

Cheat sheets help your customers get their hands dirty with your product and learn about its features interactively. This tutorial shows you how to develop interactive tutorials, called cheat sheets, for your Eclipse-based product or plug-in.

Section 1. Before you start

This tutorial is written for developers who want to provide interactive tutorials explaining complex Eclipse tasks. You will learn how to make use of the cheat sheets technology and get the best out of it for your tutorial.

About this tutorial

The goal of this tutorial is to get you started with Eclipse cheat sheets, an emerging technology that allows you to create interactive tutorials within Eclipse. The tutorial will give a general overview of the features of cheat sheets and how to implement them.

System requirements

To run the examples, you need to have Eclipse V3.0 or higher installed on your computer. Get more advice on choosing the correct materials for your needs in [Resources](#).

Section 2. Introduction to cheat sheets

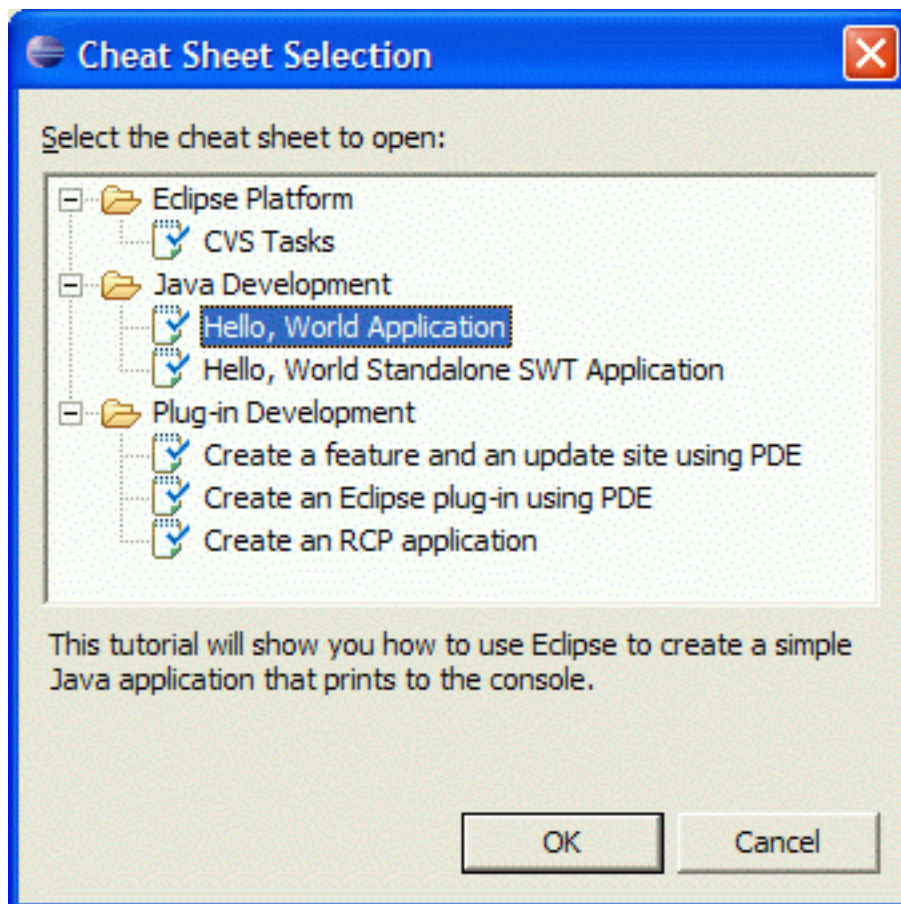
You probably know the following scenario. You are providing a new Eclipse plug-in or even an Eclipse-based product and need the users to get their hands dirty with it. So you start writing some step-by-step examples explaining the new capabilities and features of your product. Your tutorial may be delivered as a .pdf document or as part of the Eclipse help. The user now has to switch between things -- either the sheet of paper in his hand and Eclipse or between the Eclipse help and Eclipse. Wouldn't it be better if there was some view in Eclipse telling the user what the next step is to perform while they stay within the Eclipse workbench? Well there is: cheat sheets.

What are cheat sheets?

Cheat sheets is a new emerging technology within Eclipse V3.0 that is meant to guide a developer through a series of complex tasks to achieve some overall goal. Some tasks can be performed automatically, such as launching the required tools for the user. Other tasks need to be completed manually by the user. Cheat sheets tasks can be hooked up with the Eclipse help so no long search for documentation is required. You will find some sample tutorials in Eclipse under **Help > Cheat Sheets**.

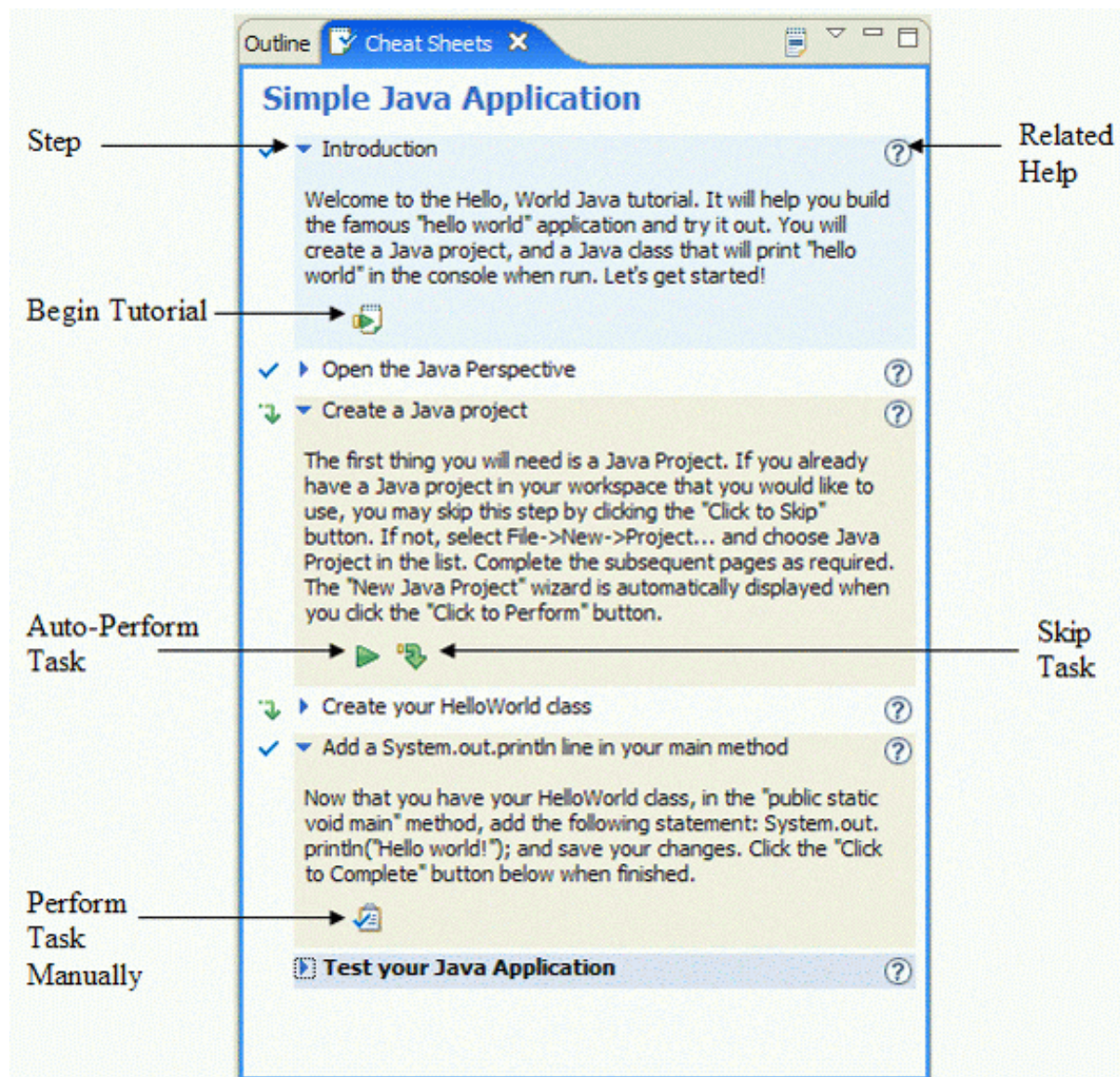
A good example is the "Hello World" cheat sheet of the Java™ Tools in Eclipse, which guides a user through the creation of his first Java program. Clicking on **Help > Cheat Sheets** will open a dialog where you can choose a tutorial.

Figure 1. Cheat sheets selection dialog



Select **Hello, World Application** and click **OK**. Eclipse will now open the cheat sheet view, most likely on the right-hand side of the workbench, and show the selected tutorial.

Figure 2. The cheat sheets view



You will see a short introduction into the tutorial and a couple of steps. Once started, you can perform the listed task one after another. Some tasks can be performed automatically. Tools like the wizard for new Java projects are launched automatically, or a perspective is opened automatically. Other steps must be completed manually and marked as completed in the cheat sheet.

Why use cheat sheets?

Tutorials are actually one of the best and most convenient ways of documentation. Users get their hands dirty very quickly and learn a lot about your product. Cheat sheets integrates tutorials and how-tos with your plug-in or Eclipse-based product. This will speed up the training curve and make the users feel more comfortable learning about it. Cheat sheets tops off the integration of your functionality into Eclipse because the tutorial becomes a part of your product.

Section 3. A first cheat sheet

In this part, you will learn how to create and run a new cheat sheet with a few basic steps. First, we will set up a small scenario to have context for our new tutorial.

The library model tutorial

Suppose you want to teach a user how to create a simple Java API and package it using the Eclipse Java Tools. Let's take a simple library model as a sample our users should develop. The user needs to develop three classes:

1. A Library class, which has a name field, a list of writers, and a list of books
2. A Book class, which has a name field, a reference to a writer, and a number of pages field
3. A Writer class, which has a name field and a list of books that the writer has written

Now, the idea is to let the user learn about the Java Tools, so we want to teach as much as possible and use as many features as we can. Let's set up a basic outline of the tutorial:

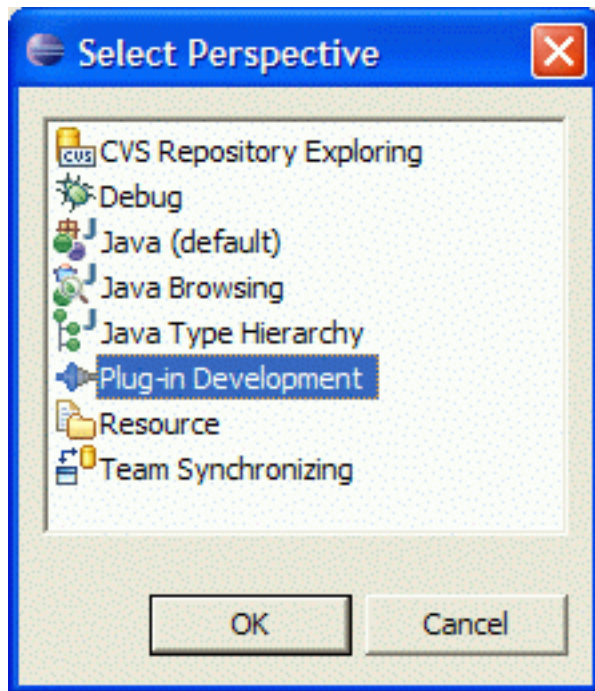
1. Opening the Java Perspective
2. Creating a Java Project
3. Creating a Java Package
4. Creating the three Java classes
5. Create a JAR file, including all three classes

Now that we have our first outline set, we can start creating our first simple cheat sheet.

Creating the cheat sheet plug-in

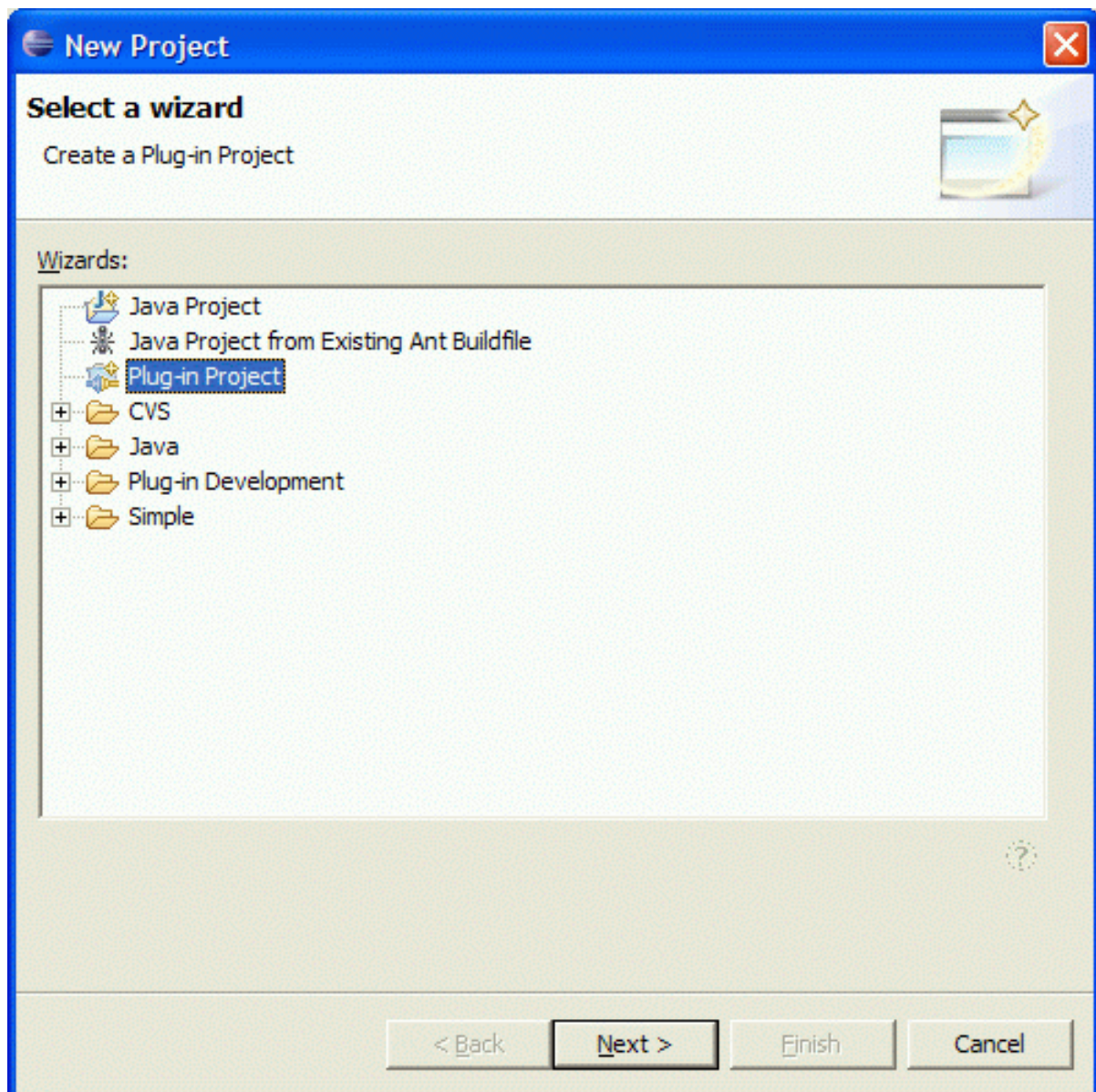
The first thing to do is to switch to the Plug-in Development perspective in Eclipse via **Window > Open Perspective > Other ... > Plug-in Development**.

