

Platform Plug-in Developer Guide

Programmer's Guide

Table of Contents

<u>Welcome to Eclipse</u>	1
<u>Who needs a platform?</u>	1
<u>End users</u>	1
<u>Software developers</u>	2
<u>The holy grail</u>	2
<u>What is Eclipse?</u>	2
<u>Open architecture</u>	2
<u>Platform structure</u>	3
<u>Platform runtime</u>	3
<u>Resource management (workspace)</u>	3
<u>Workbench UI</u>	4
<u>Help system</u>	4
<u>Version and configuration management (VCM)</u>	4
<u>Out of the box</u>	4
<u>Platform architecture</u>	5
<u>Platform SDK roadmap</u>	5
<u>Runtime core</u>	6
<u>Resource management</u>	6
<u>Workbench UI</u>	6
<u>Help System</u>	7
<u>Version and Configuration Management (VCM)</u>	7
<u>Java Development Tooling (JDT)</u>	7
<u>Plug-in Development Environment (PDE)</u>	7
<u>Plug it in: Hello World meets the workbench</u>	9
<u>A minimal plug-in</u>	9
<u>Hello world view</u>	10
<u>The hello world plug-in</u>	11
<u>Plug-in ids</u>	12
<u>Installing and running the plug-in</u>	12
<u>Beyond the basics</u>	15
<u>Resources overview</u>	17
<u>Resources and the workspace</u>	17
<u>A sample resource tree</u>	17
<u>Resources and the local file system</u>	18
<u>Our sample tree on disk</u>	18
<u>Our sample tree in code</u>	19
<u>Mapping resources to disk locations</u>	20
<u>Resource API and the file system</u>	21
<u>Resource properties</u>	21
<u>Beyond the basics</u>	21
<u>Plugging into the workbench</u>	23
<u>Quick tour of the workbench</u>	23
<u>Views</u>	24
<u>Editors</u>	24
<u>Workbench under the covers</u>	24
<u>Workbench</u>	24

Table of Contents

Page	25
Views and editors	26
Basic workbench extension points	27
org.eclipse.ui.views	28
org.eclipse.ui.viewActions	31
org.eclipse.ui.editors	32
Editor action contributors	33
Editors and content outliners	34
org.eclipse.ui.editorActions	35
org.eclipse.ui.popupMenus	36
org.eclipse.ui.actionSets	38
The plug-in class	40
Plug-in definition	40
AbstractUIPlugin	41
Workbench menu contributions	41
Menu and toolbar paths	42
Named groups	42
Fully qualified menu and tool paths	43
Externalizing UI labels	43
Adding new menus and groups	43
More workbench extensions	45
org.eclipse.ui.perspectives	45
Perspectives	45
Workbench part layout	46
org.eclipse.ui.perspectiveExtensions	46
org.eclipse.ui.elementFactories	47
IAdaptables and workbench adapters	47
Element factories	48
org.eclipse.ui.resourceFilters	48
Beyond the basics	49
Dialogs and wizards.....	51
Standard dialogs	51
Application dialogs	51
Dialog settings	53
Wizards	53
Wizard dialog	54
Wizard	54
Wizard page	54
Workbench wizard extension points	55
org.eclipse.ui.newWizards	55
Pages	57
Wizard	58
org.eclipse.ui.importWizards	59
org.eclipse.ui.exportWizards	60
Using wizard dialogs	61
Multi-page wizards	62
Validation and page control	62

Table of Contents

<u>Preferences and properties</u>	65
<u>Preferences</u>	65
<u>org.eclipse.ui.preferencePages</u>	65
<u>Preference Page</u>	67
<u>Defining the page</u>	67
<u>Plug-in preference store</u>	68
<u>Retrieving and saving preferences</u>	69
<u>Field editors</u>	70
<u>Property pages</u>	70
<u>org.eclipse.ui.propertyPages</u>	70
<u>Properties page</u>	72
<u>JFace: UI framework for plug-ins</u>	73
<u>JFace and the workbench</u>	73
<u>JFace and SWT</u>	73
<u>Viewers</u>	73
<u>Standard viewers</u>	74
<u>List-oriented viewers</u>	74
<u>Text viewer</u>	75
<u>Viewer architecture</u>	75
<u>Input elements</u>	75
<u>Content viewers</u>	75
<u>Viewers and the workbench</u>	76
<u>Actions and contributions</u>	76
<u>Actions</u>	76
<u>Contribution items</u>	77
<u>Contribution managers</u>	77
<u>User interface resources</u>	77
<u>Image descriptors and the registry</u>	78
<u>Image descriptor</u>	78
<u>Image registry</u>	79
<u>Plug-in patterns for using images</u>	79
<u>Specifying the image in the plugin.xml</u>	79
<u>Explicit creation</u>	79
<u>Image registry</u>	79
<u>Label providers</u>	79
<u>Plug-in wide image class</u>	80
<u>Font registry</u>	80
<u>JFaceResources</u>	80
<u>Long-running operations</u>	80
<u>Runnables and progress</u>	81
<u>Modal operations</u>	81
<u>Standard Widget Toolkit</u>	83
<u>Portability and platform integration</u>	83
<u>Consistency with the platform</u>	83
<u>Widgets</u>	84
<u>Widget application structure</u>	84
<u>Display</u>	85
<u>Shell</u>	85

Table of Contents

<u>Parents and children</u>	85
<u>Widget life cycle</u>	85
<u>Widget creation</u>	85
<u>Style bits</u>	85
<u>Resource disposal</u>	86
<u>Controls</u>	86
<u>Events</u>	89
<u>Untyped events</u>	91
<u>Custom Widgets</u>	91
<u>Native implementation</u>	92
<u>Extending an existing widget</u>	93
<u>Custom drawn implementation</u>	93
<u>Layouts</u>	94
<u>Widget layout concepts</u>	95
<u>Fill Layout</u>	95
<u>RowLayout</u>	96
<u>Grid Layout</u>	96
<u>Custom layouts</u>	97
<u>Threading issues for clients</u>	98
<u>Native event dispatching</u>	98
<u>Toolkit UI threads</u>	98
<u>SWT UI thread</u>	99
<u>Executing code from a non–UI thread</u>	100
<u>The workbench and threads</u>	100
<u>Error handling</u>	101
<u>IllegalArgumentException</u>	101
<u>SWTException</u>	101
<u>SWTError</u>	101
<u>Graphics</u>	101
<u>Graphics context</u>	102
<u>Fonts</u>	102
<u>Colors</u>	102
<u>Images</u>	102
<u>Graphics object lifecycle</u>	103
<u>Creation</u>	103
<u>Painting</u>	103
<u>Disposal</u>	104
<u>Editors</u>	105
<u>Workbench editors</u>	105
<u>Text editors and JFace text</u>	106
<u>JFace text</u>	106
<u>Content outliners</u>	107
<u>Resource and workspace API</u>	111
<u>Resource markers</u>	111
<u>Marker operations</u>	111
<u>Marker creation</u>	111
<u>Marker deletion</u>	112
<u>Marker attributes</u>	112

