Oracle® Fusion Middleware

Developing Extensions for Oracle JDeveloper

12*c* (12.2.1.4.0)
F16403-01
September 2019

Oracle Fusion Middleware Developing Extensions for Oracle JDeveloper, 12*c* (12.2.1.4.0)

F16403-01

Primary Author: Walter Egan

Contents

2 Developing Extensions in Oracle JDeveloper

Preface

Welcome to the *Developing Extensions for Oracle JDeveloper*.

Audience

This document is intended for developers that develop or use Oracle JDeveloper extensions.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Documents

- *Installing Oracle JDeveloper*
- *Developing Applications with Oracle JDeveloper*
- *Developing Fusion Web Applications with Oracle Application Development Framework*
- *Developing Web User Interfaces with Oracle ADF Faces*
- *Oracle JDeveloper 12c Online Help*
- *Oracle JDeveloper 12c Release Notes*, link included with your Oracle JDeveloper installation, and on Oracle Technology Network

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.

Convention	Meaning
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

What's New in This Guide

The following topics introduce the new and changed features of Oracle JDeveloper and other significant changes that are described in this guide, and provides pointers to additional information. This document is the new edition of *Developing Extensions for Oracle JDeveloper*.

New and Changed Features for 12*c* (12.2.1.4)

Oracle JDeveloper 12*c* (12.2.1.4) includes the following new and changed features for this document:

- No major feature changes in this release.

1

Introduction to Developing Oracle JDeveloper Extensions

Developing extensions to the JDeveloper integrated development environment (IDE) lets you add additional features to JDeveloper.
The following sections provide an overview of using and developing JDeveloper extensions:

About Developing Oracle JDeveloper Extensions

Oracle JDeveloper **Extensions** implement the most of the functionality in the JDeveloper IDE. You can add existing extensions into JDeveloper, or you can build on JDeveloper's native functionality by creating extensions to provide new features tailored to your organization's development requirement.

A few of the functionalities include:

- To streamline your work flow.
- To use a third-party team development tool.
- To use software packages unique to your team.
- To help implement development standards and best practices with audit extensions.

There are a wide variety of extensions readily available for you to download and use. Some of these have been developed by Oracle, and some by third parties. These provide new features to JDeveloper and integrate it with other applications that are used by customers in their own development environments. For example, many of the version control and team working systems available in JDeveloper are written and distributed as extensions. The open architecture of JDeveloper means that you can download and install them from **Check for Updates** on the JDeveloper **Help** menu. Alternatively, you can install extensions:

- From the Oracle JDeveloper Extensions page on the Oracle Technology Network (OTN) at `http://www.oracle.com/technetwork/developer-tools/jdev/index-099997.html`.

- From a properly constructed ZIP file with a manifest that is in a location accessible from your machine.

> **Note:**
>
> The recommended way of installing extensions to JDeveloper is to use **Check for Updates** on the JDeveloper **Help** menu because only extension versions that match the version of JDeveloper you are using will be available.

This same open architecture also makes it possible for you to write your own extensions if you have specific needs or you would like to integrate JDeveloper with some external process or tool that your development team uses. You can add menu items, create context menus, integrate features into the JDeveloper toolbar, create dockable windows that provide a view into your data objects, and more.

> **Note:**
>
> The way that JDeveloper handles extensions changed in Oracle JDeveloper 11.1.2.*nn*. For information about extensions for earlier versions of JDeveloper, see the online help in your version of JDeveloper.
>
> Oracle extensions for earlier versions of JDeveloper, including the Extension SDK, and third-party extensions for earlier versions of JDeveloper are available from the Oracle JDeveloper Extensions page at `http://www.oracle.com/technetwork/developer-tools/jdev/index-099997.html`.
>
> Information about migrating extensions written for earlier versions of JDeveloper is in Migrating Extensions from Previous Releases.

You can use JDeveloper's native features to develop your own extensions. See Developing Extensions in JDeveloper .

For more advanced development, the JDeveloper Extension Software Development Kit (Extension SDK) has been developed by the JDeveloper development team. It includes a collection of projects containing sample code, and the javadoc-generated documentation for the Extension SDK API which provides reference for classes and other features used in extension development. It is available from **Check for Updates** on the JDeveloper **Help** menu, and it contains much more complete documentation and examples. If you are planning to do serious extension development, either to provide wide support to an internal team or to develop an extension as part of a product or third-party application to pass on to your own customers, it is highly recommended that you download the Extension SDK. See Developing with the Extension SDK .

You can use the Oracle JDeveloper page on the OTN forum to ask a question about developing an extension, contribute to a discussion, or interact with other users. The Oracle JDeveloper page on the OTN forum is located at `https://forums.oracle.com/forums/forum.jspa?forumID=83`.

In contrast to the Extension SDK, JDeveloper also allows you to use external tools without any coding:

- You can invoke command line interfaces
- You can pass parameters
- You can add menus to JDeveloper

For information about developing extensions for JDeveloper, see Working with Extensions in *Developing Applications with Oracle JDeveloper*.

For information about External Tools, see Adding External Tools to JDeveloper.

Developing Extensions with OSGi

JDeveloper Extensions conform to the Open Services Gateway Initiative (OSGi).

The Open Services Gateway Initiative (OSGi) is a specification for building service platforms running on top of a Java Runtime environment. JDeveloper is built as a set of extensions which conform to OSGi, and any extensions that you write must also conform to this standard.

For more information about OSGi, see the OSGi Service Platform Core Specification Release 4, Version 4.2 which is available from `http://www.osgi.org`.

The OSGi Framework can be divided into two main elements, both described below:

- Service/Component Platform, that is the service providers, service requesters, and service registry.
- Deployment Infrastructure, that is the service bundles which contain implementation classes, resources, bundle metadata, and manifest file.

Service/Component Platform

The Service/Component Platform allows an OSGi implementation to activate, de-activate, update and de-install existing services and to install new services dynamically.

The Service/Component platform supports the interaction between 3 actors:

1. Service providers, which provide specific services and publish service descriptions.
2. Service requesters, which discover services and bind to the service providers.
3. Service registry, which manages the publication and discovery of services based on the registered service descriptions.

An OSGi service is a Java class or interface (service interface), along with a variable number of attributes (service properties), which are name and value pairs. Service properties allow differentiation between service providers which use the same service interface.

The service registry allows services to be discovered by service requesters using LDAP.

Service requesters can receive events signalling changes, for example, publication or retrieval of a service in the service registry, using notification mechanisms.

Deployment Infrastructure

Services are packaged into service bundles which contain service implementation classes and resources, along with the manifest file, `manifest.mf`, which contains the bundle metadata. The manifest file contains information such as the service name, version and dependencies.

Service providers and service requesters are part of a service bundle that is both a logical and a physical entity. The bundle is responsible for run-time service dependency management activities which include:

- Publication
- Discovery
- Binding
- Adapting to changes resulting from the dynamic availability—the arrival or departure—of services that are bound to the bundle.

When you create an extension in JDeveloper, it is composed into an OSGi service bundle.

How JDeveloper Extensions Work

Discover about the concepts used in JDeveloper extension development.

Only those extensions load during the JDeveloper startup that are required for the user to enhance the startup performance and overall memory usage. Some extensions related terms and concepts are described below:

- JDeveloper users choose a role in which to work, and the role determines which JDeveloper extensions are loaded because roles list the set of possible extensions for that role. For example, the JDeveloper default role lists all extensions in the Studio Edition.
- Initialization is an operation that involves calling the extension initialization code `Addin.initialize()`. Extension initialization gives the extension an opportunity to initialize whatever it needs to function properly at the moment the end user is about to exercise that extension's functionality.

  The initialization list is a list of extensions previously initialized. These same extensions are initialized when the user reopens a project recording this list.
- Trigger hooks are a set of declarative integration hooks that provide the mechanism to trigger extension initialization. They are defined in the `<trigger-hooks>` section of the extension manifest `extension.xml`, and they are processed when JDeveloper starts, even though your extension may not have been initialized.

  Any information the JDeveloper IDE needs about your extension at startup, such as the gallery items it contributes, or the Java libraries it defines, or the node recognizers it defines, must be provided in trigger hooks.

  By contrast, the `<hooks>` section of the extension manifest is processed later when your extension is initialized.
- Registration is an operation that involves reading and caching the trigger hooks section of the extension manifest.

- Lazy initialization is the term for extensions not being initialized until their trigger hook is activated.
- Extension versioning is managed by OSGi. Two or more versions of the same extension can be loaded at the same time, and if an extension is activated all dependent extensions are also loaded. OSGi handles the situation of an extension depending on a certain version or range of versions.
- Extensions are class loaded in their own classloader. The extension lists the set of packages that it exports to other extensions. The OSGi bundle manifest file defines the Java packages that are made available to other bundles. Access to the restricted classes is not possible, even using reflection, since OSGi protects them by means of the extension's Java classloader.
- The OSGi service infrastructure dynamically manages service providers and requesters.
- Extensions can have dependencies on other extensions. When an extension is loaded, any other extension marked as a dependency in the OSGi bundle manifest is also loaded. You may have already experienced this in JDeveloper, when you have to click **Load Feature**, for example, in the Application Properties, Project Properties, or Preferences dialog to start up a feature that you have not accessed before, and have to wait while the relevant extensions load.
- JDeveloper extensions conform to the JSR 198 specification, which provides a standard extension API for IDEs. See`http://jcp.org/en/jsr/detail?id=198`.

How Extensions are Processed

There are two distinct phases to the way that JDeveloper processes extensions:

- Extension registration. This happens as JDeveloper starts up. It involves registering all extensions available for the selected role. No extension code is executed except for those extensions required to start JDeveloper.
- Extension initialization. This happens at any time the user accesses a functional area registered by an extension during the extension registration phase. Therefore you can see that an extension can be initialized at any time when JDeveloper is being used.

Registering Extensions and Using Trigger Hooks

Extension registration is the JDeveloper IDE startup phase during which the extension registry processes the `<trigger hooks>` section of the extension manifest `extension.xml` from all extensions available. During this processing no extension-specific code is executed with the exception of translatable resources such as Java resource bundles.

Extensions can define extension specific trigger hooks using the `<trigger-hook>` integration hook. If they do so, they must use the `oracle.ide.extension.HashStructureHook` handler since the IDE will not execute extension specific code unless that extension is initialized. The extension specific trigger hook data will be available for retrieval in the Extension Registry. The extension owning the trigger hook can retrieve that data once that extension is initialized. For more information, see How to Register a <trigger-hook-handler>.

By contrast, other items that do not belong in the `<trigger hooks>` section of the extension manifest `extension.xml` are initialized declaratively in the initialization phase of the extension.

There are two types of trigger hooks, system and custom, and every extension is initialized by:

- Being called from a system trigger hook
- Being called from a custom trigger hook
- Being explicitly initialized by another, already initialized, extension

The types of trigger hook defined by the JDeveloper IDE are:

- **Menus:** Only menus that qualify as an entry point to an extension should be menu trigger hooks, and menus that are used to support an already initialized extension should not be trigger hooks. For example, in the JDeveloper View menu, the Breakpoints menu item is a trigger hook since a user will want to set breakpoints before running the debugger, but the debugger subMenu in the View menu is not a trigger hook since everything in it is only used once the extension is initialized. From this you can see that menu items for uninitialized extensions are not visible in JDeveloper.
- **Context Menus:** Extensions can create custom trigger hooks that use context menus as the trigger. There are no system-level context menu trigger hooks.
- **IDE Actions and Controllers:** Only actions and controllers to support menu trigger hooks should be registered. All other actions and controllers should execute in the initialization phase. Any controller registered for a trigger hook menu will not be able to run code to determine if their menu is disabled or not; it must use declarative controller logic to specify when the menu is shown.
- **Gallery Items:** Extensions can list items in the JDeveloper New Gallery using the gallery item trigger hook. When the user opens a dialog or wizard from the New Gallery it causes the extensions for that item to initialize, and any extension marked as a dependency in the OSGi bundle manifest of this extension is also loaded.
- **Technology Scopes:** When a technology scope is added to a project, the extension that registered that technology scope is initialized, along with any other extension marked as a dependency in the OSGi bundle manifest.
- **NodeFactory Recognizers:** This allows extensions to identify the nodes they own. When a node becomes visible in the Applications window, if the extension for that node is registered, it will be initialized. NodeFactory recognizers also handle things like dragging a file from the desktop into JDeveloper.
- **IDE Preferences/Settings, Application Preferences/Settings, Project Preferences/Settings:** Uninitialized extensions will only show as a category listing in the Preferences dialog (available from the Tools menu). It is not until a user clicks on the category that they will be asked if they want to initialize the extension at that point.
- **Singleton Registration:** Singleton classes register as a trigger hook. When a client requests a service from a singleton, the framework will check to see if that singleton's extension has been initialized. If it has not, it will call for the extension to be initialized. An example is the Log Manager.
- **Annotations:** Extensions can register themselves as needing an annotation trigger hook. They list out the annotation class names and when the user types

one of those annotations, we will initialize that extension at that time. The annotation trigger hooks are used during Applications window node expansion to determine icon overlays. When a parent node is expanded, the framework will look for any annotations that are registered and if found, will supply the appropriate icon for that node. The extension will not be initialized in the case where the node icon overlay is applied.

- **Application and project migrators:** These migrators must be defined declaratively. Migrators specify the current versions supported. If the application or project file lists an older version of the migrator, or no version at all, JDeveloper triggers the initialization of the extension in order to perform the migration of that extension's data.
- **Library and Tag Library:** When a library or tag library is added to a project, the associated extension will trigger to initialize. In addition to this, when a project is opened (not loaded), JDeveloper looks for libraries of that project and automatically loads the extensions that are associated with that library or tag library if they are not already loaded.
- **Custom Trigger Hooks:** An extension can define its own trigger hook. This is useful for the situation that an extension wants to allow clients to plug into it. An extension defines its trigger hook in the same manner that it defines a regular hook. Client extensions can then add a trigger hook registration (either system or custom) into their extension manifest. When registering a trigger hook, the extension specifies which extension(s) need to be loaded in order for the trigger to take effect. These are called hook dependencies. JDeveloper ensures that this hook is only processed when all the extensions listed as hook dependencies are initialized.

How Lazy Initialization Works

To avoid initializing all extensions every time JDeveloper starts, lazy extension initialization is used.

- During JDeveloper startup, the IDE only executes `Addin.initialize()` code if the extension that owns the Addin has been previously used by the end user in the currently opened application or project. For more information, see Understanding Node Recognizers.
- Each extension identifies a type associated with their trigger hook so that this hook:
  - Can only be triggered when there is an application workspace open.
  - Can only be triggered when there is a project open.
  - Can be initialized at any time in the way that, for example, the HTTP Analyzer can be.
- As users work on an application in JDeveloper, the user's usage of extension functionality is recorded in a project-specific extension initialization list. That way, when the user stops working on a project by exiting JDeveloper and re-starting at a later time, JDeveloper only initializes the extensions listed in the project's initialization list.

  For extensions that do not require a project, the information is stored in the IDE preferences for the user.

When you are developing your extension you need to consider how the integration points deal with lazy initialization. For example:

- Extensions cannot assume that integration points are static; extensions must be ready at any time during their life cycle to process and integrate new data associated with their integration point.

  For example, extension A has an API or a custom extension hook that allows other extensions to register some listener class. Extension A cannot assume that all such listeners are registered by the time extension A is initialized; other extensions that depend on A can be lazy initialized later in the life cycle of ABC. Therefore, extension A must update its event firing code associated with these listeners to deal with this effect of extension lazy loading. The way this is done is described in the next point.

- Extensions should use `oracle.ide.extension.HashStructureHook` to implement custom integration hooks. This hook processing class has a property change listener mechanism that lets clients know when new additional data is registered. Your extension should listen for these property change events.

- You should avoid having a deep dependency tree since this will cause the initialization of all the extensions it depends on, which can diminish the performance gains of lazy extension loading. If you find that your extension has a deep dependency tree, you need to refactor the extensions to reduce the number of extensions it depends on. For more information, see Understanding Large Extensions.

Migrating Extensions from Previous Releases

Extensions developed for 11g and earlier versions of JDeveloper have to be updated to run in in JDeveloper 12c.

If you have an extension that you developed for an earlier version of JDeveloper, you will need to change it to run in the current release of JDeveloper. Section How Extensions Work describes how extensions now work in JDeveloper, so you should start by being familiar with the concepts described there.

The coding areas you need to consider are:

- The `Addin.initialize` method is no longer called at IDE startup. You must initialize your extension using a trigger hook.Trigger hooks are defined in the extension manifest extension.xml, and you can use one of the trigger hooks types defined by the JDeveloper IDE, or create a custom trigger hook. For more information, see Defining and Using Trigger Hooks.

- Almost all `<hooks>` properties in the extension manifest `extension.xml` should now be placed in `<trigger hooks>`.

- In earlier versions of JDeveloper, role files were exclusionary, meaning that by default all extensions were loaded and the role file told JDeveloper what not to load. Now role files are inclusive; only extensions listed in a role are able to be loaded when their trigger hook is activated.

- All extensions have separate classloaders. As a result, code that calls package-protected methods from classes in the same package but different extensions will not work.

- If your extension has custom integration points, these must be changed to handle on-demand extension initialization. For more information, see How Lazy Initialization Works.

- Code that uses reflection to instantiate classes in an extension that it does not have explicit dependencies on will not work.

There are a number of use cases in Use Cases for Developing Extensions that describe situations you may encounter when you are converting your extensions to use declarative trigger hooks support lazy initialization.

Getting Started With Extension Development

Begin developing an extension by creating an application.

Whether you use the native features of JDeveloper to create an extension, described in Developing Extensions in , or download the Extension SDK to develop an extension, described in Developing with the Extension SDK , you start by creating an application in JDeveloper along with one or more projects configured for extension development.

How to Create an Application and Project for Extension Development

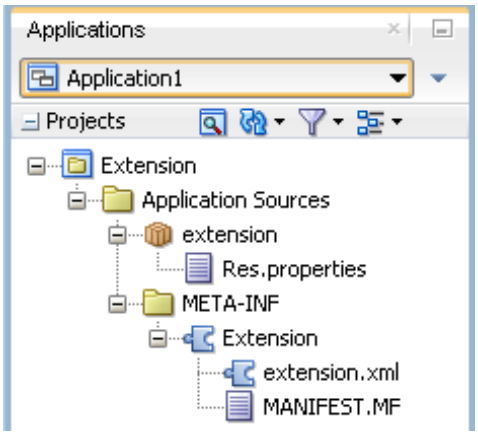
An application is the control structure for one or more projects. A project is a logical container for a set of files that defines a program or part of a program. See "About Working with Applications and Projects" in *Developing Applications with Oracle JDeveloper*.

To create an application and project:

1. Open the New Gallery by choosing **File > New**.
2. In the New Gallery, in the Categories tree, under **General** select **Applications**.
3. In the Items list, double-click **Extension Application**. The Create Extension Application wizard opens. For help with the wizard, press F1 or click **Help**.
4. Enter a name for the application and choose a location and an application package prefix and click **Next**. The project, which is configured for extension development, has a default name of `Extension`. If necessary, change the project name, then click **Next**.
5. On the Configure Java Settings page you can set the default package, the Java source path and the output directory. Make the changes you want, then click **Next**.
6. On the Configure Extension Settings page you can set options for the extension.

The extension project is created in the Applications window, and contains the elements shown in Figure 1-1.

Figure 1-1 Extension Project in the Applications Window



The extension project is created, along with a copy of the extension manifest `extension.xml` and the OSGi manifest `manifest.mf`. The extension manifest is opened in the overview editor.

If for some reason you do not want to create an Extension application directly, you can create a Custom application and select Extension Development as the project features on the Project page of the wizard.

How to Develop for a Different JDeveloper Version

If you are developing an extension for a different version of JDeveloper, you choose the platform when you create the Extension application.

> **Note:**
>
> You can only develop an extension for more than one release of JDeveloper when the versions are both minor releases of the same major release. For example, JDeveloper version 11.1.2.*nn* can work with other 11.1.2.*nn* versions. If you are using 12.1.2.*nn* then you can only develop extensions that will work with other 12.1.2.*nn* releases.
>
> The way that JDeveloper extensions are written changed in JDeveloper 11g release 11.1.2.*nn*. To develop extensions for earlier versions of JDeveloper, see the documentation in that version of JDeveloper.

To choose the JDeveloper version the extension is for:

1. Create an extension application.
2. On the Configure Extension Settings page, choose a different Target Platform. If necessary, click the Manage Extension Platforms icon and choose the platform in the Manage Extension Platforms dialog. For more help, press F1 or click **Help** in the dialog.

Next Steps

Once you have created an application and project you can begin to develop the extension:

- To continue by using the JDeveloper IDE to develop, for example, by creating an extension `Addin`, see Developing Extensions in .
- To use the samples in the Extension SDK to create your extension, or to use the Extension SDK API, see Developing with the Extension SDK .

Working with the Extension Manifest

Learn about the extension manifest for JDeveloper extensions.