Figure 3. The Perspective Selection dialog



Now we will create a new Eclipse Plug-in. Open the New Project Wizard via **File > New > Project ...** and select **Plug-in Project**.

Figure 4. The New Project wizard



Clicking **Next** will take you to the Plug-in Project wizard, where you enter a project name and continue to the next page via **Next**.

NOTE: Choose 3.0 as target version for your plug-in if you are developing with Eclipse V3.0, as opposed to V3.1 or V3.1.1.

Figure 5. The New Plug-in Project wizard

New Plug-in Project

Plug-in Project
Create a new plug-in project

Project name:

Project contents

☒ Use default

Directory:

Project Settings

☒ Create a Java project

Source Folder Name:

Output Folder Name:

Plug-in Format

What version of Eclipse is this plug-in targeted for?

☒ Create an OSGi bundle manifest

< Back Next > Finish Cancel

On the next page, you may enter a provider name and click **Finish**.

Figure 6. Plug-in content page

New Plug-in Project

Plug-in Content
Enter the data required to generate the plug-in.

Plug-in Properties

Plug-in ID:

Plug-in Version:

Plug-in Name:

Plug-in Provider:

Classpath:

Plug-in Class

☒ Generate the Java class that controls the plug-in's life cycle

Class Name:

☒ This plug-in will make contributions to the UI

Rich Client Application

Would you like to create a rich client application? ☐ Yes ☒ No

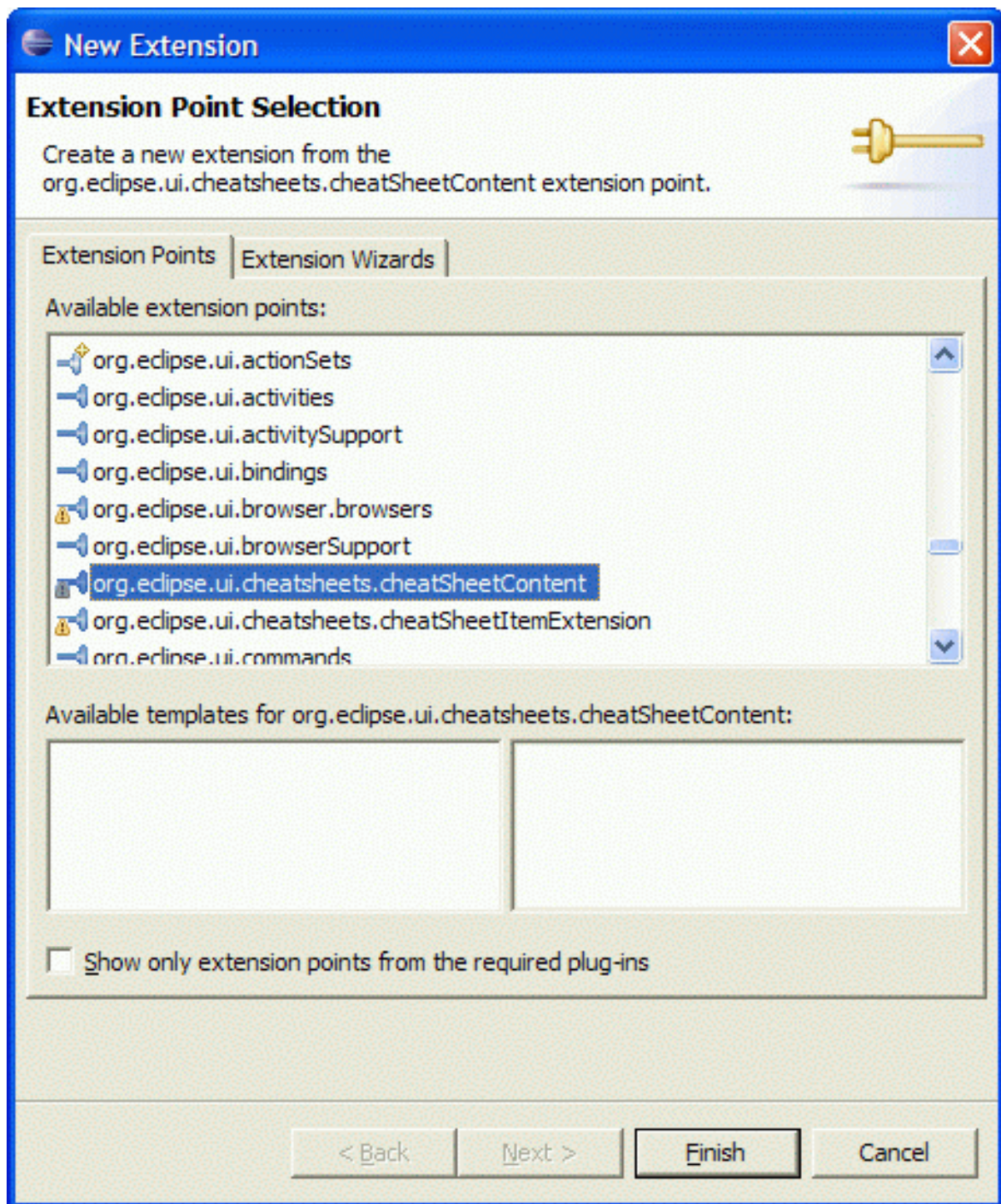
< Back Next > Finish Cancel

The wizard has created a new plug-in and will open the manifest file in an editor. We can continue to create our first cheat sheet.

Creating a new cheat sheet extension

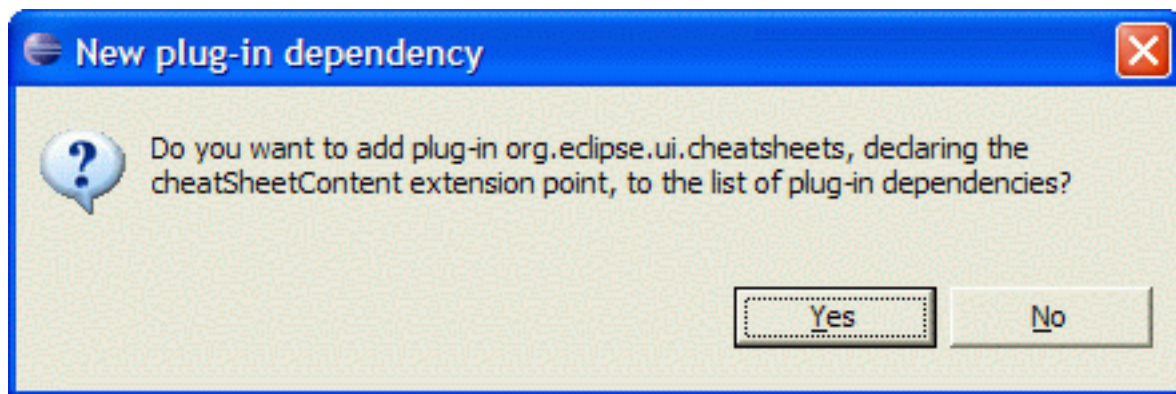
The next step is to create an extension. Eclipse provides an extension point called `org.eclipse.ui.cheatsheets.cheatSheetContent` that can be used to register a new cheat sheet. To create an extension, go to the **Extensions** tab on the bottom of the manifest editor and click **Add**. This will bring up a list of extension points.

Figure 7. The extension point selection dialog



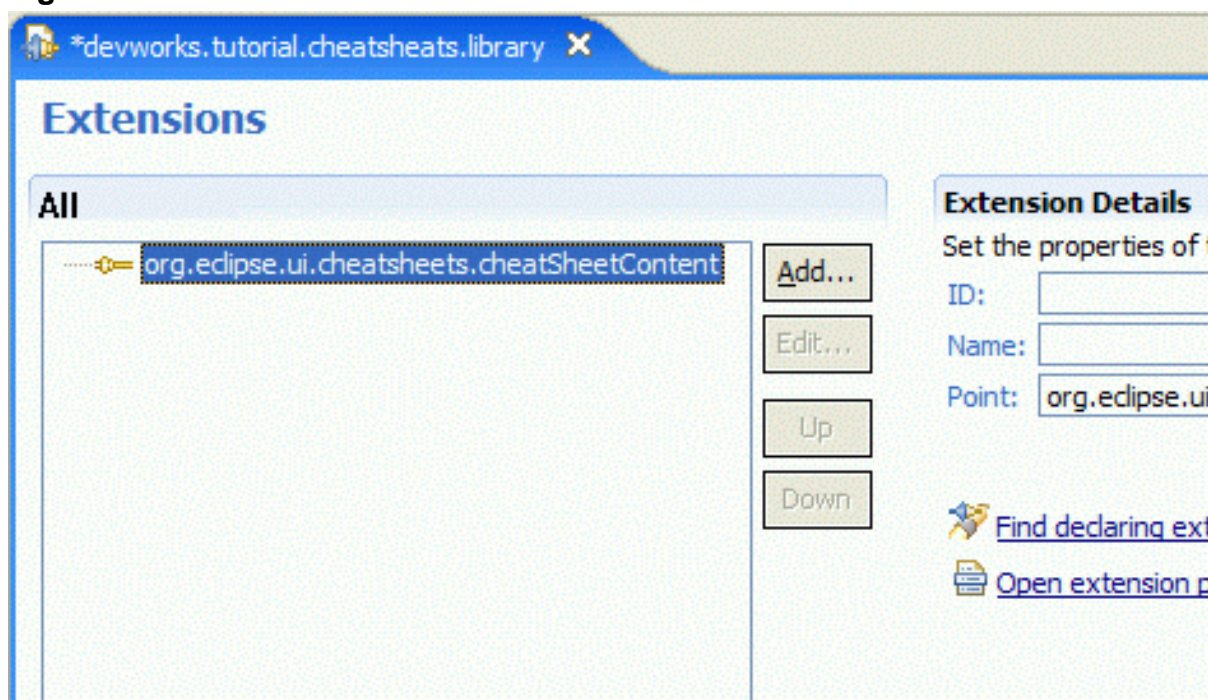
Uncheck the checkbox at the bottom and select the `org.eclipse.ui.cheatsheets.cheatSheetContent` extension point before you click **Finish**. If asked to add the `org.eclipse.ui.cheatsheets` plug-in to your dependency list, select **Yes**.

Figure 8. Add to dependencies dialog



You have now created your first cheat sheet extension, and your manifest file should look like Figure 9.