Table of Contents

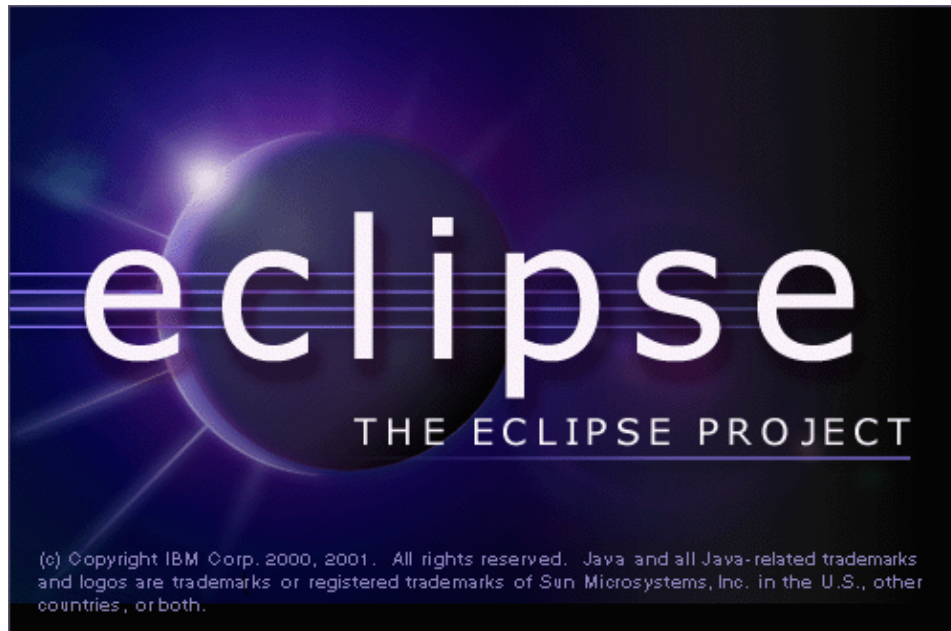
<u>Querying markers</u>	113
<u>Marker persistence</u>	113
<u>Extending the platform with new marker types</u>	113
<u>Tracking resource changes</u>	115
<u>Resource change processing</u>	115
<u>Batch changes and resource deltas</u>	115
<u>Resource change events</u>	116
<u>Implementing a resource change listener</u>	117
<u>Incremental project builders</u>	119
<u>Invoking a build</u>	119
<u>Defining an incremental project builder</u>	120
<u>Full build</u>	121
<u>Incremental build</u>	121
<u>Associating an incremental project builder with a project</u>	121
<u>Workspace save participation</u>	122
<u>Implementing a save participant</u>	122
<u>Using previously saved state</u>	125
<u>Accessing the save files</u>	125
<u>Processing resource deltas between activations</u>	125
<u>Project natures</u>	126
<u>Defining a nature</u>	126
<u>Associating the nature with a project</u>	127
<u>Scripting a user interface</u>.....	129
<u>Concepts</u>	129
<u>Limitations of scripts</u>	129
<u>Writing a batch script</u>	130
<u>Writing user interface scripts</u>	130
<u>Body</u>	130
<u>Form</u>	131
<u>UI elements</u>	131
<u>Layout controls</u>	131
<u>Scripting</u>	131
<u>Registering scripts with the workbench</u>	132
<u>Using UI scripting with Java</u>	132
<u>Explicit use of script adapters</u>	133
<u>References</u>	133
<u>Plugging in help</u>.....	135
<u>Building a help plug-in</u>	135
<u>Defining the help topics</u>	136
<u>topics Concepts.xml</u>	136
<u>topics Tasks.xml</u>	136
<u>topics Ref.xml</u>	137
<u>Creating the infoset</u>	137
<u>infoset SampleGuide.xml</u>	137
<u>Top level wiring</u>	137
<u>Wiring actions</u>	138
<u>Integrating the topics</u>	139
<u>Completing the plug-in manifest</u>	140

Table of Contents

Documentation integration	141
infoSet_Guide.xml	142
actions_All.xml	142
actions_View_Contents.xml	142
Externalizing Strings	142
Help server and zip files	142

Welcome to Eclipse

Welcome to the Eclipse platform!



The following sections discuss the issues and problems with building integrated tool suites, and how the Eclipse tooling platform can help solve these problems.

Who needs a platform?

On any given day, you can probably find an announcement about a strategic alliance, an open architecture, or a commercial API that promises to integrate all your tools, seamlessly move your data among applications, and simplify your programming life.

Down in the trenches, you're trying to apply enough import/export duct tape to let marketing say "suite" with a straight face.

Where is all this integration pressure coming from? Why is everyone trying to integrate their products into suites or build platforms to support open integration? Who needs these platforms?

End users

Let's face it. End users do not call the support line to say, "What I really need is an open tools platform."

But they do ask why your product doesn't integrate with their other tools. They ask for features outside of the scope of your application because they can't get their data to a tool that would do the job better. They run into problems importing and exporting between different programs. They wonder why their programs have completely different user interfaces for doing similar tasks. Doesn't it seem obvious that their web site design tool should be integrated with their scripting program?

Your users want the freedom to pick the best tool for the task. They don't want to be constrained because your software only integrates with a few other programs. They have a job to do, and it's not managing the flow of files and data between their tools. They're busy solving their own problems. It's your job to make the tools

work, and even better if you can make them work together.

Software developers

Meanwhile, you are slaving on your tool implementing the next round of critical features, fixing bugs, and shipping releases. The last thing you need is another emergency import feature added to your list.

Wouldn't it be nice if you could just publish enough hooks to make integrating with your tool everyone else's problem? Unfortunately, unless you work for one of the giants, you just don't have enough clout to get away with that.

The holy grail

What we all want is a level of integration that magically blends separately developed tools into a well designed suite. And it should be simple enough that existing tools can be moved to the platform without using a shoehorn or a crowbar.

The platform should be open, so that users can select tools from the best source and know that their supplier has a voice in the development of the underlying platform.

It should be simple to understand, yet robust enough to support integration without a lot of extra glue.

It should provide tools that help automate mundane tasks. It should be stable enough so that industrial strength tools can build on top of it. And it should be useful enough that the platform developers can use it to build itself.

These are all goals of Eclipse. The remainder of this programming guide will help you determine how close Eclipse has come to delivering on these ideals.

What is Eclipse?

Eclipse is a platform that has been designed from the ground up for building integrated web and application development tooling. By design, the platform does not provide a great deal of end user functionality by itself. The value of the platform is what it encourages: rapid development of integrated features based on a **plug-in** model.

Eclipse provides a common user interface (UI) model for working with tools. It is designed to run on multiple operating systems while providing robust integration with each underlying OS. Plug-ins can program to the Eclipse portable APIs and run unchanged on any of the supported operating systems.

At the core of Eclipse is an architecture for dynamic discovery of plug-ins. The platform handles the logistics of the base environment and provides a standard user navigation model. Each plug-in can then focus on doing a small number of tasks well. What kinds of tasks? Defining, testing, animating, publishing, compiling, debugging, diagramming...the only limit is your imagination.

Open architecture

The Eclipse platform defines an open architecture so that each plug-in development team can focus on their area of expertise. Let the repository experts build the back ends and the usability experts build the end user tools. If the platform is designed well, significant new features and levels of integration can be added without impact to other tools.

The Eclipse platform uses the model of a common workbench to integrate the tools from the end user's point of view. Tools that you develop can plug into the workbench using well defined hooks called extension points.

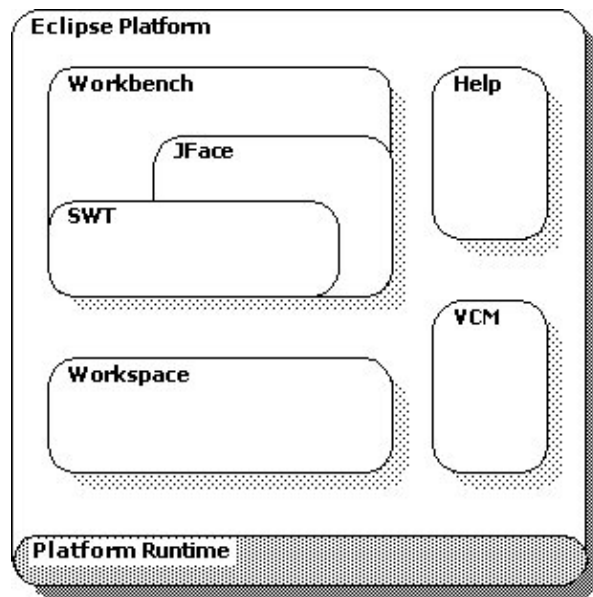
The underlying platform runtime uses the same extension model to allow plug-in developers to add support for additional file types and customized installations, such as web servers, workgroup servers, and repositories. The artifacts for each tool, such as files and other data, are coordinated by a common platform resource model.

The platform gives the users a common way to work with the tools, and provides integrated management of the resources they create with plug-ins.

Plug-in developers also gain from this architecture. The platform manages the complexity of different runtime environments, such as different operating systems or workgroup server environments. Plug-in developers can focus on their specific task instead of worrying about these integration issues.

Platform structure

The Eclipse platform itself is structured as subsystems which are implemented in one or more plug-ins. The subsystems are built on top of a small runtime engine.



