Oracle® Fusion Middleware

Developer's Guide for Oracle JDeveloper Extensions

11*g* Release 2 (11.1.2.1.0)

E20067-02

September 2011

ORACLE

Oracle Fusion Middleware Developer's Guide for Oracle JDeveloper Extensions 11*g* Release 2 (11.1.2.1.0)

E20067-02

Primary Author: Catherine Pickersgill

Contributing Author: John Brock

Contents

B Uninstalling the Extension SDK

Preface

Welcome to the *Developer's Guide for Oracle JDeveloper Extensions*.

Audience

This document is intended for developers that develop 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 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

- *Oracle Fusion Middleware Installation Guide for Oracle JDeveloper*
- *Oracle Fusion Middleware User Guide for Oracle JDeveloper*
- *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*
- *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*
- *Oracle Fusion Middleware Java EE Developer's Guide for Oracle Application Development Framework*
- Oracle JDeveloper *11g Online Help*
- Oracle JDeveloper *11g Release Notes*, link included with your Oracle JDeveloper 11*g* 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.
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.

What's New in This Guide in Release 11.1.2.1.0

For Release 11.1.2.1.0 the *Oracle Fusion Middleware Developer's Guide for Oracle JDeveloper Extensions* replaces the non-context sensitive online help that was available in previous releases of Oracle JDeveloper.

For changes made to Oracle JDeveloper and Oracle ADF for this release, see the What's New page on the Oracle Technology Network at http://www.oracle.com/technetwork/developer-tools/jdev/documentation/index.html.

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:

1.1 About Developing Oracle JDeveloper Extensions

Oracle JDeveloper is an integrated development environment (IDE) for building applications using the latest standards for Java, XML, Web services, and SQL. Most of the basic functionality in JDeveloper is implemented as extensions—software packages which add features and capabilities to the basic 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 requirements, for example:

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

> **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.

For more information, see "Working with Extensions" in *Oracle Fusion Middleware User Guide for Oracle JDeveloper*.

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 has 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 Section 1.5, "Migrating Extensions from Previous Releases."

You can use JDeveloper's native features to develop your own extensions. For more information, see Chapter 2, "Developing Extensions in Oracle 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. For more information, see Chapter 3, "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 http://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 more information about External Tools, see "Adding External Tools to JDeveloper" in *Oracle Fusion Middleware User Guide for Oracle JDeveloper*.

1.2 Developing Extensions with 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.

1.2.1 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.

1.2.2 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.

1.3 How JDeveloper Extensions Work

This section describes the essential features of how extensions work in JDeveloper. In order to help with startup performance, overall memory use, and how JDeveloper performs day to day, only extensions that the user is using are loaded. Some terms and concepts that you may be unfamiliar with 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.

  So, 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 classloaded 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. For more information, see `http://jcp.org/en/jsr/detail?id=198`.

1.3.1 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.

1.3.2 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 Section 2.5.1, "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 a navigator, 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 navigator 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.

1.3.3 How Lazy Initialization Works

In order 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 Section 2.2.4, "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 because having extensions with deep dependency trees, What will happen is that the extension 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 Section 2.2.6, "Understanding Large Extensions."

1.4 Guidelines for Writing JDeveloper Extensions

This section is an outline of good development practice for JDeveloper extensions.

- Write declarative trigger hooks to initialize the model of your extension and declare them in the extension manifest. Extension initialization code happens in the OSGi activator, and the first time that a class is classloaded from an extension, OSGi automatically activates the bundle by calling its activator.

You can find more information to guide you in Section 2.2, "Use Cases for Developing Extensions."

1.5 Migrating Extensions from Previous Releases

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 Section 1.3, "How JDeveloper 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 Section 2.5, "How to Define and Use 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 Section 1.3.3, "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 Section 2.2, "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.

1.6 Getting Started With Extension Development

Whether you use the native features of JDeveloper to create an extension, described in Chapter 2, "Developing Extensions in Oracle JDeveloper," or download the Extension SDK to develop an extension, described in Chapter 3, "Developing with the Extension SDK," you start by creating an application in JDeveloper along with one or more projects configured for extension development.

1.6.1 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. For more information, see "About Working with Applications and Projects" in *Oracle Fusion Middleware User Guide for Oracle JDeveloper*.

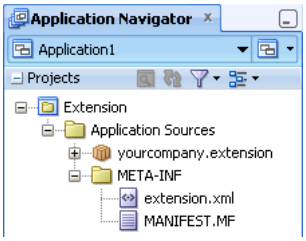
To create an application and project:

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 **Custom Application**. The Create Custom 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 **Finish**.

   The application is created with a default project called `Project1` that you will not use and can delete.
5. Next, create a project that is configured for extension development. 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 **Extension Project**. The Create Extension Project dialog opens. For help with the dialog, press F1 or click **Help**.
8. Enter a name and other details for the extension project. When you are done, click **Finish**.

The extension project is created in the Application Navigator, and contains the elements shown inFigure 1–1.

Figure 1–1 Extension Project in the Application Navigator



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.

1.6.2 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 project.

> **Note:** You can only develop extensions for other versions of JDeveloper 11.1.2.0.0.

When you create the extension project, as in Section 1.6.1, "How to Create an Application and Project for Extension Development", when you have the Create Extension Project dialog open (step 7), click Manage to open the Manage Extension Platforms dialog.

In this dialog, enter details of the JDeveloper version that you are developing the extension for. For more help, press F1 or click **Help** in the dialog.

1.6.3 How to Create an Empty Extension Project

If you are familiar with creating extensions, and want to start with an empty project, you can create a custom application with a project that is set up for extension development but which does not contain the extension manifest `extension.xml` and the OSGi manifest `manifest.mf`.

To create an extension application with an empty 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 **Custom Application**. The Create Custom 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.
5. Click **Next**, and enter a name for the project. On the Project Features tab, select `Extension Design Time` and shuttle it to the **Selected** list, then click **Finish**.

In this case, an empty project is created in the Application Navigator, and you need to create extension.xml separately in the same way that you would create any XML file. For more information, see "Creating XML Files in Oracle JDeveloper" in *Oracle Fusion Middleware User Guide for Oracle JDeveloper*.

1.6.4 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 Section 2, "Developing Extensions in Oracle JDeveloper."
- To use the samples in the Extension SDK to create your extension, or to use the Extension SDK API, see Section 3, "Developing with the Extension SDK."

1.7 Working with the Extension Manifest

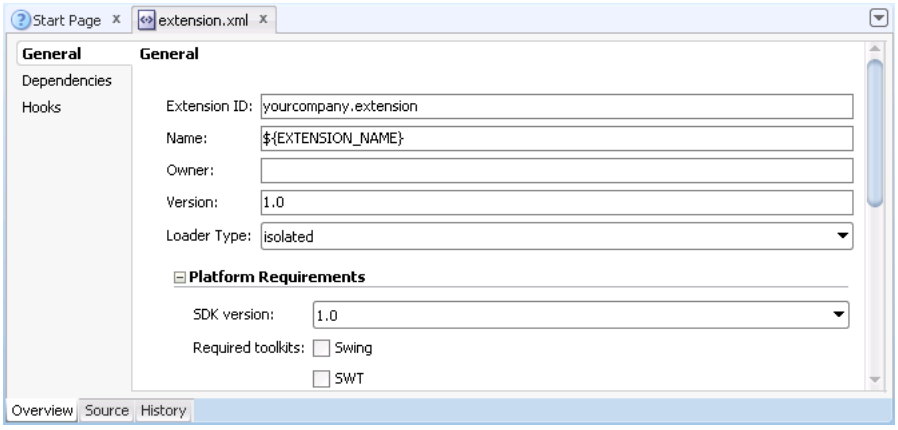
The extension manifest, `extension.xml`, controls many aspects of the extension. Before you can deploy an extension, you must complete `extension.xml` to, for example, register trigger hooks. There can only be one `extension.xml` per project, and it must always be in a directory called `meta-inf`.

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 Section 1.8.1, "Understanding Dependencies."

The template code for `extension.xml` created by creating an extension project contains placeholders for the main elements, shown in Example 1–1.

Example 1–1 Template Code in extension.xml

```
<extension id="yourcompany.extension" version="1.0" esdk-version="1.0"
rsbundle-class="yourcompany.extension.Res"
          xmlns="http://jcp.org/jsr/198/extension-manifest">
  <name>${EXTENSION_NAME}</name>
  <trigger-hooks xmlns="http://xmlns.oracle.com/ide/extension">
    <!-- TODO Declare triggering functionality provided by extension:
yourcompany.extension -->
    <triggers/>
  </trigger-hooks>
  <hooks>
```

```
    <!-- TODO Declare functionality provided by extension: yourcompany.extension
-->
  </hooks>
</extension>
```

1.7.1 Editing the Extension Manifest in the Overview Editor

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 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 Section 1.8.1, "Understanding Dependencies."