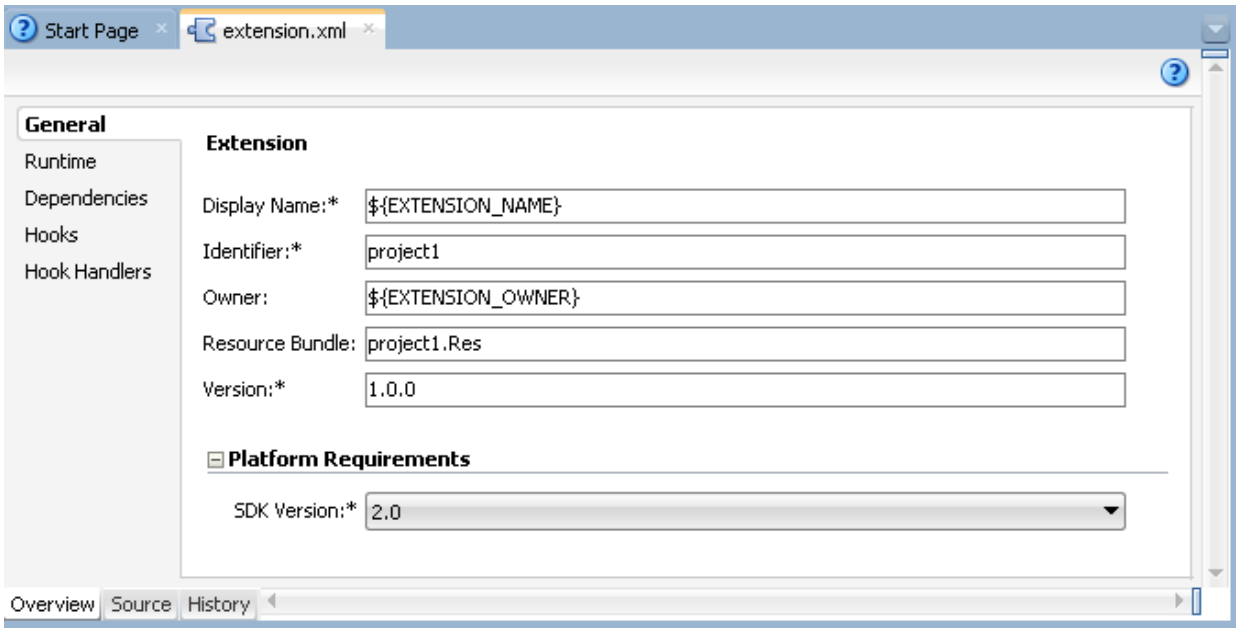
The extension manifest, `extension.xml`, controls many aspects of the extension. Before you can deploy an extension, you must complete `extension.xml` to, for example, register trigger hooks. There can only be one `extension.xml` per project, and it must always be in a directory called `META-INF`.

The extension manifest conforms to the JSR-198 specification, which is available at `http://jcp.org/aboutJava/communityprocess/final/jsr198/index.html`.

Hooks and trigger-hooks are set in the relevant sections of the extension manifest. The `<hooks>` section of extension.xml is processed when the extension is initialized. The `<trigger-hooks>` section is processed when JDeveloper starts up.

JDeveloper has a dedicated overview editor for `extension.xml`, illustrated in Figure 1-2.

Figure 1-2 extension.xml in Overview Editor



You edit `extension.xml` using the overview editor, where you enter information such as details about dependencies into fields, and choose from lists of available objects, and where you can use the Structure window and Property Inspector. To edit information in `extension.xml` that is not available in the overview editor, you can either use the Structure window or work in the `extension.xml` source, which you access by clicking the Source tab.

For more information about dependencies, see Understanding Dependencies.

The template code for `extension.xml` created by creating an extension project contains placeholders for the main elements, shown in the following example:

```
<extension id="extension" version="1.0.0" esdk-version="2.0" rsbundle-
class="extension.Res"
          xmlns="http://jcp.org/jsr/198/extension-manifest">
  <name>${EXTENSION_NAME}</name>
  <owner>${EXTENSION_OWNER}</owner>
  <trigger-hooks xmlns="http://xmlns.oracle.com/ide/extension">
    <!-- TODO Declare triggering functionality provided by extension: extension -->
    <triggers/>
  </trigger-hooks>
  <hooks>
    <!-- TODO Declare functionality provided by extension: extension -->
  </hooks>
</extension>
```

Editing the Extension Manifest in the Overview Editor

The overview editor for `extension.xml` updates the extension manifest `extension.xml` and the OSGI manifest `manifest.mf` declaratively.

On the General tab of the overview editor for `extension.xml` you enter information about the extension such as the extension name, the SDK version, and specify features such as an icon and copyright information. This information populates the `feature-hook` and `platform-info` elements.

The Runtime tab updates `manifest.mf` and allows you to specify runtime values.

The Dependencies tab is where you can specify that this extension is part of another extension, or alternatively specify that this extension is parent to another extension. This populates the `dependencies` element. The Hooks tab is where you details of additional extensions to contribute functionality to your extension. For more information, see Understanding Dependencies.

Use the Hooks tab to specify how the extension binds to the IDE.

The Hook Handlers tab allows you to define trigger hooks in your extension that can be used as integration points by other extensions.

Detailed help, which describes the content of each field, is available by pressing F1 from any tab in the overview editor.

Editing the Extension Manifest in the Source Editor

You edit the extension manifest in the source editor using the XML editor, a specialized schema-driven editor which includes a number of editing features including Code Insight and XML validation. For more information, see "Using the XML Editors" in *Developing Applications with Oracle JDeveloper*.

Working with the OSGi Manifest

Learn how to use the OSGi manifest when creating JDeveloper extensions.

The OSGi manifest, `MANIFEST.MF`, lists the packages that the extension bundle exports. There can only be one `MANIFEST.MF` per project, and it must always be in a directory called `META-INF`.

Although you can edit the OSGi manifest in the source editor, it is preferable to edit it declaratively in the overview editor. For more information, see Editing the Extension Manifest in the Overview Editor.

When a new extension project is created, a default `MANIFEST.MF` is created, and the initial content is shown in the following example:

```
Bundle-ManifestVersion: 2.0
Bundle-Version: 1.0.0
Bundle-SymbolicName: extension
Bundle-ClassPath: .
Require-Bundle: oracle.ide
```

For detailed information about the content of the OSGi manifest, see `http://www.osgi.org`.

As part of the packaging process for an extension, JDeveloper updates `manifest.mf` to provide:

- **Require-Bundle:** Classes from another OSGi bundle that are required by your extension are listed. Comes from the `require-bundles` elements in the extension manifest.
- **Export-Package:** When you want to make Java packages in your extension available to other extensions, these packages are listed.
- **Bundle-ClassPath:** If your extension needs to reference classes from a JAR which is not an OSGi bundle it is listed. Comes from `<dependencies>` elements in the extension manifest.

These values are used in the OSGi Bundle Profile dialog. For more information, see How to Create the Deployment Profile.

Understanding Dependencies

Extensions can depend on other extensions or JAR files. When you are developing extensions for JDeveloper, you need to understand the dependencies upon the extension, as well as the extension tree. You need to know the dependencies between your extension and other extensions, that is whether:

- Your extension is part of another extension
- One or more extensions are part of your extension

You also need to know the libraries and JAR files that need to be added. For example, if you add a dependency on `oracle.jdeveloper.maven.jar`, you must also add dependency libraries for those JAR files delivered with the external Maven module.

Once you have worked this out, you set dependencies in the extension manifest, and ensure that any libraries and additional JAR files are part of the extension bundle.

How to Set Dependencies in the Extension Manifest

When you create an extension project, the extension manifest is created and opened in the overview editor. For more information, see Working with the Extension Manifest.

The order in which extensions are loaded depends on the entries in the extension manifest.

To set dependencies in the Extension Manifest:

1. If necessary, open `extension.xml` in the overview editor by double-clicking on `extension.xml` under `META-INF` in the Applications window. Select the Dependencies tab.
2. To specify that other extensions are dependent on this extension, use the Imported Packages section.

   For more help at any time, press F1 or click **Help** from the dialog.

   The extensions you select as listed as `<import>` elements in the `<dependencies>` section of the extension manifest.
3. To specify extension requires classes from another OSGi bundle, use the Require Bundles section to specify one or more bundles that depend on this extension.

   The extensions you select as listed as `<bundle>` elements in the `<require-bundles>` section of the extension manifest. When searching for a class, OSGi will search in the order they are listed, so be sure to add the bundle to the proper location in the list of bundles.

How to Set Dependencies in the OSGi Bundle Profile

As part of packaging up your extension, you create an OSGi bundle profile which is used to determine the generated bundle manifest. For more information, see How to Create the Deployment Profile.

The important entries to make on the OSGi Bundle Profile dialog are:

- **Package Exports:** This is where you specify file groups that are contributors to the Export-Package section of the generated bundle manifest. For example, if there are Java packages in your extension which you want to make accessible to other extensions, list the packages in this section.
- **Package Imports:** This is where you specify file groups, library dependencies, and profile dependencies that are contributors to the Import-Package section of the generated bundle manifest.
- **Require Bundle:** This is where you specify library dependencies and profile dependencies that are contributors to the Require-Bundle section of the generated bundle manifest.

The Library Dependencies page allows you to check the library dependencies for the bundle, and the Profile Dependencies page allows you to examine and if necessary change dependencies on other JAR deployment profiles in the application.

2

Developing Extensions in Oracle JDeveloper

This chapter provides an introduction to some of the features available to JDeveloper extension writers, developers, and users, and offers examples of several of the features that extension developers ask about most frequently. It describes the native features of JDeveloper that you can begin using immediately to develop features for an extension you plan to develop.
This chapter includes the following sections:

About Developing Extensions in Oracle JDeveloper

Learn about the open archictured used by JDeveloper extensions.

JDeveloper uses an open architecture that makes it possible for you to write your own **extensions**, if you have specific needs or you would like to integrate JDeveloper with some external process or tool that your development team uses. You can add menu items, create context menus, integrate features into the JDeveloper toolbar, create dockable windows that provide a view into your data objects, and more.

Use Cases for Developing Extensions

Use these use cases to help you develop JDeveloper extensions.

This section outlines a series of use cases that you may encounter when planning your extension or if you are converting an existing extension written for an earlier version of JDeveloper to use declarative trigger hooks and support lazy initializations. Follow the recommendations here to create extensions that load and execute quickly.

Understanding Rules Based Menu Sensitivity

Extensions control menu sensitivity by setting the enabled state of the action associated with a menu item, which is determined by one of the `oracle.ide.controller.Controllers` associated with that action. Since actions are trigger hooks, `oracle.ide.controller.Controller` implementations associated with

the action are not called unless the extension that owns the controller has been initialized. For an example of setting simple rules, see How to Define Simple Rules.

Consider the following general situation: Menu item M1 is associated with action A1, which in turn is associated with several controllers. These controllers, in turn, are allowed to indicate whether they handle action A1 for the given context. The first controller that handles the selected action sets the action's enabled state.

As a specific example, EditCopy is associated with `action IdeConstants.COPY_CMD_ID`, which in turn is associated with a controller that knows how to copy text, and another controller that knows how to copy visual objects, such as UI components.

In the past, use cases following this pattern have led to performance problems, when menus are shown or when toolbar buttons change their enabled state as JDeveloper changes context—for example, when the user changes the current selection, changes the current view, or changes the current active project.

To prevent time-consuming operations in these situations, develop your extensions' menus using rule-based action sensitivity control. Controllers are defined declaratively in the extension manifest file. For more information, see Working with the Extension Manifest.

Part of the controller definition specifies action update rules, which control the action-enabled state:

- Enable "Always"
- Enable when "Active application present"
- Enable when "Active project present"
- Enable when "Context has a node"
- Enable when "Context has a node of type X"
- Enable when "Context has selection"
- Enable when "Context has single selection"
- Enable when "Context has multiple selection"

Controller can define multiple rules, and the most specific rules must appear first in the rule definition list.

JDeveloper analyses the controller rule-based sensitivity before the extension that owns the controller is initialized. Once the extension is initialized, the extension's `Controller.update()` method handles the action-enabled state. If no controller does so, extension operation follows the rules as defined. The recommended way is to have all `Controller.update()` methods return `false` so as not to interfere with GUI responsiveness.

How to Avoid Complex Controller.update() Implementations

Controllers specify the rules that determine the enabled state of an action declaratively, using the action-enabling rules described in Understanding Rules Based Menu Sensitivity. If your menu is enabled and the user executes it, the extension can then perform the operations needed to ensure that menu execution works properly.

Earlier implementations of `Controller.update()` used to execute complex and time-consuming code in order to determine the action enabled state. The current menu

sensitivity model allows the enabled state of an action to be determined declaratively, and reduces the time involved in choosing the appropriate action to perform.

The rules that control controller behavior are as follows.

When the extension is initialized:

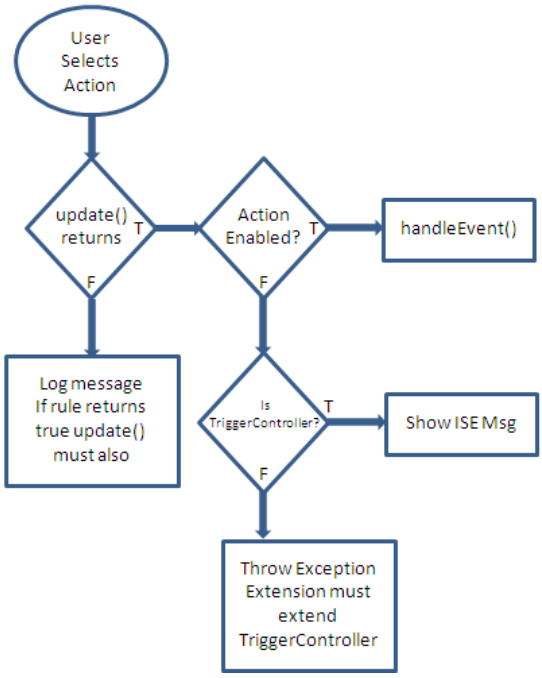
- A menu is about to be shown:
  - If it is a rules base controller, the rule will be evaluated. If `false`, the menu is grayed out by the framework. If `true` or not a rules-based controller, see the next bullet point.
  - `Controller.update()` is called. If it returns `true`, no other controllers are processed. The controller must set the enabled/disabled state of the action inside the `update()` method.
- The user selects an active action. `Controller.update()` is called. If it returns `true`, no other controllers are processed. The controller must set the enabled/disabled state of the action inside the `update()` method.

When the extension is not initialized:

- A menu is about to be shown. The rule for the controller is evaluated. If `false`, the menu is grayed out by the framework. If `true` it is activated by the framework.
- The user selects an active menu on an extension that has not been initialized:
  - `Controller.update()` is called. If `update()` returns `false` an exception will be thrown to say that when the controller's rule is evaluated to `true`, the controller's `update()` method must also return `true`, and a log message is displayed to remind you. If `update()` returns `true`, go to the next bullet point.
  - The action will be checked to see if it is enabled/disabled. If the action is enabled, `Controller.handleEvent()` will be called. If the action is disabled, go to the next bullet point.
  - If the controller is a `TriggerController`, the framework will display a message to the user telling them why the action cannot be performed. If it is not a `TriggerController`, an exception will be thrown telling the you that this controller needs to be a `TriggerController`.

  Therefore if the conditions in the first or third bullet points occur, you need to address them declaratively.

Figure 2-1 Flow Diagram for Unitialized Extension



How to Use Dynamic Menu Labels and Icons

Dynamic menu labels and icons are menu items that vary depending on which application, project or node the user selects. From within your extension, dynamic menu labels execute code when selected in order to modify the menu label string and/or icon that is displayed when the user selects the menu.

How to Append the Short Name to the Menu Label

When the user selects a dynamic menu, JDeveloper appends the selected element short label to the menu item label. You can see this in:

Run > **Run *myproject.jpr***

In general, these menus depend on the selected application, project, node, or element. Using the controller hook, extensions can request that JDeveloper update the label based on the action update rules. For more information, see Understanding Rules Based Menu Sensitivity.

This updates the menu label, adding the short name associated with the element specified by the rule. This element can be an application, project, or node.

How to Construct Dynamic Top Menus

When constructing dynamic top level menus, such as the **Source** top level menu, define them declaratively using the same trigger hooks as top level menus. For more information, see How to Create and Modify Menus.

What determines whether a menu item is included in the dynamic top level menu are both:

- Editor type
- Menu type

When these two conditions are met, the dynamic menu declaratively defined for the active editor and node opened is displayed as a top level menu. When the extension user selects this menu, menu item sensitivity is handled by setting the enabled state of the action associated with a menu item. For more information, see Understanding Rules Based Menu Sensitivity.

Understanding Node Recognizers

Node recognizers are responsible for recognizing the `oracle.ide.model.Node` subclass associated with a resource pointed to by a specific URL.

JDeveloper provides two standard recognizers:

- One that can recognize node types such as file extensions, based on information provided by the URL itself.
- One that can recognize XML node types, based on the content of an XML file.

In previous versions, JDeveloper also allowed custom recognizer implementations to be registered. This is now deprecated.

The `Addin.initialize()` method registers a custom implementation of the oracle.ide.mode.Recognizer class using one of these methods:

- `Recognizer.registerRecognizer(String, Recognizer)`
- `Recognizer.registerRecognizer(String[], Recognizer)`
- `Recognizer.registerLowPriorityRecognizer(Recognizer)`

Extensions that provide custom recognizers must adjust their node recognition in such a way that it can be handled by the recognizers in JDeveloper.

If you are rewriting an extension developed for an earlier version of JDeveloper that used the IDE standard recognizers, you need to remove the recognizer registration code from the `Addin.initialize()` method and use the recognizer trigger hook to register recognition rules in the extension manifest file, `extension.xml`. For more information, see `Recognizer` in the *Java API Reference for Oracle Extension SDK* and Working with the Extension Manifest.

Understanding Content Sets

JDeveloper uses content sets to define the categories that are displayed as nodes in the Applications window. The categories in installed JDeveloper include:

- Application Sources
- Web Sources
- Resources

Content sets are registered declaratively, but do not trigger initialization of the extension that introduced the content set, because content sets require no custom behavior. You must provide the content set ID, label, and default URLs that indicate where to look for content in your extension.

Instances of `ContentLevelFilter` provide client behavior that control what is shown under a content set.

A `ContentLevelFilter` is responsible for filtering the breadth-first traversal implemented by the `ContentLevel` class to provide a virtual representation of each level that differs from its physical representation. A `ContentLevelFilter` may:

- Add new elements to be displayed.
- Remove elements from the display.
- Remove subdirectories from the display.
- Cause a subdirectory to be displayed even if it is empty.

Instances of `ContentLevelFilter` are registered statically via the static method `addContentLevelFilter(ContentLevelFilter)` in the class `ContentLevel`.

After a new application is created or opened in the Applications window, if the extension user expands the top-level folder of a project, before adding all the folders and files in such project (which uses method `open` in `BaseTreeExplorer`) the method `applyContentLevelFilters(Context, List, List)` in class `ContentLevel` is called. This method:

1. Gets all the source root directories.
2. Checks that, for each of the instances of `ContentLevelFilter` registered, whether a filter can be applied to the given `Context`. If it can, it calls the method `updateDir(URLPath, String, List, List, Context context)` which can modify the lists of elements and subdirectories for each level before they are displayed. Whenever there are multiple instances of `ContentLevelFilter` applying changes to the same level, later filters will see the effect of earlier filters and, if appropriate, perform further filtering on them.
3. Does the same as in the previous step for instances of `AsynchronousContentLevelFilter`, `ContentLevel`. It also obtains a `Callable` from such a filter, which performs an asynchronous request to get additional elements.

> **Note:**
>
> To find out whether a `ContentLevelFilter` can be applied to a `Context`, `ContentLevel` retrieves all the content set keys from that filter and checks to see whether the context has at least one of the keys as property.

`ContentLevelFilter` does not need declarative registration. Before filtering takes place, nodes are created using `NodeFactory`, which uses recognizers to create nodes from URLs. Since recognizers are trigger hooks, by the time filtering takes place, related extensions are already initialized.

In general, extensions that register content level filters also register recognizer rules; therefore, they are always initialized if a node type is recognized by the recognizer rules they registered. For example, the Applications window ensures that as the user expands folders in the Applications window, it creates instances of nodes for all files found under that folder, which triggers extension initialization. At that point, JDeveloper invokes the content level filters giving these a chance to filter out nodes or add other nodes to display under the folder being expanded.

Understanding Large Extensions

Extensions that provide a wide set of functionality and have a deep dependency tree are considered large extensions. In general, initializing large extensions causes the pre-initialization of a number of other extensions due to their deep dependency tree. In many cases extension users may only be accessing a subset of the wide functionality provided by a large extension but, inadvertently, they get more than they require due to the monolithic structure of this type of extension.

To improve the performance of a large extension, it is recommended that you refactor large extensions following one of these models:

- Identify the extension's main functional areas and refactor prominent areas into separate extensions; or
- Keep abstract functional areas in a single extension with a shallow dependency tree, while moving functional implementation details to separate extensions.

The first model works best when functional areas have simple dependencies between them. For example, consider an extension, E1, which has two functional areas: X and Y. A simple dependency means that refactoring extension E1 into extensions E2 and E3 introduces no bi-directional dependencies between E2 and E3, or if it does, these can be resolved by introducing a support module, (for example, module M1), which provides the functionality both E2 and E3 need. It is important that M1 not have the same dependency tree as E1; this defeats the purpose of the refactoring exercise, which was to reduce the number of extensions that need to be pre-initialized before the functionality provided by E1 can be exercised. Ideally, extensions E2 and E3 do not depend on each other and the original dependency tree from extension E1 is now evenly distributed between extensions E2 and E3.

The second model leaves a version of extension E1 that has a very shallow dependency tree. In order to reduce the dependency tree, the refactoring operation must move the implementation of functional areas E2 and E3 to extensions E2impl and E3impl. For this pattern to work, it is very important that E1 does not depend on extensions E2impl and/or E3impl. To support this pattern, use the `singleton-service-hook` which makes sure that the right implementation, say the E2impl extension, is initialized when functionality E2 is accessed through extension E1. By using the `singleton-service-hook`, E2impl is initialized even though E1 does not depend on that extension. For more information, see the section in Singleton registration in Defining and Using Trigger Hooks.

Understanding Technology Scopes

When the user selects a technology scope, the extension that introduced that technology scope along with all the other extensions it depends on is initialized. The deployment dependencies are stated in their projects' deployment profiles. For more information, see "Deployment Profiles" in *Developing Applications with Oracle JDeveloper*.

If the user's selection makes it necessary to initialize extensions outside the dependency tree of the extension introducing the technology scope, the recommended method of handling this is to wait for the user to use some functional aspect of your extension that triggers extension initialization. This avoids the concept of the feature set, which would require initializing all members of the feature set and defeat the purpose of lazy initialization.

Getting the JDeveloper Look and Feel

Create a JDeveloper extensions that matches the look and feel of JDeveloper.

You can develop extensions that blend seamlessly with JDeveloper by following the guidelines in this section. For example, you can use standard spacing and alignment for common components by using the UI constants published in the `oracle.javatools.ui.LayoutConstants` to achieve the recommended spacing. The spacing can be used for all components, whether they are in dialogs, wizards or modeless overview editors.