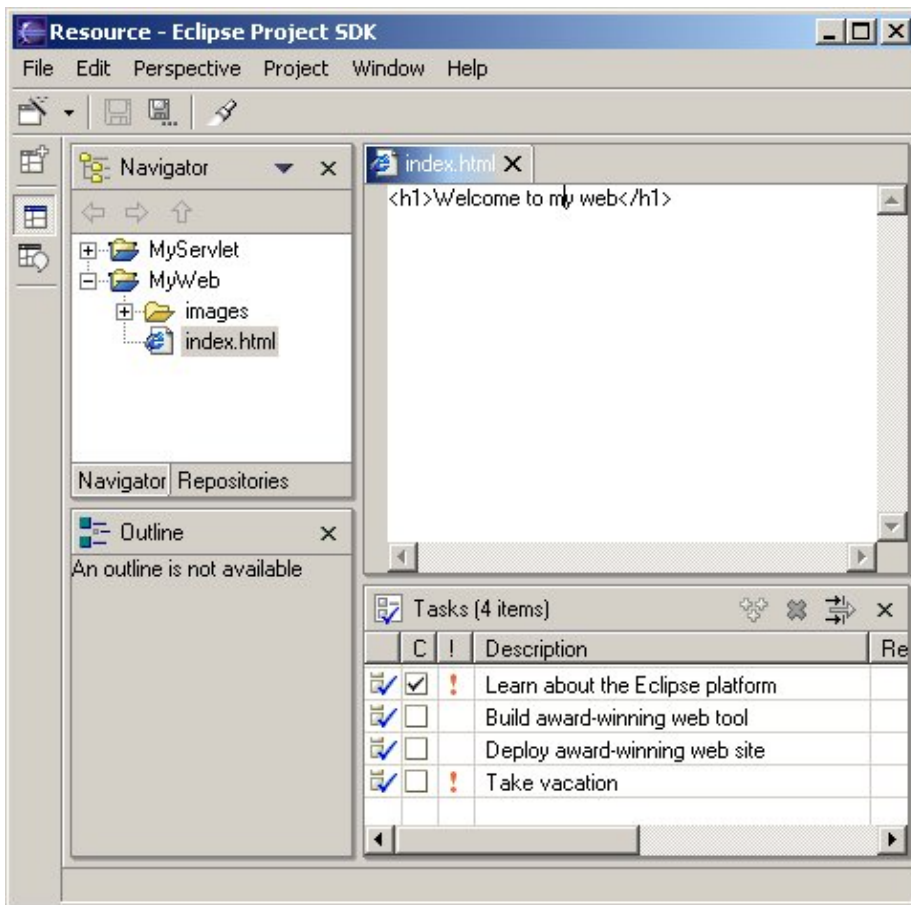
The subsystems define extension points for adding behavior to the platform. The following table describes the major runtime components of the platform that are implemented as one or more plug-ins.

Platform runtime	Defines the extension point and plug-in model. It dynamically discovers plug-ins and maintains information in a platform registry. Plug-ins are started up when required according to user operation of the platform.
Resource management (workspace)	Defines API for creating and managing resources (projects, files, and folders) that are produced by tools and kept in the file system.
	Implements the user cockpit for navigating the resources and using the tool plug-ins. It defines extension points for adding UI components such as views or menu actions. It includes additional toolkits (JFace and SWT) for building user interfaces.

Workbench UI	
Help system	Defines extension points for plug-ins to provide help or other documentation as browsable books.
Version and configuration management (VCM)	Defines a team programming model for managing and versioning resources.

Out of the box

Out of the box – or off the web – the basic platform is an integrated development environment (IDE) for anything (and nothing in particular).



It's the plug-ins that determine the ultimate functionality of the platform. That's why the Eclipse SDK ships with additional plug-ins to enhance the functionality of the SDK.

Your plug-ins can provide support for editing and manipulating additional types of resources such as Java files, C programs, Word documents, HTML pages, and JSP files.

Platform architecture

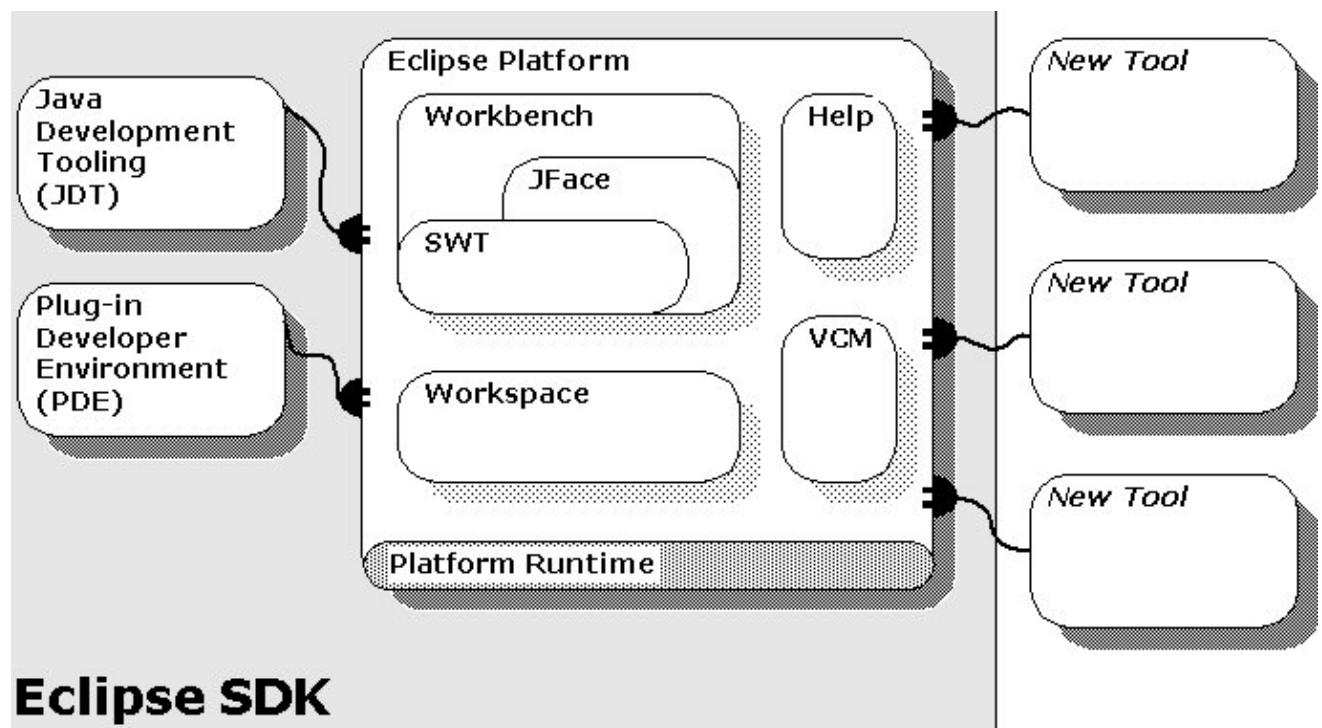
The Eclipse platform is structured around the concept of **extension points**. Extension points are well-defined places in the system where other tools (called **plug-ins**) can contribute functionality.

Each major subsystem in the platform is itself structured as a set of plug-ins that implement some key function and define extension points. The Eclipse system itself is built by contributing to the same extension points that third party plug-in providers can use. Plug-ins can define their own extension points or simply add **extensions** to the extension points of other plug-ins.

The platform subsystems typically add visible features to the platform and provide APIs for extending their functionality. Some of these components supply additional class libraries that do not directly relate to an extension point, but can be used to implement extensions. For example, the workbench UI supplies the JFace UI framework and the SWT widget toolkit.

The Eclipse SDK includes the basic platform plus two major tools that are useful for plug-in development. The Java development tooling (JDT) implements a full featured Java development environment. The Plug-in Developer Environment (PDE) adds specialized tools that streamline the development of plug-ins and extensions.

These tools not only serve a useful purpose, but provide a great example of how new tools can be added to the platform by building plug-ins that extend the system.



Platform SDK roadmap

Runtime core

The platform runtime core implements the runtime engine that starts the platform base and dynamically discovers plug-ins. A **plug-in** is a structured component that describes itself to the system using a manifest (**plugin.xml**) file. The platform maintains a registry of installed plug-ins and the function they provide.

Function is added to the system using a common extension model. **Extension points** are well-defined function points in the system that can be extended by plug-ins. When a plug-in contributes an implementation for an extension point, we say that it adds an **extension** to the platform. Plug-ins can define their own extension points, so that other plug-ins can integrate tightly with them.

The extension mechanisms are the only means of adding function to the platform and other plug-ins. All plug-ins use the same mechanisms. Plug-ins provided with the Eclipse SDK do not use any private mechanisms in their implementation.

Extensions are typically written in Java using the platform APIs. However, some extension points accommodate extensions provided as platform executables, ActiveX components, or developed in scripting languages. In general, only a subset of the full platform function is available to non-Java extensions.

A general goal of the runtime is that the end user should not pay a memory or performance penalty for plug-ins that are installed, but not used. A plug-in can be installed and added to the registry, but the plug-in will not be activated unless a function provided by the plug-in has been requested according to the user's activity.