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

1.7.2 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 *Oracle Fusion Middleware User Guide for Oracle JDeveloper*.

1.8 Working with the OSGi Manifest

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`.

When a new extension project is created, a default `manifest.mf` is created, as shown in Example 1–2.

Example 1–2 Default of manifest.mf

```
Manifest-Version: 1.0
Bundle-ClassPath: .
```

For detailed information about the content of the OSGi manifest, see `http://www.osgi.org/javadoc/r4v42/overview-summary.html`.

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

- **Required-Bundle:** Classes from another OSGi bundle that are required by your extension are listed. Comes from the `required-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 set in the OSGi Bundle Profile dialog. For more information, see Section 5.2.1, "How to Create the Deployment Profile."

1.8.1 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
- Or, 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.

1.8.1.1 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 Section 1.7, "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. In necessary, open `extension.xml` in the overview editor by double-clicking on `extension.xml` under `META-INF` in the Application Navigator. Select the Dependencies tab.
2. To specify that other extensions are dependent on this extension, click **Add Extension Import**. In the Select Extension dialog, choose one or more extensions and click **OK**.

   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, click **Add Bundle Entry**. In the Select Required Bundles dialog, choose one or more bundles that depend on this extension and click **OK**.

   The extensions you select as listed as `<bundle>` elements in the `<required-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.

1.8.1.2 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 Section 5.2.1, "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.

1.8.1.3 Duplication Between extension.xml and manifest.mf

If you edit the OSGi manifest by hand rather than letting JDeveloper create it as part of the Deployment process, you can enter similar information about the extension in the extension manifest **extension.xml** and in the OSGi manifest **manifest.mf**, for example:

extension.xml	manifest.mf
<dependencies>	Required-Bundles
<extension-id>	Bundle-SymbolicName

Where there is duplication, the information in `manifest.mf` is used, however the extension load order is always taken from the `<dependencies>` section of `extension.xml`.

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:

2.1 About Developing Extensions in Oracle JDeveloper

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.

2.2 Use Cases for Developing 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.

2.2.1 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 Section 2.5.4.2, "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 Section 1.7, "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.

2.2.1.1 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 Section 2.2.1, "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:

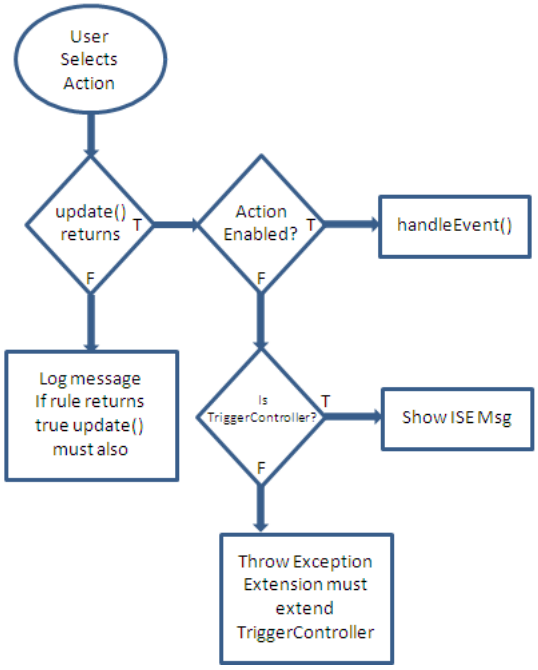
- A menu is about to be shown:
  - If it is a rules base controller, the rule will be evaluated. If `false`, the menu is grayed out by the framework. If `true` or not a rules-based controller, see the next bullet point.
  - `Controller.update()` is called. If it returns `true`, no other controllers are 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: This is described in

- 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



2.2.2 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.

2.2.2.1 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 Section 2.2.1, "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.

2.2.3 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 Section 2.4.2, "How to Create and Modify Menus."

The editor type and the node type control whether a menu item is included in the dynamic top level menu.

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 Section 2.2.1, "Understanding Rules Based Menu Sensitivity."

2.2.4 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 JDeveloper's recognizers.

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 Section 1.7, "Working with the Extension Manifest."

2.2.5 Understanding Content Sets

JDeveloper uses content sets to define the categories that are displayed as nodes in the Application Navigator. 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 Application Navigator, 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 Application Navigator ensures that as the user expands folders in the navigator, 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.

2.2.6 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 Section 2.5, "How to Define and Use Trigger Hooks."

2.2.7 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 *Oracle Fusion Middleware User Guide for 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.

2.3 Getting the JDeveloper Look and Feel

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

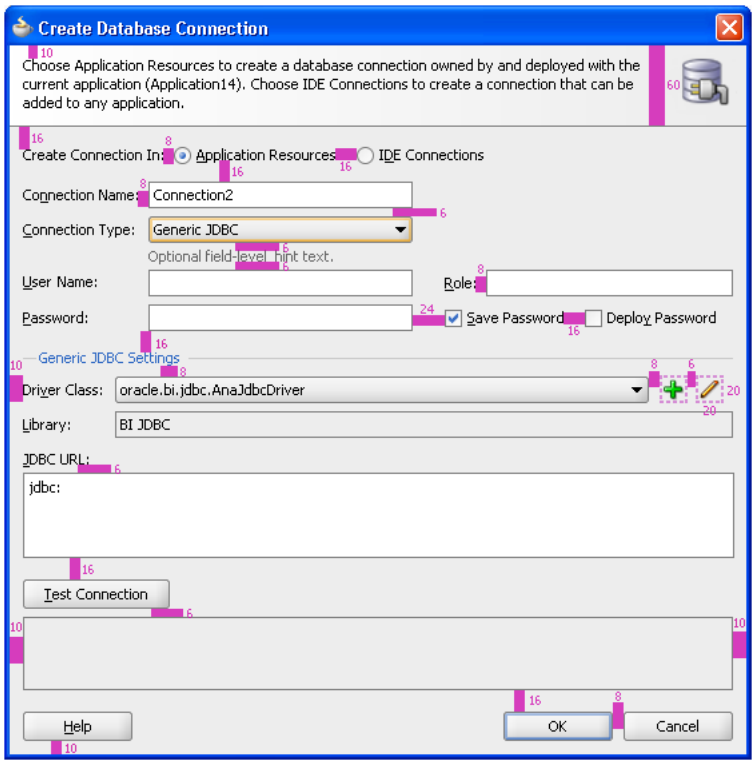
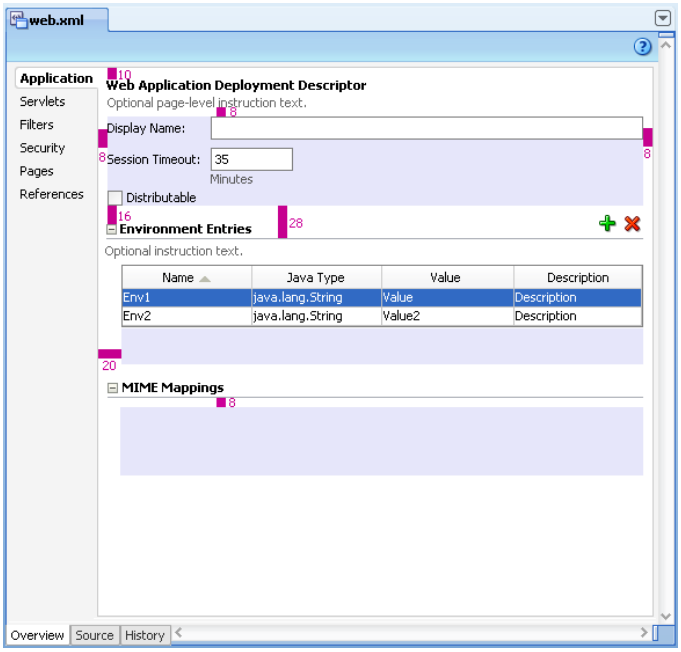


Figure 2–3 shows the spacing used for an overview editor in JDeveloper.

Figure 2–3 Overview Editor Showing Spacing



2.4 How to Create JDeveloper Elements

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
- Component palettes
- JDeveloper preferences
- Making changes undoable

2.4.1 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 more information about Actions, see Section 2.4.5.3, "How to Define an Action."
- Addins, which can perform programmatic initialization and registration for an extension while JDeveloper is starting up. For more information about Addins, see Section 2.4.5.1, "How to Implement the Addin Interface."
- Data models, which subclass **HashStructureAdapter** to store the persistent data model for preferences or project properties. For more information about data models, see Section 2.4.9.2, "How to Implement the Data Model."
- Wizards that can be invoked from the New Gallery or Tools menu. For more information about wizards, see Section 2.4.4, "How to Develop Wizards."
- New panels in the Preferences dialog. For more information about working with the Preferences dialog, see Section 2.4.9, "Understanding Preferences."
- New panels in the Project Properties dialog. For more information about project properties, see Section 2.4.10, "Understanding Project Properties."