Figure 2-2 shows the standard spacing used for elements in a JDeveloper dialog.

Figure 2-2 JDeveloper Dialog Showing Spacing

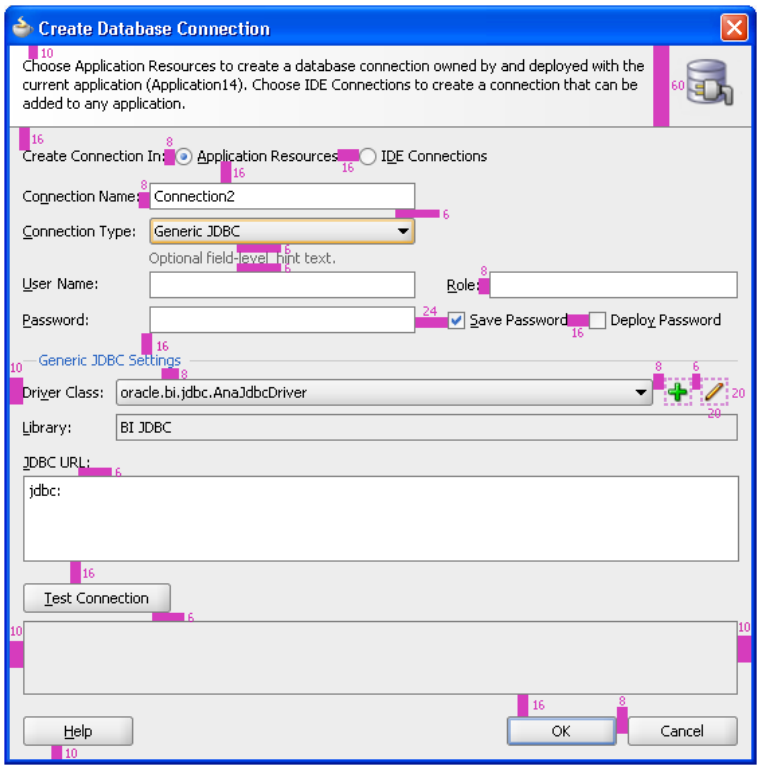
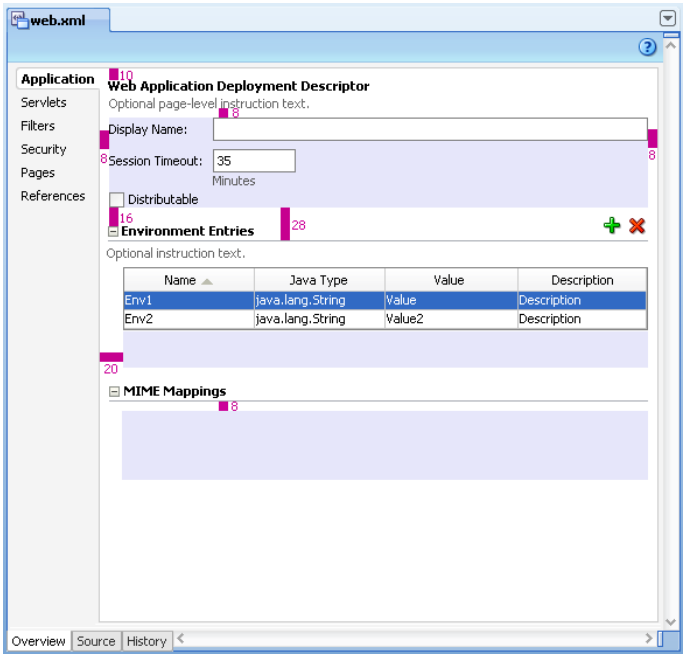


Figure 2-3 shows the spacing used for an overview editor in JDeveloper.

Figure 2-3 Overview Editor Showing Spacing



Creating JDeveloper Elements

Use elements such as menus, windows, wizards, editors, explorers for your JDeveloper extension.

This section describes how to work with and develop various JDeveloper elements:

- Menus
- Windows and views
- Wizards that can be invoked from the New Gallery or from the Tools menu
- JDeveloper commands
- Source editors
- Explorers
- File types
- Components window
- JDeveloper preferences
- Making changes undoable

The Extension SDK provides examples of these and other elements. See Developing with the Extension SDK .

How to Quickly Create JDeveloper Elements

There are some JDeveloper elements that you can create quickly from the New Gallery:

- Actions, that is operations that the extension user may invoke in the IDE. For information about Actions, see How to Define an Action.
- Addins, which can perform programmatic initialization and registration for an extension while JDeveloper is starting up. For information about Addins, see How to Implement the Addin Interface.
- Data models, which subclass `HashStructureAdapter` to store the persistent data model for preferences or project properties. For information about data models, see How to Implement the Data Model.
- Wizards that can be invoked from the New Gallery or Tools menu. For information about wizards, see How to Develop Wizards .
- New panels in the Preferences dialog. For information about working with the Preferences dialog, see Understanding Preferences.
- New panels in the Project Properties dialog. For information about project properties, see Understanding Project Properties.
- Freemarker file templates that you can use in an extension hook defined file template.

To quickly create a JDeveloper element for the New Gallery:

1. Open the New Gallery by choosing **File > New**.
2. In the New Gallery, in the Categories tree, under **General** select **Applications**.
3. In the Items list, double-click **Extension Application**. The Create Extension Application wizard opens. For help with the wizard, press F1 or click **Help**.
4. Enter a name for the application and choose a location and an application package prefix and click **Next**. The project, which is configured for extension development, has a default name of `Extension`. If necessary, change the project name, then click **Next**.
5. Open the New Gallery by choosing **File > New**.
6. In the New Gallery, in the Categories tree, under **Client Tier** select **Extension Development**.
7. In the Items list, double-click the JDeveloper element you want to create. The appropriate dialog opens. For more help, press F1 or click **Help** from the dialog.

How to Create and Modify Menus

Because so much of a user's interaction with JDeveloper is through the menus, JDeveloper lets you modify existing menus and create new ones. In addition to adding selections to the items in JDeveloper's menu bar, you can also add context menus that give you finer control over user interactions with the functions provided by your extension.

Menus interact with the command extension, which in turn relies on the `Addin` interface. For more information, see How to Develop Commands .

Understanding Menus

Menus are an important way to integrate your extension's functionality into JDeveloper. You may wish to add a major feature—selecting a versioning system, for example, or accessing an internal code-snippet database—to JDeveloper's menu

system, so that your extension can be accessed or controlled through a selection from the menu or tool bar. On the other hand, some features of your extension—checking out a file from your organization's versioning software or verifying code with your organization's preferred tool—might best be handled through a context menu that pops up when the user clicks the right mouse button in the appropriate place. JDeveloper's extension system lets you do both. For more information, see How to Create a Context Menu.

JDeveloper includes a class to support the declarative creation of context menus, which can then be hooked to the main JDeveloper functionality by a series of listeners and controllers. This lets you create menu-based extensions that fit within the structure and user interface of JDeveloper, while adding features and capabilities beyond the basic IDE. For more information, see How to Invoke an Addin From a Context Menu.

Alternatively, your extension might have features that already fit within the existing menu structure of JDeveloper, but add unique functions to each of the current menus. In this case, your extension needs to add specific new options to existing menus. For more information, see How to Add Menu Items to an Existing Menu.

You can also use a similar technique to add new selections to a toolbar. For more information, see How to Add a Drop-down Button to a Toolbar.

How to Create a Context Menu

Creating a menu declaratively using the context menu listener classes lets you use standard classes. This helps make your extension easy to update, adds consistency across all menus, helps your extension load quickly, and can require no custom code to develop. You can use the existing context menu listener classes, by registering them, using the callback interface that lets you add menus to your extension, and finally by using the controller associated with the specific view for which the menu is created. For more information, see How to Invoke an Addin From a Context Menu.

To create a menu with several selections:

- Register the context menu listener classes in your extension manifest using the sample code in the following example:

```
<extension xmlns="http://jcp.org/jsr/198/extension-manifest"
  ...
      <!--
        Define an action that will be used in various menus and toolbars.
        You can set any Swing action property on the action. The controller
        class implements oracle.ide.controller.Controller and determines
        when the action is enabled and what it does.

        You can use an ampersand (escaped as &amp;) in the Name property
        to specify the mnemonic for any menu items containing the action.
      -->
      <actions xmlns="http://xmlns.oracle.com/jdeveloper/1013/extension">
        <action id="oracle.ide.extsamples.first.invokeAction">
          <properties>
            <property name="Name">&amp;Extension SDK First Sample</property>
             <!-- use an icon, stealing this one from predefined icons
                  make sure you have oracle.icon as a required bundle to
                  pick up this icon at run time from the icons module  -->
            <property name="SmallIcon">${OracleIcons.LABEL}</property>
          </properties>
        </action>
      </actions>
```

```
      <controllers xmlns="http://xmlns.oracle.com/ide/extension">
        <controller class="oracle.ide.extsamples.firstsample.SimpleController">
          <update-rules>
            <update-rule rule="always-enabled">
              <action id="oracle.ide.extsamples.first.invokeAction">
                <label>ESDK Sample Action</label>
              </action>
            </update-rule>
          </update-rules>
        </controller>
      </controllers>
      <!--
        Install context menus in the navigator, editor, and structure pane
(explorer)
        which use the same action defined above.
      -->
      <context-menu-hook rule="always-enabled">
        <site idref="navigator, editor, explorer"/>
          <menu>
            <!-- 'weight' is used to control where the section appears in
             the menu. Weights are sorted in ascending order, each unique
             value is its own section. -->
            <!-- section element must be in namespace
             http://jcp.org/jsr/198/extension-manifest. -->
            <section xmlns="http://jcp.org/jsr/198/extension-manifest"
id="MY_CUSTOM_MENU_SECTION_ONE" weight="1.0">
              <!-- For menu items and submenus, weight determines the
               order within a menu section. Items are sorted in ascending
               order or in the order they are added if the weight is not
               unique. -->
             <item action-ref="oracle.ide.extsamples.first.invokeAction"></item>
           </section>
         </menu>
      </context-menu-hook>

      <menu-hook xmlns="http://jcp.org/jsr/198/extension-manifest">
        <!--
          Add the action in its own separator group at the top of the File
          menu.
        -->
        <menus>
          <menubar id="javax.ide.view.MAIN_WINDOW_MENUBAR_ID">
            <menu id="javax.ide.FILE_MENU_ID">
              <section id="esdksample.customsection"
before="javax.ide.NEW_SECTION_ID">
                <item action-ref="oracle.ide.extsamples.first.invokeAction"></
item>
              </section>
            </menu>
          </menubar>
        </menus>


        <!--
          Add the action as toolbar button in its own separator group
immediately after the
          existing "save" toolbar buttons.
        -->
        <toolbars>
          <toolbar id="javax.ide.view.MAIN_WINDOW_TOOLBAR_ID">
```

```
            <!-- Add a new section after the first section in the toolbar. -->
            <section id="javax.ide.tck.toolbar.testsection"
                   after="javax.ide.NEW_SECTION_ID">
              <item action-ref="oracle.ide.extsamples.first.invokeAction"></item>
            </section>
          </toolbar>
        </toolbars>
      </menu-hook>

    </triggers>
  </trigger-hooks>

  <hooks>
   <!--
      The feature hook controls the display of the extension on the
      Extensions page of preferences.
    -->
    <feature-hook>
      <description>Demonstrates installing menu, toolbar and gallery items.</
description>
      <optional>true</optional>
      <part-of>oracle.ide.extsamples.allsamples</part-of>
    </feature-hook>
  </hooks>
</extension>
```

How to Improve Performance by Registering a Context Menu Listener

Registering a context menu listener makes sure your context menus pop up quickly.

Context menu listeners should do the minimum amount of work so that they do not delay the popping-up of context menus. Currently, there are close to 250 menu listeners for the Applications window context menu, therefore, on average listeners have to take less than 4 milliseconds for the context menu to pop up in less than a second.

Here are some tips for improving performance:

- If your extension is interested in certain nodes only, your context menu listeners should be registered for that specific node type. This guarantees that your listener is only called when the user of your extension clicks on a node of that type.
- When your listener's `menuWillShow` method is called, your only action should be to add your menu item. Protect against non-applicable contexts when your menu item action is executed.
- Your listener's `menuWillHide` method should do nothing.

The following things have a negative impact on context-menu performance:

- File I/O operations.
- Parsing the contents of files.
- Iterating over a list of things looking for something.
- Searching for something on the file system.
- Iterating over the menus already added to the context menu and trying to rename them or remove them.

How to Add Menu Items to an Existing JDeveloper Menu

Sometimes, a feature of your extension can be handled simply by adding an item, or a group of items, to an existing JDeveloper menu or toolbar.

To add menus:

1. To add a new context menu in a window, editor, or explorer (such as the Structures window) install listeners, as shown in the following example:

```
 <context-menu-hook rule="always-enabled">
        <site idref="navigator, editor, explorer"/>
          <menu>
            <!-- 'weight' is used to control where the section appears in
             the menu. Weights are sorted in ascending order, each unique
             value is its own section. -->
            <!-- section element must be in namespace
             http://jcp.org/jsr/198/extension-manifest. -->
            <section xmlns="http://jcp.org/jsr/198/extension-manifest"
id="MY_CUSTOM_MENU_SECTION_ONE" weight="1.0">
              <!-- For menu items and submenus, weight determines the
               order within a menu section. Items are sorted in ascending
               order or in the order they are added if the weight is not
               unique. -->
             <item action-ref="oracle.ide.extsamples.first.invokeAction"></item>
           </section>
         </menu>
      </context-menu-hook>
```

2. To add a menu to an existing menubar, see the following example:

```
<menu-hook xmlns="http://jcp.org/jsr/198/extension-manifest">
        <menus>
          <menubar id="javax.ide.view.MAIN_WINDOW_MENUBAR_ID">
            <menu id="MY_CUSTOM_MENU" weight="20.0">
              <label>${TRANSLATED_MENU_LABEL}<\label>
            </menu>
          </menubar>
        </menus>
```

3. To add a menu item to a menu, see the following example:

```
<menu-hook xmlns="http://jcp.org/jsr/198/extension-manifest">
        <!--
          Add the action in its own separator group at the top of the File menu.
        -->
        <menus>
          <menubar id="javax.ide.view.MAIN_WINDOW_MENUBAR_ID">
            <menu id="javax.ide.FILE_MENU_ID">
              <section id="esdksample.customsection" weight="1.0">
                <item action-ref="oracle.ide.extsamples.first.invokeAction"></
item>
              </section>
            </menu>
          </menubar>
        </menus>
```

How to Add a Drop-down Button to a Toolbar

In some cases, your extension will operate through a window which may itself have a toolbar with specific options unique to your extension, or to the specific data object viewed in your extension's window. For more information, see How to Invoke an Addin From a Main Window Menu.

To add a drop-down button to a toolbar:

The code in the subsequent example, when included as a method to a window unique to your extension, adds a drop-down button, with three actions, to a toolbar. When the user clicks the button, the menu that appears contains three actions.

```
void doit(Toolbar toolbar, IdeAction dropDownIdeAction, IdeAction ideAction1,
IdeAction ideAction2, IdeAction ideAction3) {
   ActionMenuToolButton actionMenuToolButton =
toolbar.addActionMenuButton(dropDownIdeAction);
   Action[] actions = new Action[] {
      ideAction1,
      ideAction2,
      ideAction3,
   };
   actionMenuToolButton.setMenuActions(actions);
}
```

Working with Windows and Views

The JDeveloper data model uses two important concepts for displaying and monitoring the content of the information you are working with: windows and views. Windows (often dockable windows, which occupy a specific location in the IDE framework) display information and take command inputs. Views define how JDeveloper accesses the information objects (such as files or database records) that your extension generates, manipulates or displays.

Understanding Dockable Windows

Dockable windows allow you to create a dedicated area for your extension at a specified location in JDeveloper. Using dockable windows involves two key steps:

1. Creating a dockable window.
2. Positioning a dockable window.

Once you have created and positioned the window to be used by your extension, you need to make sure that your extension is aware of that window, and in particular, that it is aware when the user has selected your window for input. The mechanism that JDeveloper uses for this is the IDE listener. Adding an IDE listener to your window connects your extension to the window you created. For more information, see How to Add an IDE Listener to a View.

Now that you have created, positioned, and assigned an IDE listener to your dockable window, you are ready to make sure that your extension is aware when the user has selected it. When a window is selected, it has the active view. You can now instruct the IDE listener you added to listen for the active view. For more information, see How to Listen for the Active View.

How to Create Simple Dockable Windows

Creating a dockable window involves three steps: creating a `DockableFactory`, creating a dockable window using the class `MyDockableWindow`, and then installing the `DockableFactory` during the addin initialization.

To create a dockable window:

1. Create a `DockableFactory`, as shown in the following example:

```
public class MyDockableFactory implements DockableFactory
{
 static final String VIEW_TYPE = "MY_VIEW_TYPE";
 public void install()
 {
   final DockingParam dockingParam = new DockingParam();
   dockingParam.setPosition(IdeConstants.SOUTH);
   DockStation.getDockStation().dock(
         new MyDockableWindow(),
         dockingParam);
}
 public Dockable getDockable(ViewId viewId)
{
   if (viewId.getName().equals(MyDockableWindow.VIEW_ID))
      return new MyDockableWindow();
  {
    return null;
  }
}
```

2. Create a `DockableWindow`, as shown in the following example:

```
public class MyDockableWindow extends DockableWindow
{
  static final String VIEW_ID = "MY_VIEW_ID";
   private JLabel _ui;
   public MyDockableWindow()
  {
     super(MyDockableFactory.VIEW_TYPE + "." + VIEW_ID);
  }
  public String getTabName()
  {
  return "ShortName";
  }
  public String getTitleName()
  {
  return "The Long Name Comes Here";
  }
  public String getTitleName()
  {
  return "The Long Name Comes Here";
  }
 public Component getGUI()
 {
   if (_ui == null)
   {
    _ui = new JLabel("The UI is here");
   }
   return _ui;
   }
```

```
 public int getDefaultVisibility(Layout layout)
  {
    return DEFAULT_VISIBILITY_VISIBLE;
  }
}
```

3. Install the factory during the addin initialization, as shown in the following example:

```
DockStation.getDockStation().registerDockableFactory(
   MyDockableFactory.VIEW_TYPE,
   new MyDockableFactory()
);
```

How to Position Dockable Windows

The following example shows how to center a dockable window with the Applications window. If the Applications window extension is not loaded, the window will be docked on the left (WEST).

```
dockingParam = new DockingParam();
final NavigatorManager applicationNavigatorManager =
NavigatorManager.getApplicationNavigatorManager();
final NavigatorWindow navigatorWindow =
applicationNavigatorManager.getNavigatorWindow();
dockingParam.setPosition(
   navigatorWindow,
   IdeConstants.CENTER,
   IdeConstants.WEST
);
```

How to Add an IDE Listener to a View

To add a selection listener to the active view, you need to listen for the active view changes. If you added your listener to the view becoming inactive, you need to remove your listener from that view.

To add an active listener:

Create code based on the following example:

```
import oracle.ide.view.ActiveViewListener;
import oracle.ide.view.ActiveViewEvent;
import oracle.ide.view.ViewSelectionListener;
class MyActiveViewListener implements ActiveViewListener
 {
    private ViewSelectionListener _selectionListener = new ViewSelectionListener()
    {
       public void viewSelectionChanged(ViewSelectionEvent e)
        {
         //Your code responding to view selection changes goes here.
        {
    };
```

To change the active listener:

Create code based on the following example:

```
public void activeViewChanged(ActiveViewEvent e)
 {
   View view = e.getOldView();
```

```
   if (view != null) view.removeViewListener(_selectionListener);
   view = e.getNewView();
   //While this example adds a ViewSelectionListener to any active view,
   //it is strongly recommended that you add your view selection listener
   //to views your extension is interested in only.
   view.addViewListener(_selectionListener);
  }
}
```

How to Listen for the Active View

The JDeveloper IDE architecture's model/view/controller model requires that your extension keep track of which view—that is, which representation of the data is being displayed in a given window—is active. (You may be familiar with windowing systems that refer to active views as having input focus.) The JDeveloper IDE architecture requires your extension to listen for the active view, as a way of ensuring that commands executed by the view are applied to the appropriate data.

To listen for the active view:

Create code based on the following example:

```
Ide.getMainWindow().addActiveViewListener(new ActiveViewListener()
  {
     public void activeViewChanged(ActiveViewEvent e)
    {
       final View view = e.getNewView();
       System.out.println(view.getId() + " has been activated");
     }
});
```

How to Develop Wizards

A wizard is an extension that is invoked to perform some task. A typical wizard is invoked through the UI to open a user interface consisting of one or a sequence of dialog boxes, in which the user specifies task parameters. A typical task is the creation of a document or other data object. Specialized invocation mechanisms are provided for wizards that are installed in the **New Gallery** or the **Tools** menu. In these cases the Wizard Manager manages the invocation details.

How to Set Up a Wizard Project

The wizard project's properties specify paths, libraries, and other settings for the wizard project. When you have set up your project you can add your source files to it, and then debug and deploy your wizard.

A wizard project usually has three main components:

- A wizard class. This class deals with the wizard's appearance in the user interface, and with its invocation. This class must implement the `Wizard` interface.
- A modal dialog. The dialog interacts with the user to collect data required for the function of the wizard.
- A data object. The wizard applies the data collected by the dialog to create or modify a data object.

`HelloX`, a sample project, available as part of the Extension SDK implements wizard directly and uses a JDialog as its user interface. Wizards that conform to JDeveloper's look and feel use the JEWT wizard framework.

For more information about the Extension SDK, see Developing with the Extension SDK ..

How to Implement the Wizard Interface

Extensions that are to be invoked from the user interface to perform some modal task should implement the `oracle.ide.wizard.Wizard` interface. This interface provides for the extension's installation, and integrates it with JDeveloper's user interface.