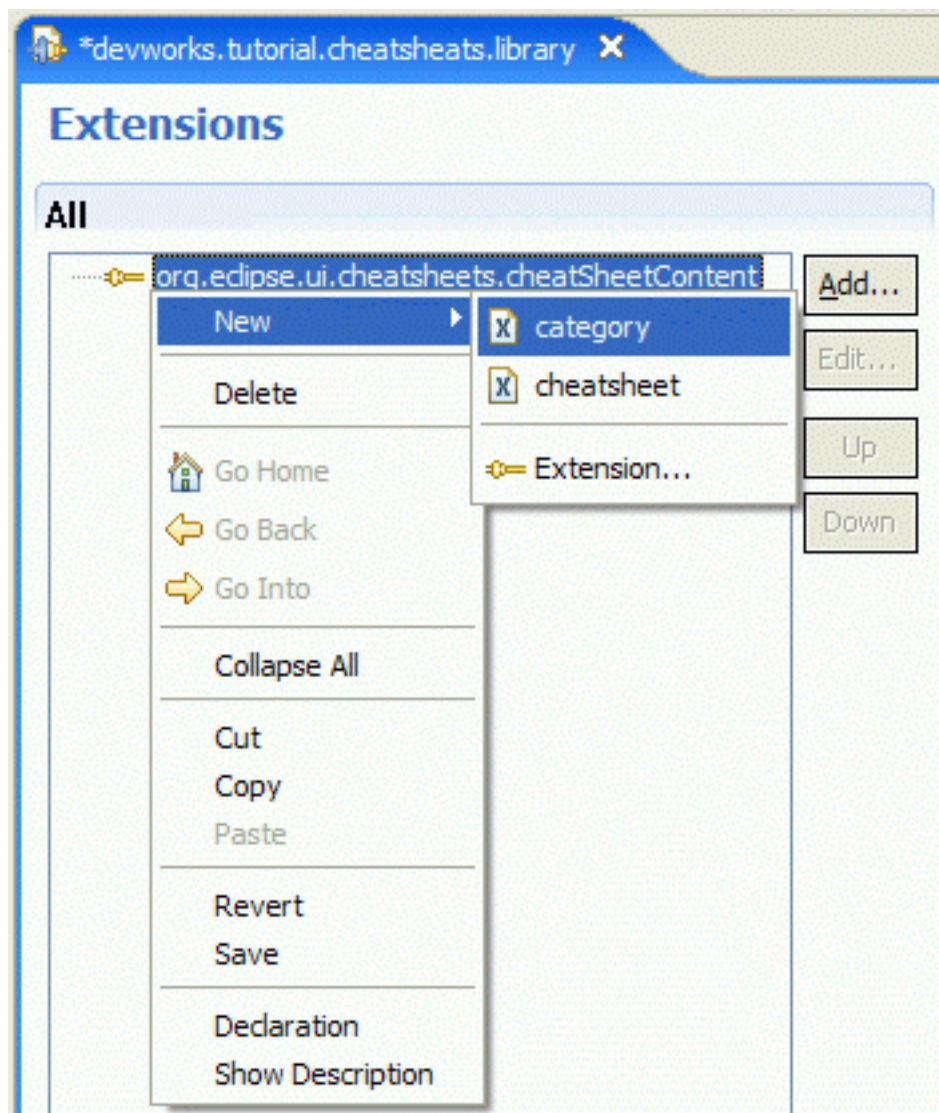
Figure 9. New cheat sheet extension



Creating a category

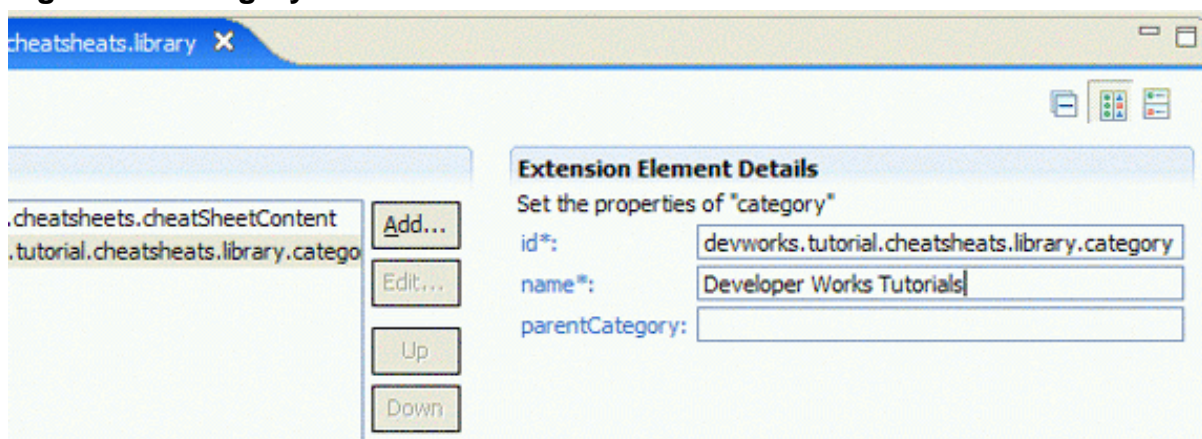
Cheat sheets can be located in categories. You can add your cheat sheet to an existing category or create a new one. If neither are done, your cheat sheet will be located in a default category. To create a new category, right-click the newly created extension in your manifest editor and select **New > category**.

Figure 10. Creating a new category



Now you can enter a category ID and a name for your new category.

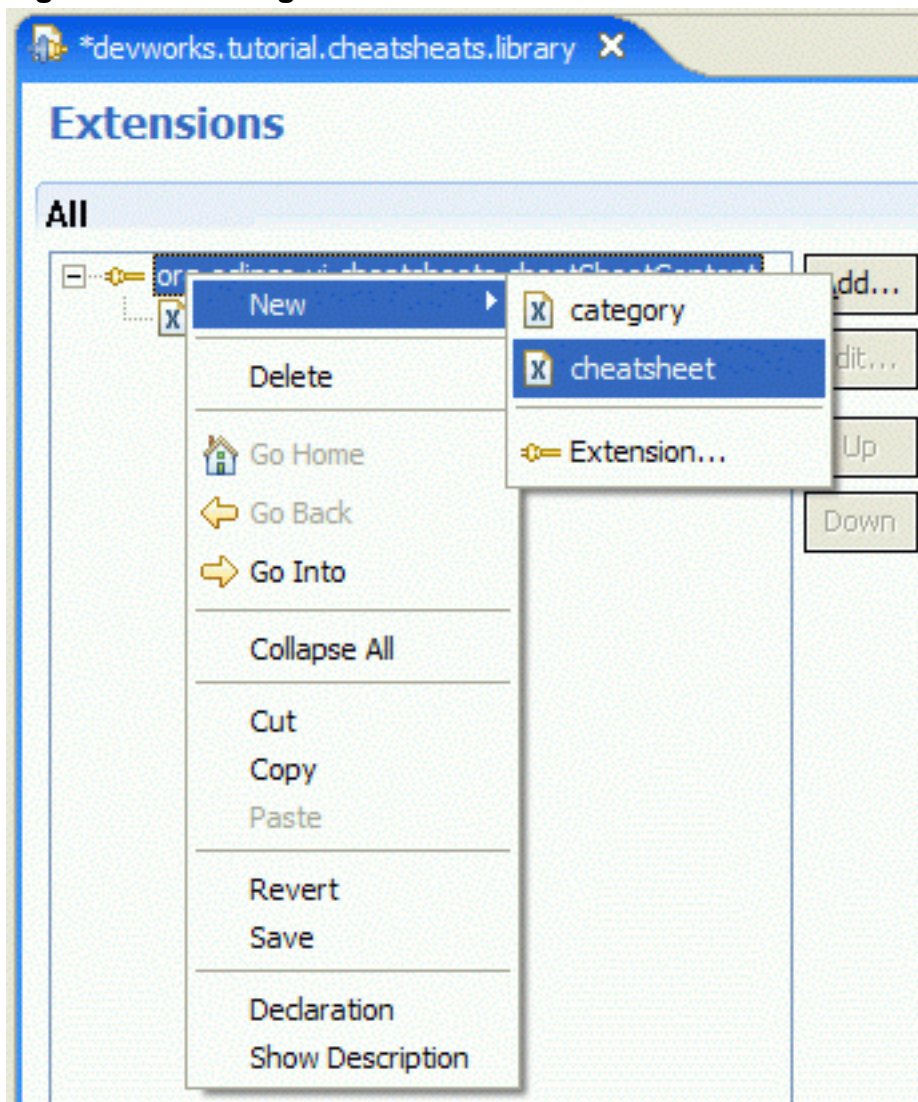
Figure 11. Category attributes



Creating the cheat sheet content

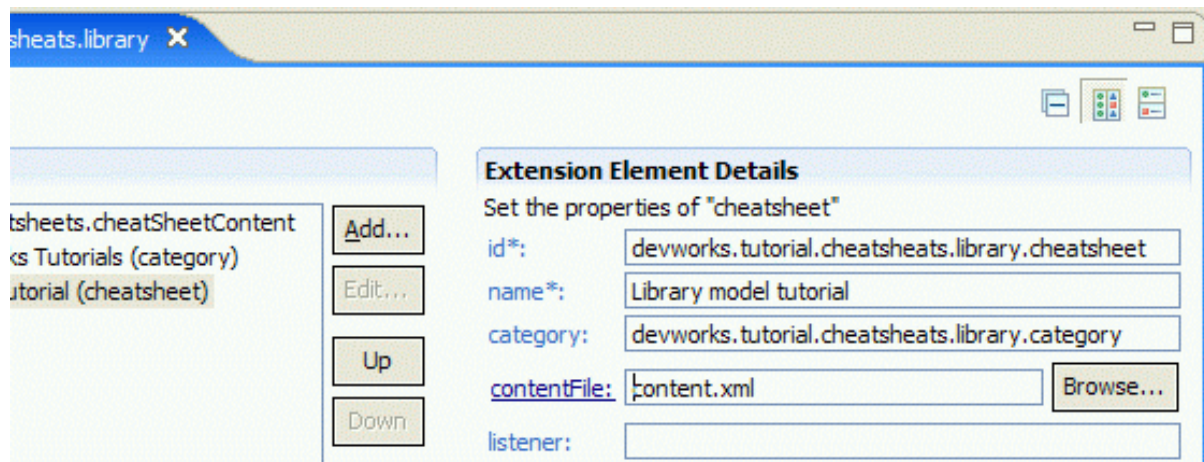
Now that we have a category, we want to add a new cheat sheet. Right-click on the extension and select **New > Cheat Sheet**.

Figure 12. Creating a new cheat sheet



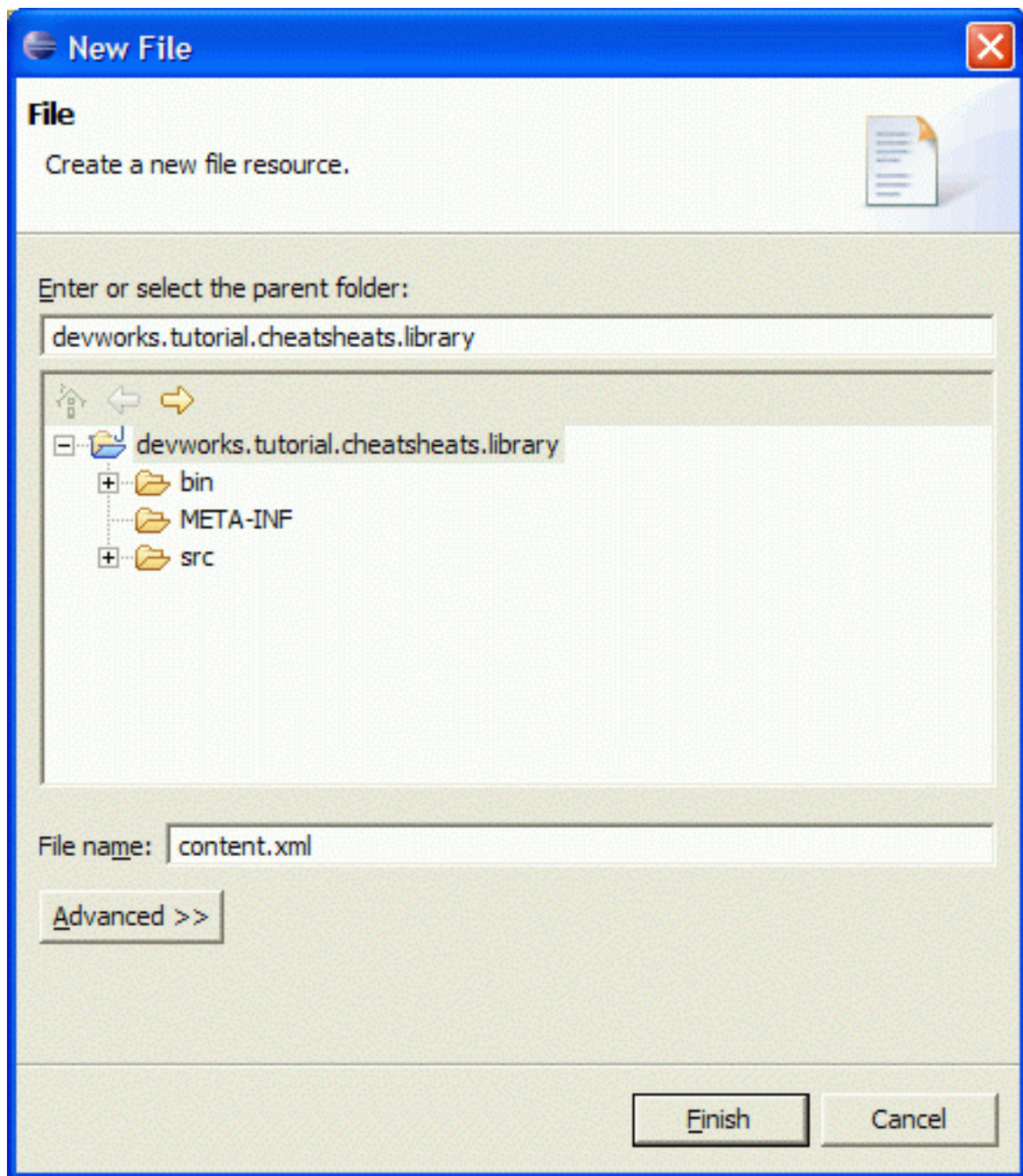
Enter a unique ID for the cheat sheet and a name. For the category, type in the ID of the category you created before. Last but not least, specify the file that includes the content of the cheat sheet.

Figure 13. Cheat sheet attributes



As the content file does not exist, yet we will create it now. Select **File > New > File**, enter the file name and select the plug-in project in the tree before clicking **Finish**.

Figure 14. The New File wizard



The new file will be opened in the Eclipse text editor automatically. You may use an XML editor if you have one installed in Eclipse.

Now we are ready to enter the content. Think of the steps we have defined before. You might enter something like Listing 1; you can simply copy and paste the content into your file.