To quickly create a JDeveloper element from the New Gallery:

1. Follow the steps in Section 1.6.1, "How to Create an Application and Project for Extension Development" to create an application and extension project.
2. Open the New Gallery by choosing **File > New**.
3. In the New Gallery, in the Categories tree, under **Client Tier** select **Extension Development**.
4. 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.

2.4.2 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 Section 2.4.5, "How to Develop Commands."

2.4.2.1 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 Section 2.4.2.2, "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 Section 2.4.5.5, "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 Section 2.4.2.4, "How to Add Menu Items to an Existing JDeveloper Menu."

You can also use a similar technique to add new selections to a toolbar. For more information, see Section 2.4.2.5, "How to Add a Drop-down Button to a Toolbar."

2.4.2.2 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 Section 2.4.5.5, "How to Invoke an Addin From a Context Menu."

To create a menu with several selections:

1. Register the context menu listener classes in your extension manifest using the sample code in Example 2–1.

Example 2–1 Registering Context Menu Listener Classes

```
<extension xmlns="http://jcp.org/jsr/198/extension-manifest" ...>
 ...
  <hooks>
   <jdeveloper-hook xmlns="http://xmlns.oracle.com/jdeveloper/1013/extension">
    <actions>
      <!--  Action to show the File List -->
      <action id="MY_CMD_ID">
```

```
      <properties>
        <property name="Name">My Action</property>
      </properties>
        <controller-class>oracle.ide.extsamples.basic.SimpleController
        </controller-class>
    </action>
  </actions>
    <context-menu-listeners>
      <site idref="navigator"> <!-- or "editor", or "explorer" -->
          <listener-class>oracle.ide.extsamples.basic.
            SimpleContextMenuListener</listener-class>
      </site>
  </context-menu-listeners>
 </jdeveloper-hook>
</hooks>
...
</extension>
```

2. Implement `ContextMenuListener`, as shown in Example 2–2.

   This is the callback interface that allows extensions to add menu items and submenus to the context menu:

Example 2–2 ContextMenuListener

```
package oracle.ide.extsamples.basic;
import oracle.ide.Context;
import oracle.ide.controller.ContextMenu;
import oracle.ide.controller.ContextMenuListener;
import oracle.ide.controller.IdeAction;
/**
  * ContextMenuListeners add items to context menus.
*/
public final class SimpleContextMenuListener
 implements ContextMenuListener
{
  public void menuWillShow(ContextMenu contextMenu)
  {
    // Add my menu to the context menu only when user clicked on SomeNode class.
    if (contextMenu.getContext().getNode() instanceof
oracle.ide.extsamples.basic.SomeNode.class)
    {
       IdeAction action = IdeAction.find( SimpleController.SAMPLE_CMD_ID );
       contextMenu.add( contextMenu.createMenuItem( action ) );
    }
  }
  public void menuWillHide(ContextMenu contextMenu)
  {
      // Most context menu listeners will do nothing in this method. In
      // particular, you should *not* remove menu items in this method.
   }
  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;
   }
}
```

3. Implement `oracle.ide.controller.Controller`, as shown in Example 2–3.

Each `View` has an associated `Controller`. Controllers receive requests to handle the commands associated with user actions. The controller's handleEvent method gets called with the appropriate Command specified. If the Controller does not handle the requested command, it delegates the Command to a supervising Controller. Controllers are also responsible for determining the availability of a specific command, by calling the update method.

Example 2–3 oracle.ide.controller.Controller

```
package oracle.ide.extsamples.basic;
import oracle.ide.Context;
import oracle.ide.controller.Controller;
import oracle.ide.controller.IdeAction;
/**
  * ContextMenuListeners add items to context menus.
 */
public final class SimpleController
  implements Controller
{
   public static final int MY_CMD_ID = Ide.findCmdID( "MY_CMD_ID" );
   public boolean handleEvent( IdeAction action, Context context )
    {
      if (IdeAction.getCommandId() == MY_CMD_ID )
       {
         // Do some action
         return true;
         }
        return false;
       }
    public boolean update( IdeAction action, Context context )
     {
       if (IdeAction.getCommandId() == MY_CMD_ID )
        {
         // Enable action
         action.setEnabled(true);
         return true;
        }
        action.setEnabled(false);
        return false;
       }
 }
```

2.4.2.3 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 Application Navigator 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.

2.4.2.4 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. For more information, see Section 2.4.5.4, "How to Invoke an Addin From a Main Window Menu."

To add a new menu item to the Tools menu:

1. Import `oracle.ide.controller.*`, then create the action (`_myActionID`) you plan to call in the menu, as shown in Example 2–4.

Example 2–4 oracle.ide.controller.*

```
import oracle.ide.controller.*;
  // Create the action
  _myActionID = Ide.createCmdID("FirstAction");
  IdeAction firstAction = IdeAction.get(
      _myActionID,
      null,
      "My First Action",
      IdeMainWindow.ACTION_CATEGORY_TOOLS,
      new Integer('F'),
      null, null, true);
```

2. Set the current item as the controller:, as shown in Example 2–5.

Example 2–5 Controller

```
// Set ourselves as the controller
 firstAction.addController(this);
```

3. Add the menu item associated with `firstAction` to the menu, as shown in Example 2–6.

Example 2–6 firstAction Menu Item

```
// Add a menu entry in the Tools menu
final Menubar menubar = Ide.getMenubar();
final JMenuItem firstActionMenu = menubar.createMenuItem
 firstAction,MenuConstants.WEIGHT_UNDEFINED);
 menubar.add
 (firstActionMenu,MenuManager.getJMenu(IdeMainWindow.MENU_TOOLS));
```

To add multiple menu items to an existing menu:

Example 2–7 adds three menu items to an existing menu.

Example 2–7 Sample Code to Add Three Menu Items to Menu

```
//  First menu of the first section
JMenuItem menu1 = contextMenu.createMenuItem(myAction1, 1f);
contextMenu.add(menu1, 1f);

//  Second menu of the first section
final JMenuItem menu2 = contextMenu.createMenuItem(myAction2, 2f);
contextMenu.add(menu2, 1f);

// <- First menu of the second section
final JMenuItem menu3 = contextMenu.createMenuItem(menu3, 1f);
contextMenu.add(menu3, 2f);
```

2.4.2.5 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 Section 2.4.5.4, "How to Invoke an Addin From a Main Window Menu."

To add a drop-down button to a toolbar:

The code in Example 2–8, 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.

Example 2–8 Adding a Drop-down Button to a Toolbar

```
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);
}
```

2.4.3 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.

2.4.3.1 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 Section 2.4.3.4, "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 Section 2.4.3.5, "How to Listen for the Active View."

2.4.3.2 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 Example 2–9.

Example 2–9 DockableFactory

```
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 Example 2–10.

Example 2–10 DockableWindow

```
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 Example 2–11.

Example 2–11 Install Factory

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

2.4.3.3 How to Position Dockable Windows

Example 2–12 shows how to center a dockable window with the Application Navigator. If the application navigator extension is not loaded, the window will be docked on the left (WEST).

Example 2–12 Centering a Dockable Window

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

2.4.3.4 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 Example 2–13.