To Implement the wizard interface:

- Define the `invoke` method
- Define the `getIcon` method
- Define the wizard in the extension manifest

This method embodies the wizard's functionality. Wizards generally open a dialog to obtain parameters from the user, and then use those parameters to create or modify a document or other data object.

This method is called when the wizard's UI element is selected by the user through the **New Gallery** or **Tools** menu. The `context` parameter identifies the currently selected objects that the wizard might affect.

To define the invoke method:

Create code based on the following example:

```
public boolean invoke(Context ctx) {
    if (!isAvailable(ctx))
        return false;
    String greetee = null;
    greetee =
            JOptionPane.showInputDialog(Ide.getMainWindow(), "Enter your name:",
                                        WIZARD_NAME,
                                        JOptionPane.OK_CANCEL_OPTION);]
    if (greetee == null) // User canceled
        return false;
    JOptionPane.showMessageDialog(Ide.getMainWindow(),
                                  "Hello " + greetee + "!", WIZARD_NAME,
                                  JOptionPane.INFORMATION_MESSAGE);
    return true;
}
```

This method is called to obtain the wizard's New Gallery or Tools menu icon. If the wizard does not require an icon, this method should return `null`.

To define the getIcon method:

Create code based on the following example:

```
public Icon getIcon() {
       if(image == null) {
           image = new ImageIcon(HelloX.class.getResource(ICON_NAME));
       }
       return image;
   }
```

How to Add a Wizard to the New Gallery

Wizards can be installed in the New Gallery by including a gallery wizard description for them in the extension manifest file. All of the details of wizard registration and event processing are handled by the Wizard Manager.

If the wizard is to be invoked only from the New Gallery, the constructor for the wizard class is not called until the extension user first opens its category. The Gallery Manager reads the description file when the category is first opened, and then instantiates the wizard and constructs the item using the icon and label derived from the instance.

There are a number of things that you define in the extension manifest. See the following example:

```
<trigger-hooks xmlns="http://xmlns.oracle.com/ide/extension">
    <triggers>
      <gallery xmlns="http://xmlns.oracle.com/jdeveloper/1013/extension">
        <folders>
          <name label="ESDK-Gallery Example">ESDK-Gallery Example</name>
          <category>General</category>
        </folders>
        <item rule="always-enabled">
          <!-- name is the fully-qualified name of an oracle.ide.wizard.Invokable  --
>
          <name>oracle.ide.extsamples.hellox.HelloX</name>
          <!-- id is optional, and if specified will be stored in the Context
               so it can be retrieved when the wizard is invoked. -->
          <id>oracle.ide.extsamples.hellox</id>
          <!-- description is the short label of the gallery item -->
          <description>Say Hello Wizard</description>
          <!-- help is the long label for the gallery item -->
          <help>Prompts the user to enter his name, and repeats it. Part of the
Extension SDK, this is a trivial example.</help>
          <icon>/oracle/ide/extsamples/hellox/HelloX.gif</icon>
          <category>General</category>
          <folder>ESDK-Gallery Example</folder>
        </item>
      </gallery>
    </triggers>
  </trigger-hooks>
```

`<gallery>` contains the information about the wizard in the New Gallery.

- `<name>` defines the human-readable name for the wizard.
- `<category>` specifies the New Gallery category for the wizard. Within the category, entries are alphabetical.
- `<folder>` specifies the folder in the category.
- `rule` in specifies when the wizard is available.
- The wizard class is identified in `<id>`.
- `<description>` is the short description of the wizard in the New Gallery.

  `<help>` is the long description for the entry displayed when the item is selected, or when Show All Descriptions in the New Gallery is selected.
- `<icon>` specifies the icon in the New Gallery.

By running the extension application, `extensionsdk`, you can see how the code in the previous example affects what is shown in the New Gallery, see

Figure 2-4 HelloX Wizard in New Gallery

How to Develop Commands

An extension that adds a user-interface element such as menu item or toolbar icon, or customizes an existing element for a new purpose, should encapsulate the functionality in a command extension.

How to Implement the Addin Interface

Most extensions should implement the `Addin` interface. This interface provides for the extensions installation at the time of JDeveloper's startup.

To implement the `Addin` interface, first define the constructor. The constructor should do as little as possible; defer initialization tasks to the initialize method.

Then, define the `initialize` method. This method is called by the Addin Manager after the instance has been created. The tasks that should be performed at initialization are:

- creation of UI elements and controllers.
- Registration with managers.

Other tasks, such as the creation of data structures that are not needed until and if the addin is invoked, should be deferred, so that JDeveloper's startup is not unnecessarily delayed.

How to Implement a Command

If your extension adds a menu item or toolbar icon, you should implement its functionality as a command.

An extension that defines specialized behavior for an existing control should use the action and command class already provided for it, rather than define a custom action or implement a custom command class. Fields defined in the `Ide` class give the class names and IDs of the IDE's standard commands.

A command for an extension involves these tasks:

- Handling the event
- Defining the undo method
- Defining other methods

The code examples in this section are taken from the `FirstSample` sample project, available as part of the Extension SDK. For more information, see Developing with the Extension SDK ..

Handling the Event

The code in the following example shows how to handle the event, which is triggered when you invoke the command.

```
/**
  * ContextMenuListeners add items to context menus.
  */
public final class SimpleContextMenuListener implements ContextMenuListener {
    public void menuWillShow(ContextMenu contextMenu) {
        // First, retrieve our action using the ID we specified in the
        // extension manifest.
        IdeAction action = IdeAction.find(SimpleController.SAMPLE_CMD_ID);
        // Then add it to the context menu.
        contextMenu.add(contextMenu.createMenuItem(action));
     }
      public boolean handleDefaultAction(Context context) {
        // You can implement this method if you want to handle the default
        // action (usually double click) for some context.
        return false;
     }
}
```

For long blocks of code, you can use the `doIt` method.

How to Define the undo Method

This method must be defined only if the constructor specifies that the controller is of the `NORMAL` type. The `undo` method generally has two tasks:

- Undo the `doit` methods effect by restoring a checkpointed state with the value obtained from the `getData` method, or by performing the inverse of the `doit` method's operation.
- Notify observers that the modification has taken place.

Return `OK` if the command is successful, or `CANCEL` or some other non-zero value if not. The default implementation returns `CANCEL` and has no side-effects.

How to Define Other Methods

These methods' default implementations can be overridden:

- `getId` returns the command ID passed to the constructor.
- `getType` returns the "command type" constant passed to the constructor, or `NO_CHANGE` if this argument was not given.
- `getName` returns the name string passed to the constructor, or the empty string if this argument was not given.

- `getAffectedNodes` returns `null` by default.
- `setContext` and `getContext` respectively write and read a protected `Context` variable. The default value is `null`.
- `setData` and `getData` respectively write and read a private `Object` variable. The default value is `null`.

How to Define an Action

An extension that implements new command classes should define actions to contain them. An action serves as the link between a menu item, or other user-interface control, and the command that is executed when the menu item is selected. Actions are instances of `IdeAction`.

An extension that defines specialized behavior for an existing UI element should use the action already provided for it, rather than define a custom action. Fields defined in the `Ide` class give the class names and IDs of the standard commands. For example, any editor that provides a 'save' operation should use the predefined `Ide.SAVE_CMD` and `Ide.CUT_SAVE_ID` values and the action that is associated with them.

Actions are typically defined as fields of the Addin class that installs the menu items or toolbar icons they are associated with. Create and configure actions as part of the extension's initialization.

The code examples in this section are taken from the `FirstSample` sample project, available as part of the Extension SDK. For more information, see Developing with the Extension SDK .

How to Obtain an Action

Actions are cached by command ID. If an action already exists for the command you require, you should generally use it instead of creating a new one.

Do not use the `IdeAction` contstructors to create actions. Instead, use the following static methods to retrieve a cached action or create a new one:

- The `find` method returns an action that contains the given command id, if one has already been cached.
- The various `get` methods return the action matching the given command id, if such an action has been cached, but the properties of this action may or may not match the parameters given. If no action is found for the command ID a new action is created from the given parameters, cached, and returned.
- The `create` methods create and return a new action without caching it. A action obtained from create is not available to subsequently loaded extensions.

How to Set Action Values

An action may have various properties, such as those that determine appearance, which are accessed through string keys by the `putValue` and `getValue` methods. Some of the properties are set when an action is created by a create or get static method. Keys recognized by the IDE are defined in the ToggleAction superclass. For more information, see `ToggleAction` in the *Java API Reference for Oracle Extension SDK*.

How to Extend an Action's Controller

An action that is independent of a view, such as 'open', must specify a controller to update the action and handle its events. An action that is invoked in the context of a view need not have an action controller.

The behavior of a command, for all IDE features that use that command, can be extended to perform some custom operation by replacing the controller of the command's action, or by giving the action a controller if it did not already have one. However, care must be taken to avoid disrupting default behavior.

To replace an existing controller:

1. Implement the new controller class by extending the old class.
2. In the new controller's handling of the command that is to be extended, perform the `custom` operation, and then invoke the old controller's `handleEvent` method for the command, so that the original behavior is preserved.

To add a controller to an action that does not have one:

1. Implement a new controller class.
2. In the new controller's handling of the command that is to be extended, perform the `custom` operation, and then invoke the supervisor's `handleEvent` method for the command, so that the original behavior is preserved.

Use the `getController` and `addController` methods to access an action's controller.

Extending an Action's Command Class

The `getCommand` method returns the name of the action's command class. The `setCommand` method can be used to replace it. However, doing so replaces it globally, affecting all IDE features that handle the action. To avoid unwanted side effects, replace an original command class only with a class that extends it.

How to Invoke an Addin From a Main Window Menu

To allow an addin to be invoked from a main window menu, add a menu item for it to one of the main window menus. The item can be added to any menu, however, if the extension is to be invoked from the **Tools** menu, it should be installed as a wizard, rather than (or in addition to) being installed as an addin: the Wizard Manager takes care of the details of adding and handling menu item in the **Tools** menu.

The IDE menus are represented by a singleton instance of `Menubar`, which can be accessed using the `getMenubar` method.

Extensions can add menus, submenus, and menu items to the IDE menus. Extensions that are invoked, such as wizards, add their own items to menus when they are installed. Alternatively, extensions can define their own behavior for standard menu items. For example, effect of the items in the Edit menu depend on the editor in use.

How to Invoke an Addin From a Context Menu

Some views, such as the Applications window and the source editor have context menus that pop-up when the user right-clicks in the window. Extensions can add items to context menus.

When the user right-clicks, the context menu is reconstructed. Selected context menu listeners — those associated with the subject of the right-click — are polled, and given the opportunity to contribute items or submenus to the context menu. For example, when a node representing a document is right-clicked, editors and designers that are registered as viewers for that document's type are allowed to add their menu items to the context menu.

The following example shows how to install context menus in a window such as the Applications window. The code is taken from the ClassBrowser project in the extension samples that are installed as part of the Oracle IDE Extension SDK.

```
 <!--
        Install context menus in the navigator, editor
        which use the same action defined above. -->

      <context-menu-hook rule="always-enabled">
        <site idref="navigator,editor"/>
          <menu>
            <!-- 'weight' is used to control where the section appears in
             the menu. Weights are sorted in ascending order, each unique
             value is its own section. -->
            <!-- section element must be in namespace
             http://jcp.org/jsr/198/extension-manifest.  -->
            <section xmlns="http://jcp.org/jsr/198/extension-manifest"
id="MY_CUSTOM_MENU_SECTION_ONE" weight="1.0">
              <!-- For menu items and submenus, weight determines the
               order within a menu section. Items are sorted in ascending
               order or in the order they are added if the weight is not
               unique.  -->
              <item action-ref="esdksample.showClassBrowser"></item>
            </section>
          </menu>
      </context-menu-hook>
```

How to Develop Editors

An editor is a view that displays an object for the user to modify. Editors usually display text; a designer is a non-textual editor. Editors are opened for a document through its node's context menu or the **View** menu. The editors available for a document are those registered for the document's type with the IDE's Editor Manager. Editors are usually used in conjunction with structure explorers, but this is not required.

The `CustomEditor` project in the Extension SDK illustrates how to do this by implementing a custom `XMLQueryEditor` which extends `oracle.ide.editor.Editor` and implements `java.beans.PropertyChangeListener`. For more information, see Developing with the Extension SDK .

Using Asynchronous Editors

An asynchronous editor loads its contents in a worker thread, outside the UI thread (event dispatch thread in Swing,) resulting in a more responsive user experience. While loading the contents of an asynchronous editor, the editor framework:

- Shows a panel with the message "Loading Editor" and an animation. Both of these are configurable.
- Prevents the Components window from loading and blocking the UI.

Loading of the editor starts when a `Context` is set. This action triggers loading of the model that the editor's UI needs from the UI thread. To do so, an asynchronous editor creates a worker thread to perform the loading. While the model loading is taking place, the IDE asks the editor for its GUI component. If the editor's real UI is not loaded yet, a panel with a message (for example, "Loading Editor") appears and the Components window is not loaded. Once the editor's real UI is created, the asynchronous editor automatically swaps it.

The editor framework gives a lot of freedom as to how editors can be implemented. For more information, see `AsynchronousEditor` in *Java API Reference for Oracle Extension SDK*.

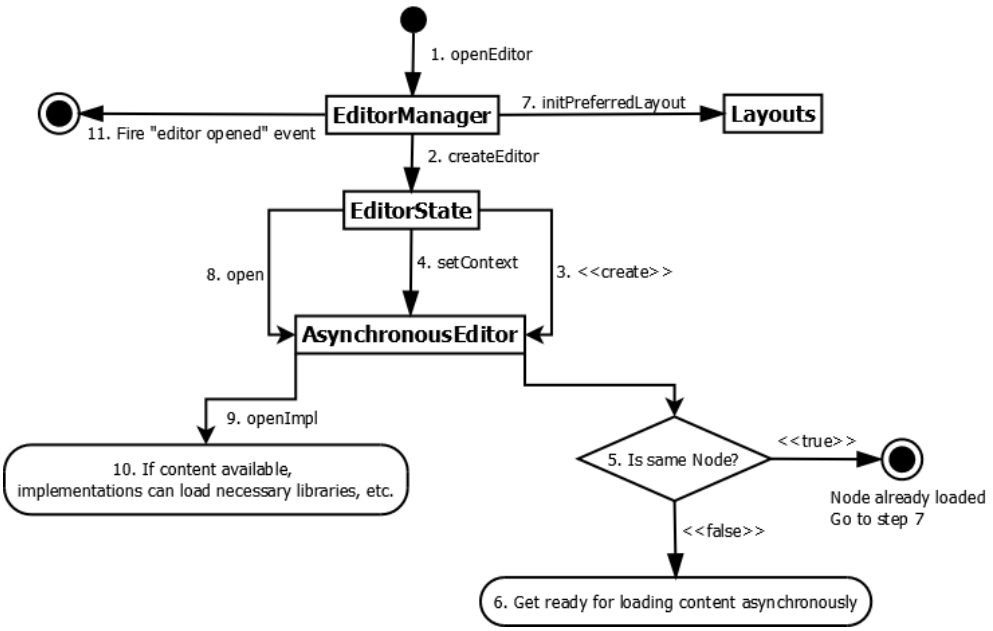
How Asynchronous Editors Work

The first step is that `EditorManager` creates and opens the editor, for example when a user double-clicks a node in the Applications window:

1. `EditorState` creates a new asynchronous editor (a subclass of `oracle.ide.editor.AsynchronousEditor`.)
2. When the asynchronous editor is created, it sets a property that lets other views in the IDE know that it is an asynchronous editor and the "real" editor is not loaded yet. An example of a view interested in this property is the Components window. The Components window only loads its contents once the editor is fully loaded.
3. `EditorState` sets the Context in the new asynchronous editor.
4. The asynchronous editor waits for the appropriate flags indicating that its contents are not loaded yet before loading the asynchronous editor content model. This is done asynchronously (off the UI event thread.)
5. `EditorManager` sets the preferred layout for the editor
6. `EditorState` opens the editor, which eventually causes a call to `getGUI`.

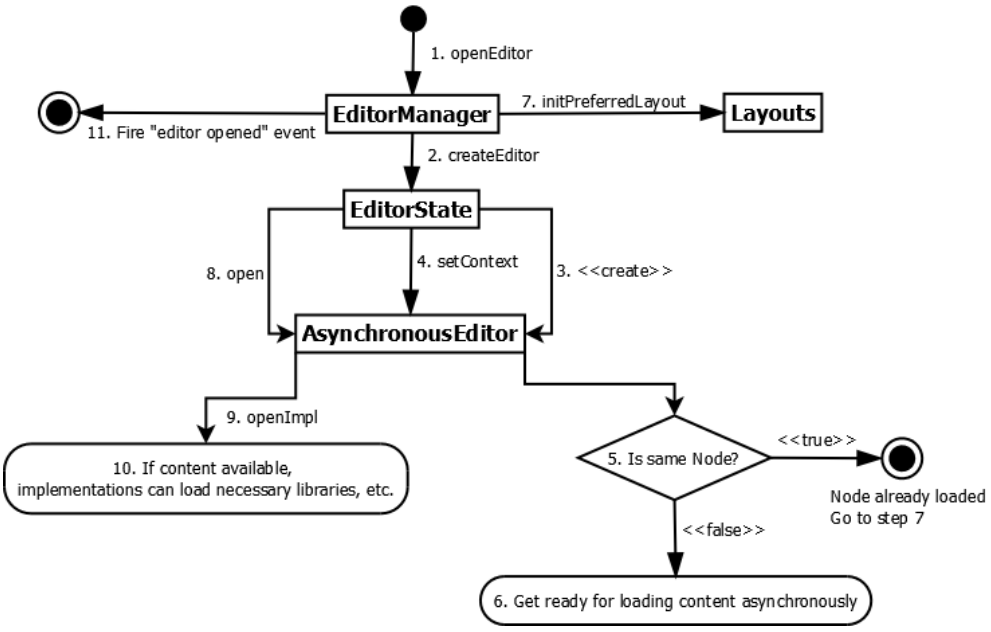
This is illustrated in Figure 2-5.

Figure 2-5 Opening the Editor



Once the editor is open. the editor's method `getGUI` is called. This is the moment that asynchronous editors show a "loading editor" message, while scheduling a task off the UI event thread to load the content model asynchronously, as illustrated in Figure 2-6.

Figure 2-6 Getting the Editor's GUI



At this point, the content model has been loaded asynchronously, and:

- `AsynchronousEditor` calls the methods `openImpl`, `showImpl` and `activateImpl`, to load the editor's GUI, show it, and activate it.
- `AsynchronousEditor` switches the "waiting" page with the "real" editor GUI
- `AsynchronousEditor` notifies listeners such as the Components window that it is fully loaded, so they can load their respective contents.

To make a regular editor an asynchronous editor:

1. Extend `oracle.ide.editor.AsynchronousEditor` instead of `oracle.ide.editor.Editor`.
2. Implement the abstract method `getEditorContext(Context)`. This method is the responsible of loading the model that the editor's UI needs from the given `Context`.
3. Implement the abstract method `doSetContext(Context)`. This method is expected to have the same functionality as the original method `setContext(Context)` in `Editor`. `setContext(Context)` is now final (to guarantee correct behavior of asynchronous functionality.)
4. Implement the abstract method `isContentModelLoaded()`. This method indicates the asynchronous editor framework if the model used by your editor is loaded or not.
5. Implement the abstract method `openImpl(boolean)`. To ensure correct behavior of asynchronous editors, the abstract method `open()` in `Editor` is now final. `openImpl` passes a boolean argument that indicates whether the model for the editor's UI has been loaded or not. In most cases, you would only need to move

your existing implementation of `open()` to `openImpl` when the given argument is `true`.

6. Move all the code in the `Editor` lifecycle events to the new methods. To guarantee that asynchronous content loading works correctly, `AsynchronousEditor` makes the editor life-cycle final, giving subclasses implement them through "`impl`" methods.

   The life-cycle methods made final are:

   - `open`
   - `close`
   - `editorShown`
   - `editorHidden`
   - `activate`
   - `deactivate`

   Subclasses of AsynchronousEditor need to implement these methods instead:

   - `openImpl(boolean)`
   - `closeImpl(boolean)`
   - `editorShownImpl(boolean)`
   - `editorHiddenImpl(boolean)`
   - `activateImpl(boolean)`
   - `deactivateImpl(boolean)`

   where the boolean argument specifies whether the content model of the editor has been loaded or not.

   The complexity of this task is related to the number levels in the class hierarchy for a specific editor, especially when overriding life-cycle methods. If a chain of classes override `close()`, now all of them need to override `closeImpl(boolean)` instead.

How to Develop Explorers

A structure explorer displays the organization of a document, usually as a tree. JDeveloper provides explorers for various common document types, and allows custom explorers to be installed and registered with the Explorer Manager.

How to Create an Explorer

A structure explorer displays the organization of a document, usually as a tree.

A structure explorer is a `View` of a `Document`. When an editor or designer is given focus, its explorer is shown in the structure window. An explorer is essentially an index or table of contents for the document: the user uses the explorer to navigate in the content displayed in the document view. The hierarchy of the document is displayed in the Structure window, and is updated as the document is edited.

JDeveloper provides explorers for various common document types, and allows custom explorers to be installed and registered with the Explorer Manager. When a

document is opened in JDeveloper, the Explorer Manager provides an appropriate explorer for it, which then parses the document and displays the resulting structure.

An explorer addin has the following components:

- A class that performs the registration. This class implements `Addin`. A single instance of this class is created by the addin manager when JDeveloper is launched, and that instance performs the registration. The addin class does not need to have any other purpose.
- A class that provides the explorer's user interface. This class is a specialized viewer. When the document's structure is to be represented by a tree most of the required functionality can be provided by extending `TreeExplorer` or one of its subclasses. This class handles mouse clicks on elements, tracks changes in the editor, and invokes the parser to rebuild the tree when necessary.
- An element model, which contains one or more instances of `Element`. The model is the structural representation of the explorer view of a document. Elements are associated with specific locations in the content of the document, and when selected, scroll the viewer to that location.
- A parser that generates a structure of elements from the document. The parser reduces the document to a hierarchy of elements which map to viewer coordinates.