Listing 1. content.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<cheatsheet title="Library model tutorial">
  <intro>
<description>This tutorial guides you through the creation of a simple Java
  model with three Java classes.
```

```

</description>
</intro>
<item title="Open the Java Perspective">
<description>Select Window > Open Perspective > Java in the menubar at the
top of the workbench. This step changes the perspective to set up the
Eclipse workbench for Java development.
</description>
</item>
<item title="Create a Java project" skip="true">
<description>The first thing you will need is a Java Project. If you
already have a Java project in your workspace that you would like to use,
you may skip this step by clicking the "Click to Skip" button. If not,
select File > New > Project... and choose Java Project in the list.
Complete the subsequent pages as required.
</description>
</item>
<item title="Create a Java package" skip="true">
<description>You should now have a Java project in your workspace. The next
thing to do is creating a package. Use the Eclipse tools by selecting
File > New > Package action. Give the package a name for example
"tutorial.library.model" and click the "Finish" button. If you already
have a project with a package you might as well skip this step.
</description>
</item>
<item title="Create the library model classes">
<description>Now you should be set up for creating your library model. The
library model consists of three Java classes, a library class, a writer
class and a book class. Use the Java class wizard by selecting
File > New > Class. Repeat this for every class.
</description>
</item>
<item title="Package your classes into an archive">
<description>In the last step of this tutorial you will package the created
classes into a Java archive or JAR file. Therefore, right-click your Java
project and select the "Export..." action. In the wizard select "JAR file"
and click the "Next" button. On the next page specify a location and name
for the JAR file and click "Finish". You have now successfully created a
little Java model and packaged that into a JAR file.
</description>
</item>
</cheatsheet>

```

Let's have a look at the content in detail. The root element is `cheatsheet`. A cheat sheet must have an introduction, represented by the `intro` element; and one or more steps, represented by `item` elements. In the content file above, you can see one `intro` element and five `item` elements representing the five steps of the tutorial.

The `intro` element contains a `description` element, which is used to enter the introduction text. You should shortly explain what the user will learn and what the result of the cheat sheet is.

Every `item` element contains a `description` element, as well. Here, the description describes what has to be done in this step and how. In the `title` attribute of the `item` element, you can enter a step title. In some cases, the user has performed a step before starting a cheat sheet; for example, in our case, they might have created a Java project already. You can set the `skip` attribute of the `item` element to `true` if you want the user to be able to skip a step.

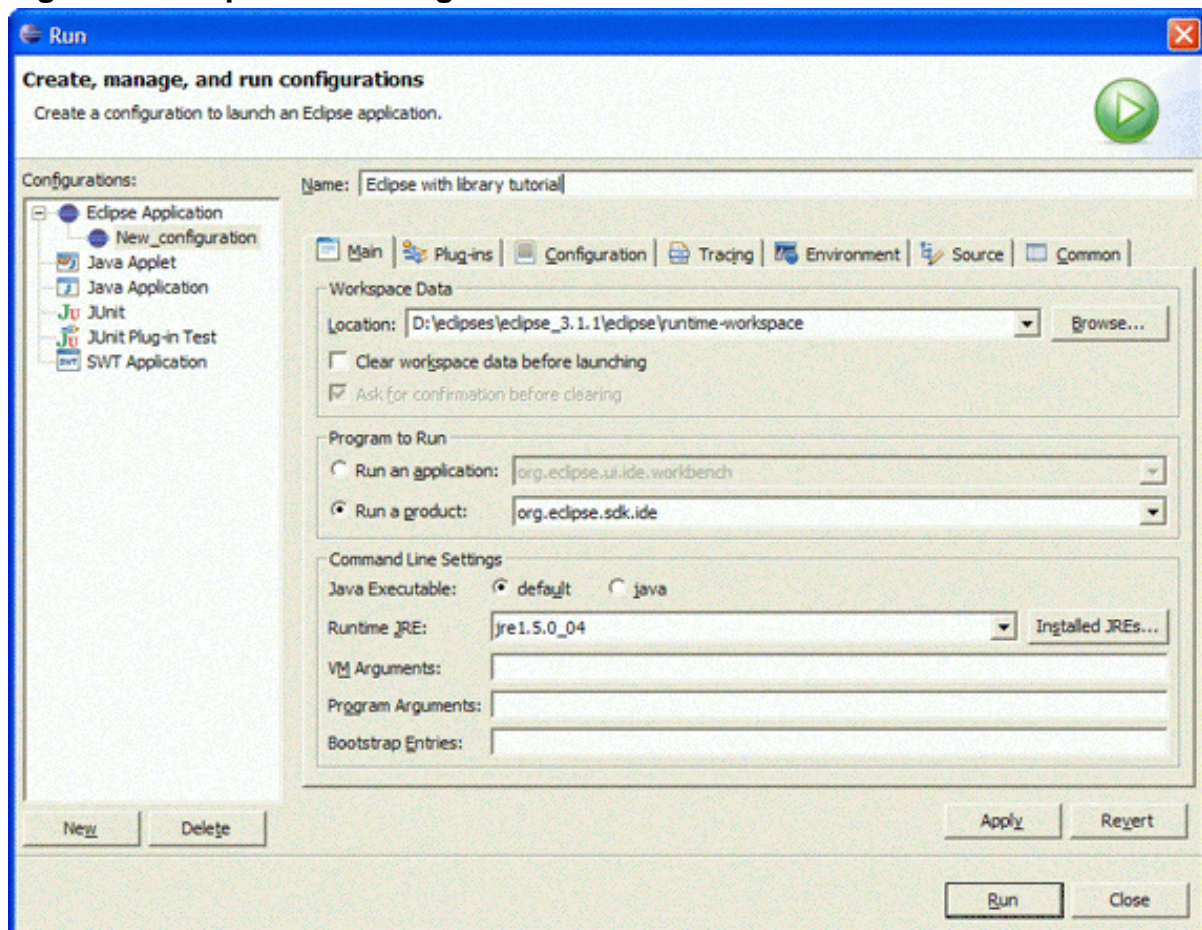
After saving the file, we can try out our first simple cheat sheet.

Testing the cheat sheet

To test our cheat sheet, we have to run Eclipse with our plug-in. This is as easy as launching a Java application from within Eclipse. We just use the Eclipse run-time workbench instead.

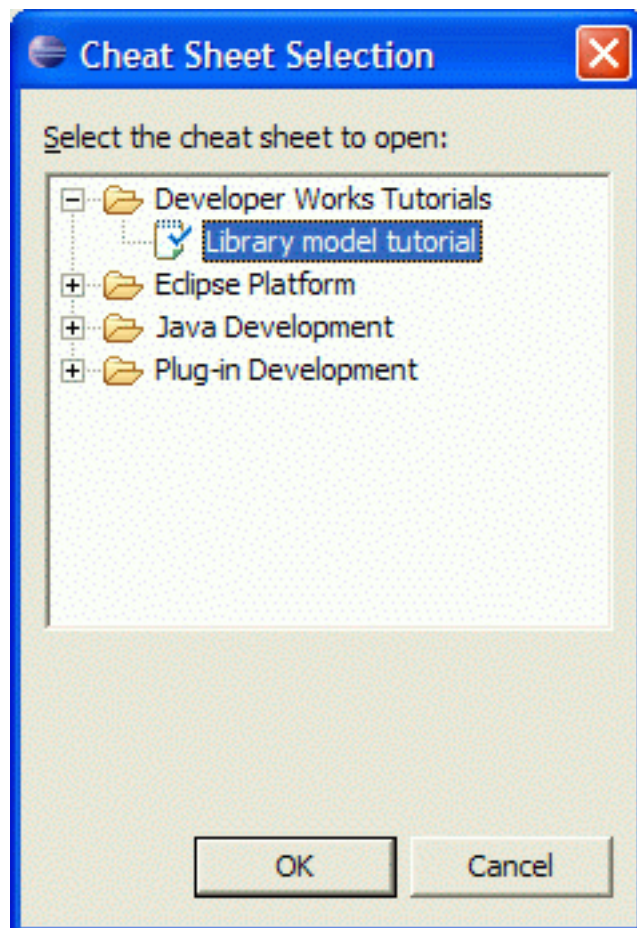
First, open the Run dialog by navigating to **Run > Run....** Select **Eclipse Application** and **New**. This will open a configuration page on the right side of the selection. If desired, you can enter a name for your run configuration. To run Eclipse with our cheat sheet, simply click **Run**.

Figure 15. Eclipse Run dialog



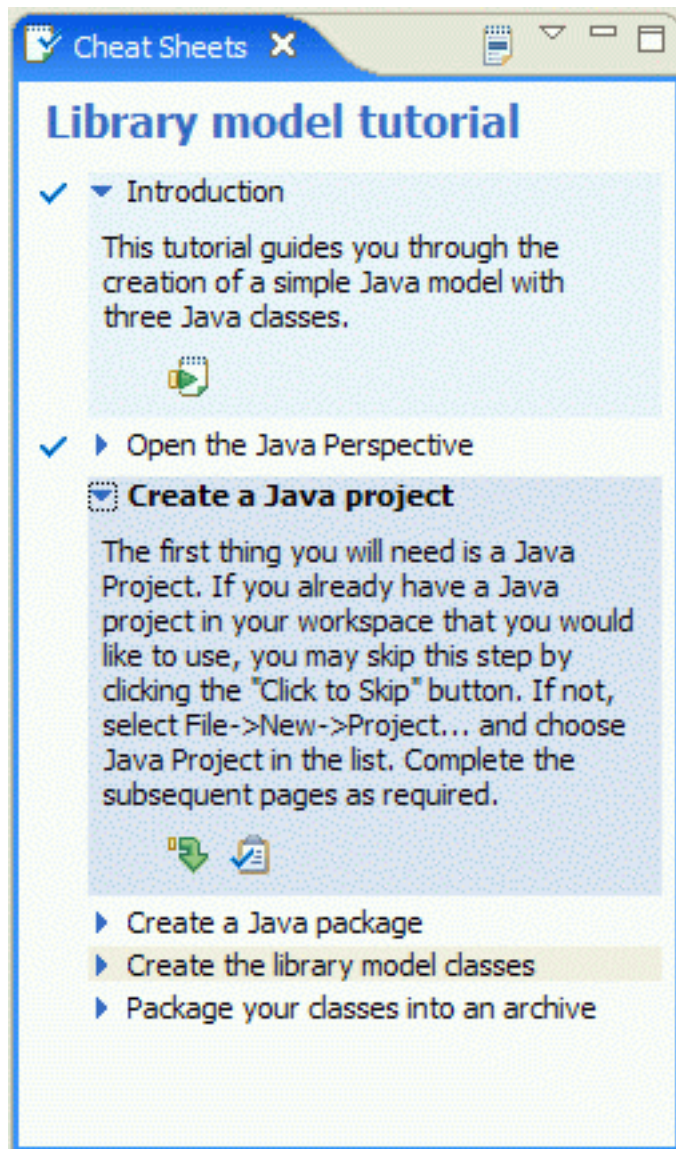
Eclipse launches a new run-time workbench, including our cheat sheet plug-in. When the workbench is launched, click **Help > Cheat Sheets...**, and the cheat sheet selection dialog will pop up.

Figure 16. Cheat sheet Selection with new tutorial



You should now see the new category, including your Library model tutorial cheat sheet. Click **OK** to open it.

Figure 17. Library Model cheat sheet



The result should look something like this. If so, congratulations! You have created your first cheat sheet with five simple steps.

Section 4. Automating your cheat sheet

Now that you have created your first cheat sheet, we want to extend it a little bit and make the user feel more comfortable. Cheat sheets have the possibility of automatically performing a step. In this section, you will learn how. We will automate as many steps as possible.

Let's try to automate the first step: opening the Java Perspective. We need to add an `action` element to our `item`. The `action` element requires two attributes:

1. `class` -- The name of the class that implements the action
2. `pluginId` -- The ID of the plug-in that contains the Java class

The class that implements the action to be executed must implement the interface `org.eclipse.jface.action.IAction`. If the action we implement also implements the `org.eclipse.ui.cheatsheets.ICheatSheetAction`, it will be invoked via its `run(String[], ICheatSheetManager)` method. With the `String` array, the action gets passed parameters from the cheat sheet. To specify parameters for the action, you can use the attributes `param1` to `param9` in the `action` element.

Linking steps with existing actions

So, for our step, we have to create a new class implementing the action interface and make sure the Java Perspective is open when the action is executed. Luckily, there are other plug-ins that have implemented actions like this before, so we can reuse them. In our case, this is the cheat sheet plug-in that implements an action to open a perspective. The perspective to be opened can be defined by a parameter. So we add the following `action` element to our first `item` element.

Using nonpublic APIs

In the sample above, we made use of the class `OpenPerspective`, defined in the package `org.eclipse.ui.internal.cheatsheets.actions`. Classes defined in such internal packages (all packages that contain an "internal" segment) are non-public APIs and, thus, their use is discouraged. You should not use those classes in your application. However, for the purpose of this tutorial, we can overlook it because we want to easily reuse existing functionality without reimplementing.

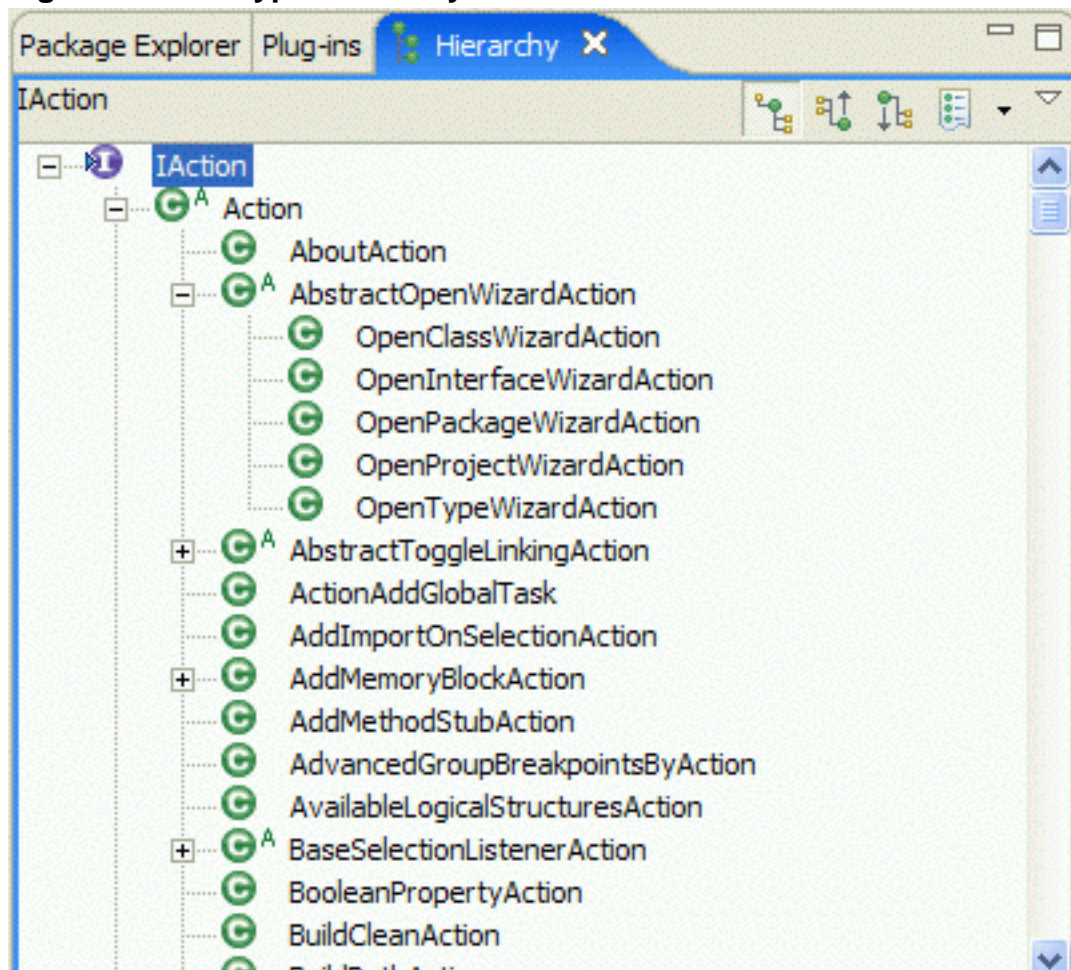
Listing 2. Adding an action