Example 2–13 Adding an Active Listener

```
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 Example 2–14.

Example 2–14 Changing an Active Listener

```
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);
  }
}
```

2.4.3.5 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 onExample 2–15.

Example 2–15 Listening for the Active View

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

```
});
```

2.4.4 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.

2.4.4.1 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.

The code examples shown in this section are taken from the `HelloX` sample project, available as part of the Extension SDK. For more information, see Chapter 3, "Developing with the Extension SDK.".

`HelloX` 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.

2.4.4.2 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 constructor
- Define the `invoke` method
- Define the `getMenuSpecification` method
- Define the `isAvailable` method
- Define the `getIcon` method
- Define the `getName` method

2.4.4.2.1 How to Define the Constructor A wizard's constructor is invoked only once, when the wizard is loaded. The constructor should be lightweight, meaning that it should not create and hold object references, and you should defer such operations to the `initialize` method.

2.4.4.2.2 How to Define the Invoke Method

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. The `params` parameter is empty when the wizard is called from Tools menu, and contains the value set of the wizard's `wizardParameters` tag from the `extension.xml` file.

To define the invoke method:

Create code based on Example 2–16.

Example 2–16 Defining the Invoke Method

```
public boolean invoke(oracle.ide.addin.Context context, java.lang.String[] params)
{
  if ( !this.isAvailable(context) )
    return false;
  String greetee = null;
  JProject project = (JProject) context.getProject();
  // Get the parameter from the user.
  greetee = JOptionPane.showInputDialog
    (new JDialog(), prompt, wizName, JOptionPane.OK_CANCEL_OPTION);
  if ( greetee == null )
    return false;
  // Create the document and the node that represents it.
  if ( !createNode(project, greetee) )
    return false;
  return true;
}
```

2.4.4.2.3 How to Define the getMenuSpecification Method

This method is called to determine the appearance of the wizard's item in the **Tools** menu, when it is to be displayed. If the wizard is not installed in the **Tools** menu this method should return `null`.

To define the getMethodSpecification method:

Create code based on Example 2–17.

Example 2–17 Defining the getMethodSpecification Method

```
public MenuSpec getMenuSpecification()
 {
   if ( menuSpec == null)
     {
     Icon icon = getIcon();
     menuSpec = new MenuSpec(wizName, new Integer((int)'X'), (KeyStroke)null,
icon);
     }
    return menuSpec;
  }
```

2.4.4.2.4 How to Define the isAvailable Method

This method is called to determine if the wizard's New Gallery entry or Tools menu item should be enabled or disabled, given

the current context. For example, a wizard that operates on project nodes must only be enabled only when the current node is a project node.

To define the isAvailable method:

Create code based on Example 2–18.

Example 2–18 Defining the isAvailable Method

```
public  boolean isAvailable(oracle.ide.addin.Context context)
 {
   Project p = context.getProject();
   if ( (p != null) && (p instanceof JProject) )
     return true;
   return false;
}
```

2.4.4.2.5 How to Define the getIcon Method 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 Example 2–18.

Example 2–19 Defining the getIcon Method

```
public Icon getIcon()
{
 if (image == null)
 {
   image = GraphicsUtils.createImageIcon(
   GraphicsUtils.loadFromResource(imageName, this.getClass()));
 }
 return image;
}
```

2.4.4.2.6 How to Define the getName Method This method provides a human-readable name for the wizard, so enter an appropriate string.

2.4.4.3 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.

2.4.4.4 How to Add a Wizard to the Tools Menu

The Wizard Manager provides special support for wizards that are installed in the Tools menu. The Wizard Manager takes care of the details of adding the menu item and handling the user's selection of it.

Wizards can also be installed elsewhere in the user interface, such as other menu, context menus, or the tool bar, but in these cases a command must be defined, and installed and handled explicitly. In the extension manifest file, include an addin description for the wizard.

To install a wizard in the Tools menu:

1. Your wizard should implement `getMenuSpecification()`, as shown in Example 2–20. In this method, create a new instance of `oracle.ide.util.MenuSpec`, passing in the label, mnemonic, and icon of the required menu item.

   The code example is taken from the `HelloX` and `ConfigPanel` sample project, available as part of the Extension SDK. For more information, see Chapter 3, "Developing with the Extension SDK.".

Example 2–20 Installing a Wizard in the Tools Menu

```
public MenuSpec getMenuSpecification()
{
  if ( menuSpec == null)
    {
    Icon icon = getIcon();
    menuSpec = new MenuSpec(wizName, new Integer((int)'X'),
(KeyStroke)null, icon);
    }
 return menuSpec;
}
```

2. Define the wizard's `getMenuSpecification` method to return an icon and label. (This method may return null if the wizard is invoked only from the New Gallery).
3. Deploy the extension to install it. For more information, see Chapter 5, "Packaging and Deploying Extensions."

2.4.5 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.

2.4.5.1 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.

2.4.5.2 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 Chapter 3, "Developing with the Extension SDK.".

2.4.5.2.1 Handling the Event The code in Example 2–21 shows how to handle the event, which is triggered when you invoke the command.

Example 2–21 Handling the Event

```
/**
  * 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.

2.4.5.2.2 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.

2.4.5.2.3 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`.

2.4.5.3 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 Chapter 3, "Developing with the Extension SDK.".

2.4.5.3.1 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.

2.4.5.3.2 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 *Oracle Fusion Middleware Java API Reference for Oracle Extension SDK*.

2.4.5.3.3 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.

2.4.5.3.4 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.

2.4.5.4 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.

Menu items are associated with `IdeAction` objects.

When the state of the IDE changes, the enabled/disabled status of all menu items are reset as dictated by the active view's controller. When the user then selects a menu item or enters a keyboard shortcut for an item, the command named by the item's action is executed under the current context.

To define a menu item, provide these components:

- A command ID representing the command that is to be executed when the menu item is selected.

- An action that is associated with the command ID, and acts as the link between user actions and the controller. For more information, see Section 2.4.5.3, "How to Define an Action."
- A controller responsible for enabling and disabling the menu item and handling its events, and ultimately invokes the feature installed by the addin. For more information, see Section 2.4.6.3, "How to Implement a Controller."
- The menu item, which serves as the link between user actions and the controller.

Call the menu bar's `createMenuItem` method to create the menu item. The menu bar is a component of the IDE.

```
return Ide.getMenubar().createMenuItem(action);
```

Menus are static members of the Main Window. Call a menu's add method to add a menu item. Menu items should be immediately available when JDeveloper is launched, so create and install them in the addin's `initialize` method. The following adds a menu item to the Navigate menu.

```
public void addMenuItem()
 {
    Environment.getJMenu(IdeMainWindow.MENU_EDIT).
add(createMenuItem(contextInfoAction));
}
```

2.4.5.5 How to Invoke an Addin From a Context Menu

Some views, such as the Application Navigator 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.

User interface elements that are represented by objects of any class that implements `ContextMenuListener` may have a context menu. Context menus can be used almost anywhere in JDeveloper: most of the user interface's elements ultimately either implement this class, or are subcomponents of implementors.

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.

A context menu listener is polled through these methods:

- `poppingUp`, called when the context menu is being constructed. The listener should contribute its menu items at this time.
- `poppingDown`, called when the context menu is dismissed.
- `handleDefaultAction`, called on double-clicks, in which case exactly one of the polled listeners should return true, indicating that the action associated with its menu item should be invoked.

To allow an extension to be invoked from a context menu, add a menu listener to the view's context menu. The menu listener is given an opportunity to install the menu item when the context menu is recreated, which occurs whenever it pops-up.

To define a context menu item, provide these components:

- A command ID representing the command that is to be executed when the menu item is chosen.
- An action that is associated with the command ID, and acts as the link between user actions and the controller. For more information, see Section 2.4.5.3, "How to Define an Action."
- A controller responsible for enabling and disabling the menu item and handling its events, and ultimately invokes the feature installed by the addin. For more information, see Section 2.4.6.3, "How to Implement a Controller."
- A context menu listener, which provides a listener instance for each context menu to which the item can be added. The various views each manage their own context menus.

Example 2–22 adds a listener to the Application Navigator's context menu.

Example 2–22 Adding a Listener to Context Window

```
public void createCtxMenuListeners(ContextInfoController controller) {
     ContextMenu menu;
     // Add a listener to the Explorer's context menu.
     // This form will work for any manager or view that defines getContextMenu.
     menu = EditorManager.getEditorManager().getContextMenu();
     menu.addContextMenuListener(new ContextInfoMenuListener(controller));
     // Add a listener to the Navigator's context menu.
     NavigatorManager.getWorkspaceNavigatorManager().addContextMenuListener
(new ContextInfoMenuListener(controller), null);
}
```

The context menu listener should be added when the addin is loaded, so this task should be performed in the addin's initialize method. The creation of the controller and action can be deferred until the context menu is opened; the first time it calls the listener's method.

2.4.6 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.

2.4.6.1 How to Implement the EditorAddin Class

Editor extensions implement the `EditorAddin` class, which integrates the extension with the Editor Manager, and provides for the extension's installation at the time of JDeveloper's startup.

To implement the `EditorAddin` class:

- Define the constructor.
- Define the `initialize` method.
- Define the `getEditorClass` method.
- Define the `isDefault` method.
- Define the `getMenuSpecification` method.

The code examples shown here are taken from the `CustomEditor` sample project, available as part of the Extension SDK. For more information, see Chapter 3, "Developing with the Extension SDK.".

2.4.6.1.1 How to Define the Constructor The constructor should do as little as possible. Defer initialization tasks to the initialize method.

2.4.6.1.2 How to Define the initialize Method This method is called by the Addin Manager after the instance has been created. Register the editor with the Editor Manager here.

Editors must be registered with the IDE's EditorManager. A registration associates an editor with one or more node classes. When the user attempts to open an editor for a node, through either its context menu or the **View** menu, only editors that are registered for that node's class are enabled.