Keep these facts in mind while designing your explorer:

- An `Explorer` is a view which provides a visual, structured, representation of the data contained in a `Document`.
- Multiple explorer implementations may exist for a single document.
- Implement the `Explorer` interface to create a structure explorer.
- `ChildFilter` is an optional interface to use in conjunction with `TreeExplorer` to filter which children of a given container node are made visible to the user when the user expands that container node.

How to Register and Initialize a Structure Explorer

Explorers must be registered with the IDE's `ExplorerManager`. Registration associates an explorer with a node class, or with a node/viewer pair.

When the user selects a viewer for a node, the IDE sends a request to the `ExplorerManager` for the explorer that best satisfies that node and viewer. The resulting viewer will be one registered for that node and viewer, or else one registered for the node alone. If no such registrations have been made, ExplorerManager tries to find a viewer that is registered for a node class that converts to the requested node class.

Registration operations should be performed in an addin class' initialize method, as shown in the following example. The code example is taken from the StructurePane project, one of the sample projects available as part of the Extension SDK. For more information, see Developing with the Extension SDK .

```
public void initialize()
{
  Class editorClass = PropFileEditor.class;
  ExplorerManager.getExplorerManager().register(PropFileSourceNode.class,
                     PropFileExplorer.class,
```

```
                        PropFileEditor.class );
}
```

How to Create a Structure Explorer Element Model

The element model of an Explorer is the structural representation, illustrated graphically, of data in a given Document.

Each model is made up of one or more instances of `Element`. The model is typically rendered visually in the Explorer as a tree, but this is not a requirement. The element model is not a visual object, but can be rendered visually using graphical controls, for example, `javax.swing.tree.TreeModel` rendered by `javax.swing.JTree`.

Implementations of Explorer are responsible for ensuring that their model, and by extension their visual representation of that model, receive changes made to the data in the Document by any method (for example, in the source editor, Property Inspector, hand editing of the document outside of JDeveloper). You can use the JDeveloper event messaging mechanism to achieve this.

How to Update the Structure Explorer

A structure explorer should be notified whenever the content of the of the associated document is modified, so that it can update its state. Use the notification mechanism to transmit information about changes in the viewer to the explorer. The structure explorer should implement `Observer`, providing an update method, and register with the object representing the viewer's document.

How to Add New Components Window Pages

The Component Manager lets you add individual pages to the Components window that users of your extension can employ to build a consistent user interface or perform other standardized tasks. The Components window provides pages of components, from which your users can select the components to add to the content they are building with your extension, or for your extension.

The ability to create and manage component pages is also available programmatically, through the methods of the `PaletteManager` instance. For more information, see `PaletteManager` in the *Java API Reference for Oracle Extension SDK*.

This section includes examples for declaratively adding static and dynamic components to your Components window, as well as providing additional help and augmenting the user's ability to search for specific items in your individual component pages.

Understanding Components Window Pages

The Components window lets users add commonly used data structures to a project, application, or other content in JDeveloper. The Components window provides pages, from which users can select individual components to add to the content they are building with or for your extension. The specific components can be as simple as a copyright statement or as complicated as the settings for connecting to a remote repository. Furthermore, in JDeveloper, the available components from any given Components window page varies depending on the type of file selected. For example, if you are editing an HTML file, the Components window page might display a list of available HTML components such as anchors, email links, and other commonly-used

HTML components. If you are editing a Java page, a completely different set of components are at your disposal.

To see the Components window, select **View** > **Components**.

At the extension developer's level, the Components window provides you with a declarative way to add pages to the Components window, to make specific components available to users of your extension. These pages can offer whatever standard components (both static and dynamic) you wish to provide. For example, to help your users build a consistent user interface, or to perform any standardized tasks that draw on the components maintained by JDeveloper's Components manager, you can add one or more pages of components to the Components window, using the Component Manager.

When you develop extensions, you can declaratively populate your extension with component pages, static and dynamic components, and support for them. You can:

- Declare static components on a Components window.
- Declare dynamic components, populating the Components window when it is loaded.
- Help make your components available to your users.
- Make it easier for users to search for your components.

If you use these declarative tools, users of your extension can add commonly used elements to their own projects and applications using the extension you are creating. In addition, the help and search features makes it easier for your users to find and apply these Components window components.

You can also add component pages programmatically, using the Components Manager.

How to Declare Static Content for a Components Window

In many cases, the Components window's content is constant: a set of connection parameters to a source repository, or other fixed data that does not change regardless of where it is used in your extension. For cases like this, you can define static content in an extension manifest file, using `palette-hook`, as shown in the subsequent example.

The shape of the Components window is defined as a simple four-tier taxonomy:

- The top level, referred to as Page, groups components according to technology; Swing, AWT, ADF Swing, ADF Faces, Java Server Faces (JSF), JavaServer Pages (JSP), and so on. The combo box selections at the top of the Components window are where Pages display in the UI.
- Within each Page are Groups. Groups contain a small number of functional categories. For example, page Swing may have groups Common Controls and Layout. Groups are the dockable windows within the Components window.
- Within each Group are Sections, sections provide the means to cluster common components, display a separator line in the UI similar to menu separators, and alphabetically sort components by name.
- Finally, within each Section are Items (or components). Items are made up of attributes that enable components to be displayed by the Components window, for example name, description, icon, etc.

The following example, which is taken from the `CPPageProvider` project in the Extension SDK, demonstrates how to declare a Components window page with a single component. For more information about the Extension SDK, see Developing with the Extension SDK .

```
<?xml version="1.0" encoding="UTF-8" ?>
<extension xmlns="http://jcp.org/jsr/198/extension-manifest"
           id="oracle.ide.extsamples.cppageprovider"
           version="11.1.1"
           esdk-version="1.0">
  <name>ESDK Sample - Component Palette Page Provider</name>
  <owner>Oracle</owner>
  <feature-member id="esdk-samples"
 xmlns="http://xmlns.oracle.com/ide/extension"/>
  <hooks>
    <palette-hook xmlns="http://xmlns.oracle.com/jdeveloper/1013/extension">
      <pageProvider>
        <providerClassName>oracle.ide.extsamples.pageprovider.SamplePageProvider</
provider
ClassName>
      </pageProvider>
      <page>
        <name>Sample Components (ESDK Sample)</name>
        <pageId>SampleStatic</pageId>
        <showForTypes>
          <type>java</type>
        </showForTypes>
        <technologyScopes>
          <technologyScope>Java</technologyScope>
        </technologyScopes>
        <type>java</type>
        <group>
         <name>Components</name>
         <groupId>SampleStatic-Components</groupId>
          <showForTypes>
            <type>java</type>
          </showForTypes>
          <technologyScopes>
            <technologyScope>Java</technologyScope>
          </technologyScopes>
          <type>java</type>
          <section>
             <sectionId>SampleStatic-Components-Section1</sectionId>
             <name/>
             <item>
               <name>Table</name>
               <description>Sample Table</description>
               <icon>${OracleIcons.TABLE}</icon>
               <info/>
               <type>JavaBean</type>
               <itemId>SampleStatic-Components-Section1-Item1</itemId>
               <technologyScopes>
                 <technologyScope>Java</technologyScope>
               </technologyScopes>
             </item>
           </section>
        </group>
      </page>
    </palette-hook>
    <feature-hook>
      <part-of>oracle.ide.extsamples.allsamples</part-of>
```

```
      <description>Demonstrates how to integrate a component palette page provider
into the IDE.</description>
      <optional>true</optional>
    </feature-hook>
  </hooks>
</extension>
```

How to Declare a Dynamic Component for a Components Window Page

At times, the content of a Components window cannot be identified before loading. For the case when the Components window content can not be defined beforehand it is necessary to define a Components window Page Provider. This approach provides considerable flexibility to client developers in that the Components window provides the display context to the provider and the provider is given an opportunity to return components to be displayed in the Components window.

Components window page providers are developed by client developers using the published Components window API and the page providers class name is provided to the Components window in an extension manifest file.

For a description of the Components window API, see `oracle.ide.palette2` in the *Java API Reference for Oracle Extension SDK*.

The extension manifest entry uses hook's `element pageProvider`.

The following example shows an extension manifest that identifies a Components window Page Provider class name.

```
<?xml version="1.0" encoding="windows-1252" ?>
<extension xmlns="http://jcp.org/jsr/198/extension-manifest"
           id="oracle.ide.samples.pageprovider"
           version="1.0"
           esdk-version="1.0">
  <name>Component Palette Page Provider Sample</name>
  <owner>Oracle</owner>
  <dependencies>
    <import>oracle.ide.palette2
  </dependencies>
  <hooks>
    <palette-hook xmlns="http://xmlns.oracle.com/jdeveloper/1013/extension">
      <pageProvider>
        <providerClassName>oracle.ide.samples.pageprovider.SamplePageProvider</
providerClassName>
      </pageProvider>
    </palette-hook>
  </hooks>
</extension>
```

The subsequent example shows a Components window Page Provider that adds a Components window page to the Components window. This class extends `PalettePageProvider`.

The element `pageProvider.providerClassName` in `extension.xml` registers this class with the Components window as a page provider. The Components window calls `createPalettePages()` with the current Context.

The class `SamplePages`, which extends `PalettePages`, is constructed with the current context, means that it should have sufficient information to assemble a list of palette pages when `getPages()` is called by the Components window.

```
package oracle.ide.samples.pageprovider;
SamplePageProvider.java
import oracle.ide.Context;
import oracle.ide.palette2.PalettePageProvider;
import oracle.ide.palette2.PalettePages;
/**
 * SamplePageProvider
 * <p>
 * This is an example of a PalettePageProvider that adds a palette page to the
 * Component Palette(CP).  As required this class extends PalettePageProvider.
 * </p>
 * <p>
 * Element pageProvider.providerClassName in extension.xml registers
 * this class with the CP as a page provider.
 * </p>
 * <p>
 * The CP will call method createPalettePages() with the current Context.
 * Class SamplePages which extends PalettePages is constructed with the current
context.
 * Using the current context SamplePages should have sufficient information to
 * assemble a list of palette pages when method getPages() is called by the CP.
 * </p>
 *
 * @see SamplePages
 * @see SamplePalettePage
 * @see SamplePaletteGroup
 * @see SamplePaletteItem
 */
public class SamplePageProvider extends PalettePageProvider  {
   /**
     * Default constructor.
     *
     */
    public SamplePageProvider() {
    }
    /**
     * Override the default, returns SamplePages if context applies.
     *
     * @param context
     * @return PalettePages if context is relevant, otherwise null.
     */
    public PalettePages createPalettePages(Context context) {
        if ( checkRelevantContext( context ) )  {
          SamplePages pages = SamplePages.getInstance();
          pages.initialize(context);
          return pages;
        }
        else
          return null; // no pages to provide for this context.
    }
}
```

The following example shows an instance of `SamplePages.java`. This class creates a Components window page with a single group that contains a single section that contains a single item when the `pageType` is `java`. As required this class extends `PalettePages`.

```
package oracle.ide.samples.pageprovider;
import java.net.URL;
import java.util.ArrayList;
import java.util.Collection;
```

```
import java.util.Collections;
import java.util.List;
import oracle.ide.Context;
import oracle.ide.net.URLFileSystem;
import oracle.ide.palette2.DefaultPaletteSection;
import oracle.ide.palette2.PaletteItem;
import oracle.ide.palette2.PalettePage;
import oracle.ide.palette2.PalettePages;
import oracle.ide.palette2.PalettePagesListener;
/**
 * SamplePages
 * <p>
 * This class creates a palette page with a single group that contains a
 * single section that contains a single item when the pageType is "java".
 * As required this class extends PalettePages.
 * </p>
 * <p>
 * TODO: palettePagesListener
 * </p>
 */
public class SamplePages extends PalettePages {
    private final static String SAMPLEPROVIDER_ID
=SamplePageProvider.class.getName();
     /**
     * Singleton
     */
    private static SamplePages _singleton = new SamplePages();
   /**
     * Composite for PalettePagesListener
     */
    protected List<PalettePagesListener> palettePagesListeners;
    /**
    * Composite for PalettePage
    */
    protected List<PalettePage> palettePages;
    // Default constructor
    public SamplePages() {
    }
    /**
     * Returns the singleton instance of SamplePages.
     * @return the singleton instance of SamplePages
     */
    public static SamplePages getInstance() {
       return _singleton;
    }
    /**
     * Initialize palettePages.
     * @param context
     */
    public void initialize(Context context) {
        URL url = context.getNode().getURL();
        final String pageType = getSuffix( url );
    // Only interested in java.
      if( pageType.equals("java") ) {
          if( palettePages != null ) {
             palettePages.clear();
          }
      // create a PalettePage
      SamplePalettePage page = new SamplePalettePage
"oracle.ide.samples.pageprovider.SampPage01" // pageId
             , "My Sample Page Component"  // description
```

```
            , null                  // icon
            , "java"                // type
            , "java"                // showForTypes
            , "Java;JavaBeans" );   // technologyScope
      // create a PaletteGroup
      SamplePaletteGroup group = new SamplePaletteGroup
"oracle.ide.samples.pageprovider.SampGroup01" // groupId
           , "My Sample Group"         // name
           , "My Sample Group Component"  // description
           , "java" );                // type
      // add group to page
          page.addGroup(group);
      // create a section. Make the name null since a separator is not neeeded.
       DefaultPaletteSection section = new DefaultPaletteSection
"oracle.ide.samples.pageprovider.SampSection01"  // sectionId
      // add section to group
       group.addSection(section);
      // create an item. TODO: use SampleBean.java
       SamplePaletteItem item = new SamplePaletteItem
("oracle.ide.samples.pageprovider.SampItem01"  // itemId
           , SAMPLEPROVIDER_ID  // provider id
           , "My Sample Bean"    // name
           , "My Sample Bean Description"  // description
           , "/oracle/ide/samples/pageprovider/snapshot.png"   // icon
           , "java");           // type
      // add item to section
       section.addItem(item);  // add item to section
      // add page
      addPage(page);
     }
  else {
        palettePages.clear();
       }
   }
    /**
    * getPages
    */
    public Collection<PalettePage> getPages() {
     return Collections.unmodifiableList(palettePages);
}
/**
  * Returns the PaletteItem identified by itemId.  providerId is used to
  * determine whether this item is owned by this page provider.
  * The provider returns the matching PaletteItem only if the PaletteItem
  * is within the current context. Null is returned if the PaletteItem is
  * within the current context or not recognized by this provider.
  * </p>
  * @return PaletteItem
  */
 public PaletteItem getItem( String providerId, String itemId ) {
     if( providerId == null || providerId.length() == 0
         || itemId == null || itemId.length() == 0 ) {
          return null;
     }
       PaletteItem paletteItem = null;
       if( providerId.equals(SAMPLEPROVIDER_ID)) {
         for( PalettePage palettePage : palettePages ) {
           SamplePalettePage sampPage = (SamplePalettePage) palettePag
           paletteItem = sampPage.getItem(itemId);
           if( paletteItem != null ) {
               break;
```

```
            }
          }  // end of for
         }
          return paletteItem;
        }
        /**
         * addPalettePagesListener
        public void addPalettePagesListener(PalettePagesListener listener) {
           if( palettePagesListeners == null ) {
               palettePagesListeners = new ArrayList<PalettePagesListener>();
            }
            palettePagesListeners.add(listener)
          }
        /*
         * add pages to palettePages.
        /*
       private void addPage( SamplePalettePage sampPage ) {
           if (palettePages == null ) {
           palettePages = new ArrayList<PalettePage>();
           }
           palettePages.add(sampPage);
           }
           */
           * Return suffix
           * @param title Title of EditorFrame
           */
           private String getSuffix( URL url )
           (
             final String suffix = URLFileSystem.getSuffix( url );
             int period = suffix.lastIndexOf( "." );
             if( period != -1 )
             {
              // Is a selected file
              return suffix.substring( period + 1 );
               }
             return "";
           }
}
```

The following example shows an instance of `SamplePalettePage.java`.

```
package oracle.ide.samples.pageprovider;
import java.util.ArrayList;
import java.util.List;
import java.util.StringTokenizer;
import oracle.ide.palette2.DefaultPalettePage;
/**
 * SamplePalettePage
 * <p>
 * This class extends DefaultPalettePage. DefaultPalettePage comprises
 * the attributes and methods that this example requires. A constructor is added
 * to accomodate the data values for this sample.
 * </p>
 */
public class SamplePalettePage extends DefaultPalettePage {
    public SamplePalettePage(String pageId, String pageName, String pageDescription
             , String pageIcon, String pageType, String pageShowForTypes
             , String pageTechnologyScope) {
      setName( pageName );
      setDescription( pageDescription );
      setIcon( pageIcon );
```

```
      setData(PAGE_PAGEID, pageId );
      setData(PAGE_TYPE, pageType);
     // make delimited string into a List of strings
     List<String> showForTypes = new ArrayList<String>();
     final StringTokenizer tknTypes = new StringTokenizer pageShowForTypes, ";" ); //
NOTRANS
      while( tknTypes.hasMoreTokens() )
      {
         String token = (String)tknScopes.nextToken();
         technologyScope.add(token);
      }
    setData(PAGE_TECHNOLOGYSCOPES, technologyScope );
  }
}

package oracle.ide.samples.pageprovider;
import oracle.ide.palette2.DefaultPaletteGroup;
/**
 * SamplePaletteGroup
 * <p>
 * This sample extends DefaultPaletteGroup.DefaultPaletteGroup comprises the
 * attributes and methods that this example requires. A constructor is added to
 * accomodate the data values for this sample.
 * </p>
 *
 */
public class SamplePaletteGroup extends DefaultPaletteGroup {
    public SamplePaletteGroup(String groupId, String groupName, String
groupDescription, String groupType) {
       setName( groupName );
       setDescription( groupDescription );
       setData(GROUP_GROUPID, groupId);
       setData(GROUP_TYPE, groupType);
     }
}
```

The following example shows an instance of `SamplePaletteItem.java`.

```
package oracle.ide.samples.pageprovider;
import oracle.ide.palette2.DefaultPaletteItem;
 * SamplePaletteItem
 * <p>
 * This example extends DefaultPaletteItem. DefaultPaletteItem comprises
 * the attributes  and methods that this example requires. A constructor is
 * added to accommodate the data values for this sample.
 * </p>
 *
 */
public class SamplePaletteItem extends DefaultPaletteItem
 {
   public SamplePaletteItem(String itemId, String itemProviderId, String itemName,
String itemDescription,
 String itemIcon, String itemType) {
    setName( itemName );
    setDescription( itemDescription );
    setIcon( itemIcon );
    setItemId( itemId );
    setProviderId(itemProviderId);
    setData(ITEM_TYPE, itemType);
  )
)
```

How to Provide Help for a Components Window Page

There are two ways to provide help on a component:

- The Components window item description is displayed as a tool tip for the component, visible when your user rolls the mouse over the component in the Components window. You can use the "\n" newline character to break the description into multiple lines.
- The user can right-click a Components window component and get help from its context menu. The Help option appears on the context menu for the component only if a Helpable is available for the component. If the helpable class cannot be instantiated, the string is used as the helpId for the IDE help system.

The following example demonstrates how to do this.

```
  <item>
    <name>Table</name>
    <description>Sample Table</description>
    <icon>/oracle/ide/samples/pageprovider/table.png</icon
    <info/>
    <type>JavaBean</type>
    <itemId>SampleStatic-Components-Section1-Item1</itemId>
    <technologyScopes>
      <technologyScopes>
      <technologyScope>JavaBeans</technologyScope>
     </technologyScopes>
    <helpable>oracle.samples.SampleHelpable</helpable>
 </item>
```

Alternatively, you can provide the helpid directly, as shown in the following example:

```
 <item>
   <name>Table</name>
   <description>Sample Table</description>
   <icon>/oracle/ide/samples/pageprovider/table.png</icon>
   <info/>
   <type>JavaBean</type>
   <itemId>SampleStatic-Components-Section1-Item1</itemId>
   <technologyScopes>
     <technologyScope>Java</technologyScope>
     <technologyScope>JavaBeans</technologyScope>
   </technologyScopes>
  <helpable>help_topic_id</helpable>
</item>
```

The programmatic equivalent of this would be to implement `getHelpable()` in the `PaletteItem` interface.

How to Augment the Search for a Components Window Item

By default, when the user types a word or phrase into the Components window search box, JDeveloper looks for items that have the search string in their Name or Description.

To make the Components window search on other embedded data that is not visible to the user (such as tag attributes), a `searchTextContext` can be provided along with the item.

Declaratively, you indicate this in the Components window extension hook as shown in the following example:

```
  <item>
    <name>Table</name>
    <description>Sample Table</description>
    <icon>/oracle/ide/samples/pageprovider/table.png</icon>
    <info/>
    <type>JavaBean</type>
    <itemId>SampleStatic-Components-Section1-Item1</itemId>
    <technologyScopes>
       <technologyScope>Java</technologyScope>
       <technologyScope>JavaBeans</technologyScope>
    </technologyScopes>
    <searchTextContext>MySearchContext</searchTextContext>
 </item>
```

`SearchContext` then implements the `PaletteSearch` interface, as shown in the following example:

```
class MySearchContext implements PaletteSearch
 {
   boolean searchItemContainsText(String itemId, String searchString)
   {
     boolean found = false;
    // Search internal data
    return found;
   }
}
```

Understanding Preferences

Making a way for your users to modify and store their preferences adds productivity and flexibility to the process of using your JDeveloper extensions.

How to Implement Preferences

You manage product-level preferences with the `Preferences` class. For more information, see `Preferences` in the *Java API Reference for Oracle Extension SDK*.

The data structure used to store preferences is `HashStructure`. The `Preferences` class saves preferences in only one extension directory—the extension that represents the entire product. In JDeveloper, for example, the product directory follows the format `jdev-user-directory/system/oracle.JDeveloper.12.1.3.x.y`, where "12.1.3" represents the version number and "x.y" represents the build number. The file holding the preferences is named `preferences.xml`.

To incorporate a new set of preferences into the IDE:

1. Implement the data model for storing the preferences. For more information, see How to Implement the Data Model.
2. Implement the data model for storing the preference. For more information, see How to Implement a UI Panel.
3. Register the UI panel with the Preferences dialog, which is available from the Tools menu. For more information, see How to Register a UI Panel.
4. Obtain the preferences model. For more information, see How to Obtain the Preference Model.