```
...
<item title="Open the Java Perspective">
  <action pluginId="org.eclipse.ui.cheatsheets"
    class="org.eclipse.ui.internal.cheatsheets.actions.OpenPerspective"
    param1="org.eclipse.jdt.ui.JavaPerspective"/>
  <description>Select Window > Open Perspective > Java in the menubar at the top
    of the workbench. This step changes the perspective to set up the Eclipse
    workbench for Java development.
  </description>
</item>
...
```

The action to be performed is implemented in `org.eclipse.ui.internal.cheatsheets.actions.OpenPerspective`. We have to specify the ID of the containing plug-in, which is `org.eclipse.ui.cheatsheets`, pointing to the cheat sheet plug-in. And finally, to tell the action which perspective to open, we hand over a parameter with the ID of the perspective.

Tip: To find out which actions exist, simply open the Java™ Type Hierarchy for the IAction interface and browse.

Figure 18. The Type Hierarchy



NOTE: Cheat sheets can only use action implementations that provide a no-argument constructor. Eclipse V3.2 will hopefully provide some help to find available actions.

For steps 2 and 3, we add the following actions to the items.

Listing 3. Open the Java Project wizard

```
<item title="Create a Java project" skip="true">
  <action pluginId="org.eclipse.jdt.ui"
    class="org.eclipse.jdt.internal.ui.wizards.OpenProjectWizardAction"/>
  <description>The first thing you will need is a Java Project. If you already
    have a Java project in your workspace that you would like to use, you may
    skip this step by clicking the Click to Skip" button. If not, select
    File > New > Project... and choose Java Project in the list. Complete the
    subsequent pages as required.
  </description>
</item>
```

Listing 4. Open the Java Package wizard

```
<item title="Create a Java package" skip="true">
```

```

<action pluginId="org.eclipse.jdt.ui"
class="org.eclipse.jdt.internal.ui.wizards.OpenPackageWizardAction"/>
<description>You should now have a Java project in your workspace. The next
thing to do is creating a package. Use the Eclipse tools by selecting
File > New > Package action. Give the package a name for example
"tutorial.library.model" and click the Finish button. If you already have
a project with a package you might as well skip this step.
</description>
</item>

```

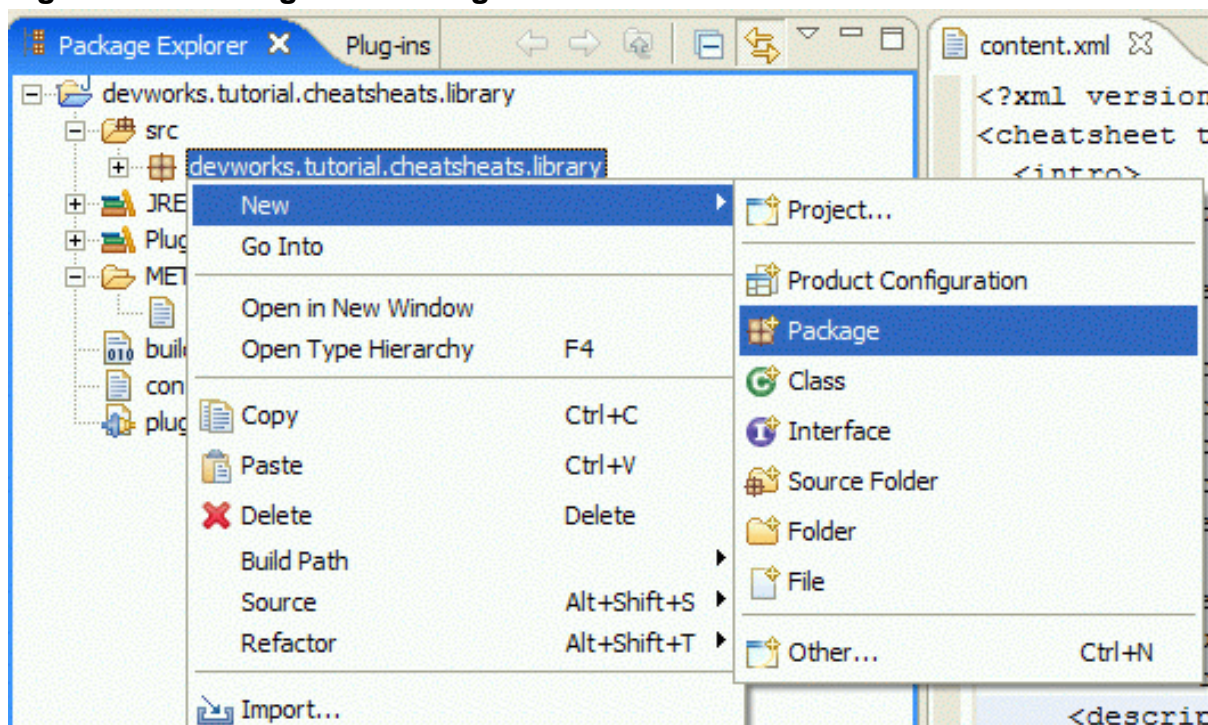
We will take a look at the automation of creating the three Java classes next.

For the last step, we would like to open the Export JAR file wizard. Unfortunately, there is no existing action we can make use of, so we have to implement our own.

Implementing a cheat sheet action

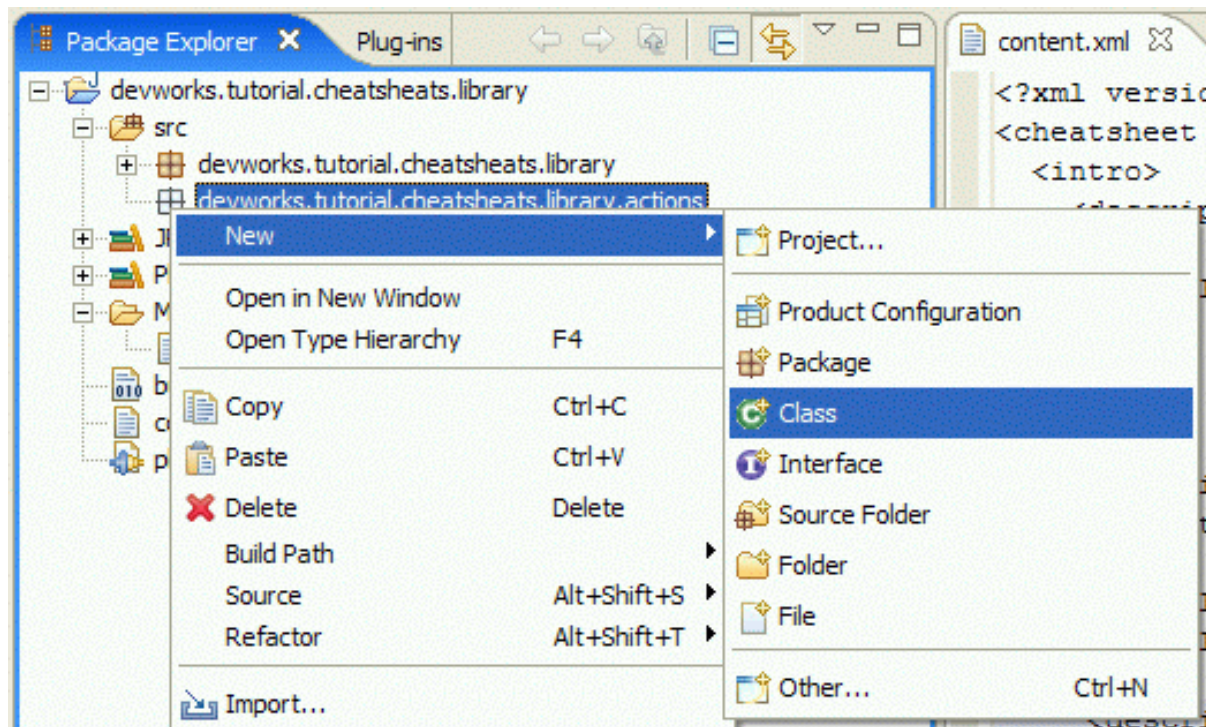
First, we want to create a new package where our action will reside. Therefore, right-click on your project and select **New > Package**, which will open the New Package Wizard.

Figure 19. Creating new Package



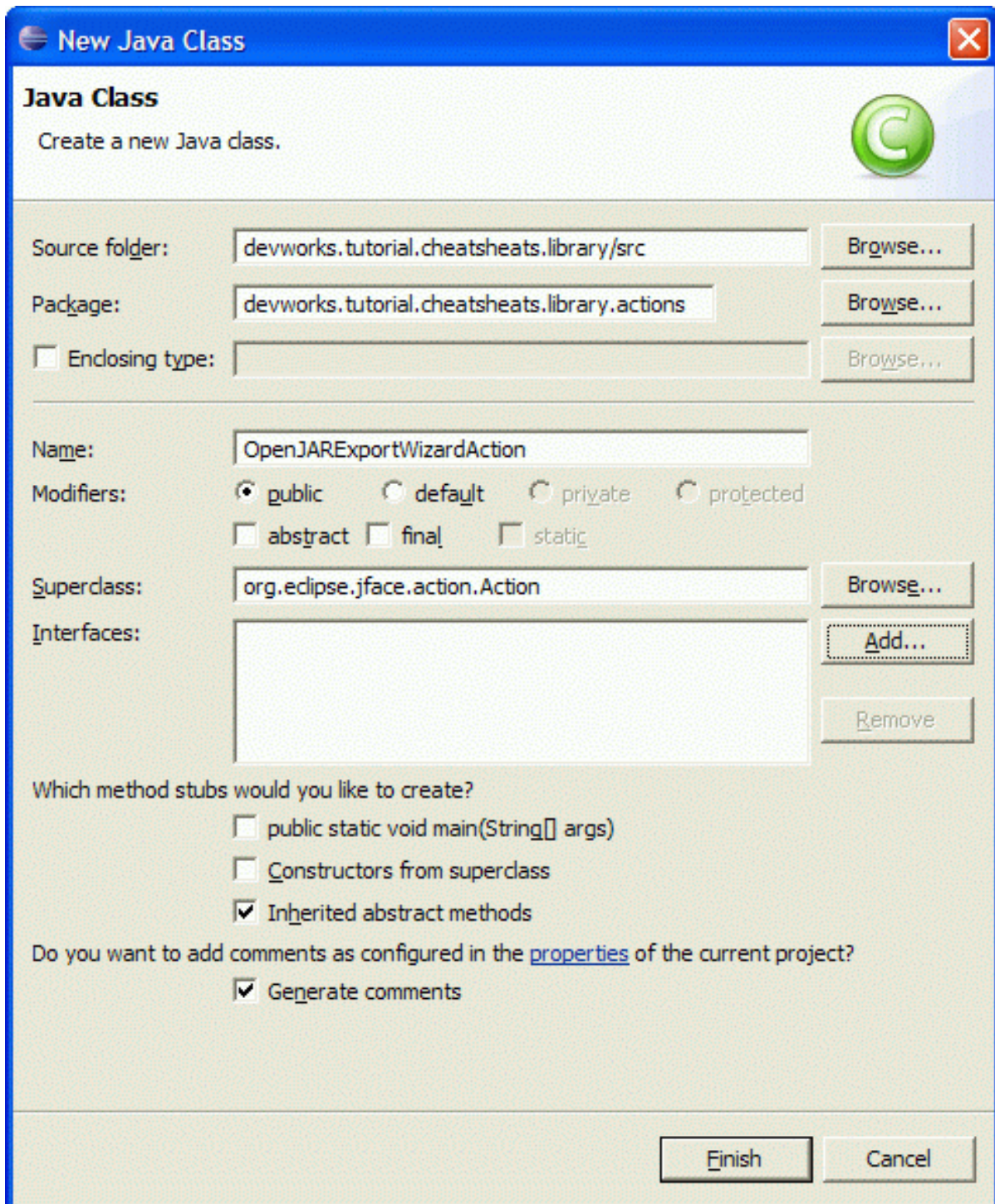
Enter a package name -- for example, `devworks.tutorial.cheatsheets.library.actions` -- and click **Finish**. The next step is to create the action class. Right-click the new package and select **New > Class**.

Figure 20. Creating a new Class



This will open the New Class Wizard. Type a name for your class -- for example, OpenJARExportWizard.

Figure 21. The New Class wizard



New Java Class

Java Class
Create a new Java class.

Source folder:

Package:

☐ Enclosing type:

Name:

Modifiers: ☒ public ☐ default ☐ private ☐ protected
☐ abstract ☐ final ☐ static

Superclass:

Interfaces:

Which method stubs would you like to create?

☐ public static void main(String[] args)

☐ Constructors from superclass

☒ Inherited abstract methods

Do you want to add comments as configured in the [properties](#) of the current project?

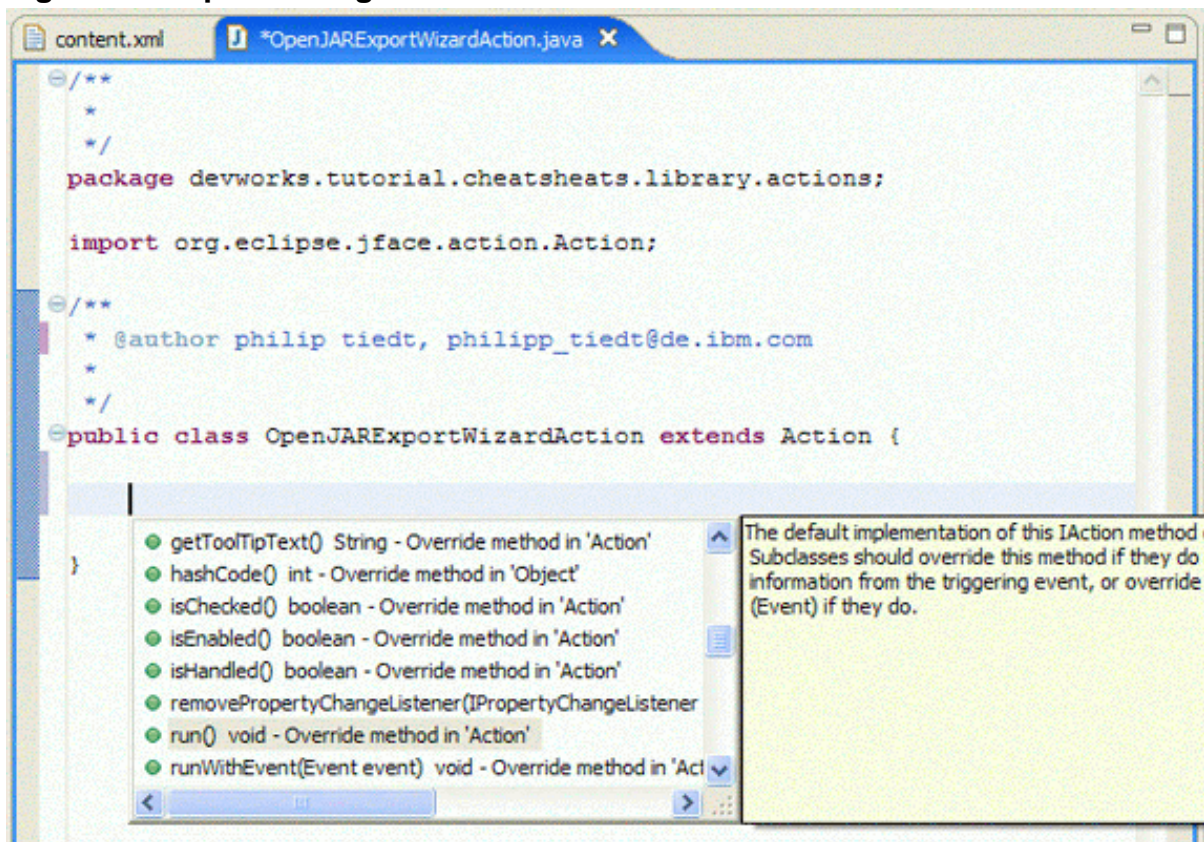
☒ Generate comments

We will let our action class extend the `org.eclipse.jface.action.Action` class, which implements the required `IAction` interface. You can type the class name directly into the superclass field and use the code assistant (`CTRL+SPACE`) to resolve the full qualified name or you can browse for the action class using **Browse**. Click **Finish**, and the new class is created and opened in the Java editor.

We could now add this action to our cheat sheet, but nothing would happen when the action would be executed as the default implementation of the `IAction.run()` method does nothing. So we need to override the `run()` method. In the Java Editor,

press **CTRL+SPACE** and select the `run()` method in the content assistant.

Figure 22. Implementing the run method



The assistant will create a method body. Your code should look similar to this now.

Listing 5. Auto-generated run() method