2.4.6.1.3 How to Define the getEditorClass Method This method names the class, an implementation of Editor which acts as the editable view.

2.4.6.1.4 How to Define the isDefault Method A node class can have a default editor; when the user double-clicks on the node it is opened in the default editor. To declare an editor to be the default for its registered node types, implement this method to return true.

2.4.6.1.5 How to Define the getMenuSpecification Method The Editor Manager adds the menu specification for the icon and text which appear at the top of each navigator, identifying it. The menu specification describes the appearance of the editors menu item.

2.4.6.2 How to Define an Editor Class

An editor is a view that displays an object for the user to modify. Create your own editor by extending `Editor`. The editor class is instantiated whenever a node of a type registered for the editor is opened for viewing.

An editor is associated with these other components:

- The Editor Manager, a component of the IDE, selects an editor class and instantiates it when the user opens a node.
- An `EditorAddin` serves as the go-between the editor and the Editor Manager. When JDeveloper starts, the addin registers the editor class.
- An object representing the data being edited, usually an implementation of `Node`.
- An editor component, an implementation of `JEditorPane`, which performs the actual modification of the data.
- A `Controller` that interprets user interface editing commands and invokes editor component methods.
- An `Explorer` that displays the structure of the data object. The explorer responds to user actions by calling editor methods.

An editor class must be responsible for:

- Instantiating the editor
- Initializing the editor
- Accessing the Controller
- Getting the root GUI component

- Responding to explorer events
- Responding to editor component events
- Generating update messages

The code examples shown here are taken from the CustomEditor sample project, available as part of the Extension SDK. For more information, see Chapter 3, "Developing with the Extension SDK.".

2.4.6.2.1 How to Instantiate the Editor The Editor Manager instantiates an editor class when the user opens a node in the Application Navigator. The Editor Manager calls the default constructor, so no context-specific parameters are available to it, as shown in Example 2–23.

Example 2–23 Editor Manager Calling the Default Constructor

```
public PropFileEditor()
 {
     systemClipboard = Toolkit.getDefaultToolkit().getSystemClipboard();
}
```

2.4.6.2.2 How to Initialize the Editor The Editor Manager initializes a editor instance by calling its setContext method. This method should extract the document to be edited from the context, and initialize the editor component, as shown in Example 2–24.

Example 2–24 Initializing the Editor Component

```
public synchronized void setContext( Context context )
{
   if ( context != null )
   {
      Element element = context.getElement();
      // A sanity check: if the context is bad the editor pane will be empty.
      if ( ( element != null ) &&
           ( element instanceof TextDocument ) )
      {
       super.setContext( context );
       document = (TextDocument) context.getElement();
       initializeGraphics( context );
      }
   }
}
```

2.4.6.2.3 How to Access the Controller The editor has an associated controller instance that intercepts and interprets user events. The IDE calls the editor's `getController` method to acquire the controller, which should create the controller when first called, as shown in Example 2–25.

Example 2–25 Calling the Editor's getController Method

```
public Controller getController()
{
  if ( editorController == null )
  {
    editorController = new PropFileEditorController( this );
  }
  return editorController;
```

```
}
```

2.4.6.2.4 How to Get the Root GUI Component The editor has an associated root graphical user interface component, and return the panel for this editor instance, as shown in Example 2–26.

Example 2–26 Getting the rootGUI component

```
public Component getGUI()
{
  return mainPanel;
}
```

2.4.6.2.5 How to Respond to Explorer Events An explorer view of a data object provides a means of navigating in a document in an editor: when the user selects an element in the explorer, the explorer instructs the editor to display the corresponding part of the data.

If the editor is intended to be integrated with an explorer it must provide one or more 'goto' methods. (This capability is not required by the Editor interface.) Example 2–27 is a method that causes the editor's cursor to move to a specified line of a text file.

Example 2–27 Method that Moves Cursor

```
public void setCaretPosition( int offset )
{
   editorComponent.setCaretPosition( offset );
}
```

2.4.6.2.6 How to Respond to Editor Component Events The editor component, as an implementation of `JEditorPane`, generates action events when its data changes. The editor should implement an interface such as `KeyListener`, and during initialization should call the editor component's `addKeyListener` method to register itself to receive the events. Example 2–28 contains an example of the `KeyListener` method `keyTyped` triggers a method that alerts other IDE objects, such as the recognizer, of the change.

Example 2–28 KeyListener Method

```
public void keyTyped( KeyEvent e )
{
 timer.restart();
}
```

2.4.6.2.7 How to Generate Update Messages When the data object's state changes other IDE components — such as the document's explorer — must be informed. This is accomplished through the notification mechanism. The editor creates an `UpdateMessage` instance and broadcasts it to the document's observers, as shown in Example 2–29.

Example 2–29 UpdateMessage Instance

```
public void keyTyped( KeyEvent e )
{
  timer.restart();
```

```
}
```

2.4.6.3 How to Implement a Controller

If your extension implements a view class it should also implement a controller class to handle the view's events. To implement the controller, use the `Controller` interface.

To implement a controller class provide definitions for the following members:

- Defining the `handleEvent` method
- Defining the `update` method
- Define command constants

The code examples shown here are taken from the FirstSample and ClassSpy projects, two of the sample projects available as part of the Extension SDK. For more information, see Chapter 3, "Developing with the Extension SDK."

2.4.6.3.1 How to Define the handleEvent Method The `handleEvent` method is a switch statement which handles actions referring to selected commands, and defers the rest to its supervisor. You define this method to handle commands defined for the extension, and others that the extension must override, as shown in Example 2–30.

Example 2–30 Defining the handleEvent method

```
public boolean handleEvent(IdeAction action, Context context) {
 int cmdId = action.getCommandId();
 // Handle actions containing this command:
 if (cmdId == CONTEXT_INFO_CMD_ID) {
     CommandProcessor cmdProc = CommandProcessor.getInstance();
     String commandName = action.getCommand();
     Command command = cmdProc.createCommand(commandName, context);
     // Use command processor to execute command.
     try {
       cmdProc.invoke(command);
       }
     catch (Exception e) {
       System.err.println(e.toString());
       }
     finally {
       return true;
       }
     }
 // Let the IDE try to find another Controller to handle this action.
 return false;
}
```

An event may be handled directly by calling methods that act on the data. Alternatively, an event may be handled indirectly by creating a command object and invoking it through the command processor. Some cases where the latter option is preferable are:

- The event has an applicable default behavior. If the action's command class is compatable with the data then there is no need to re-implement the command. For example, `Ide.SAVE_CMD` is the default command class for the 'Save' action and is defined for any data class that implements the `Document` interface.
- The event is to be undoable. The command processor manages undo stacks.

- The event interacts with the system or is otherwise computationally intensive. The command process invokes commands in their own threads.

2.4.6.3.2 How to Define the update Method The update method is essentially a switch statement which enables and disables selected actions, and defers the rest to its supervisor. You define this method for actions defined for the extension, and others that the extension must override, as shown in Example 2–31.

Example 2–31 Defining the update Method

```
public boolean update(IdeAction action, Context context) {
  int cmdId = action.getCommandId();
  // Set the enabled status for relevant actions.
  if ( cmdId == CONTEXT_INFO_CMD_ID ) {
     action.setEnabled(enableContextInfo(context));
     return true;
  }
  // Let the IDE try to find another Controller to update this action.
  return false;
}
```

2.4.6.3.3 How to Define Command Constants Define the commands required by your extension by giving them command IDs. The `Ide.findOrCreateCmdID` static method assigns unique command IDs, as shown in Example 2–32.

Example 2–32 Ide.findOrCreateCmdID Method

```
public static final int CONTEXT_INFO_CMD_ID =
    Ide.findOrCreateCmdID("ContextInfoController.CONTEXT_INFO_CMD_ID");
```

2.4.6.4 How to Specify an Editor Layout

Layouts are a feature of JDeveloper that allow the dockable windows to be arranged to accommodate a particular task. Layouts are modified by user actions, saved when the user changes to another task, and restored when the user returns.

Generally, users define custom layouts in the Environment page of the Preferences dialog (available from the Tools menu), or using layout managers for the Java Visual Editor. However, an editor extension can specify a layout for that editor.

The layout is specified by the methods of `LayoutSelector`, which is a subclass of `Editor`. Editors should override these methods:

- `getPreferredLayoutURL` names the properties file in which the state of the layout is preserved between sessions. The layout is extracted from this file when the editor is first used in a session, and it is written back to the file when JDeveloper is closed.
- `onPreferredLayoutActivate` is called when the layout file is not found, to initialize the layout. This method should open, close, and arrange dockable windows.

2.4.6.5 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 Component Palette 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 Component Palette 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 *Oracle Fusion Middleware Java API Reference for Oracle Extension SDK*.