5. Listen for changes to the preferences, so they can be stored. For more information, see How to Listen for Changes.

How to Implement the Data Model

The class that represents your preferences data should be a subclass of `HashStructureAdapter`. For more information, see `HashStructureAdapter` in the *Java API Reference for Oracle Extension SDK*.

The following example contains a typical implementation pattern, with comments for the details.

```
package oracle.killerapp.coolfeature;
import oracle.javatools.data.HashStructure;
import oracle.javatools.data.;
import oracle.javatools.data.PropertyStorage;
// Start with class being final.  You can always remove final if subclassing ever
// proves useful.  In many cases, subclassing is actually unnecessary and may get
// you into an instanceof/typecast mess.  Consider defining a separate (not
// subclass) adapter class instead.
public final class CoolFeaturePrefs extends
{
// The DATA_KEY should be a hard-coded String to guarantee that its value stays
 // constant across releases.  Specifically, do NOT
 // constant across releases.  Specifically, do NOT use
CoolFeaturePrefs.class.getName().
 // The reason is that if CoolFeaturePrefs is ever renamed or moved,
 // CoolFeaturePrefs.class.getName() will cause the DATA_KEY String to
change,which
 // introduces a preferences migration issue (since this key is used in the persisted
 // XML) that will require more code and testing to accommodate and open up your
code to
 // annoying little bugs.  Unknowing developers have been trapped by this problem
before,
 // so eliminate this cause of bugs by using a hard-coded String for DATA_KEY.
 //
 // By convention, DATA_KEY should be the fully qualified class name of the
 // .  This helps ensure against name collisions. This also makes it
 // easier to identify what piece of code is responsible for a preference when you're
 // looking at the XML in the product-preferences.xml file.  Of course, that only
works
 // as long as the adapter class itself is never renamed or moved, so avoid renaming
or
 // moving this class once it's been released in production.
  private static final String DATA_KEY =
"oracle.killerapp.coolfeature.CoolFeaturePrefs";
 // Private constructor enforces use of the public factory method below.
 private CoolFeaturePrefs(HashStructure hash)
{
   super(hash);
}

// Factory method should take a PropertyStorage (instead of HashStructure directly).
// This decouples the origin of the HashStructure and allows the future possibility
// of resolving preferences through multiple layers of HashStructure. Classes/methods
// that currently implement/return PropertyStorage:
// - oracle.ide.config.Preferences
// - oracle.ide.model.Project
// - oracle.ide.model.Workspace
// - oracle.ide.panels.TraversableContext.getPropertyStorage()
```

```
 public static CoolFeaturePrefs getInstance(PropertyStorage prefs)
{
  // findOrCreate makes sure the HashStructure is not null.  If it is null, a
  // new empty HashStructure is created and the default property values will
  // be determined by the getters below.
   return new CoolFeaturePrefs(findOrCreate(prefs, DATA_KEY));
}
-----------------------
// Like DATA_KEY, all other keys also appear in the XML, so they should not be
// changed once released into production, or else you'll have some migration  issues
// to fix
private static final String MAX_NUMBER_OF_THINGIES = "maxNumberOfThingies";  //
NOTRANS
private static final int DEFAULT_MAX_NUMBER_OF_THINGIES = 17;

public int getMaxNumberOfThingies()
{
  // Specify default in the getInt call to take advantage of HashStructure's
  // placeholder mechanism.  See HashStructure javadoc for details on placeholders.
  return _hash.getInt(MAX_NUMBER_OF_THINGIES, DEFAULT_MAX_NUMBER_OF_THINGIES);
}
 public void setMaxNumberOfThingies(int maxNumberOfThingies)
{
 _hash.putInt(MAX_NUMBER_OF_THINGIES, maxNumberOfThingies);
}
//-----------------------------------------
 private static final String THINGIE_NAME = "thingieName";   //NOTRANS
 private static final String DEFAULT_THINGIE_NAME = "widget";  //NOTRANS
 public String getThingieName()
{
 return _hash.getString(THINGIE_NAME, DEFAULT_THINGIE_NAME);
}
 public void setThingieName(String thingieName)
{
 return _hash.putString(THINGIE_NAME, thingieName);
}
 // etc..
}
```

How to Implement a UI Panel

The class that implements your preferences panel should be a subclass of `DefaultTraversablePanel`. For more information, see `DefaultTraversablePanel` in the *Java API Reference for Oracle Extension SDK*.

The following example shows a typical implementation pattern.

```
package oracle.killerapp.coolfeature;
import oracle.ide.panels.DefaultTraversablePanel;
// You should keep the panel class package-private and final unless there
// is a good reason to open it up.  In general, preferences panels are not
// supposed to be part of a published API, so the class modifiers should
// enforce that.
final class CoolFeaturePrefsPanel extends DefaultTraversablePanel
{
 // But, the no-arg constructor still needs to be public.
 public CoolFeaturePrefsPanel()
  {
   // Layout the controls on this panel.
  }
  public void onEntry(TraversableContext tc)
```

```
 {
   final CoolFeaturePrefs prefs = getCoolFeaturePrefs(tc);
  // Load prefs into the panel controls' states.
 }
   public void onExit(TraversableContext tc)
 {
  final CoolFeaturePrefs prefs = getCoolFeaturePrefs(tc);
 // Save the panel controls' states to prefs.
}
  private static CoolFeaturePrefs getCoolFeaturePrefs(TraversableContext tc)
{
  // If you've implemented CoolFeaturePrefs according to the typica
  // implementation pattern given above, this is how you attach the
  // adapter class to the defensive copy of the preferences being
  // edited by the Tools->Preferences dialog.
  return CoolFeaturePrefs.getInstance(tc.getPropertyStorage());
 }
}
```

How to Register a UI Panel

The following example contains an XML fragment for the extension manifest (extension.xml) that registers the panel shown above with the Preferences dialog, which is available from the Tools menu.

```
<extension ...>
  <hooks>
    <settings-ui-hook xmlns="http://xmlns.oracle.com/ide/extension">
      <page id="CoolFeaturePrefs" parent-idref="/preferences">
         <label>${SOME_RES_KEY}</label>
           <traversable-class>oracle.killerapp.coolfeature.CoolFeaturePrefsPanel
           </traversable-class>
         </page>
     </settings-ui-hook>
   </hooks>
</extension>
```

How to Obtain the Preference Model

Use the `oracle.ide.config.Preferences` class to obtain preferences if you need to read or write preferences in code other than preference dialog code. Do not use this technique in preference pages. Any changes you make to the preference object using this code takes immediate effect, which makes it unsuitable for the Preferences dialog, which should always be cancelable.

```
Preferences p = oracle.ide.config.Preferences.getPreferences();
CoolFeaturePrefs myPrefs = CoolFeaturePrefs.getInstance( p );
```

How to Listen for Changes

You can listen for changes to preferences by attaching an `oracle.javatools.data.StructureChangeListener` to the hash structure underlying your preferences model object. Usually, a good approach is to expose methods for attaching a listener to your model object as shown in the following example:

```
public final class CoolFeaturePrefs extends
{
 //...
 public void addStructureChangeListener( StructureChangeListener l )
```

```
  {
    _hash.addStructureChangeListener( l );
  }
   public void removeStructureChangeListener( StructureChangeListener l )
  {
    _hash.removeStructureChangeListener( l );
   }
 //...
}
```

Understanding Project Properties

Project properties are similar to project preferences, see Understanding Preferences.

You manage project properties with the `Project` class. For more information, see `Project` in the *Java API Reference for Oracle Extension SDK*.

How to Share Project Properties

When working on an application, it is beneficial to have projects within an application share property settings. Sharing project properties can be set up either through the Run/Debug configuration or by customizing an project template by adding a project-creation listener, which is called whenever that template is used to create a new project. It allows the extension writer to modify the project's run configurations and other properties. This section describes setting the Run/Debug configuration to enable the use of common settings among projects within an application. For a discussion on adding a project-creation listener to a template, see TBD.

You can configure a project to use a shared run configuration at one of the following times:

- When you create the project through the creation listener, which is called when a project is created with a project template
- When a technology is added to the project
- During project migration

JDeveloper provides the `SharedProjectPropertiesManager` API that you can add to a run configuration file to enable sharing run configuration properties:

- `SharedPropertiesManager` - Manages shared properties.
- `SharedPropertiesManager getInstance()` - Obtains an instance the shared property.
- `HashStructure getProperties(Project` *project*, `String` *propertyDataKey*`)` - Returns the appropriate properties hash for either a workspace or project.
- `propertiesAreShareable(String)` - Specifies the project properties to share.
- `isUsingSharedProperties(Project, String)`- Checks if the project uses the specified shared properties.
- `setUsesSharedProperties(Project, String, boolean)` - Sets the shared properties as being used by the project or not.

Use `SharedProjectPropertiesManager` to configure a project to use shared run configuration properties, as shown in the following example:

```
if
(SharedProjectPropertiesManager.getInstance().propertiesAreShareable( RunConfiguratio
ns.DATA_KEY ))
{
  Project project = ...;
  SharedProjectPropertiesManager.getInstance().setUsesSharedProperty( project,
RunConfigurations.DATA_KEY, true );
}
```

Alternatively, you can set a project to use shared run configuration properties through the IDE.

To Use A Shared Run Configuration:

1. Right-click the project in the Projects window and select **Project Properties**.
2. Select **Run/Debug** in the Project Properties dialog.
3. Select **Use Shared Settings** on the Run/Debug page.

   When you select this option, the shared run configuration is stored in the application (the `.jws` file). Any project within the application can use these property settings.
4. Select the run configuration to use in the Run Configurations list.

   You can add new shareable run configurations or edit existing ones by clicking **Edit Shared Settings** (this option is only available if **Use Shared Settings** is selected).

How to Make Changes Undoable

Users who follow the rapid, iterative model of application development rely on the ability to easily back out changes as they move from solution to solution. Adding the ability to make your changes undoable provides this functionality for your users.

How to Make Text Changes Undoable

If your extension uses any form of text entry—for example, if you are implementing a custom text editor with specific features used by your organization—your users will expect to be able to undo changes they make while entering text. `UndoableEdit` allows your extension to do and undo the changes made to text by the user, as shown in the following example:

```
import javax.swing.undo.UndoableEdit;
import oracle.ide.Context;
import oracle.ide.controller.Command;
import oracle.ide.controller.CommandProcessor;
import oracle.ide.model.TextNode;
import oracle.javatools.buffer.TextBuffer;
public class MyCommand extends Command
{
   private UndoableEdit _undoableEdit;
   public MyCommand(Context context)
  {
    super(-1, Command.NORMAL, "Insert Hello");
    setContext(context);
  }
  public int doit() throws Exception
  {
```

```
    if (_undoableEdit == null)
    {
       final TextNode textNode = (TextNode) context.getNode();
       final TextBuffer textBuffer = textNode.acquireTextBuffer();
         textBuffer.beginEdit();
         textBuffer.insert(0, "Hello World".toCharArray());
         _undoableEdit = textBuffer.endEdit();
         textNode.releaseTextBuffer();
      } else
      {
        _undoableEdit.redo();
      }
      return OK;
      }
      public int undo() throws Exception
      {
       _undoableEdit.undo();
       return OK;
   }
}
```

How to Make Commands Undoable

If you want to make undoable changes to a document, you have to implement a command that knows how to do and undo the changes, as shown in the following example:

```
import oracle.ide.controller.Command;
import oracle.ide.Context;
import oracle.ide.model.Node;
public class MyCommand extends Command
{
  public MyCommand(Context context, Node affectedNode)
  {
    super(-1, Command.NORMAL, "My Changes");
   // The context usually already contains the node so this would not be necessary
   final Context contextCopy = new Context(context);
   contextCopy.setNode(affectedNode);
   setContext(contextCopy);
 }
   public int doit() throws Exception
{
    final Node affectedNode = context.getNode();
    // Do the changes to the node here
    return OK;
 }
public int undo() throws Exception
   final Node affectedNode = context.getNode();
   // Undo the changes to the node here
  return OK;
 }
}
```

Defining and Using Trigger Hooks

Use trigger hooks to control when JDeveloper loads the extension.

The `<trigger-hooks>` element is in the `http://xmlns.oracle.com/ide/extension` namespace.

The `<trigger-hooks>` element contains three child elements:

- `<registry>` - used to register trigger hook handlers
- `<triggers>` - where all trigger hooks are placed
- `<rules>` - defines conditions that can be used to conditionally execute a set of triggers

The IDE provided trigger-hooks are:

- Actions
- Controllers
- Menus
- Context Menus
- Feature Hook
- Editor Menu
- Accelerators/Shortcut Keys
- Gallery Items
- Technology Scopes
- Editors
- NodeFactory Recognizers
- IDE Preferences/Settings
- Application Preferences/Settings
- Project Preferences/Settings
- Content Set Providers
- Singleton Registration
- Application and Project Migrators
- URLFileSystem Hook
- ImportExport Hook
- Menu Customizations Hook
- Bridge Extensions Hook
- On Project Open Hook
- Help Callbacks Hook
- Help Hook
- Dockable Hook
- Historian Hook
- External Tools Hook

How to Register a <trigger-hook-handler>

To define your own trigger-hook, you register a trigger hook handler as shown in the following example:

```
<trigger-hooks xmlns="http//xmlns.oracle.com/ide/extension/myExtension">
    <registry>
      <trigger-hook-handler
        tag-name="my-hook"
        handler-class="oracle.ide.extension.HashStructureHook"
        namespace="http://xmlns.oracle.com/ide/extension"
        schema-location="my-hook.xsd"
        register-as-hook="true"/>
    </registry>
  </trigger-hooks>
```

The `register-as-hook` attribute controls whether or not the extension hook is also allowed in the <hooks> section. The default value is `false`; only set it to `true` if it makes sense to support the same element in <hooks>.

If you have an existing declarative hook and you want it to now be a trigger hook, you need to remove your current hook handler registration and instead register a trigger-hook-handler.

If you are updating an existing extension built to previous standards used by JDeveloper, for backward compatibility, you can keep the same namespace you used. If your existing hook is a child of the old `jdeveloper-hook` and you would also like to support it as a trigger hook, set the attribute `register-as-jdeveloper-hook` to `true`.

Extensions outside of the core IDE must use a `hook-handler` class that is part of the core IDE. There are two options you can use:

- `oracle.ide.extension.HashStructureHook` as the handler-class. If your trigger-hook is also in <hooks>, or if it can be used in conditional trigger sections, then new elements are processed by the `HashStructureHook` after you observe the `HashStructure`. In this situation, your code must listen to the `HashStructureHook` events.
- `DeferredElementVisitorHook`, allows you to use a custom `ElementVisitor` class to process your hook's XML data, so if you have already written a custom hook handler you can reuse your code.

How to Define Trigger Hooks for your Extension

To use a trigger hook, use syntax similar to the following example:

```
<triggers>
    <singleton-provider-hook>
      <singleton base-class="oracle.bali.xml.addin.JDevXmlContextFactory"
                 impl-class="oracle.bali.xml.addin.JDevJavaXmlContextFactory" />
    </singleton-provider-hook>
  </triggers>
</trigger-hooks>
```

How to Retrieve Parsed Information from the ExtensionRegistry

When you define a `trigger-hook-handler` for an element name, a single instance of that `ExtensionHook` handler class is created that processes all the usages of that trigger hook.

To retrieve that `ExtensionHook` instance, call `ExtensionRegistry.getHook(elementName)`. In earlier versions of JDeveloper extension development, this API was used to retrieve the `ExtensionHook` for a regular

declarative hook. All the processed trigger hooks and all the currently loaded hooks sections contribute to the same bucket of `ExtensionHook` instances.

If your `trigger-hook-handler` is `HashStructureHook`, you can retrieve the `HashStructure` from the returned `HashStructureHook` instance. It contains the XML information for all the usages of your element name in the `HashStructure`, as well as stored information about what extensions the different sections of the `HashStructure` came from. You can use a `HashStructureAdapter` subclass to pull information out of the `HashStructure`.

How to Define Rules and Condition Triggers Sections

This section describes how to define rules and conditions in the extension manifest.

How to Define Rules

Rules and rule-types are defined in the `<rules>` section of `<trigger-hooks>` in the extension manifest, `extension.xml`.

A `rule-type` represents a rule function implemented in Java and describes the parameters accepted by that function (such as a method signature). There is a set of built-in rule-types provided by the IDE.

A rule represents a call to a rule function passing specific values for the parameters. Each rule is given a globally unique id, and then referenced by id from the hooks that support rules.

The rule framework is `oracle.ide.extension.rules` in the IDE module.

To define a rule type you must supply:

- An id by which it is referenced,
- An implementation class which is a subclass of `oracle.ide.extension.rules.RuleFunction`, and list the supported parameters. You can indicate whether a parameter is optional or required.

The IDE module's `extension.xml` defines several built-in rule types, illustrated in the following example:

```
<trigger-hooks xmlns="http://xmlns.oracle.com/ide/extension">
  <rules>
    <rule-type id="always-enabled"
class="oracle.ide.extension.rules.functions.AlwaysEnabled" />

    <rule-type id="any-selection-has-attribute"
class="oracle.ide.extension.rules.functions.AnySelectionHasAttribute">
      <supported-parameters>
        <param name="element-attribute" required="true"/>
      </supported-parameters>
    </rule-type>
    <rule-type id="context-has-element"
class="oracle.ide.extension.rules.functions.ContextHasElement">
      <supported-parameters>
        <param name="element-class" required="true"/>
      </supported-parameters>
    </rule-type>
    <rule-type id="context-has-node"
class="oracle.ide.extension.rules.functions.ContextHasNode">
```

```
      <supported-parameters>
        <param name="node-class" required="true"/>
      </supported-parameters>
    </rule-type>
    <rule-type id="context-has-project"
class="oracle.ide.extension.rules.functions.ContextHasProject" />
   <rule-type id="context-has-view"
class="oracle.ide.extension.rules.functions.ContextHasView">
      <supported-parameters>
        <param name="view-class" required="true"/>
      </supported-parameters>
    </rule-type>
    <rule-type id="context-has-workspace"
class="oracle.ide.extension.rules.functions.ContextHasWorkspace" />
    <rule-type id="element-has-attribute"
class="oracle.ide.extension.rules.functions.ElementHasAttribute">
      <supported-parameters>
        <param name="element-attribute" required="true"/>
      </supported-parameters>
    </rule-type>
    <rule-type id="on-extension-init"
class="oracle.ide.extension.rules.functions.ExtensionInitialized">
      <supported-parameters>
        <param name="extension-id" required="true"/>
      </supported-parameters>
    </rule-type>
    <rule-type id="extension-is-enabled"
class="oracle.ide.extension.rules.functions.ExtensionEnabled">
      <supported-parameters>
        <param name="extension-id" required="true"/>
      </supported-parameters>
    </rule-type>
    <rule-type id="on-single-selection"
class="oracle.ide.extension.rules.functions.SingleSelection">
      <supported-parameters>
        <param name="element-class" required="false"/>
      </supported-parameters>
    </rule-type>
    <rule-type id="on-multiple-selection"
class="oracle.ide.extension.rules.functions.MultipleSelection">
      <supported-parameters>
        <param name="element-class" required="false"/>
      </supported-parameters>
    </rule-type>
    <rule-type id="node-is-dirty"
class="oracle.ide.extension.rules.functions.NodeIsDirty" />
    <rule-type id="project-has-techscope"
class="oracle.ide.extension.rules.functions.ProjectHasTechScope">
      <supported-parameters>
        <!-- Comma-separated list of technology keys -->
        <param name="technology-keys" required="true" />
        <!-- Specify 'all' or 'any' for match, to specify if all keys should exist
or any one key suffices -->
        <param name="match" required="false" />
      </supported-parameters>
    </rule-type>
  </rules>
</trigger-hooks>
```

The handler for <rule-type> stores this information for use during parsing of <rule> and the runtime evaluation of rules. The rule class is not loaded until the last possible moment, when a rule needs to be evaluated.

You can introduce your own rule types however there are some restrictions that are important to understand:

- The IDE never loads a rule type class from an extension that is not fully loaded.
- Rule evaluation never triggers loading an extension.

Consider the case that extension E1 introduces R1-rule-type and T1-trigger-hook. Imagine that T1-trigger-hook supports using rules. Only when extension E1 is fully loaded will it attempt to consume the data from T1-trigger-hook and evaluate any rule referenced within, so it is perfectly correct to use rules of R1-rule-type with T1-trigger-hook.

However, if you try to use a rule of type R1-rule-type with an ide-core trigger hook such as gallery, there is no guarantee that extension E1 would be loaded when that rule was evaluated, in which case an error is logged.

How to Define Simple Rules

A rule is defined in the <rules> section of <trigger-hooks>. To define a rule type you must supply an id by which it is referenced, a type attribute that identifies the rule-type, and values for any parameters required by the rule-type.

The handler for <rule> verifies that, as shown in the subsequent example.

- The id is unique.
- The value of type matches the id of a rule-type defined in this extension.xml (or in the extension.xml of a dependency).
- All required parameters have values.
- All provided parameters match parameter names defined by the rule type.

```
<rules>
  <rule id="context-has-text-node" type="context-has-node">
    <parameters>
      <param name="node-class" value="oracle.ide.model.TextNode" />
    </parameters>
  </rule>
  <rule id="context-has-source-node-1" type="context-has-node">
    <parameters>
      <param name="node-class" value="org.product.SourceNode1" />
    </parameters>
  </rule>
 <rule id="context-has-source-node-2" type="context-has-node">
    <parameters>
      <param name="node-class" value="org.product.SourceNode2" />
    </parameters>
  </rule>
  <rule id="on-xxx-init" type="on-extension-init">
    <parameters>
      <param name="extension-id" value="org.product.MyXxxExtension"/>
    </parameters>
  </rule>
 <rule id="on-yyy-init" type="on-extension-init">
    <parameters>
      <param name="extension-id" value="org.product.MyYyyExtension"/>
```

```
      </parameters>
    </rule>
    <rule id="on-text-node-single-selection" type="on-single-selection">
      <parameters>
        <param name="element-class" value="oracle.ide.model.TextNode" />
      </parameters>
    </rule>
  </rules>
</trigger-hooks>
```

Implicitly Available Rules

Each rule-type that has no required parameters is automatically registered as a rule (using the rule-type ID as the rule ID). For example, `context-has-project` and `node-is-dirty` are examples of rule-types with no required parameters, and those ids can be used anywhere a rule is referenced without needing to explicitly add a `<rule>` to an `extension.xml`.