```

package devworks.tutorial.cheatsheets.library.actions;

import org.eclipse.jface.action.Action;

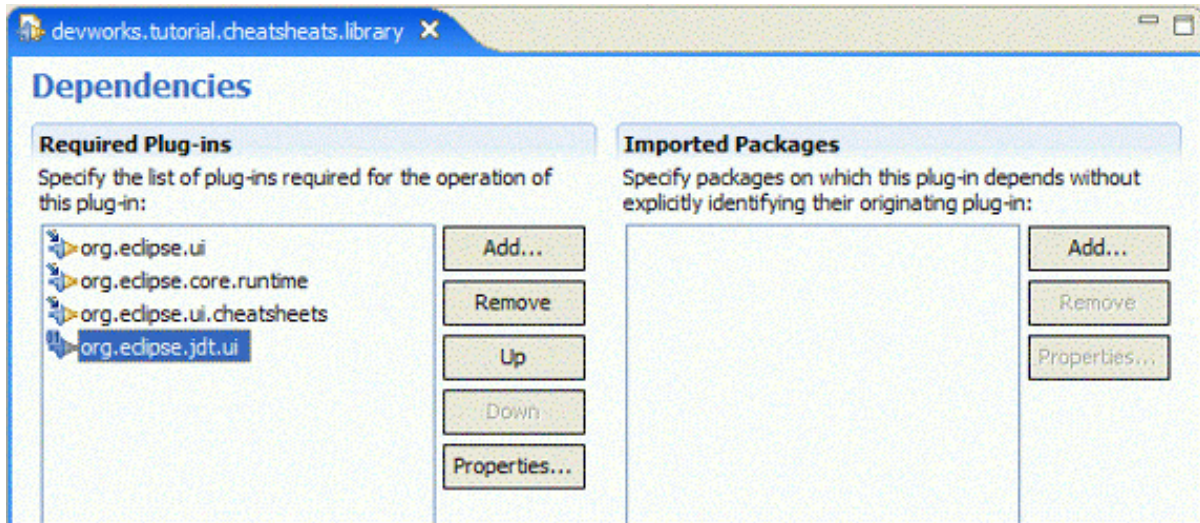
/**
 * @author philip tiedt, philipp_tiedt@de.ibm.com
 */
public class OpenJARExportWizardAction extends Action {

    public void run() {
        // TODO Auto-generated method stub
        super.run();
    }

}

```

We will implement the `run()` method now so it opens the JAR Export Wizard. The wizard is part of the JDT UI plug-in, so at first, we need to add a dependency to `org.eclipse.jdt.ui` in our plug-in manifest file. Go to the Dependencies tab in the manifest editor and click **Add**. In the dialog, type `org.eclipse.jdt.ui` and click **OK**.

Figure 23. Adding the JDT dependency

The implementation for the `run()` method looks like Listing 6.

Listing 6. Implemented `run()` method

```

        public void run() {
            IWorkbench workbench = PlatformUI.getWorkbench();
            Shell shell = workbench.getActiveWorkbenchWindow().getShell();
            JarPackageWizard wizard= new JarPackageWizard();
            wizard.init(workbench,new StructuredSelection());
            WizardDialog dialog= new WizardDialog(shell, wizard);
            dialog.create();
            dialog.open();
            //did the wizard succeed?
            notifyResult(dialog.getReturnCode()==Dialog.OK);
        }

```

In code in Listing 6 works as follows:

1. We get a reference to the workbench and to the current active shell on the workbench.
2. Then we construct the `JarPackageWizard` and initialize it with the workbench.
3. The second argument is a selection the wizard can use to pre-select a project to be packaged. We keep it simple and pass an empty selection.
4. Now we create and open a wizard dialog that displays the wizard.
5. Finally, we check the return code of the dialog to see if the wizard was actually finished or canceled. We notify the success of the wizard using `notifyResult(boolean)`. This will, for example, notify our cheat sheet of success or otherwise so it can mark the step as done or not.
6. Don't forget to list the import statements for the classes you use. The whole implementation of the class is shown below.

Listing 7. The OpenJARExportWizardAction

```

    /**
 *
 */
package devworks.tutorial.cheatsheets.library.actions;

import org.eclipse.jdt.internal.ui.jarpackager.JarPackageWizard;
import org.eclipse.jface.action.Action;
import org.eclipse.jface.dialogs.Dialog;
import org.eclipse.jface.viewers.StructuredSelection;
import org.eclipse.jface.wizard.WizardDialog;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.ui.IWorkbench;
import org.eclipse.ui.PlatformUI;

/**
 * @author philip tiedt, philipp_tiedt@de.ibm.com
 */
public class OpenJARExportWizardAction extends Action {

    public void run() {
        IWorkbench workbench = PlatformUI.getWorkbench();
        Shell shell = workbench.getActiveWorkbenchWindow().getShell();
        JarPackageWizard wizard= new JarPackageWizard();
        wizard.init(workbench,new StructuredSelection());
        WizardDialog dialog= new WizardDialog(shell, wizard);
        dialog.create();
        dialog.open();
        //did the wizard succed ?
        notifyResult(dialog.getReturnCode()==Dialog.OK);
    }
}

```

Finally, you need to register your action with your item in the `context.xml` file.

Listing 8. Adding the action to the cheat sheet

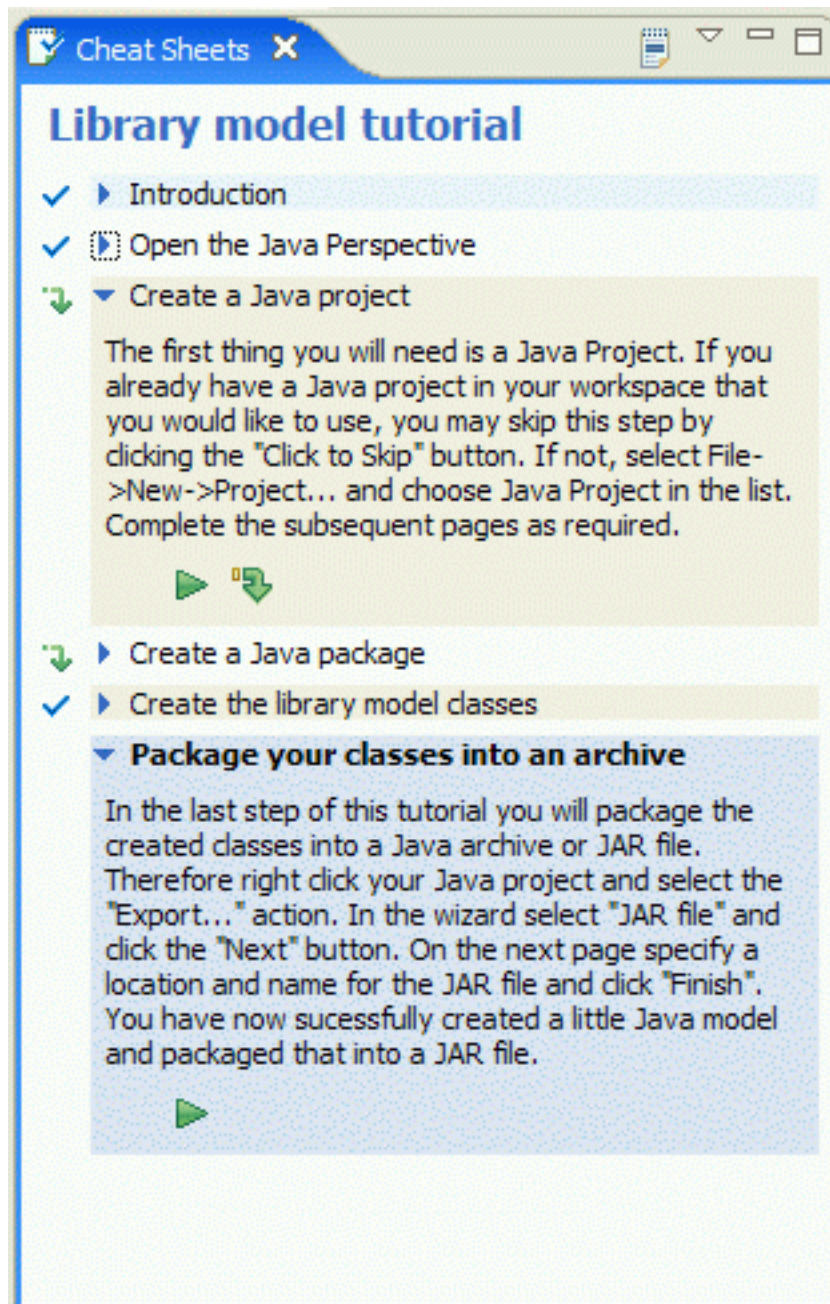
```

        <item title="Package your classes into an archive">
            <action pluginId="devworks.tutorial.cheatsheets.library"
class="devworks.tutorial.cheatsheets.library.actions.OpenJARExportWizardAction"/>
            <description>In the last step of this tutorial you will package the created
classes into a Java archive or JAR file. Therefore right-click your Java
project and select the "Export..." action. In the wizard select "JAR file"
and click the "Next" button. On the next page, specify a location and name
for the JAR file and click "Finish". You have now successfully created a
little Java model and packaged that into a JAR file.
            </description>
        </item>

```

If you run the run-time workbench now, your cheat sheet should look similar to this. Check your actions by clicking the green arrow buttons.

Figure 24. Cheat sheet with actions



We have now automated our cheat sheet. We will now have a look at how to structure the content.

Section 5. Structuring and composing the cheat sheet

You will learn how to define substeps for a step in your tutorial. Also, there are steps that can be repeated or executed on special conditions. You will learn how to use them, as well.

Subitems

Let's start with simple subitems. Suppose you want the user to set up the workbench before starting with creating projects, packages, and classes. We first will create a new step using the `item` element.

Listing 9. Adding a new item