The best way to get a feel for the runtime system is to build a plug-in. See [Plug it in: Hello World meets the workbench](#) to get started by building a plug-in.

Resource management

The resource management plug-in defines a common resource model for managing the artifacts of tool plug-ins. Plug-ins can create and modify **projects**, **folders**, and **files**, as well as define specialized types of resources.

[Introduction to resources](#) provides an overview of the resource management system.

Workbench UI

The workbench UI plug-in implements the workbench UI and defines a number of extension points that allow other plug-ins to contribute menu and toolbar actions, drag and drop operations, dialogs, wizards, and custom views and editors.

[Plugging into the workbench](#) introduces the workbench UI extension points and API.

The workbench UI plug-in also provides frameworks that are useful for user interface development. These frameworks were used to develop the workbench itself. Using the frameworks not only eases the development of a plug-in's user interface, but ensures that plug-ins have a common look and feel and consistent level of workbench integration.

The **Standard Widget Toolkit** (SWT) is a low-level, operating system independent toolkit that supports platform integration and portable API. It is described in [Standard Widget Toolkit](#).

The **JFace** UI framework provides higher-level application constructs for supporting dialogs, wizards, actions, user preferences, and widget management. The functionality in JFace is described in [Dialogs and wizards](#), [Preferences and properties](#), and [JFace: UI framework for plug-ins](#).

Help System

The Help plug-in implements a platform optimized help web server and document integration facility. It defines extension points that plug-ins can use to contribute help or other plug-in documentation as browsable books. The documentation web server includes special facilities to allow plug-ins to reference files by using logical, plug-in based URLs instead of file system URLs.

Additional features are provided for integrating help topics in product level documentation configurations.

The help facility is described in [Plugging in Help](#).

Version and Configuration Management (VCM)

The Version and Configuration Management (VCM) plug-ins define a versioning and team programming model for platform resources. The Eclipse SDK includes early versions of the VCM API and adapters that map the VCM model onto well known repository and team versioning tools. Since the API for VCM is still evolving, it is not discussed in this programmer's guide.

Java Development Tooling (JDT)

The Java development tooling (JDT) plug-ins extend the platform workbench by providing specialized features for editing, viewing, compiling, debugging, and running Java code.

The JDT is installed as a set of plug-ins that are included in the SDK. The JDT User Guide describes how to use the Java tools. The JDT Plug-in Developer Guide describes the structure and API of the JDT.

Plug-in Development Environment (PDE)

The Plug-in Development Environment (PDE) supplies tools that automate the creation, manipulation, debugging, and deploying of plug-ins.

The PDE is installed as a set of plug-ins that are included in the SDK. The PDE Guide describes how to use the environment.

Plug it in: Hello World meets the workbench

The Eclipse platform is structured as a core runtime engine and a set of additional features that are installed as platform **plug-ins**. Plug-ins contribute functionality to the platform by contributing to pre-defined **extension points**. The workbench UI is contributed by one such plug-in. When you start up the workbench, you are not starting up a single Java program. You are activating a platform runtime which can dynamically discover registered plug-ins and start them as needed.

When you want to provide code that extends the platform, you do this by defining system extensions in your plug-in. The platform has a well-defined set of extension points – places where you can hook into the platform and contribute system behavior. From the platform's perspective, your plug-in is no different from basic plug-ins like the resource management system or the workbench itself.

So how does your code become a plug-in?

- Decide how your plug-in will be integrated with the platform.
- Identify the extension points that you need to contribute in order to integrate your plug-in.
- Implement these extensions according to the specification for the extension points.
- Provide a manifest file (plugin.xml) that describes the extensions you are providing and the packaging of your code.

Creating a plug-in is best demonstrated by implementing an old classic, "Hello World," as a plug-in. We'll gloss over some details at first in order to get it running. Then we'll look at extension points in more detail, see where they are defined, and learn how plug-ins describe their implementation of an extension.

A minimal plug-in

We all know what "Hello World" looks like in plain old Java without using any user interface or other framework.

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

What happens to this old standard in the context of the Eclipse platform? Instead of thinking of Hello World as a self-contained program, we recast it as an extension of the platform. Since we want to say hello to the world, we need to figure out how to extend the workbench to include our greeting.

When we get deeper into the platform user interface components, we'll do an exhaustive review of the ways that you can extend and customize the workbench UI. For now, let's start with one of the simplest workbench extensions – a view.

You can think of the workbench window as a frame that presents various visual parts. These parts fall into two major categories: views and editors. We will look at editors later. **Views** provide information about some object that the user is working with in the workbench. Views often change their content as the user selects different objects in the workbench.

Hello world view

For our hello world plug-in, we will implement our own view to greet the user with "Hello World."

The package [org.eclipse.ui](#) and its sub packages contain the public interfaces that define the workbench user interface (UI) API. Many of these interfaces have default implementation classes that you can extend to provide simple modifications to the system. In our hello world example, we will extend a workbench view to provide a label that says hello.

The interface of interest is [IViewPart](#), which defines the methods that must be implemented to contribute a view to the workbench. The class [ViewPart](#) provides a default implementation of this interface. In a nutshell, a view part is responsible for creating the widgets needed to show the view.

The standard views in the workbench often display some information about an object that the user has selected or is navigating. Views update their contents based on actions that occur in the workbench. In our case, we are just saying hello, so our view is quite simple.

```
package org.eclipse.examples.helloworld;

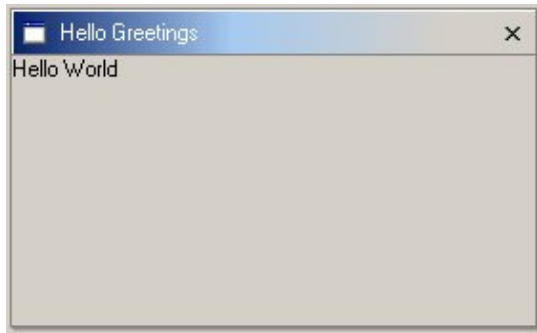
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.SWT;
import org.eclipse.ui.part.ViewPart;

public class HelloWorldView extends ViewPart {
    Label label;
    public HelloWorldView() {
    }
    public void createPartControl(Composite parent) {
        label = new Label(parent, SWT.WRAP);
        label.setText("Hello World");
    }
    public void setFocus() {
        // set focus to my widget. For a label, this doesn't
        // make much sense, but for more complex sets of widgets
        // you would decide which one gets the focus.
    }
}
```

The view part creates the widgets that will represent it in the **createPartControl** method. In this example, we create an SWT label and set the "Hello World" text into it.

We're done with the coding part! We can compile our new class (don't forget to make sure that the platform JAR files are visible in your IDE or compiler environment so that you can compile the plug-in), but we still have to figure out how to run our new view.

Our new view should look something like this:



How do we add this view to the platform?

The hello world plug-in

We need to inform the platform that we want to contribute a view. This is done by extending the `org.eclipse.ui.views` extension point. We register our extension by providing a manifest file, **plugin.xml**, which describes our plug-in, including where its code is located, and the extension we are adding.

```
<?xml version="1.0" ?>
<plugin
  name="Hello World Example"
  id="org.eclipse.examples.helloworld"
  version="1.0"
  provider-name="Object Technology International, Inc.">
  <requires>
    <import plugin="org.eclipse.ui" />
  </requires>
  <runtime>
    <library name="helloworld.jar" />
  </runtime>
  <extension point="org.eclipse.ui.views">
    <category
      id="org.eclipse.examples.helloworld.hello"
      name="Hello" />
    <view
      id="org.eclipse.examples.helloworld.helloworldview"
      name="Hello Greetings"
      category="org.eclipse.examples.helloworld.hello"
      class="org.eclipse.examples.helloworld.HelloWorldView" />
  </extension>
</plugin>
```

In this file, we define the **name**, **id**, **version**, and **provider-name** for our plug-in.

We also list our required plug-ins. Since we use workbench and SWT API in our plug-in, we must list **org.eclipse.ui**. We must also describe where our executable code is located. In our case, we will package the code in **helloworld.jar**.

Finally, we declare what extension point our plug-in is contributing. The [org.eclipse.ui.views](#) extension has several different configuration parameters. Let's review the parameters we have specified in our manifest.

We first declare a **category** for our view extension. Categories can be used to group related views together in the workbench **Show View** dialog. We define our own category, "Hello," so that it will display in its own group.

We declare a unique **id** for our view and specify the name of the **class** that provides the implementation of the view. We also specify a **name** for the view, "Hello Greetings" which will be shown in the Show View dialog and in our view's title bar.