Guidelines for Rules

You can avoid ID duplication and maximize reuse by carefully choosing which `extension.xml` a rule should go into, and the rule name.

The simple rule of thumb is find the extension that contains the class name you are passing as a parameter, and put it in that extension's `extension.xml`. To define the `context-has-source-node-1` rules in the example in "How to Define Simple Rules" find the extension that contains the `SourceNode1` class and look to see if there is an existing rule that meets your needs. If there is not, add the rule to that `extension.xml`.

The ID should be descriptive and based on the rule-type ID, for example:

- context-has-xxx-node
- context-has-xxx-node
- context-has-xxx-element
- context-has-xxx-view
- on-xxx-single-selection
- on-xxx-init

How to Define Composite Rules

You can define a rule that is composed of other rules and boolean operators. The supported boolean operators are <or>, <and>, and <not>. The <and> and <or> boolean operator elements accept any number of children, and the <not> element accepts one child. Each child of a boolean operator must be either a rule-reference or another boolean operator, as illustrated in the following example:

```
<trigger-hooks xmlns="http://xmlns.oracle.com/ide/extension">
  <rules>
    <composite-rule id="context-has-xxx-or-yyy-node">
      <or>
        <rule-reference id="context-has-jsp-node" />
        <rule-reference id="context-has-zzz-node" />
      </or>
    </composite-rule>
```

```
   <composite-rule id="on-aaa-and-bbb-init">
      <and>
        <rule-reference id="on-aaa-init" />
        <rule-reference id="on-bbb-init /">
      </and>
    </composite-rule>
   <composite-rule id="more-complicated-composite-rule">
      <or>
        <rule-reference id="rule-a" />
        <and>
          <rule-reference id="rule-b" />
          <not>
            <rule-reference id="rule-c" />
          </not>
       </and>
     </or>
    </composite-rule>
  </rules>
</trigger-hooks>
```

How to Reference Rules From Hooks

All rules are defined in the `<rules>` section of an `extension.xml` and then referenced by id in hooks that support rules.

A trigger hook that wants to support rules only needs to change its syntax to accept the id of the rule. By convention, the rule id should be supplied in an attribute named "rule". These are illustrated in the three subsequent examples:

Example 1

```
  ...
  <item rule="always-enabled">
    <name>oracle.jdeveloper.template.wizard.TemplateWizard</name>
    <id>Application</id>
    <description>${NEW_APPLICATION_TEMPLATE_GALLERY_ITEM}</description>
    <help>${MANAGE_TEMPLATES_WIZARD_DESCRIPTION}</help>
    <folder>Applications</folder>
    <technologyKey>General</technologyKey>
    <icon>/oracle/javatools/icons/apptemplate.jpg</icon>
  </item>
</gallery>
```

Example 2

```
<controllers xmlns="http://xmlns.oracle.com/ide/extension">
  <controller class="com.random.FooController">
    <update-rules>
      <update-rule rule="on-text-node-selected">
        <action id="some.action.id1">
          <label>Perform Action on {0}</label>
          <label-param>${node.name}</label-param>
        </action>
      </update-rule>
    </update-rules>
  </controller>
  <controller class="com.random.BarController">
    <update-rules>
      <update-rule rule="on-jsp-node-selected">
        <action id="some.action.id3" />
```

```
        <action id="some.action.id4" />
        <action id="some.action.id5" />
      </update-rule>
    </update-rules>
  </controller>
</controllers>
```

Example 3

```
<context-menu-hook rule="context-has-xxx-or-yyy-node">
  <site ref-id="navigator"/>
  <menu>
    <section xmlns="http://jcp.org/jsr/198/extension-manifest"
id="MY_CUSTOM_MENU_SECTION_ONE" weight="1.0">
      <item action-ref="myTriggerActionId" weight="1.0" />
      <item action-ref="myOtherTriggerActionId" weight="2.0" />
     </section>
   </menu>
 </context-menu-hook>
```

How to Validate Rule References and Evaluate Rules

If you are implementing a trigger hook, and you'd like to support rules in your hook, in your XML syntax add a rule attribute that accepts the ID of a rule (the XML Schema type for the rule attribute should be `xsd:NCName`).

In your hook handler implementation, when you retrieve the value of the rule attribute, you should call the following method on RuleEngine to do parsing-time validation of the rule reference:

```
public static boolean validateRuleReference(String id, Set<String>
expectedRuleTypes, ElementContext referenceContext
```

This method validates that there is a known rule with that ID (in the same `extension.xml` or in a dependency), and if you pass the optional `expectedRuleTypes` it validates that the rule is one of those types (if it is a composite rule all the particles must be of the expected types). If the optional `referenceContext` parameter is passed, it logs all the problems to the `ElementContext`'s logger.

When it was time to actually evaluate the rule, you would pass the rule id to a method on the Rule Engine along with an IDE Context (if available). The Rule Engine instantiates the rule-type class, pass the parameters from the rule definition, along with the context, and return true or false.

```
public static boolean evaluateRule(String id, Context ideContext)
```

The first method logs any exceptions that occur during rule evaluation and returns false if there were any problems evaluating the rule. The latter method also logs any exceptions, but throws the exception to the caller (in case the caller wants to take a different action).

Adding Online Help Support

Create JDeveloper extensions that have online help.

The help system used by JDeveloper provides context-sensitive F1 topics integrated into the structure of the IDE, where users access help by clicking the Help button in a wizard panel or a dialog, or by pressing the F1 key.

You can provide similar context-sensitive F1 help topics for your extension to make it easier for the users of your extension to learn about it and to use it effectively.

How to Create the Help System

You create a help system by providing a helpset, which is a Java archive, containing a number of HTML help topics.

For information about creating a help system, see *Developing Help Systems with Oracle Help*. For additional resources and to download the Oracle Help for Java Developer's Kit, see http://www.oracle.com/technetwork/developer-tools/help/index-083946.html.

To create the help system

1. For each panel of a wizard, or each window or editor, create an HTML topic file describing the panel.
2. Create a map control file for the helpset.
3. Create a helpset (`.HS`) control file.
4. Assemble the topic and control files into a Java archive. The archive must have the same root name as the HS file, and the HS file must be placed in the archive's root directory.
5. Make an association between a component and a help topic. For each panel, provide a call to `registerTopic` to register the topic ID. For example, if `panel` is the name of a wizard panel, and `topic` is the topic ID for its documentation, the call is:

   ```
   HelpSystem.getHelpSystem().registerTopic((JComponent)panel, topic);
   ```

6. Include the helpset's Java archive in the extension for deployment, so that when the extension is loaded the help topics are available for users.

How to Register the Help System

The help system for the extension should be available to users before the extension has loaded so that they can find out about an area of functionality. Therefore the help system has to be registered as shown in the following example:

```
<trigger-hooks xmlns="http://xmlns.oracle.com/ide/extension">
  <triggers>
    <help xmlns="http://xmlns.oracle.com/jdeveloper/1013/extension">
      <item>
          <helpName>ejb</helpName>
          <helpURL>../doc/$edition/ohj/helpset.jar!/helpset.hs</helpURL>
          <relativeTo>using_diagrams</relativeTo>
        <relativePosition>after</relativePosition>
      </item>
    </help>
  </triggers>
</trigger-hooks>
```

Adding Print Support

Create JDeveloper extensions that support printing.

JDeveloper provides a default implementation of `DocumentPrintFactory` that can create a `Pageable` object based on a `Component` or a `TextNode`.

How to Register DocumentPrintFactory

Printing is supported by declaratively registering a `DocumentPrintFactory` in extension.xm, as shown in the following example:

```
<hooks>
 <print-hook xmlns="http://xmlns.oracle.com/ide/extension">
     <documentPrintFactory id="oracle.jdevimpl.help.HelpTopicDocumentPrintFactory"

class="oracle.jdevimpl.help.HelpTopicDocumentPrintFactory" />
 </print-hook>
</hooks>
```

How to Register a View Class to Enable Printing

In order to enable printing from a view you need to register the view class and the `DocumentPrintFactory` that is used to create a `Pageable` object when print is called on the view.

The following example shows how to register a `printHelper` for `oracle.ide.navigator.NavigatorWindow` and specifies that the `DocumentPrintFactory` that is registered with the id `oracle.ide.print.DocumentPrintFactory` is used to construct the `Pageable` object.

```
<hooks>
    <print-hook xmlns="http://xmlns.oracle.com/ide/extension">
      <printHelper
documentPrintFactoryId="oracle.ideri.navigator.NavigatorPrintFactory"
                  view-class="oracle.ide.navigator.NavigatorWindow" />
    </print-hook>
</hooks>
```

If you want to print something else or enhance default printing you have to extend `DocumentPrintFactory` to create the `Pageable` object for your view and register your implementation of the `DocumentPrintFactory`. The following example shows how to use the same `DocumentPrintFactory` for multiple views.

```
<print-hook xmlns="http://xmlns.oracle.com/ide/extension">
     <documentPrintFactory id="oracle.jdevimpl.help.HelpTopicDocumentPrintFactory"

class="oracle.jdevimpl.help.HelpTopicDocumentPrintFactory" />
     <printHelper
documentPrintFactoryId="oracle.jdevimpl.help.HelpTopicDocumentPrintFactory"
                  view-class="oracle.jdevimpl.help.HelpTopicEditor" />
     <printHelper
documentPrintFactoryId="oracle.jdevimpl.help.HelpTopicDocumentPrintFactory"
                   view-class="oracle.jdevimpl.help.HelpWindow" />
     <printHelper
documentPrintFactoryId="oracle.jdevimpl.help.HelpTopicDocumentPrintFactory"
                  view-class="oracle.jdevimpl.help.HelpContentPanel" />
    <printHelper
documentPrintFactoryId="oracle.jdevimpl.help.HelpTopicDocumentPrintFactory"
                 view-class="oracle.jdevimpl.help.HelpCenterWindow$HelpCenterView" />
    </print-hook>
</hooks>
```

How to Register a PageableFactory implementation for Handling a Node

JDeveloper provides a default PageableFactory implementation to handle printing `TextNode` data.

In some cases you may want to override the standard `PageableFactory` implementation with one that provides the output you want. For example, the `DocumentPrintFactory` registered for the `NavigatorWindow` is the `NavigatorPrintFactory`. When it needs to create a `Pageable` object to print it calls `PrintManager.createPageableForObject()` which attempts to look up the best `PageableFactory` for the selected element. You can register a `PageableFactory` that is used to handle your node class.

```
<print-hook xmlns="http://xmlns.oracle.com/ide/extension">
      <textNodePageableFactory node-class="oracle.mypackage.MyNodeClass"
               weight="0.5">oracle.mypackage.MyPageableFactory</
textNodePageableFactory>
</print-hook>
```

In the previous example the node-class identifies the node that you want to be handled by the `PageableFactory` specified (in this case `oracle.mypackage.MyPageableFactory`). The weight is used to order the list of registered `PageableFactory` implementations when checking for a match for the current element.

Creating an Application or Project Template

Create custom application and project templates for JDeveloper.

If the application or project templates provided by JDeveloper do not meet your specific needs, you can define your own application or project template. Application and project templates are defined declaratively in a module's `extension.xml` file and are loaded automatically in the New Gallery under **General** > **Applications** and **General** > **Projects**, respectively. A dynamic wizard is generated from the template definition that includes an application or project definition page and, optionally, additional configuration pages, one per project technology.

Defining an Application or Project Template

You define an application or project gallery template using the template-hook in the `extension.xml` file.

Defining Template Hooks

You use the template hook to create a project and application templates. The `<template-hook>` element contains the child elements, `<projectTemplate>` and `<applicationTemplate>` and is in the `http://xmlns.oracle.com/ide/extension` namespace.

Use the `<projectTemplate>` element to define a project template that is displayed in the project gallery `<projectTemplate>` contains the following child elements:

- `<name>` - Specifies the name of the project as shown in the gallery.
- `<templateId>` - Specifies a unique project ID.

- <unsorted> - (Optional) Sorts the template list, putting this item at the top of the list in the gallery. Not recommend as this element is intended to be used to place the default template at the top of the list.
- <galleryFolder> - In addition to the project, adds a gallery item in the folder.
- <description> - Brief description of the project.
- <packageName> - Displays the default package name.
- <toolTip> - (Optional) Displays the path of the project when the cursor hovers over the project name or icon in the IDE's project window.
- <icon> - (Optional) Specifies an icon for the project in the gallery.
- <projectName> - Specifies the file name of the project. If no name is specified, a default name (Project1, Project2, ...) is provided.
- <creationListener> - Specifies the creation listener element.
- <excludeFromGallery> - Specifies whether this project template is used only in the context of the given application template. Setting this element to true, hides the project from the list of available projects in the gallery.

Use the <applicationTemplate> element to define an application template that is displayed in the application gallery. <applicationTemplate> contains the following child elements:

- <name> - Specifies the name of the application as shown in the gallery.
- <templateId> - Specifies a unique application ID.
- <unsorted> - (Optional) Sorts the template list, putting this item at the top of the list in the gallery. Not recommend as this element is intended to be used to place the default template at the top of the list.
- <galleryFolder> - In addition to the application, adds a gallery item in the folder.
- <description> - Brief description of the application.
- <packageName> - Displays the default package name.
- <toolTip> - (Optional) Displays the path of the application when the cursor hovers over the application name or icon in the IDE's application window.
- <icon> - (Optional) Specifies an icon for the application in the gallery.
- <applicationName> - Specifies the file name of the application. If no name is specified, a default name (Application1, Application2,...) is provided.
- <projectTemplates> - Contains <projectTemplate> elements, which specify the projects within the application.

How to Create a Project Template

You declaratively define a project template in a module's extension.xml file. The template is loaded automatically in the New Gallery under **General** > **Applications** or **General** > **Projects**. A dynamic wizard is generated from the template definition that includes a project definition page and, optionally, additional configuration pages, one per project technology.

The following example shows the template definition for a Java Project template:

```
<projectTemplate>
  <name>${TEMPLATE_JAVA_PROJECT}</name>
  <description>${TEMPLATE_JAVA_PROJECT_DESC}</description>
  <packageName>${TEMPLATE_JAVA_PROJECT_PACKAGE}</packageName>
  <projectName>${TEMPLATE_JAVA_PROJECT_JPR_NAME}</projectName>
  <templateId>javaProjectTemplate</templateId>
  <technologyScope>
    <technologyKey>Java</technologyKey>
    <technologyKey>Swing/AWT</technologyKey>
    <technologyKey>JavaBeans</technologyKey>
  </technologyScope>
</projectTemplate>
```

How to Create an Application Template

Similarly, you declaratively define an application template in a module's `extension.xml` file. The application template is also loaded automatically in the New Gallery. A dynamic wizard is generated from the template definition that includes an application definition page.

The following example shows the template definition for a Java Application template:

```
<applicationTemplate>
  <name>${TEMPLATE_JAVA_APP}</name>
  <templateId>javaAppTemplate</templateId>
  <unsorted>true</unsorted>
  <galleryFolder>General</galleryFolder>
  <description>${TEMPLATE_JAVA_APP_DESC}</description>
  <projectTemplates>
    <projectTemplate>
      <templateId>javaProjectTemplate</templateId>
    </projectTemplate>
  </projectTemplates>
</applicationTemplate>
```

Registering a Template

Sometimes an extension requires you to customize a project to support a technology, independent of the technologies the project currently has. To customize a project, you can modify the project template to register a creation listener, which is notified when a project is created from the modified template.

Registering a Creation Listener

By registering a `creationListener` element, the creation listener is called when a project is created from the template (identified by its ID) in which the listener is defined.

The `creationListener` element is optional. The listener is not called if it is not registered with the template. If called, the listener is called after the project is created as shown in the following example:

```
<template-hook>
  <projectTemplate>
    <templateId>testCallbackProjectTemplate</templateId>
    <name>Creation Listener Test Project</name>
    <description>A project for testing creation listener callbacks.</description>
    <projectName>sylvia</projectName>
```

```
    <creationListener>oracle.myfeature.wizard.MyWizardCallback</creationListener>
    <excludeFromGallery>true</excludeFromGallery>
  </projectTemplate>
  <applicationTemplate>
    <templateId>testCallbackAppTemplate</templateId>
    <name>Creation Listener Test App</name>
    <description>Creates an application for testing wizard creation listener
implementation.</description>
    <applicationName>WizardCallbackApp</applicationName>
    <projectTemplates>
      <projectTemplate>
        <templateId>testCallbackProjectTemplate</templateId>
      </projectTemplate>
    </projectTemplates>
  </applicationTemplate>
</template-hook>
```

How to Add a Page for a Technology to the New Application or Project Wizard

Project templates can have a page that enables a user to include specific technologies to a project. Optionally, for each of these technologies, you can define a configuration page that displays in the template creation wizard.

To define a wizard page for a particular technology scope, you need to add the page in the declaration of the technology scope. The following example shows the definition of the Java technology scope. This technology scope adds a page to the project creation wizard. Such wizard pages must extend `oracle.ide.panels.DefaultTraversablePanel` and implement `oracle.jdeveloper.wizard.template.WizardTraversable`.

```
<technology-hook xmlns="http://xmlns.oracle.com/ide/extension">
  <technology>
    <key>Java</key>
    <name>${JAVA_TECH}</name>
    <description>${JAVA_TECH_DESC}</description>
    <wizard-pages>
      <page>
        <traversable-
class>oracle.jdevimpl.wizard.project.JavaTechnologyPanel        </traversable-class>
      </page>
    </wizard-pages>
  </technology>
</technology-hook>
```

How to Add Additional Pages to an Application Template

Once you have created an application template, you might find that you want to add an additional page. For example, you might want to add a page to the template to provide a page for the user to specify a build configuration. Include the `<applicationAddonPage>` element to your template definition to add an additional page to the create application wizard.

An application template must define a list of add-on page groups that it subscribes to. It can subscribe to multiple groups. An add-on pages must define an add-on page group that it publishes to. It must be a single group. Multiple add-on pages can publish

to the same group. If the same page must be published to multiple groups, create another add-on page template hook and return the same class name.

Add-on pages are an enhancement of the Application Template API. It provides the following:

- Ability to add panels to all Application Templates or a specific Application Template
- Ability for Application Templates to opt-in to having add-on pages
- Ability to add a panel to beginning of an existing Application Template just after the application definition
- Ability to add a panel to the end of an existing Application Template

> **Note:**
>
> Use the add-on pages only if you absolutely need to and cannot modify the application template for some reason.

The following example shows an existing template-hook that lives in the `extension.xml` file to define an application template that includes the `<addonPageGroups>` field to enable templates to opt-in to add-on page groups:

```
<applicationTemplate>
  <name>${TEMPLATE_JAVA_APP}</name>
  <templateId>javaAppTemplate</templateId>
  <unsorted>true</unsorted>
  <galleryFolder>General</galleryFolder>
  <description>${TEMPLATE_JAVA_APP_DESC}</description>
  <projectTemplates>
    <projectTemplate>
      <templateId>javaProjectTemplate</templateId>
    </projectTemplate>
  </projectTemplates>
  <addonPageGroups>
    <addonPageGroup>testGroup</addonPageGroup>
  </addonPageGroups>
</applicationTemplate>
```

The `<addonPage>` element is a section in `extension.xml` of the template-hook that defines an add-on page as shown in the following example:

```
<template-hook>
<applicationAddonPage>
  <name>${TEMPLATE_TEST_ADDON_PAGE_NAME}</name>

  <pageId>testPage</pageId>

  <label>${TEMPLATE_TEST_ADDON_PAGE_LABEL}</label>

      Needs to be translated so reference a key in a resource bundle.-->
  <title>${TEMPLATE_TEST_ADDON_PAGE_TITLE}</title>

  <class>oracle.jdeveloper.MyPanel</class>

  <location>beginning</location>
```

```
  <addonPageGroup>testGroup</addonPageGroup>
</applicationAddonPage>
</template-hook>
```

The <applicationAddonPage> element contains the following child elements:

- <name> - Specifies the page name.
- <pageID> - Specifies a unique ID for the page.
- <label> - Specifies the name displayed in the left hand side of the wizard dialog to show the progress through the wizard.
- <title> - Provides the caption displayed at the top of the wizard to explain more detail about the wizard page.
- <class> - Declares the panel that you want to create an instance of. It must implement WizardTraversable and Traversable.
- <location> - (Optional) Specifies the placement of the page, at the beginning or end of the wizard. The value end is the default if this element is not provided.
- <addonPageGroup> - Specifies the add-on page group that this add-on page template is added to.

To add a page to an application template, follow these basic steps:

1. Add the <addonPageGroup> element to the template-hook in your extension.xml file:

```
<template-hook>
<applicationAddonPage>
  <name>testPage</name>
  <pageId>testPage</pageId>
  <label>dude</label>
  <title>yo dude</title>
  <class>oracle.jdeveloper.MyPanel</class>
  <addonPageGroup>testGroup</addonPageGroup>
</applicationAddonPage>
</template-hook>
```

2. Create a component that implements the WizardTraversable, Traversable and AddonPage interfaces:

```
public class MyPanel extends JPanel implements WizardTraversable, Traversable,
AddonPage
{
  public MyPanel() // The default constructor is the constructor that is called.
  {
    //...
  }

  //...
}
```

3. Make sure the Application Template has add-ons enabled:

```
<applicationTemplate>
  <!-- ... -->
  <addonPageGroups>
    <addonPageGroup>testGroup</addonPageGroup>
  </addonPageGroups>
</applicationTemplate>
```

3
Developing with the Extension SDK

This chapter describes the Extension SDK, which is available to download to JDeveloper, and how to use it to develop your own **extensions**.
This chapter includes the following sections:

About Developing with the Extension SDK

The JDeveloper Extension Software Developer Kit (SDK) includes a collection of projects containing sample code and Javadoc-generated documentation for the Extension API.