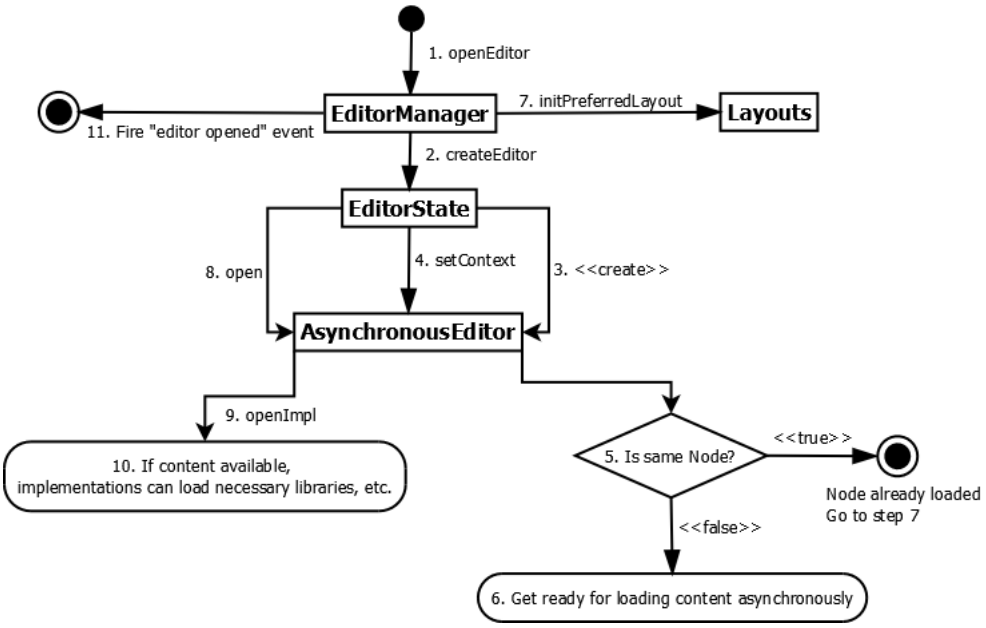
Life-cycle methods like `open`, `close`, `activate` are final. The asynchronous editor introduces new but similar life-cycle methods that take an extra argument: a flag that indicates whether the editor's UI has been loaded.

2.4.6.5.1 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 Application Navigator:

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 Component Palette. The Component Palette 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`.

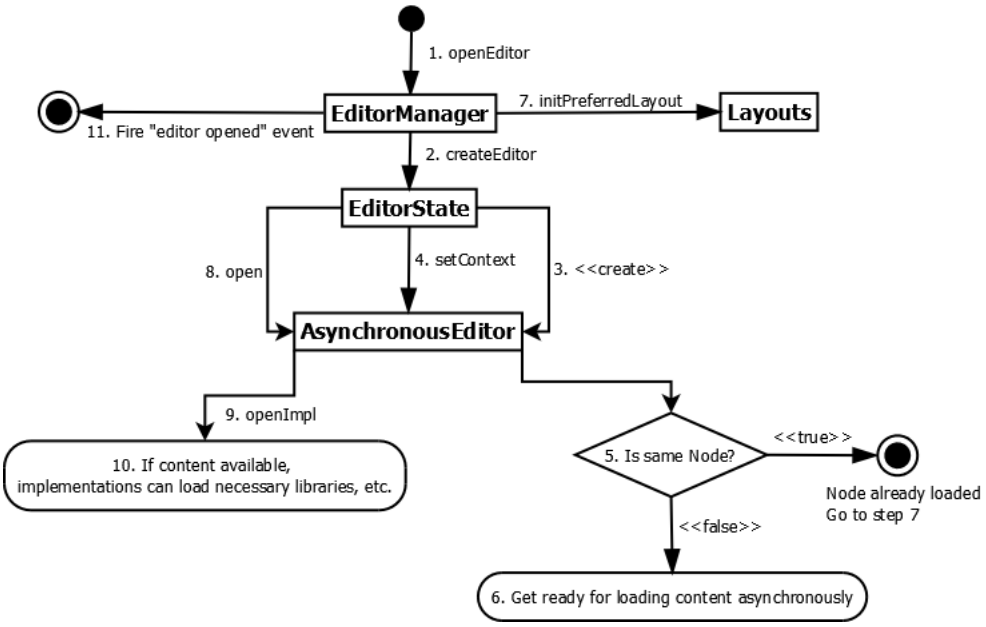
This is illustrated in Section 2–4, "Opening the Editor".

Figure 2–4 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 Section 2–5, "Getting the Editor's GUI".

Figure 2–5 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 Component Palette 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.

2.4.7 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.

2.4.7.1 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.

2.4.7.2 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 Example 2–33. 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 Chapter 3, "Developing with the Extension SDK."

Example 2–33 Registration Operation

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

2.4.7.3 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.

2.4.7.4 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.

2.4.8 How to Add New Component Palette Pages

The Component Manager lets you add individual pages to the Component Palette that users of your extension can employ to build a consistent user interface or perform other standardized tasks. The Component Palette 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 *Oracle Fusion Middleware Java API Reference for Oracle Extension SDK*.

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

2.4.8.1 Understanding Component Palette Pages

The Component Palette lets users add commonly used data structures to a project, application, or other content in JDeveloper. The Component Palette 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 palette page varies depending on the type of file selected. For example, if you are editing an HTML file, the Component Palette 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 Component Palette, select **View** > **Component Palette**.

At the extension developer's level, the Component Manager provides you with a declarative way to add pages to the Component Palette, 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 Palette Manager, you can add one or more pages of components to the Component Palette, 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 Component Palette page.
- Declare dynamic components, populating the Component Palette page when it is loaded.
- Make help 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 palette components.

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

2.4.8.2 How to Declare Static Content for a Component Page

In many cases, the extension palette'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 Example 2–34.

The shape of the Component Palette 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 Component Palette 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 Component Palette 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 Component Palette, for example name, description, icon, etc.

Example 2–34 demonstrates how to declare a palette page with a single component.

Example 2–34 Declaring a Palette Page with a Single Component

```
<?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</import>
 </dependencies>
 <hooks>
   <palette-hook xmlns="http://xmlns.oracle.com/jdeveloper/1013/extension">
     <page>
      <name>My Sample Components</name>
      <pageId>SampleStatic</pageId>
      <showForTypes>
        <type>java</type>
      </showForTypes>
      <technologyScopes>
         <technologyScope>Java</technologyScope>
         <technologyScope>JavaBeans</technologyScope>
      </technologyScopes>
      <type>java</type>
      <group>
       <name>Components</name>
        <groupId>SampleStatic-Components</groupId>
         <showForTypes>
           <type>java</type>
         </showForTypes>
         <technologyScopes>
           <technologyScope>Java</technologyScope>
           <technologyScope>JavaBeans</technologyScope>
         </technologyScopes>
         <type>java</type>
         <section>
            <sectionId>SampleStatic-Components-Section1</sectionId>
            <name/>
            <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>
            </item>
          </section>
        </group>
      </page>
   </palette-hook>
  </hooks>
</extension>
```

2.4.8.3 How to Declare a Dynamic Component for a Palette Page

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

Palette Page Providers are developed by client developers using the published Palette API and the page providers class name is provided to the Component Palette in an extension manifest file.

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

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

Example 2–35 shows an extension manifest that identifies a Palette Page Provider class name.

Example 2–35 Extension Manifest Identifying a Palette 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>
```

Example 2–36 shows a Palette Page Provider that adds a palette page to the Component Palette. This class extends `PalettePageProvider`.

The element `pageProvider.providerClassName` in `extension.xml` registers this class with the Component Palette as a page provider. The Component Palette 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 Component Palette.

Example 2–36 Palette Page Provider

```
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.
        }
}
```

Example 2–37 shows an example of `SamplePages.java`. 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`.

Example 2–37 SamplePages.java

```
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 "";
          }
}
```

Example 2–38 shows an example of `SamplePalettePage.java`.

Example 2–38 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 );
  }
}
```

Example 2–39 SamplePaletteGroup.java

```
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);
     }
}
```

Example 2–40 shows an example of `SamplePaletteItem.java`.

Example 2–40 SamplePalette Item.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);
  )
)
```

2.4.8.4 How to Provide Help for a Component Page

There are two ways to provide help on a component:

- The Palette Item description is displayed as a tool tip for the component, visible when your user rolls the mouse over the component in the Component Palette. You can use the "\n" newline character to break the description into multiple lines.
- The user can right-click a Palette 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.

Example 2–41 demonstrates how to do this.

Example 2–41 Providing Help for a Component Page

```
  <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 Example 2–42.