Plug-in ids

There are many ids used in a plug-in manifest file. Individual extension points often define configuration parameters that require ids (such as the category id used above for the views extension point). We also define a plug-in id. In general, you should use Java package name prefixes for all of your ids in order to ensure uniqueness among all the installed plug-ins.

The specific name that you use after the prefix is completely up to you. However, if your plug-in id prefix is exactly the same name as one of your packages, you should avoid using class names from that package. Otherwise it will become difficult to tell whether you are looking at an id name or a class name.

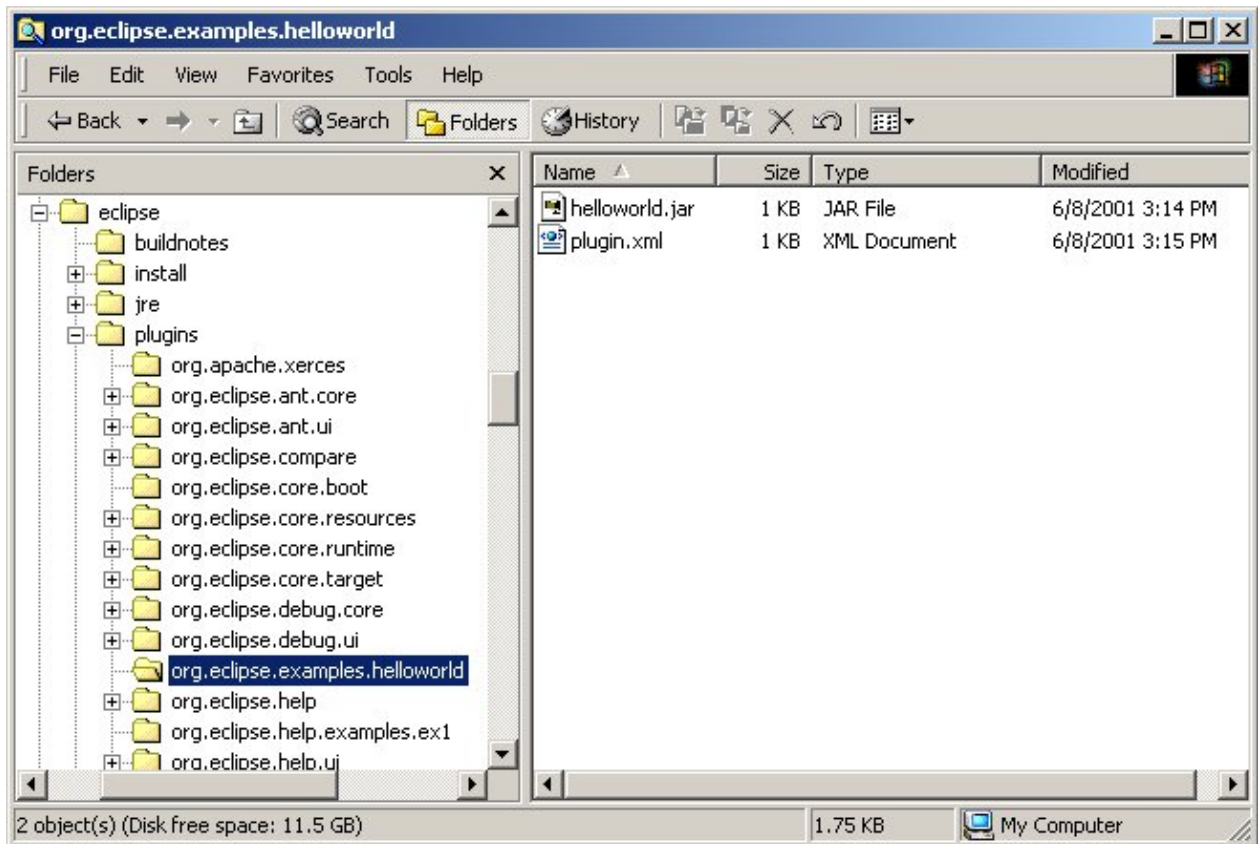
You should also avoid using the same id for different extension configuration parameters. In the above manifest, we have used a common id prefix (**org.eclipse.examples.helloworld**) but all of our ids are unique. This naming approach helps us to read the file and see which ids are related.

Installing and running the plug-in

Let's put all of this together so that we can run our new plug-in.

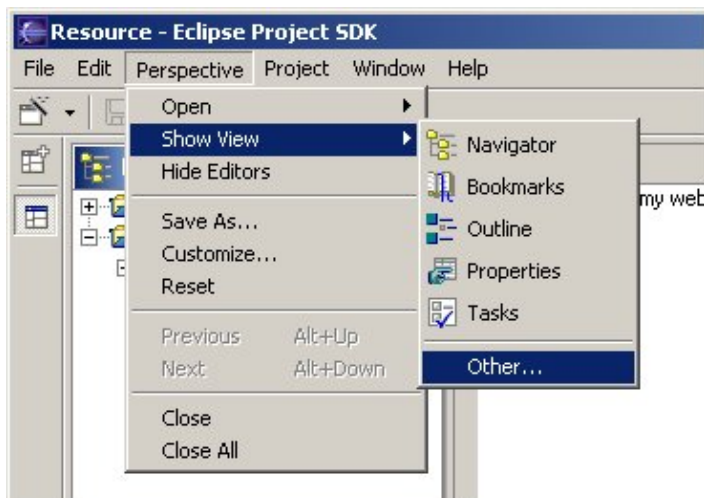
First, we need to compile our classes into a jar called **helloworld.jar**. Why? Because that's where we told the platform our plug-in could be found.

Next, we install the plug-in into a directory underneath the platform's plug-in directory. This directory corresponds to our plug-in id, which must be unique. The standard practice for plug-in directory names is to use the plug-in's fully qualified id, including the dots. In this case, we need to create a directory named **org.eclipse.examples.helloworld** inside the platform's plug-in directory. (The plug-in directory is named **plugins** and is typically located underneath the main directory where you installed the platform.) We copy the **helloworld.jar** and the **plugin.xml** to this new directory.



If you are currently running the workbench, you will need to shut it down and restart it. Why? When the platform starts, it assembles a list of all of the plug-ins installed in the system, called the plug-in registry. This registry keeps track of plug-ins and the extension points that they contribute. Restarting the workbench will cause it to find your new plug-in.

Now what? How do we run the plug-in? We can see all of the views that have been contributed by plug-ins using the **Perspective**→**Show View** menu.



This menu shows us what views are available for the current perspective. You can see all of the views that are contributed to the platform (regardless of perspective) by selecting **Other...**. This will display a list of view categories and the views available under each category.

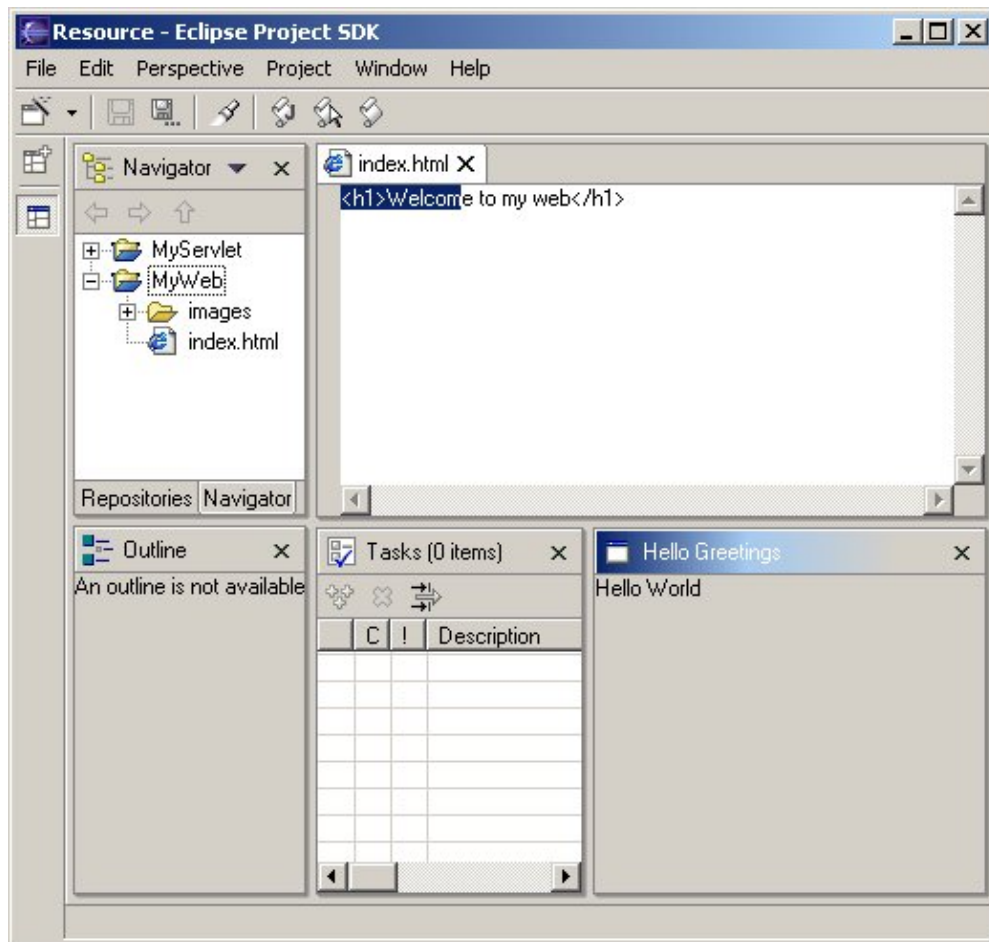
The workbench creates the full list of views by using the plug-in registry to find all the plug-ins that have provided extensions for the org.eclipse.ui.views extension point.