```
<item title="Set up
the Workbench">
  <description>Before we really get
  started,
  let's set up the Workbench.
  </description>
</item>
```

Now we will add three substeps to the task, using the `subitem` element.

Listing 10. Adding subitems

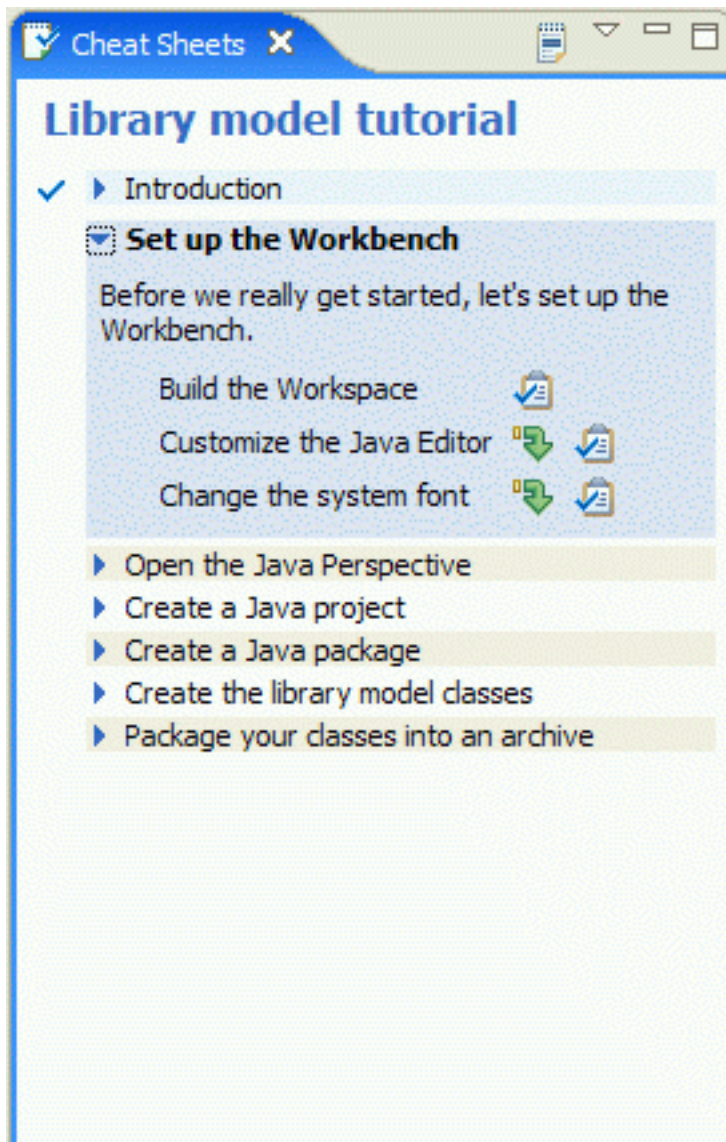
```
<item title="Set up the
Workbench">
  <description>Before we really get started,
  let's set up the Workbench.
  </description>
  <subitem label="Build the Workspace"/>
  <subitem label="Customize the Java Editor"
  skip="true"/>
  <subitem label="Change the system font"
  skip="true"/>
</item>
```

Subitems have a short label similar to a title, but no lengthy description like items. Subitems can be skipped by setting the `skip` attribute to `true`. Unlike items, subitems do not have to be performed in a strict sequence but can be executed in any order.

You can attach an optional action to a subitem to automate its execution. For example, we could add an action that invokes a whole workspace build to the first subitem. As this is similar to adding an action to an item, we will skip this here.

If you run your cheat sheet in the run-time workbench, it should look like this now.

Figure 25. Cheat sheet with subitems



Note that an item is only completed if all subitems are completed or skipped within the given item.

Repeated subitems

Let's have a look at our current item for creating the library classes. For now, we tell the user to create three classes using the New Class Wizard. We could split this up into three substeps, each creating a different one of the classes. Therefore, we would probably add three subitem elements and attach an action to each that creates the class.

Cheat sheets comes up with a better solution here. As the task for the three subitems is similar, we could put it in a loop each time that is then repeated three times. Cheat sheets offers the `repeated-subitem` element for this.

At first, we go to the `item` element that describes the task and change its

description a little bit and add a `repeated-subitem` element.

Listing 11. Adding a repeated subitem

```
<item title="Create the library model classes">
  <description>Now you should be set up for creating your library model. The
  library model consists of three Java classes, a library class, a writer class
  and a book class. In this step you will create the three classes.
  </description>
  <repeated-subitem values="Library, Book, Writer">
    <subitem label="Create the class ${this}."/>
  </repeated-subitem>
</item>
```

The `repeated-subitem` element requires one `subitem` element and the `values` attribute. The `values` attribute provides a list of comma-separated Strings, and the `subitem` element acts as a template for similar steps. You can access the current String value in the loop using the `${this}` variable.

For the sample above, the repeated subitem would expand to something equivalent to:

Listing 12. Expanded repeated subitem

```
<subitem label="Create the class Library."/>
<subitem label="Create the class Book."/>
<subitem label="Create the class Writer."/>
```

To automate the creation of each task, we can add an action to the subitem. The action needs to open the wizard to create the class. It would be even better if we could fill in the type name of the class automatically. In *Implementing a cheat sheet action*, you learned how to implement an action using the `IAction` interface. Additionally, to the `IAction` interface, our new action will also implement the `ICheatSheetAction` interface, so we can pass parameters to the `run()` method.

Use **File > New > Class** to open the New Class Wizard.

Figure 26. Creating the cheat sheet action class

New Java Class

Java Class
Create a new Java class.

Source folder:

Package:


☐ Enclosing type:

Name:

Modifiers: ☒ public ☐ default ☐ private ☐ protected
☐ abstract ☐ final ☐ static

Superclass:

Interfaces:

 org.eclipse.ui.cheatsheets.ICheatSheetAction

Which method stubs would you like to create?
☐ public static void main(String[] args)
☐ Constructors from superclass
☒ Inherited abstract methods

Do you want to add comments as configured in the [properties](#) of the current project?
☒

Fill in the name of the action -- for example, `OpenNewClassWizardAction`. For the superclass, enter `org.eclipse.jface.action.Action` or simply type `Action` and use the auto-completion. Add the `ICheatSheetAction` interface to the class and click **Finish**. The generated code should look similar to Listing 13.

Listing 13. Auto-generated cheat sheet action class

```

*                               / **
*
* /

```

```

package devworks.tutorial.cheatsheets.library.actions;

import org.eclipse.jface.action.Action;
import org.eclipse.ui.cheatsheets.ICheatSheetAction;
import org.eclipse.ui.cheatsheets.ICheatSheetManager;

/**
 * @author philip tiedt, philipp_tiedt@de.ibm.com
 */
public class OpenNewClassWizardAction extends Action implements
    ICheatSheetAction {

    /* (non-Javadoc)
     * @see org.eclipse.ui.cheatsheets.ICheatSheetAction#run(java.lang.String[],
     *      org.eclipse.ui.cheatsheets.ICheatSheetManager)
     */
    public void run(String[] params, ICheatSheetManager manager) {
        //TODO Auto-generated method stub
    }
}

```

We now have to implement the `run(String[] params, ICheatSheetManager manager)` method. As you can see, the method gets passed a `String` array and an `ICheatSheetManager`. We will focus on the `String` array, as it contains the parameters passed to this action via the cheat sheet.

Listing 14. Implementing the action

```

        public void run(String[] params, ICheatSheetManager manager) {
String typeName = null;
if (params!=null && params.length > 0)
    typeName = params[0];
    NewClassCreationWizard wizard = new NewClassCreationWizard();
    WizardDialog dialog = new
    WizardDialog(PlatformUI.getWorkbench().getActiveWorkbenchWindow().
        getShell(), wizard);
    dialog.create();
    if (typeName != null) {
        NewClassWizardPage page = \
        (NewClassWizardPage) wizard.getStartingPage();
        page.setTypeName(typeName, false);
        page.setErrorMessage(null);
    }
    dialog.open();
    //did the wizard succeed ?
    notifyResult(dialog.getReturnCode()==Dialog.OK);
}

```

At first, we check the `String` array that we get passed. If it is not `null` and not empty, we assume the first parameter is the type name. So we copy it to a `String` object.

Then we instantiate the New Class Creation Wizard and a Wizard Dialog to open it. After we create the dialog, we can be sure that the first wizard page was created, as well. The wizard has one page only, which is of type `NewClassWizardPage`, so we get that instance. At the page, we can preset the Type Name. We call `setTypeName()` with the Type Name we got from the `params` array. The second method parameter is a `Boolean` value specifying whether the text field for the type should be editable. We pass `false`, as we want the type name not to be changed by the user.

As we set the type name, the page checks if all values in the whole page are correct. We only specified a type name, but not a package or a folder to create the type in, so the page would be invalid, and the user would be presented with an error message from start on without having entered anything. So we call `setErrorMessage(null)`, which will suppress the error message and show the page description.

Finally, we open the dialog and verify that it was finished successfully. Now we have to hook this action into our cheat sheet content.

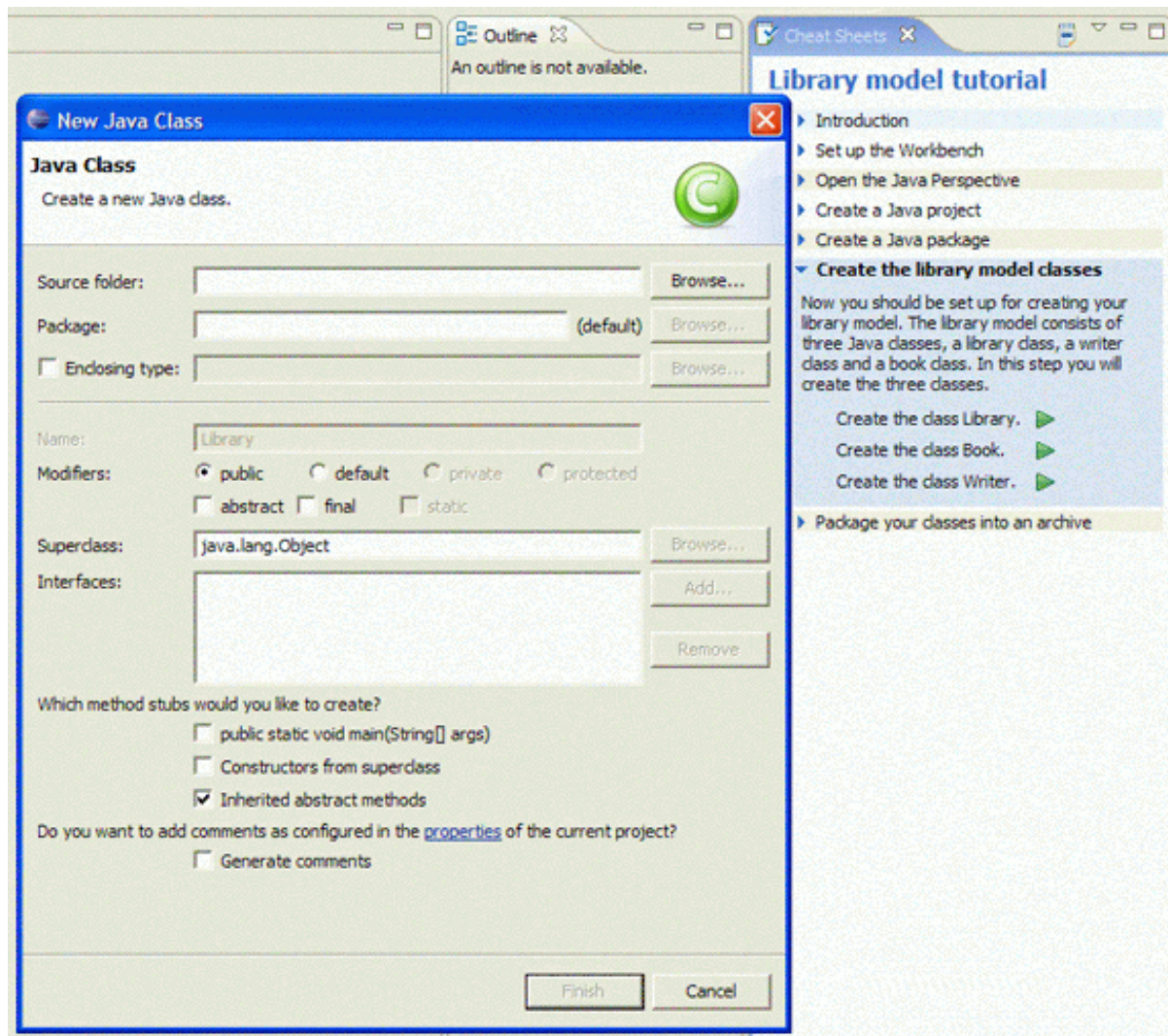
Listing 15. Hooking in the action