Example 2–42 Proving helpid directly

```
 <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.

2.4.8.5 How to Augment the Search for a Palette Item

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

To make the Component Palette 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 ComponentPalette extension hook as shown in Example 2–43.

Example 2–43 Component Palette Extension Hook

```
  <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 Example 2–44.

Example 2–44 Palette Search

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

```
}
```

2.4.9 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.

2.4.9.1 How to Implement Preferences

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

The data structure used to store preferences is `HashStructure`, also described in *Oracle Fusion Middleware Java API Reference for Oracle Extension SDK*. 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.11.2.0.x.y`, where "11.2.0" 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 Section 2.4.9.2, "How to Implement the Data Model."
2. Implement the data model for storing the preference. For more information, see Section 2.4.9.3, "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 Section 2.4.9.4, "How to Register a UI Panel."
4. Obtain the preferences model. For more information, see Section 2.4.9.5, "How to Obtain the Preference Model."
5. Listen for changes to the preferences, so they can be stored. For more information, see Section 2.4.9.6, "How to Listen for Changes."

2.4.9.2 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 *Oracle Fusion Middleware Java API Reference for Oracle Extension SDK*.

Example 2–45 contains a typical implementation pattern, with comments for the details.

Example 2–45 Implementation Pattern

```
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..
}
```

2.4.9.3 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 *Oracle Fusion Middleware Java API Reference for Oracle Extension SDK*.

Example 2–46 shows a typical implementation pattern.

Example 2–46 DefaultTraversalPanel

```
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());
 }
}
```

2.4.9.4 How to Register a UI Panel

Example 2–47 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.

Example 2–47 XML Fragment for Extension Manifest

```
<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>
```

2.4.9.5 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 );
```

2.4.9.6 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 Example 2–48.

Example 2–48 Exposing Methods for Attaching to your Model Object

```
public final class CoolFeaturePrefs extends
{
 //...
 public void addStructureChangeListener( StructureChangeListener l )
  {
    _hash.addStructureChangeListener( l );
  }
   public void removeStructureChangeListener( StructureChangeListener l )
  {
    _hash.removeStructureChangeListener( l );
   }
 //...
```

```
}
```

2.4.10 Understanding Project Properties

Project properties are similar to project preferences, see Section 2.4.9, "Understanding Preferences."

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

2.4.11 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.

2.4.11.1 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` show in Example 2–49.

Example 2–49 Undoable Edit

```
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;
    }
}
```

2.4.11.2 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 Example 2–50

Example 2–50 Doing and Undoing Changes

```
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;
 }
}
```

2.5 How to Define and Use Trigger Hooks

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

2.5.1 How to Register a <trigger-hook-handler>

To define your own trigger-hook, you register a trigger hook handler as shown in Example 2–51, "Registering a Trigger Hook Handler".

Example 2–51 Registering a Trigger Hook Handler

```
<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 as shown in Example 2–51, "Registering a Trigger Hook Handler", and set the `register-as-hook` attribute to `true`.

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.

2.5.2 How to Define Trigger Hooks for your Extension

To use a trigger hook, use syntax similar to Example 2–52, "Defining Trigger Hooks".

Example 2–52 Defining Trigger Hooks

```
    <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>
```

2.5.3 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`.

2.5.4 How to Define Rules and Condition Triggers Sections

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

2.5.4.1 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 Example 2–53.

Example 2–53 Built-in Rule Types

```
<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.

2.5.4.2 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 Example 2–54:

- 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.

Example 2–54 Simple Rule

```
  <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>
```

2.5.4.3 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`.

2.5.4.4 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`. So, for example, in order to define the `context-has-source-node-1` rules in Example 2–54, 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

2.5.4.5 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 Example 2–55.

Example 2–55 Composite Rules

```
<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>
```

2.5.4.6 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 inExample 2–56, Example 2–57, and Example 2–58.

Example 2–56 Gallery Items

```
  ...
  <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 2–57 Controllers

```
<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 2–58 Context Menus

```
<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>
```

2.5.4.7 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'd 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).

2.6 How to Add Online Help Support

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.

2.6.1 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 more information about creating a help system, see the *Oracle Fusion Middleware Developer's Guide for 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.

2.6.2 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 Example 2–59.

Example 2–59 Help Registered in <trigger-hooks>

```
<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>
```

2.7 How to Add Print Support

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

2.7.1 How to Register DocumentPrintFactory

Printing is supported by declaratively registering a `DocumentPrintFactory` in extension.xm, as shown in Example 2–60.

Example 2–60 Registering DocumentPrintFactory

```
<hooks>
```

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

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

2.7.2 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.

Example 2–61 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.

Example 2–61 Registering the View Class

```
<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`. Example 2–62 shows how to use the same `DocumentPrintFactory` for multiple views.