There we are! The view called "Hello Greetings" has been added to the **Show View** window underneath our category "Hello." The labels for our category and view were obtained from the extension point configuration markup in the **plugin.xml**.

Up to this point, we still have not run our plug-in code! The declarations in the **plugin.xml** (which can be accessed via the plug-in registry) are enough for the workbench to know that there is a view called "Hello View" available in the "Hello" category. It knows what class implements the view. But none of our code will be run until we decide to show the view.

If we choose the "Hello Greetings" view from the **Show View** list, the workbench will activate our plug-in, instantiate and initialize our view class, and show the new view in the workbench along with all of the other views. Now our code is running.



There it is, our first plug-in! We'll cover more specifics about UI classes and extension points later on.

Beyond the basics

Hopefully you've got a good idea of how you can contribute to an extension point, and package the functionality in a plug-in. We are ready to take a closer look at the resources plug-in and its API. Then we'll put everything together to implement a more complex workbench plug-in that works with a special kind of file.

A complete list of extension points can be found in the [Platform Extension Point Reference](#).

Resources overview

One of the essential plug-ins provided in the platform is the resources plug-in (named **org.eclipse.core.resources**). The resources plug-in provides services for accessing the files that a user is working with.

Resources and the workspace

The central hub for your user's data files is called a **workspace**. You can think of the platform workbench as a tool that allows the user to navigate and manipulate the workspace. The resources plug-in provides APIs for creating, navigating, and manipulating resources in a workspace. The workbench uses these APIs to provide this functionality to the user. Your plug-in can also use these APIs.

From the standpoint of a resource-based plug-in, there is exactly one workspace, and it is always open for business as long as the plug-in is running. The workspace gets opened automatically when the resources plug-in is activated, and closed when the platform is shut down. If your plug-in requires the resources plug-in, then the resources plug-in will be started before your plug-in, and the workspace will be available to you.

The workspace contains a collection of resources. From the user's perspective, there are three different types of resources: **projects**, **folders**, and **files**. A project is a collection of any number of files and folders. It is a container for organizing other resources that relate to a specific area. Files and folders are just like files and directories in the file system. A folder contains other folders or files. A file contains an arbitrary sequence of bytes. Its content is not interpreted by the platform.

A workspace's resources are organized into a tree structure, with projects at the top, and folders and files underneath. A special resource, the workspace root resource, serves as the root of the resource tree. The workspace root is created internally when a workspace is created and exists as long as the workspace exists.

A workspace can have any number of projects.

A sample resource tree

The tree below (represented in the workbench navigator view) illustrates a typical hierarchy of resources in a workspace. The (implied) root of the tree is the workspace root. The projects are immediate children of the workspace root. Each node (other than the root) is one of the 3 kinds of resource, and each has a name that is different from its siblings.



Resource names are arbitrary strings (almost — they must be legal file names). The platform itself does not dictate resource names, nor does it specify any names with special significance. (One exception is that you

cannot name a project **".metadata"** since this name is used internally.)

Projects contain files and folders, but not other projects. Projects and folders are like directories in a file system. When you delete a project, you will be asked whether you want to delete all of the files and folders that it contains. Deleting a folder from a project will delete the folder and all of its children. Deleting a file is analogous to deleting a file in the file system.

Resources and the local file system

When the platform core is running and the resources plug-in is active, the workspace is represented by an instance of [IWorkspace](#), which provides protocol for accessing the resources it contains. An [IWorkspace](#) instance represents an associated collection of files and directories in the local file system. You can access the workspace from the resources plug-in class in [org.eclipse.core.resources](#).

```
ResourcesPlugin.getWorkspace();
```

When the resources plug-in is not running, the workspace exists solely in the local file system and is viewed or manipulated by the user via standard file-based tools. Let's look at what a workspace looks like on disk as we explain the resources plug-in API.

Our sample tree on disk

When you installed the platform SDK, you unzipped the files into a directory of your choosing. We will call this directory the platform root directory. This is the directory that contains the **plugins** directory, among others. Inside the platform root directory, there is a **workspace** directory which is used to hold the resources that are created and manipulated by the platform. If you look in your **workspace** directory, you'll see separate subdirectories for each project that exists in the workspace. Within these subdirectories are the folders and files that each project contains.

If the SDK in our example is installed in **c:\MySDK**, then inside the **c:\MySDK\workspace** directory we find subdirectories named after the workspace's projects, **MyWeb** and **MyServlet**. These are called the projects' content directories. Content directories are created by the platform when the user creates a project.

Inside each directory, we find the files and folders within the project, laid out exactly the same as they are in the workspace's resource tree. All file names are the same, and the files' contents are the same whether accessed from the file system or from the workspace. No surprises.

```
C:\MySDK\workspace (workspace root)
  .metadata\ (platform metadata directory)
  MyWeb\ (project content directory for MyWeb)
    index.html
    images\
      logo.gif
  MyServlet\ (project content directory for MyServlet)
    src\
      main.java
    bin\
      main.class
      main$1.class
```

The platform has a special **.metadata** directory for holding platform internal information. The **.metadata** directory of a workspace is considered to be a "black box." Important information about the workspace

structure, such as a project's references or a resource's properties, is stored in the metadata portion of the workspace and should only be accessed by tools through the platform API. These files should never be edited or manipulated using generic file system API.

Apart from the **.metadata** directory, the folders and files in the workspace directory are fair game for other tools. The files and folders can be manipulated by non-integrated tools, such as text editors and file system utilities. The only issue is that the user must be careful when editing these files both in the workbench and externally. (This is no different than when a user edits a file using two independent stand-alone tools.) The workbench provides refresh operations to reconcile the workspace view of resources with the actual state in the file system.

Our sample tree in code

The resource API allows us to manipulate this resource tree in code. Here we will look at some code snippets for a quick taste of the resource API. The resource API is defined in a series of interfaces in [org.eclipse.core.resources](#). There are interfaces for all of the resource types, such as [IProject](#), [IFolder](#), and [IFile](#). Extensive common protocol is defined in [IResource](#). We also make use of the [org.eclipse.core.runtime](#) interface [IPath](#), which represents segmented paths such as resource or file system paths.

Manipulating resources is very similar to manipulating files using [java.io.File](#). The API is based on **handles**. When you use API like **getProject** or **getFolder**, you are returned a handle to the resource. There is no guarantee or requirement that the resource itself exists until you try to do something with the handle. If you expect a resource to exist, you can use **exists()** protocol to ensure this is the case.

To navigate the workspace from a plug-in, we must first obtain the [IWorkspaceRoot](#), which represents the top of the resource hierarchy in the workspace.

```
IWorkspaceRoot myWorkspaceRoot = ResourcesPlugin.getWorkspace().getRoot();
```

Once we have a workspace root, we can access the projects in the workspace.

```
IProject myWebProject = myWorkspaceRoot.getProject("MyWeb");  
// open if necessary  
if (myWebProject.exists() && !myWebProject.isOpen())  
    myWebProject.open(null);
```

Before we can manipulate a project, we must open it. Opening the project reads the project's structure from disk and creates the in-memory object representation of the project's resource tree. Opening a project is an explicit operation since each open project consumes memory to represent the resource tree internally and open projects participate in various resource lifecycle events (such as building) which can be lengthy. In general, closed projects cannot be accessed and will appear empty even though the resources are still present in the file system.