Each sample project contains a deployment profile for one-click deployment of the sample to the appropriate directory. You can also deploy all the projects at once.

You can use the code that provides each extension as a template to help you develop similar features.

Downloading and Installing the Extension SDK

Learn how to download and install the Extension SDK to JDeveloper.

You can download and install the Extension SDK using the Check for Updates wizard, or by downloading it from the Oracle website.

To install the Extension SDK using Check for Updates:

1. Follow the information in How to Install Extensions with Check for Updates in *Developing Applications with Oracle JDeveloper*, and on the Source page select the `Official Oracle Extensions and Updates` update center.
2. On the Updates page, choose `Extension SDK` from the list of available extensions, and click **Next**. Owing to the size of the extension, it may take a few minutes to download depending on your internet connection.
3. When you finish the wizard, JDeveloper will restart and install the Extension SDK. When it restarts, you are asked whether JDeveloper should install the sample application containing the sample projects described in this chapter. If you answer yes, the application `extensionsdk` is open in the Applications window, and the sample projects are listed.

If you are working behind a firewall, JDeveloper will not be able to connect to the update center until you enter details of your proxy server. You can either do this in the Web Browser and Proxy page of the Preferences dialog (available from the Tools

menu), or the Check for Updates wizard will time out and display the Web Browser and Proxy page.

To install the Extension SDK using a file:

1. Follow the information in How to Install Extensions Directly from OTN, and navigate to the page for Official Oracle JDeveloper Extensions.
2. Download the Extension SDK to a local location, then open the Check for Updates wizard from the Help menu.
3. On the Source page, choose `Install from Local File` and enter the location of the `esdk_bundle.zip` file and complete the wizard.
4. JDeveloper will restart and install the Extension SDK. When it restarts, you are asked whether JDeveloper should install the sample application containing the sample projects described in this chapter. If you answer yes, the application `extensionsdk` is open in the Applications window, and the sample projects are listed.

What Happens When you Install the Extension SDK

When you successfully install the Extension SDK, the sample application is optionally open in the Applications window, giving you access to the sample projects that illustrate different aspects of JDeveloper functionality. You can use the code to help you develop your own extensions.

The Oracle Fusion Middleware Java API Reference for Oracle Extension SDK is available from the Reference node in the Contents list of the online help (available from the **Help** menu).

Troubleshooting Installing the Extension SDK

If you have trouble downloading the Extension SDK and you are working behind a firewall, ensure that you have set the proxy server settings on the Web Browser and Proxy page of the Preferences dialog (available from the Tools menu).

If the Check for Updates wizard has located the update center you are using, it may take up to a couple of minutes before the list of available extensions is displayed, depending on the speed of your connection.

When you select the Extension SDK in the wizard, it downloads while the wizard is still open, and this can take a couple of minutes, depending on the speed of your connection. Once the extension has finished downloading, JDeveloper needs to restart in order to install it, and a message to this effect is displayed.

Using the Sample Extensions

JDeveloper's Extension SDK has a number of sample projects that illustrate how to use different types of JDeveloper element.

When you install the Extension SDK, you install an application extensionsdk, which contains a number of projects that give examples of how to create and use different types of JDeveloper element.

For information about running the samples, see Running the Sample Projects.

Most of the extension samples are described at:

http://www.oracle.com/technetwork/developer-tools/jdev/samples-083838.html

The following list is a sampling of the some of the available sample extension projects and a brief summary of their main features:

- `ApplicationOverview.jpr`: Demonstrates how to plug in to the Application Overview window. Adds a **Show Overview** menu item to both the Application and Project context menus. It illustrates how to:
- `AuditRefactor.jpr`: Demonstrates using the Audit framework to detect problems in Java code, and how to write a transform using the refactoring API to fix the problems automatically.
- `Balloon.jpr`: Demonstrates how to install a balloon notification into the JDeveloper status bar.
- `ClassBrowser.jpr`: Shows how to use the JDeveloper `ClassBrowser`.
- `ClassGenerator.jpr`: Shows the basic of JOT (Java Object Tool) to generate a java file in your project. Also demonstrates how to use the Finite State Machine (FSM) Wizard utility to create a multi step wizard.
- `ClassSpy.jpr`: Shows the simplest way to identify the class of a given node in the Applications window, and how to write its name in the Log window.
- `ClickableURL.jpr`: Shows how to generate a clickable URL in the Log window.
- `CodeInteraction.jpr`: Shows how to get text from the code editor, and launches a Google search on the currently selected text.
- `ConfigPanel.jpr`: Demonstrates how to store and retrieve preferences, and install a panel into the Preferences dialog.
- `CPPageProvider.jpr`: Demonstrates how to plug into the component palette using the palette2 API.
- `CreateDialog.jpr`: An example of a simple create dialog invoked from the gallery.
- `CreateStructure.jpr`: Demonstrates creating a new empty application (`.jws` file) and adding a new empty project (`.jpr` file) to it.
- `CustomEditor.jpr`: Shows how to implement your own editor for a given kind of file. To see the tool in action, open any XML file and switch to the Query page.
- `CustomExtensionHook.jpr`: Demonstrates a custom extension hook, which allows other extensions to plug into your extension.
- `CustomNavigator.jpr`: Demonstrates how to install a custom Applications window. Installs a "Favorites" window which can be used to store frequently accessed files.
- `DebugObjectPreferences.jpr`: Demonstrates how to install a custom object preferences renderer for the Data window of the debugger.
- `DockableWindow.jpr`: Shows how to implement a custom dockable window.
- `ExternalToolCreation.jpr`: Shows how to write an addin that installs an external tools shortcut for an external program. This feature is actually part of the JDeveloper IDE, and uses a wizard to set up external applications. For more information about External Tools, see the section "Adding External Tools to JDeveloper" in *Developing Applications with Oracle JDeveloper*.

- `ExternalToolImportExport.jpr`: Imports and exports the properties associated with a list of External Tools.
- `ExternalToolMacros.jpr`: Shows how to write an addin that installs a custom macro for use by external tools.
- `ExternalToolScanner.jpr`: Shows how to write a scanner for external tools that can automatically install external program shortcuts when the user clicks on the Find Tools button in the External Tools dialog, (available from the Tools menu).
- `FirstSample.jpr`: A sample that illustrates the main concepts involved in writing extensions for JDeveloper.
- `FlatEditor.jpr`: Demonstrates how to implement a form-based overview editor similar to the one used for `extension.xml`.
- `HelloX.jpr`: This is a sample that demonstrates a number of core concepts, including installing New Gallery items, actions, and menu items.
- `LayoutMenuFilter.jpr`: Shows how to filter the top level menus in the IDE when the layout changes.
- `MethodCallCounter.jpr`: Uses JOT (Java Object Tool) to count the number of occurrences of a named method in some code.
- `OpenNodes.jpr`: Implements a dockable window which shows nodes that are open in the NodeFactory, and tracks when nodes are opened and closed.

  When you run this sample, you can choose **View > ESDKSample: Open Nodes** to open an Open Node Tracker window, which shows the nodes that are open in `NodeFactory`, and tracks when nodes are opened and closed.
- `Overlay.jpr`: Shows how to use overlays in the Applications window, like those used by the version control extensions.
- `ProgressBar.jpr`: Shows how to implement a progress bar during the execution of some lengthy task.
- `ProjectSettings.jpr`: Demonstrates storing and retrieving project properties and adding custom project settings UI.
- `StructurePane.jpr`: Shows how to display your own content in the Structure Window.
- `UpdateCenter.jpr`: Demonstrates how to install a custom update center as part of an extension.

Running the Sample Projects

Learn how to run the sample JDeveloper extension projects.

You can run the sample projects in JDeveloper to see how they work.

To run a sample project:

1. The first time you run one of the sample project extensions, you have to right-click on it and choose **Deploy to Target Platform**.
2. To run the extension, right-click and choose **Run Extension**.

   This runs a second copy of JDeveloper with the sample project installed.

4

Testing and Debugging Extensions

This chapter describes how to use the tools and features provided by JDeveloper to run and debug Java programs.
This chapter includes the following sections:

About Testing and Debugging Extensions

Any extension needs to be tested and debugged for any anomaly before deployment.

You test and debug an **extension** by installing it and running it in a debug instance of JDeveloper.

Debugging Extension Code

Learn how to debug JDeveloper extensions.

JDeveloper provides you with a comprehensive debugger to test your code. For general information about debugging in JDeveloper, see "Debugging Applications" in *Developing Applications with Oracle JDeveloper*.

To debug an extension, you run it in JDeveloper. A new instance of JDeveloper opens in debug mode with the extension installed. If something is not working correctly then you can set breakpoints in your extension code to debug it.

How to Run a JDeveloper Extension

When you have developed the code for your extension, you will want to try it out. To do this you install the extension and then run JDeveloper. A new instance of JDeveloper runs in debug mode for you to test your extension.

To run an extension in JDeveloper:

- From the context menu of the project in the Applications window, or in the source editor, choose **Run Extension**.

A new version of JDeveloper starts with the extension running in it.

How to Debug a JDeveloper Extension

With the extension running in a debug version of JDeveloper, you can test the functionality and use the JDeveloper debugging features to find and fix problems. See "Debugging Applications" in *Developing Applications with Oracle JDeveloper.*

To debug an extension in JDeveloper:

1. Enter suitable breakpoints in your code.
2. From the context menu of the project in the Applications window, or in the source editor, choose **Debug Extension**.

A new version of JDeveloper starts in debug mode with the extension running in it.

Troubleshooting Debugging

Discover how to resolve debugging issues with JDeveloper extensions.

Use this list of symptoms and solutions to resolve extension issues:

- **Symptom:** Exception at startup.

  Solution: Check stack trace and debug it.

- **Symptom:** Extension does not load.

  Solution: Check the Features dialog (available from the Tools menu) to make sure the extension is enabled when JDeveloper launches. For more information, see "How to Manage JDeveloper Features" in *Developing Applications with Oracle JDeveloper*.

- **Symptom:** Class cast exception.

  Solution: For an extension that was written for an earlier version of JDeveloper, some code has not been migrated, or the extension was compiled against an older version of JDeveloper. Check to see that the extension has been migrated. For more information, see Migrating Extensions from Previous Releases.

5
Packaging and Deploying Extensions

This chapter describes how to create a package to distribute your extension. This chapter includes the following sections:

About Packaging and Deploying Extensions

Prior to the deployment of an extension, it has to be created and then packaged in an extension JAR file.

The steps to package and deploy an extension are as follows:

1. First, you create an extension package which consists of a JAR that is packaged in an extension bundle archive. The extension bundle archive is a JAR file containing the extension JAR and any supporting files used by the extension.

   The extension JAR file contains:

   - The extension manifest file `extension.xml`.
   - Compiled class files and resources in the same directory structure they had while they were being developed.

   The extension bundle archive contains:

   - One or more extension JAR files.
   - Any supporting files such as library JAR files.
2. Next, you package the extension JAR files into a `.zip` file for distribution.
3. If this is the first time you have opened the project in JDeveloper, for example if you are migrating an extension written for an earlier version of JDeveloper, right-click the project in the Applications window and choose **Deploy to Target Platform**. This generates the bundle manifest `manifest.mf` if one does not already exist. You may need to refresh the Applications window.
4. If the extension references external libraries, open the bundle manifest `manifest.mf` by locating it in the Applications window under the META-INF node, and change the line

   ```
   Bundle-ClassPath: .
   ```

   so that it says:

   ```
   Bundle-ClassPath: .,external:jdev-install/jdeveloper/jdev/extensions/library
   ```

Packaging the Extension

Learn how to package JDeveloper extensions for deployment.

When you create an extension project, an extension deployment profile to create an OSGi bundle is also created. For more information, see "Deployment Profiles" in *Developing Applications with Oracle JDeveloper*.

How to Create the Deployment Profile

You set the OSGi bundle profile parameters from the Project Properties dialog.

To edit the deployment profile:

1. In the Applications window, right-click the project, then choose **Properties**. Alternatively, choose **Project Properties** from the **Application** menu.
2. Select **Deployment** in the panel on the left of the Project Properties dialog. The extension profile `Extension (Extension JAR)` is selected. Click **Edit**.
3. In the Bundle Options page of the OSGi Bundle Profile dialog, you can enter details such as the bundle name, version, and activator. For more help at any time, press F1 or click **Help** from the dialog.

   You set dependencies and set libraries for inclusion by choosing other pages in the dialog. For more information, see Understanding Dependencies.

   Click **OK** when you are finished editing the deployment profile properties.

How to Create the OSGi Bundle

Once you have edited the deployment profile you can create the OSGi Bundle.

To create the OSGi Bundle:

1. In the Applications window, right-click the extension project and choose **Deploy > extension-profile** to open the Deploy Extension dialog.
2. The option **Deploy to OSGi Bundle** is selected. You can click **Next** to examine details of the bundle that will be created in the Summary page. When you are satisfied, click **Finish**.

The OSGi bundle containing your extension is created in the `Oracle-home/jdeveloper/jdev/extensions` directory.

You can run and debug the OSGi bundle from this location automatically. For more information, see Testing and Debugging Extensions .

Deploying an Extension

Learn how to deploy JDeveloper extensions.

Extensions can be distributed to a team by making them available on a file system, and having users install the extension using the Check for Updates wizard available from the Help menu. See "Working with Extensions".

Alternatively, you can make an extension available to a wider audience by hosting it on the Web somewhere, or you could have it hosted as an open source project so that other people can help you enhance your extension and further develop it.

Oracle hosts some JDeveloper third party extensions, which are available at `http://www.oracle.com/technetwork/developer-tools/jdev/index-099997.html`. If you would like to upload your extension to this site, post a message on the JDeveloper and ADF forum at `https://forums.oracle.com/forums/forum.jspa?forumID=83`.

How to Publish an Extension in an Update Center

Once you have written and tested an extension for JDeveloper you can make it available in a JDeveloper Update Center, from where users can download it and have it automatically install on top of their JDeveloper installation.

Update centers for JDeveloper extensions are online at http://www.oracle.com/technetwork/developer-tools/jdev/index-099997.html, including the Oracle JDeveloper Third Party Extension Exchange which lists extensions developed by third-party companies and individual developers.

For information about how users install updates using the JDeveloper Update Center, see Working with the Extension Manifest.

In order for your extension to be installable using JDeveloper's Check for Updates feature:

1. Create a `bundle.xml` file for the extension.
2. In Project Properties, create a Jar deployment profile which uses the file extension `.zip` rather than `.jar`.
3. With Profile Dependencies selected, ensure that the extension project is selected.
4. With File Groups selected, create a new file group for `bundle.xml` and specify the target directory as `META-INF`.

   When you deploy the project using the new deployment profile the ZIP file is created, and this can be used on its own to distribute the extension. However, if you want to distribute the extension using an update center you have to create an update center XML file.
5. If you want to distribute the extension using an update center you have to create an update center XML file. Then in the Check for Updates dialog (available from the Help menu) to create a new update center pointing to the URL in the update center XML file.

You can have users add the new update center to their instance of JDeveloper so that they can download and install your extension.

Extension bundle.xml Document

This describes the `bundle.xml` document which you need to deploy and distribute an extension.

bundle.xml

Create a document called bundle.xml similar to the following example.

> **Note:**
>
> The metadata in `bundle.xml` must contain the `u` namespace, as shown in the example below.

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<update-bundle version="1.0"
  xmlns="http://xmlns.oracle.com/jdeveloper/updatebundle"
  xmlns:u="http://xmlns.oracle.com/jdeveloper/update">
   <u:update id=" organization.dev.extension ">
    <u:name>Name of Extension</u:name>
    <u:version>1.0</u:version>
    <u:author>YourOrganization</u:author>
    <u:author-url> http://www.organization.com/dev/index.html/</u:author-
url>
    <u:description>Provides support for a feature.
    </u:description>
    <u:requirements>
      <u:requires-extension id="oracle.jdeveloper"
        minVersion="12.1.2"
        maxVersion="12.2.1" />
      </u:requires-agreement url="META-INF/agreement.html" />
      </u:requirements>
    <u:destination>${oracle.home}</u:destination>
  </u:update>
</update-bundle>
```

Important Elements and Attributes

Update Id
Each extension is identified by a unique id and the JDeveloper update center will refer to your extension using it.

Destination
This optional attribute is the install location and two macro values are available:

- `oracle.home`, (default) the JDeveloper directory, for example, `C:\oracle\middleware\jdeveloper`.
- `oracle.mw.home`, which can be used when you want to install extensions or patches in the middleware home directory above the jdeveloper directory, for example, `C:\oracle\middleware` .

Requires Agreement
This optional attribute which brings up the specified URL in the default browser for the user to provide agreement for using some technology.

Requires Extension minVersion and maxVersion
Use to specify the JDeveloper versions that the extension is for.

Update Center XML Document

This describes the update center XML document which you need to distribute an extension from an update center through JDeveloper's Check for Updates feature.

Update Center XML File

The update center file is similar to `bundle.xml` but it has an additional element, `bundle-url`, which specifies the location of the extension ZIP file. For more information about `bundle-url`, see .

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<update-bundle version="1.0"
  xmlns="http://xmlns.oracle.com/jdeveloper/updatebundle"
  xmlns:u="http://xmlns.oracle.com/jdeveloper/update">
   <u:update id=" organization.dev.extension ">
    <u:name>Name of Extension</u:name>
    <u:version>1.0</u:version>
    <u:author>YourOrganization</u:author>
    <u:author-url> http://www.organization.com/dev/index.html/</u:author-
url>
    <u:description>
    </u:description>
    <u:requirements>
      <u:requires-extension id="oracle.jdeveloper"
        minVersion="12.1.2"
        maxVersion="12.2.1" />
    </u:requires-agreement url="META-INF/agreement.html" />
    </u:requirements>
    <u:destination>${oracle.home}</u:destination>
     <u:bundle-url> http://www.organization.com/dev/extension.zip/</
u:bundle-url>
  </u:update>
</update-bundle>
```

A

Elements Installed with the Extension SDK

This appendix provides information about the elements installed with the Extension SDK.
The appendix contains the following sections:

Elements Installed in the File System

Learn the location of the JDeveloper Extension SDK documentation and sample application.

When you install the Extension SDK:

- The Extension SDK API documentation is installed in *jdev_install*`/jdeveloper/jdev/extensions/oracle.jdeveloper.esdk/doc`.
- The extension samples application `extensionsdk.jws` which contains the sample projects is installed in *jdev_install*`/jdeveloper/mywork/extension-samples-12.1.3.0.`*nn.nnnnnn.nnnn* (the value of *n* is the label number, for example, `extension-samples-12.1.3.0.41.131204.1931`).

Elements Installed in the IDE

Learn what is present after installing the JDeveloper Extension SDK.

After downloading and installing the Extension SDK, and restarting JDeveloper the following elements are present in the IDE:

- If you have chosen to install the samples, the application `extensionsdk.jws` is open in the Applications window. If you choose not to install the samples when JDeveloper restarts, you can open the extension SDK samples application at a later time from **Help > Open Extension Samples**.
- In the Manage Features for Studio Developer Role dialog, available from **Features** on the **Tools** menu, **ESDK Samples** is listed under the IDE node.
- The Oracle Fusion Middleware Java API Reference for Oracle Extension SDK is added to the Reference node in the JDeveloper online help Contents.

B

Uninstalling the Extension SDK

This appendix provides information about disabling and uninstalling the Extension SDK.
The appendix contains the following sections:

About Uninstalling the Extension SDK

JDeveloper extensions are easy to uninstall or just disable to optimize the system performance.

You can disable the Extension SDK JDeveloper and enable it at a later time so that you do not have to download and install the Extension SDK afresh, or you can uninstall the sample application, `extensionsdk.jws` and its projects, or you can completely remove the Extension SDK from your machine.

How to Disable the Extension SDK

Learn how to disable the JDeveloper Extension SDK.

When you work with JDeveloper you choose a role in which to work, and the role you choose determines which JDeveloper **extensions** are loaded. In general, different roles remove JDeveloper extensions that are not needed so that JDeveloper performance is optimized. Indeed, you can create your own JDeveloper role that disables the extensions you do not want to work with and modifies the menus that are loaded when the IDE starts. For more information, see "Working with JDeveloper Roles" in the *Developing Applications with Oracle JDeveloper*.

If you want to disable the Extension SDK, you can do this in the same way that you would disable any other JDeveloper extension.

To disable the Extension SDK:

1. From the **Tools** menu choose **Features** to open the Manage Features for Studio Developer Role dialog. This displays all the extensions that are available for the role in which you are working.

   For more information at any time, press F1 or click **Help** from within the dialog.

2. If necessary, expand the IDE node, and uncheck **ESDK Samples**.

If you later want to use the Extension SDK, you can enable it by navigating to the same dialog and checking **ESDK Samples**.

How to Delete the Sample Application

Learn how to delete the JDeveloper Extension SDK sample application.

You can delete the sample application, `extensionsdk.jws`, and the projects it contains in the same way that you can delete any JDeveloper application.

To delete the sample application:

1. In the Applications window, click the Application menu dropdown list (next to the name of the application, `extensionsdk`).
2. Select **Delete Application**.
3. In the Confirm Delete Application dialog, choose **Yes**.

Alternatively, right-click `extensionsdk` in the Applications window toolbar, and choose **Delete Application**.

JDeveloper will delete the `extensionsdk` application, including all its projects and their directories.

How to Uninstall the Extension SDK

Learn how to uninstall the JDeveloper Extension SDK.

When the Extension SDK is installed, it places the Extension SDK API documentation in *`jdev_install`*`/jdeveloper/jdev/extensions`, where *`jdev_install`* is the directory that JDeveloper is installed in, and the extension samples application `extensionsdk.jws` which contains the sample projects in *`jdev_install`*`/jdeveloper/mywork`.

You can completely remove the Extension SDK from your local drive by deleting:

- *`jdev_install`*`/jdeveloper/jdev/extensions/oracle.jdeveloper.esdk`
- *`jdev_install`*`/jdeveloper/mywork/extension-samples-11.1.2.0.nn.nn.nn`