Example 2–62 Using 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>
```

2.7.2.1 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.

Example 2–63 Registering a PageableFactory to Handle a 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 Example 2–63 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.

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:

3.1 About Developing with the Extension SDK

The JDeveloper Extension Software Developer Kit (SDK) includes a collection of projects containing sample code and the 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.

3.2 Downloading and Installing the Extension SDK

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 *Oracle Fusion Middleware User Guide for 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 Application Navigator, 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" in *Oracle Fusion Middleware User Guide for Oracle JDeveloper*, 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 Application Navigator, and the sample projects are listed.

3.2.1 What Happens When you Install the Extension SDK

When you successfully install the Extension SDK, the sample application is optionally open in the Application Navigator, 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.

3.2.2 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.

3.3 Using with the Sample Extensions

The sample projects available in the Extension SDK are briefly described in this section. For more information, see the comments in each project, which provides guidance on the functionality.

If you need more information about JDeveloper, see the relevant section of the *Oracle Fusion Middleware User Guide for Oracle JDeveloper*.

For information about running the samples, see Section 3.4, "How to Run the Sample Projects."

- **AllSamples.jpr**: This sample provides a deployment profile that can be used to run all the sample projects at once.
- **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:
  - Add your own custom editor.
  - Get files from an application and project.
  - Get the status of individual files in a project.
- **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.
  - Adds an option to the IDE Audit properties page.
  - Manages the public final static methods names.
  - shows how to use the refactoring features of the IDE.
- **Balloon.jpr:** Demonstrates how to install a balloon notification into the JDeveloper status bar.

  Shows how to:
  - Add an Action Listener to a dialog or Balloon element.
  - Interact with the IDE Status bar, both adding and removing items.
  - Get an icon from `OracleIcons`.
- **ClassBrowser.jpr:** Shows how to use the JDeveloper `ClassBrowser`.
  - Shows how to setup a single page wizard.
  - Adds a context menu to Navigator and Observer UI sections.
- **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.
  - Adds an item to the **View** menu.
  - Shows how to enable and disable a menu option based on what is currently selected in the Application Navigator.
  - Shows how to setup a multi-page wizard.
- **ClassSpy.jpr:** Shows the simplest way to identify the class of a given node in the Navigator, and how to write its name in the log window. Shows how to:
  - Add a context menu to the source editor, explorer, and Application Navigator using `extension.xml`.
  - Work with a Context object.
  - Display information in a log window.

  Code from this sample is used to illustrate Section 2.4.6.3, "How to Implement a Controller."
- **ClickableURL.jpr:** Shows how to generate a clickable URL in the log window.

  - Adds a context menu to the Application Navigator using the `extension.xml` file.
  - Shows interaction with a new log window.
  - Shows how to work with a file URL.
  - Works with `MessageWindow` to place data in a log.
  - Shows how to launch a browser from the IDE.
- **CodeInteraction.jpr:** Shows how to get text from the code editor, and launches a Google search on the currently selected text.
  - Adds context menu to the source editor.
  - Shows how to enable the context menu item based on a condition in the source editor.
  - Shows how to call a browser with data from the source editor.
- **ConfigPanel.jpr:** Demonstrates how to store and retrieve preferences, and install a panel into the Preferences dialog.
  - Adds a menu item to the **Tools** menu using an `Addin` class.
  - Shows interaction with the log window.
  - Shows how to use a listener to listen for changes to the IDE Preferences model.
  - Code from this project is used to illustrate Section 2.4.4.4, "How to Add a Wizard to the Tools Menu."
- **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.
  - Shows how to add a New Gallery item using `extension.xml`.
  - Shows how to use a background task with monitor.
  - Uses JEWTDialog and Panel to display data.
  - Shows how to listen for Enter key press in text fields and do something with the event.
- **CreateStructure.jpr:** Demonstrates creating a new empty application (jws file) and adding a new empty project (jpr file) to it.
  - Works with the file system IO, creating files, and directories
  - Uses oracle.ide APIs to build structure rather than JDeveloper APIs
  - Shows the use of a logs() method for sending info to the logs window
- **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.

  Code from this sample project is used to illustrate Section 2.4.6.1, "How to Implement the EditorAddin Class" and Section 2.4.6.2, "How to Define an Editor Class."
- **CustomExtensionHook.jpr:** Demonstrates a custom extension hook, which allows other extensions to plug into your extension.

  Shows how to add a menu item declaratively in `extension.xml`, specifically to the **View** menu.

- **CustomNavigator.jpr:** Demonstrates how to install a custom navigator window. Installs a "Favorites" window which can be used to store frequently accessed files.
  - Shows how to create your own navigator window.
  - Shows file IO methods for saving a file to the system directory.
  - Shows how to add a customer icon to the window.
  - Shows how to work with selected items in a menu.
  - Adds a separate controller for just one menu area.
- **DebugObjectPreferences.jpr:** Demonstrates how to install a custom object preferences renderer for the Data window of the debugger.
  - Adds its parameter to the debugger output window in the Data tab.
  - Shows how to work with the `debugger_hook` in `extension.xml`.
- **DockableWindow.jpr:** Shows how to implement a custom dockable window.
  - Shows how to work with `DockableFactory`.
  - Uses Addin initialize to add extension to **View** menu.
  - Shows how to work with `DockableWindow`.
- **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 *Oracle Fusion Middleware User Guide for Oracle JDeveloper*.

  Shows how to setup an external tool and integrate it into the JDeveloper IDE, for example, passing the current editor content to an external tool for review or processing.
- **ExternalToolImportExport.jpr:** Imports and exports the properties associated with a list of External Tools.

  Works with reading and writing `ExternalTool` properties.
- **ExternalToolMacros.jpr:** Shows how to write an addin that installs a custom macro for use by external tools.

  Shows the use of `<externaltools>` and `<macros>` tags in `extension.xml`.
- **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.
  - Shows how to work with `extension.xml` tags for `<menu-hook>`, `<gallery>`, `<context-menu-listener>`, `<toolbars>` and `<actions>`.
  - All IDE integration points call a `Wizard` class that displays a message box.
  - Code from this sample project is used to illustrate Section 2.4.5.2, "How to Implement a Command," Section 2.4.5.3, "How to Define an Action," and Section 2.4.6.3, "How to Implement a Controller."
- **FlatEditor.jpr:** Demonstrates how to implement a form-based overview editor similar to the one used for `extension.xml`.
  - Shows `<ide:editor>` in `extension.xml`.

  - Adds a custom tab to the editor window.
  - Shows how to work with `VerticalFlowLayout`.
- **HelloX.jpr:** This is a sample that demonstrates a number of core concepts, including installing New Gallery items, actions, and menu items.
  - Shows how to call a dialup window.
  - Shows how to add something to the New Gallery using `extension.xml`.
  - Code from this sample project is used to illustrate Section 2.4.4.1, "How to Set Up a Wizard Project" and Section 2.4.4.4, "How to Add a Wizard to the Tools Menu."
- **LayoutMenuFilter.jpr:** Shows how to filter the top level menus in the IDE when the layout changes.
  - Allows you to set only those menu items that you want shown.
  - Uses `ExtensionRegistry`.
  - Uses `addIdeListener` to get IDE events and act on them.
- **MethodCallCounter.jpr:** Uses JOT (Java Object Tool) to count the number of occurrences of a named method in some code.
  - Shows how to determine node type in the navigator and source editor menus.
  - Adds context menu options to the navigator and source editor.
  - Uses `showInputDialog` to allow parameter entry at runtime.
  - Uses `MessageDialog`.information to display results.
  - Shows how to use two method declarations in one class. The method used depends on parameters passed to it.
- **OpenNodes.jpr:**Implements a dockable window which shows nodes that are open in the NodeFactory, and tracks when nodes are opened and closed.
  - Shows how to create and implement a dockable window in the IDE.
  - Overrides `toString()` to set specific string formatting.
  - Uses `Jtable` inside a Jpanel, and shows setup and configuration.
  - Uses a timer to flash a color change on a table change then return to normal state.

  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 navigator, like those used by the version control extensions.
  - Shows how to set up multiple Actions in `extension.xml`.
  - Works with `OverlayCache`.
  - Works with `IdeActions`.
- **ProgressBar.jpr:** Shows how to implement a progress bar during the execution of some lengthy task.
  - Uses `oracle.ide.progressbar`.
  - Illustrates a background running process.

  - Shows an indeterminate progress bar.
  - Writes to the log window in a background process.
- **ProjectSettings.jpr:** Demonstrates storing and retrieving project properties and adding custom project settings UI.
  - Illustrates use of `<settings-ui>` tag in `extension.xml`.
  - Uses `ProjectSettingsTraversablePanel` to layout project settings page.
  - Uses `HashStructure` and `PropertyStorage` to store and retrieve project settings data.
- **StructurePane.jpr:** Shows how to display your own content in the Structure Window.
  - Shows how to register a Structure Window using `ExplorerManager`.
  - Shows how to extend `AbstractTreeExplore`.
  - Code from this sample project is used to illustrate Section 2.4.7.2, "How to Register and Initialize a Structure Explorer."
- **UpdateCenter.jpr:** Demonstrates how to install a custom update center as part of an extension.
  - Shows how to use `<update-hook>` and `<update-center>` tags in `extension.xml`.
  - Shows how your extension can add its own Update Center item to the Updates Center page of the Check for Updates wizard.
- **VersionControlRCS.jpr:** This is a sample version control integration using the version control system (VCS) API.
  - Shows implementation of VCS related tags such as `<vcs-hook>`, `<vcs-profile>` and `<vcs-menu>`.
  - Shows implementation of action tags such as <automatic-action> and `<refresh-action>`.

3.4 How to Run the Sample 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:

4.1 About Testing and Debugging Extensions

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

4.2 Debugging Extension Code

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

4.2.1 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:

- Run the extension using one of the following:
  - In the Application Navigator, select the extension project and from the main menu choose **Run Extension**.
  - From the context menu of the project in the Application Navigator, or in the source editor, choose **Run**.
  - From the **Run menu** on the JDeveloper toolbar choose **Run Extension**.

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

4.2.2 How to Debug a JDeveloper Extension

With the extension running in a debug version of JDeveloper, you can text the functionality and use the JDeveloper debugging features to find and fix problems. For more information see "Debugging Applications" in *Oracle Fusion Middleware User Guide for Oracle JDeveloper* .

To debug an extension in JDeveloper:

1. Enter suitable breakpoints in your code.
2. Run the extension in JDeveloper in debug mode using one of the following:
   - In the Application Navigator, select the project and from the main menu choose **Run > Debug Project**.
   - From the context menu of the project in the Application Navigator, or in the source editor, choose **Debug**.
   - Click the **Debug icon** on the JDeveloper toolbar.

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

Is it necessary to include

4.3 Troubleshooting Debugging

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 *Oracle Fusion Middleware User Guide for Oracle JDeveloper*.

- **Symptom:** Class case 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 Section 1.5, "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:

5.1 About Packaging and Deploying Extensions

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 Application Navigator 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 Application Navigator.
4. If the extension references external libraries, open the bundle manifest `manifest.mf` by locating it in the Application Navigator under the META-INF node, and change the line

   Bundle-ClassPath: .

   so that it says:

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

5.2 Packaging the Extension

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 *Oracle Fusion Middleware User Guide for Oracle JDeveloper*.

5.2.1 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 Application Navigator, 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 Section 1.8.1, "Understanding Dependencies."

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

5.2.2 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 Application Navigator, 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 Chapter 4, "Testing and Debugging Extensions."

5.3 Deploying an Extension

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. For more information, see "Working with Extensions" in *Oracle Fusion Middleware User Guide for Oracle JDeveloper*

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
http://forums.oracle.com/forums/forum.jspa?forumID=83.

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:

- Section A.1, "Elements Installed in the File System"
- Section A.2, "Elements Installed in the IDE"

A.1 Elements Installed in the File System

When you install the Extension SDK:

- The Extension SDK API documentation is installed in *jdev_install*/jdeveloper/jdev/extensions/oracle.jdeveloper.esdk.
- The extension samples application extensionsdk.jws which contains the sample projects is installed in *jdev_install*/jdeveloper/mywork/extension-samples-11.1.2.0.nn.nn.nn .

A.2 Elements Installed in the IDE

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 Application Navigator. 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:

B.1 About Uninstalling the Extension SDK

You can disable the Extension SDK to optimize JDeveloper performance, 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.

B.2 How to Disable the 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 *Oracle Fusion Middleware User Guide for 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**.

B.3 How to Delete the 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 Application Navigator, 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 Application Navigator toolbar, and choose **Delete Application**.

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

B.4 How to Uninstall the 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`