You'll notice that many of these resource examples pass a null parameter when manipulating resources. Many resource operations are potentially heavyweight enough to warrant progress reporting and user cancelation. If your code has a user interface, you will typically pass an [IProgressMonitor](#), which allows the resources plug-in to report progress as the resource is manipulated and allows the user to cancel the operation if desired. For now, we simply pass **null**, indicating no progress monitor.

Once we have an open project, we can access its folders and files, as well as create additional ones. In the following example we create a file resource from the contents of an external file located outside of our workspace.

```

IFolder imagesFolder = myWebProject.getFolder("images");
if (imagesFolder.exists()) {
    // create a new file
    IFile newLogo = imagesFolder.getFile("newLogo.gif");
    FileInputStream fileStream = new FileInputStream(
        "c:/MyOtherData/newLogo.gif");
    newLogo.create(fileStream, false, null);
    fileStream.close();
}

```

In the example above, the first line obtains a handle to the images folder. We must check that the folder **exists** before we can do anything interesting with it. Likewise, when we get the file **newLogo**, the handle does not represent a real file until we create the file in the last line. In this example, we create the file by populating it with the contents of **logo.gif**.

The next snippet is similar to the previous one, except that it copies the **newLogo** file from the original logo rather than create a new one from its contents.

```

IFile logo = imagesFolder.getFile("logo.gif");
if (logo.exists()) {
    IPath newLogoPath = new Path("newLogo.gif");
    logo.copy(newLogoPath, false, null);
    IFile newLogo = imagesFolder.getFile("newLogo.gif");
    ...
}

```

Finally, we'll create another images folder and move the newly created file to it. We rename the file as a side effect of moving it.

```

...
IFolder newImagesFolder = myWebProject.getFolder("newimages");
newImagesFolder.create(false, true, null);
IPath renamedPath = newImagesFolder.getFullPath().append("renamedLogo.gif");
newLogo.move(renamedPath, false, null);
IFile renamedLogo = newImagesFolder.getFile("renamedLogo.gif");

```

Many of the resource API methods include a **force** boolean flag which specifies whether resources that are out of sync with the corresponding files in the local file system will be updated. See [IResource](#) for more information.

Mapping resources to disk locations

In the sample resource tree, we've assumed that all of the project content directories are in the **workspace** directory underneath the platform root directory (**C:\MySDK\workspace**). This is the default configuration for projects. However, a project's content directory can be remapped to any arbitrary directory in the file system, perhaps on a different disk drive.

The ability to map the location of a project independent of other projects allows the user to store the contents of a project in a place that makes sense for the project and the project team. A project's content directory should be considered "out in the open." This means that users can create, modify, and delete resources by using the workbench and plug-ins, or by directly using file system based tools and editors.

Resource path names are not complete file system paths. Resource paths are always based on the project's location (usually the **workspace** directory). To obtain the full file system path to a resource, you must query its location using [IResource.getLocation\(\)](#).

You can determine the file system location of a particular resource by querying its location using `getLocation()`. However, you cannot use `IProjectDescription.setLocation` to change its location, because that method is just a simple setter for a data structure.

Resource API and the file system

When we use the resources API to modify our workspace's resource tree, the files are changed in the file system in addition to updating our resource objects. What about changes to resource files that happen outside of the platform's API?

External changes to resources will not be reflected in the workspace and resource objects until detected by the resources plug-in. Clients can use resource API to reconcile workspace and resource objects with the local file system quietly and without user intervention. The user can always explicitly force a refresh in the resource navigator view of the workbench.

Note: Many of the methods in the resource APIs include a force parameter which specifies how resources that are out of sync with the file system should be handled. The API Reference for each method provides specific information about this parameter. Additional methods in the API allow programmatic control of file system refresh, such as `IResource.refreshLocal(int depth, IProgressMonitor monitor)`. See [IResource](#) for information on correct usage and costs.

Resource properties

Resources can have properties that hold state information defined by your tool. Resource properties are declared, accessed, and maintained by various plug-ins, and are not interpreted by the platform. When a resource is deleted from the workspace, its properties are also deleted.

There are two kinds of resource properties:

- **Session properties** allow your plug-ins to cache information in key-value pairs. The values are arbitrary objects. These properties are maintained in memory and lost when a resource is deleted from the workspace, or when the project or workspace is closed.
- **Persistent properties** are used to store resource-specific information on disk. The value of a persistent property is an arbitrary string. Your plug-in decides how to interpret the string. The strings are intended to be short (under 2KB). Persistent properties are stored on disk with the platform metadata and maintained across platform shutdown and restart.

Note: If you have large persistent properties, you should expose these as resources in their own right rather than using the persistent properties API.

Note: If you follow the convention of qualifying property key names with the unique id of your plug-in, you won't have to worry about your property names colliding with those of other plug-ins.

[IResource](#) provides protocol (`getSessionProperty`, `setSessionProperty`, `getPersistentProperty`, `setPersistentProperty`) for using properties.

Beyond the basics

We've completed a quick survey of the resource API and explored how the resources and the workspace relate

to the file system.

The examples in **org.eclipse.examples.core.resources** demonstrate further use of the resource API.

It is best to look at resources in the context of a workbench plug-in. In the next few chapters, we will look at a sample plug-in, the readme tool, which manipulates resources and integrates with the workbench.

More information on resources can be found in [Resource and workspace API](#).

Plugging into the workbench

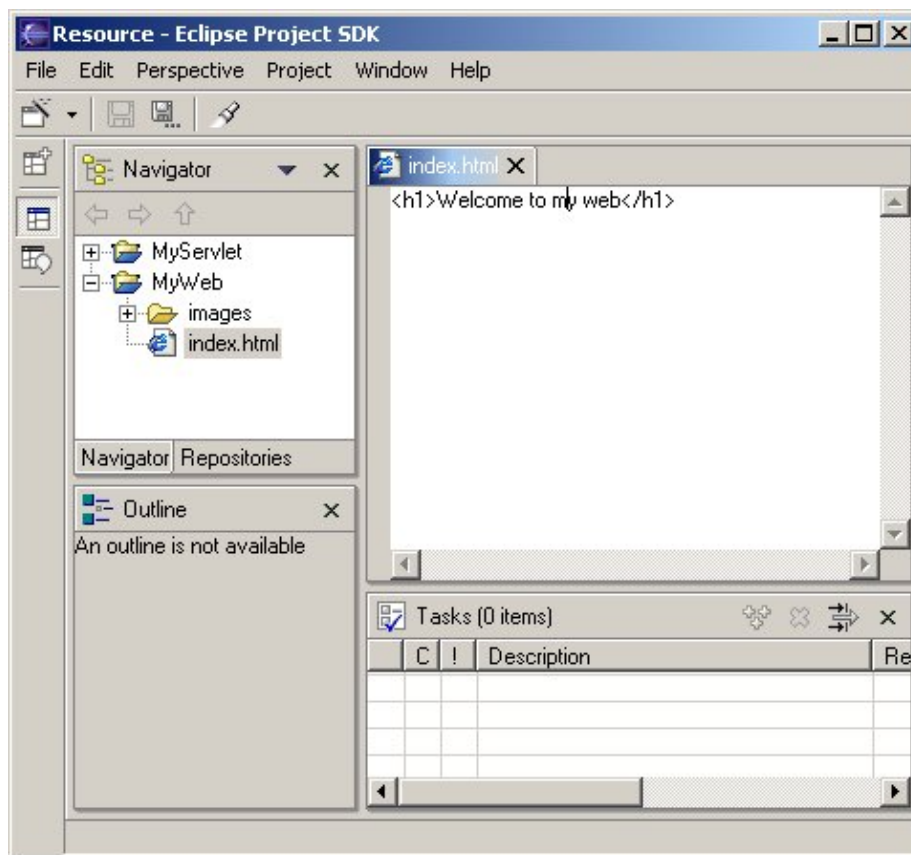
By now, you should be quite familiar with the operation of the workbench and how it uses views and editors to display information. If not, read the quick tour of the workbench below.

The sections following the quick tour will look at the user interface from an API perspective. We will show how a plug-in can contribute to the workbench UI.

Quick tour of the workbench

The workbench is the cockpit for navigating all of the function provided by plug-ins. Using the workbench, we can navigate projects, folders, and files. We can view and edit the content and properties of these resources.

When you open your workbench on a set of projects, it looks something like this.