```
<repeated-subitem values="Library,Book,Writer">
  <subitem label="Create the class ${this}.">
    <action pluginId="devworks.tutorial.cheatsheets.library"
      class="devworks.tutorial.cheatsheets.library.actions.
        OpenNewClassWizardAction" param1="${this}"/>
    </subitem>
  </repeated-subitem>
```

We add an `action` element to the subitem, and specify `pluginId` and `class`, as usual. Additionally, we pass one parameter. Therefore, we use the `param1` attribute. The value is the current class name carried by the `${this}` variable.

Run the cheat sheet in a run-time workbench, and the result should look like Figure 27.

Figure 27. Cheat sheet with repeated subitem



Note that the Name field is not editable. You have now structured your content with different types of subitems.

Conditions

There are more possibilities to structure your content and make it interactive. Cheat sheets provide a mechanism to show subitems depending on a special condition, or to perform an action if a special condition is matched. Conditions can be based on so-called cheat sheet variables. Advanced users should refer to the DTD of the cheat sheet content and the following elements:

1. conditional-subitem
2. perform-when

This tutorial will not explain those elements.

Section 6. Linking the tutorial with the Eclipse help

Usually, cheat sheets do not offer the space to write extensive instruction or information about a step in your tutorial. Details about a topic reside in the Eclipse help. With cheat sheets, you can link the different steps of your tutorial with help topics.

Help is optional and can be displayed in two ways: as a help document or as context-based help. Help only applies to the introduction and the main steps of your tutorial (the `intro` and the `item` elements).

Linking help documents

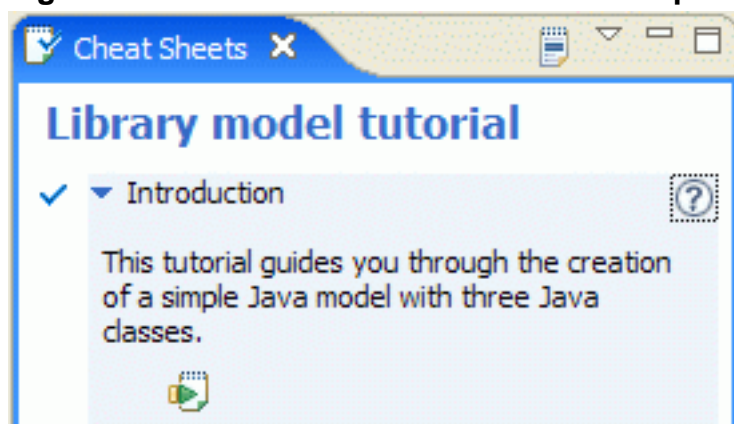
Let's start with our tutorial introduction and provide the user with a little help about cheat sheets. We want to open the Eclipse help showing the help document for cheat sheets. So we simply add a `href` attribute to our `intro` element.

Listing 16. Linking a help document

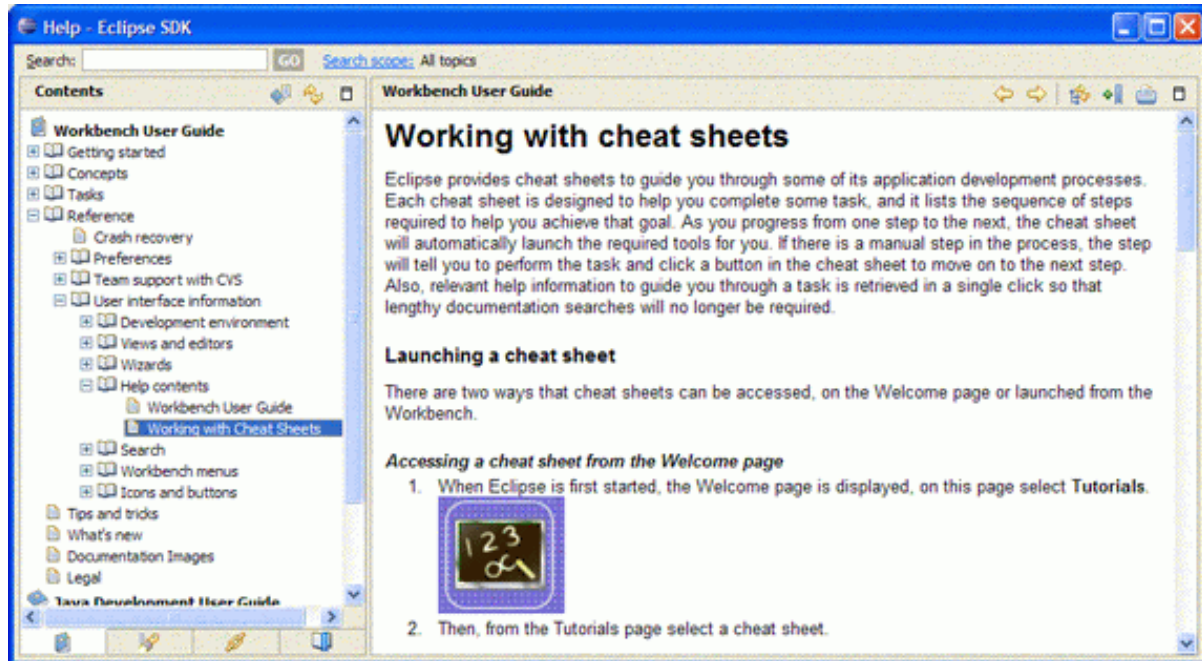
```
<intro href="/org.eclipse.platform.\
doc.user/reference/ref-cheatsheets.htm">
  <description>This tutorial guides you through \
the creation of a simple Java
model with three Java classes.
  </description>
</intro>
```

The value of the attribute must be a link to a help document in the Eclipse help. In our case, it's the cheat sheets help reference. Saving the content and running the tutorial should show a little question mark on the upper-right side of the introduction.

Figure 28. Cheat sheet introduction with help



Clicking the help icon should bring up the Eclipse help showing the expected help document.

Figure 29. Cheat sheet help document

Linking with dynamic help

Suppose you want to provide context-sensitive help for example help associated with opening the Java perspective. This can be done linking dynamic help to your introduction or a step. This help is shown in the Eclipse help view or as a pop-up, depending on your help preference settings.

Let's add context-based help to our step that opens the Java perspective. Therefore, we add the `contextId` attribute to the `item` element.

```
<item title="Open the Java Perspective"
contextId="org.eclipse.jdt.ui.open_java_perspective_action">
```

The value of the attribute must be a fully qualified help context ID. In our case, we use the context ID for the action that opens the Java perspective.

Saving the content and running the tutorial should now result in a second help icon at the step that opens the Java perspective. Clicking the icon should bring up a small pop-up or the help view.

Figure 30. Cheat sheet with help pop-up

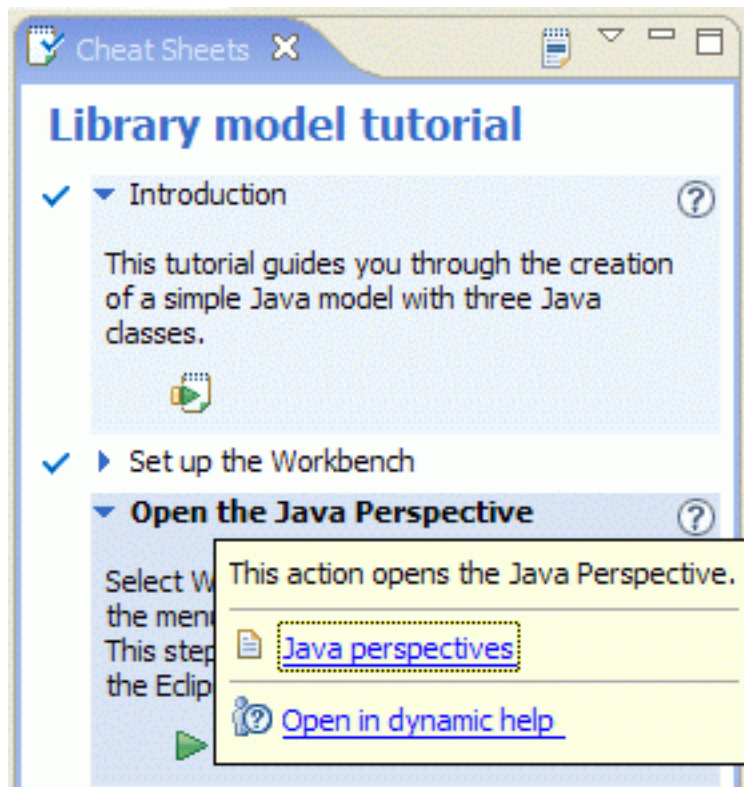
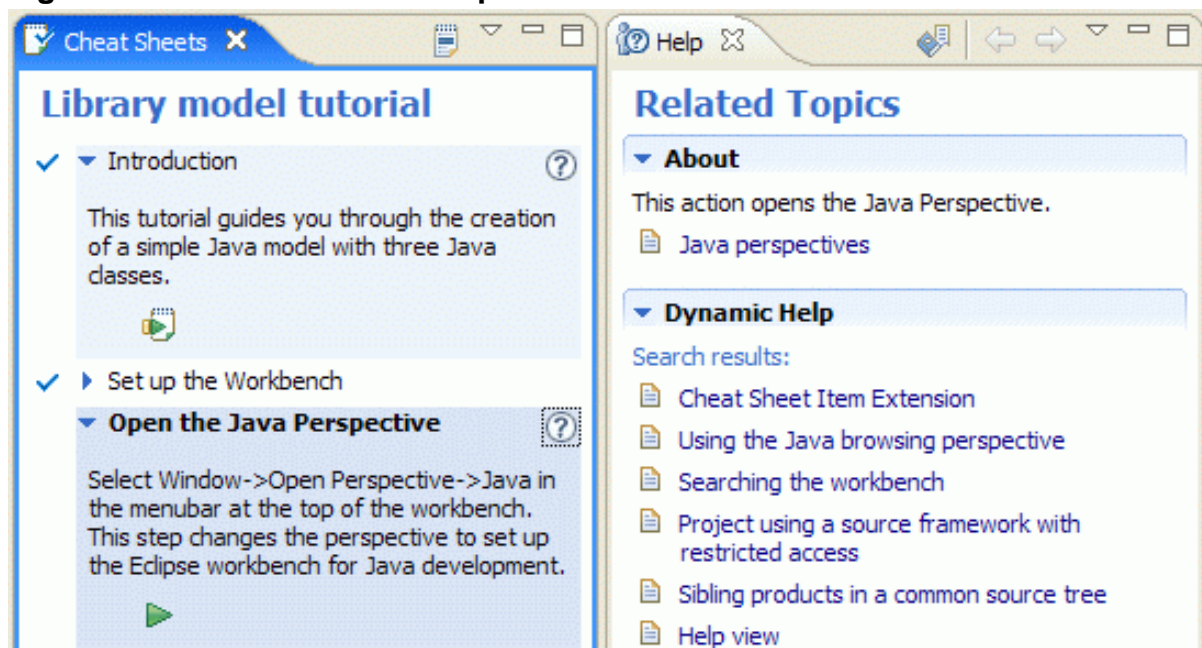


Figure 31. Cheat sheet with help view



Note that only one type of help should be supplied for a step or introduction. If specifying a `contextId` and `href` attribute the `href` attribute will be ignored.

Section 7. Summary

At this point, you should have a simple Eclipse cheat sheet running. You learned how to create a simple cheat sheet plug-in, contribute to the cheat sheet extension point and provide the content for a tutorial. The tutorial showed how to automate your cheat sheet by implementing or reusing cheat sheet actions. Finally, you learned how to structure and compose your tutorial and how to link different help mechanisms with it.

The complete sample plug-in, including all source, can be downloaded from [Downloads](#).

Downloads

Description	Name	Size	Download method
Complete sources of the sample plug-in	os-cheatsheets.zip	9KB	HTTP

[Information about download methods](#)

Resources

Learn

- For an update to this tutorial, see the developerWorks article "[Building cheat sheets in Eclipse V3.2.](#)"
- Read the "[Getting started with the Eclipse Platform](#)" to get an overview of Eclipse, including its origin and architecture, and details on how to install it and its plug-ins.
- Learn more about cheat sheets in the [Cheat sheet help](#).
- Visit developerWorks' [Eclipse project resources](#) to learn more about Eclipse.
- Check out [Eclipse projects](#), which lists a bunch of fast-growing and interesting Eclipse subprojects.
- Find out about [Contributing a cheat sheet](#).
- The specification of the cheat sheet content can be found in the [Cheat sheet content DTD](#).
- The [Cheat sheet extension point description](#) provides basic information about how to contribute to the cheat sheet extension point.
- Check out the "[Recommended Eclipse reading list](#)."
- Browse all the [Eclipse content](#) on developerWorks.
- New to Eclipse? Read the developerWorks article "[Get started with Eclipse Platform](#)" to learn its origin and architecture, and how to extend Eclipse with plug-ins.
- Expand your Eclipse skills by checking out IBM developerWorks' [Eclipse project resources](#).
- To listen to interesting interviews and discussions for software developers, check out [developerWorks podcasts](#).
- For an introduction to the Eclipse platform, see "[Getting started with the Eclipse Platform](#)."
- Stay current with developerWorks' [Technical events and webcasts](#).
- Watch and learn about IBM and open source technologies and product functions with the no-cost [developerWorks On demand demos](#).
- Check out upcoming conferences, trade shows, webcasts, and other [Events](#) around the world that are of interest to IBM open source developers.
- Visit the developerWorks [Open source zone](#) for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products.

Get products and technologies

- Download [the latest version of Eclipse](#) from [Eclipse.org](#).
- Check out the latest [Eclipse technology downloads](#) at IBM [alphaWorks](#).
- Download [Eclipse Platform and other projects](#) from the Eclipse Foundation.
- Download [IBM product evaluation versions](#), and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
- Innovate your next open source development project with [IBM trial software](#), available for download or on DVD.

Discuss

- The [Eclipse Platform newsgroups](#) should be your first stop to discuss questions regarding Eclipse. (Selecting this will launch your default Usenet news reader application and open eclipse.platform.)
- The [Eclipse newsgroups](#) has many resources for people interested in using and extending Eclipse.
- Participate in [developerWorks blogs](#) and get involved in the developerWorks community.

About the author

Philipp Tiedt

Philipp Tiedt is a software engineer in IBM's Development Lab in Boeblingen Germany. He holds a bachelor's degree in computer science from the Open University. Before joining IBM Germany in 2004, he completed his bachelor's study at IBM's T.J. Watson Research Center in Hawthorne, N.Y. His areas of interest are Eclipse, user interface design, Java technology, and service-oriented architecture.