The workbench is just a frame that can present various visual parts. These parts fall into two major categories: **views** and **editors**.

- **Editors** allow the user to edit something in the workbench. Editors are "document-centric," much like a file system based editor. They follow an open-save-close lifecycle much like file system based tools, but they are tightly integrated into the workbench.
- **Views** provide information about some object that the user is working with in the workbench. Views often change their content as the user selects different objects in the workbench. Views often support editors by providing information about the content in the active editor.

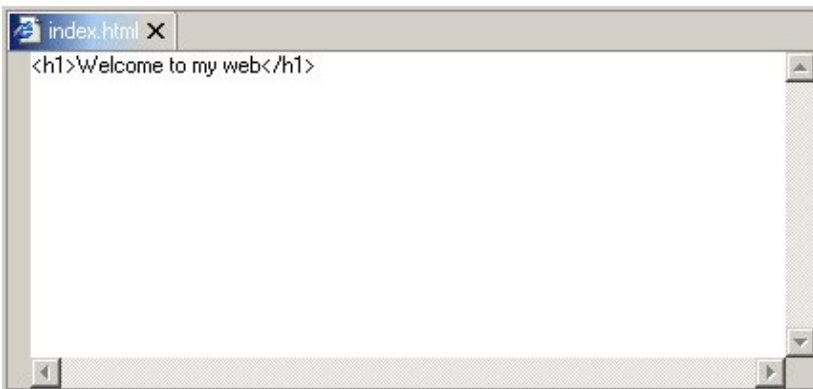
Views

The workbench provides several standard views that allow the user to navigate or view something of interest. For example, the resource navigator lets the user navigate the workspace and select resources.



Editors

Editors allow the user to open, edit, and save objects. The workbench provides a standard editor for text resources.



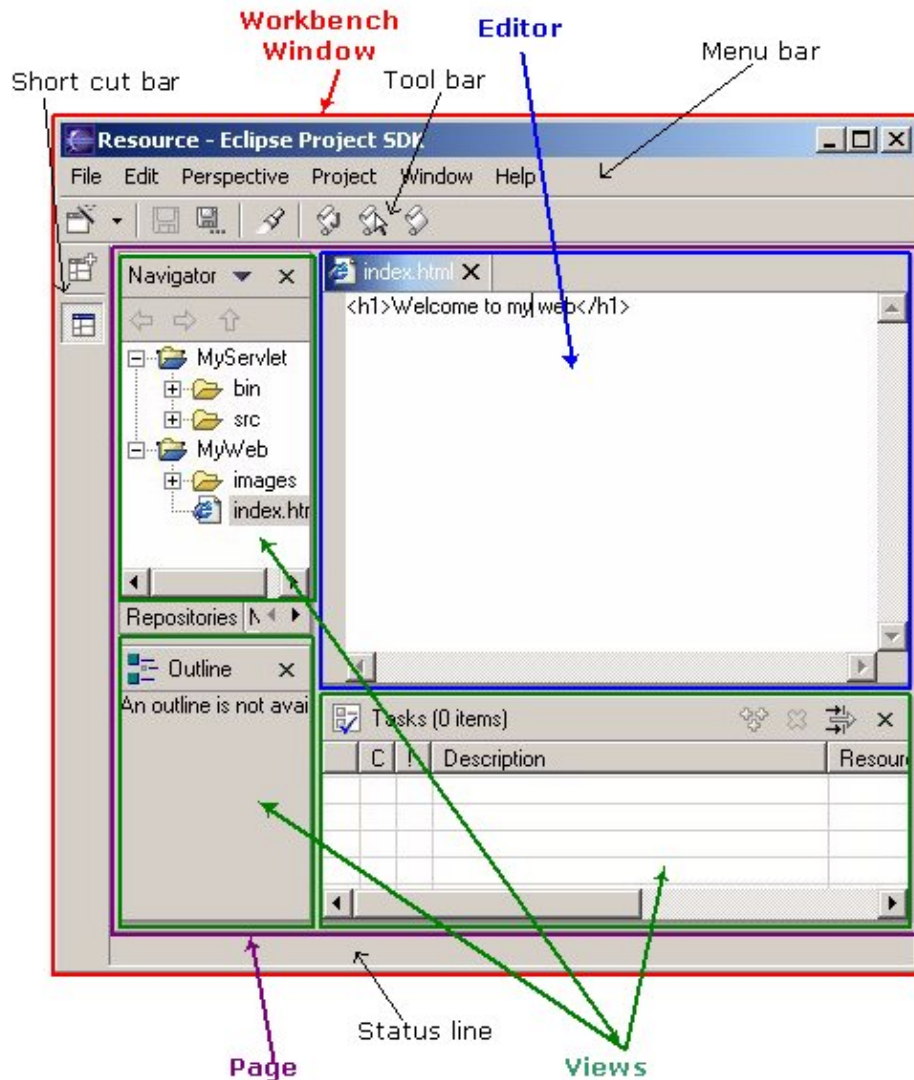
Additional editors, such as Java code editors or HTML editors, can be supplied by plug-ins

Workbench under the covers

An extensive set of classes and interfaces is available for building complex user interfaces. Fortunately you don't need to understand them all to do something simple. We'll start by looking at some concepts that are exposed in the workbench user interface and their corresponding structure under the covers.

Workbench

We've been using the term **workbench** loosely to refer to "that window that opens when you start the platform." Let's drill down a little and look at some of the visual components that make up the workbench.



For the rest of this discussion, when we use the term workbench, we will be referring to the workbench window ([IWorkbenchWindow](#)). The workbench window is the top-level window in a workbench. It is the frame that holds the menu bar, tool bar, status line, short cut bar, and pages. In general, you don't need to program to the workbench window. You just want to know that it's there.

*Note: You can open multiple workbench windows using **Perspective**→**Open**. By default, new perspectives are opened in the same workbench window. Other options are available (see the workbench preference page for details). Each workbench window is a self-contained world of editors and views, so we'll just focus on a single workbench window.*

From the user's point of view, a workbench contains views and editors. There are a few other classes used to implement the workbench window.

Page

Inside the workbench window, you'll find one or more pages ([IWorkbenchPage](#)) that in turn contain parts. Pages are an implementation mechanism for grouping parts. You typically don't need to program to the page, but you'll see it in the context of programming and debugging.

Note: Pages are used in the implementation of Perspectives. They allow workbench parts to be placed in an outer container without concern for whether the perspective is opened in the same workbench window or a new workbench window.

Views and editors

Views and editors are where we move beyond implementation details into some common plug-in programming. When you add a visual component to the workbench, you must decide whether you want to implement a view or an editor. How do you decide this?

- A **view** is typically used to navigate a hierarchy of information, open an editor, or display properties for the active editor. For example, the **navigator** view allows you to navigate the workspace hierarchy. The **properties** and **outline** views show information about an object in the active editor. Any modifications that can be made in a view (such as changing a property value) are saved immediately.
- An **editor** is typically used to edit or browse a document or input object. Modifications made in an editor follow an open–save–close model, much like an external file system editor. The platform text editor and Java editor are good examples of editors.

In either case, you will be building your view or editor according to a common lifecycle.

- You implement a **createPartControl** method to create the SWT widgets that represent your visual component. You must determine which widgets to use and allocate any related UI resources needed to display your view or editor.
- When your view or editor is given focus, you'll receive a **setFocus** notification so that you can set the focus to the correct widget.
- When the view or editor is closed, you will receive a **dispose** message to signify that the view or editor is being closed. At this point the controls allocated in **createPartControl** have already been disposed for you, but you must dispose of any graphics resources (such as cursors, icons, or fonts) that you allocated while creating the controls or responding to widget events.

Throughout this lifecycle, events will fire from the containing workbench page to notify interested parties about the opening, activation, deactivation, and closing of the views and editors.

Seem simple? It can be. That's the beauty of workbench views and editors. They're just widget holders, and can be as simple or complex as you need them to be. We saw the simplest of views earlier when we built a hello world view. Here it is again now that we've explained a bit better what's going on.

```
package org.eclipse.examples.helloworld;

import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.SWT;
import org.eclipse.ui.part.ViewPart;

public class HelloWorldView extends ViewPart {
    Label label;
    public HelloWorldView() {
    }
    public void createPartControl(Composite parent) {
        label = new Label(parent, SWT.WRAP);
        label.setText("Hello World");
    }
    public void setFocus() {
        // set focus to my widget. For a label, this doesn't
```

```
        // make much sense, but for more complex sets of widgets
        // you would decide which one gets the focus.
    }
}
```

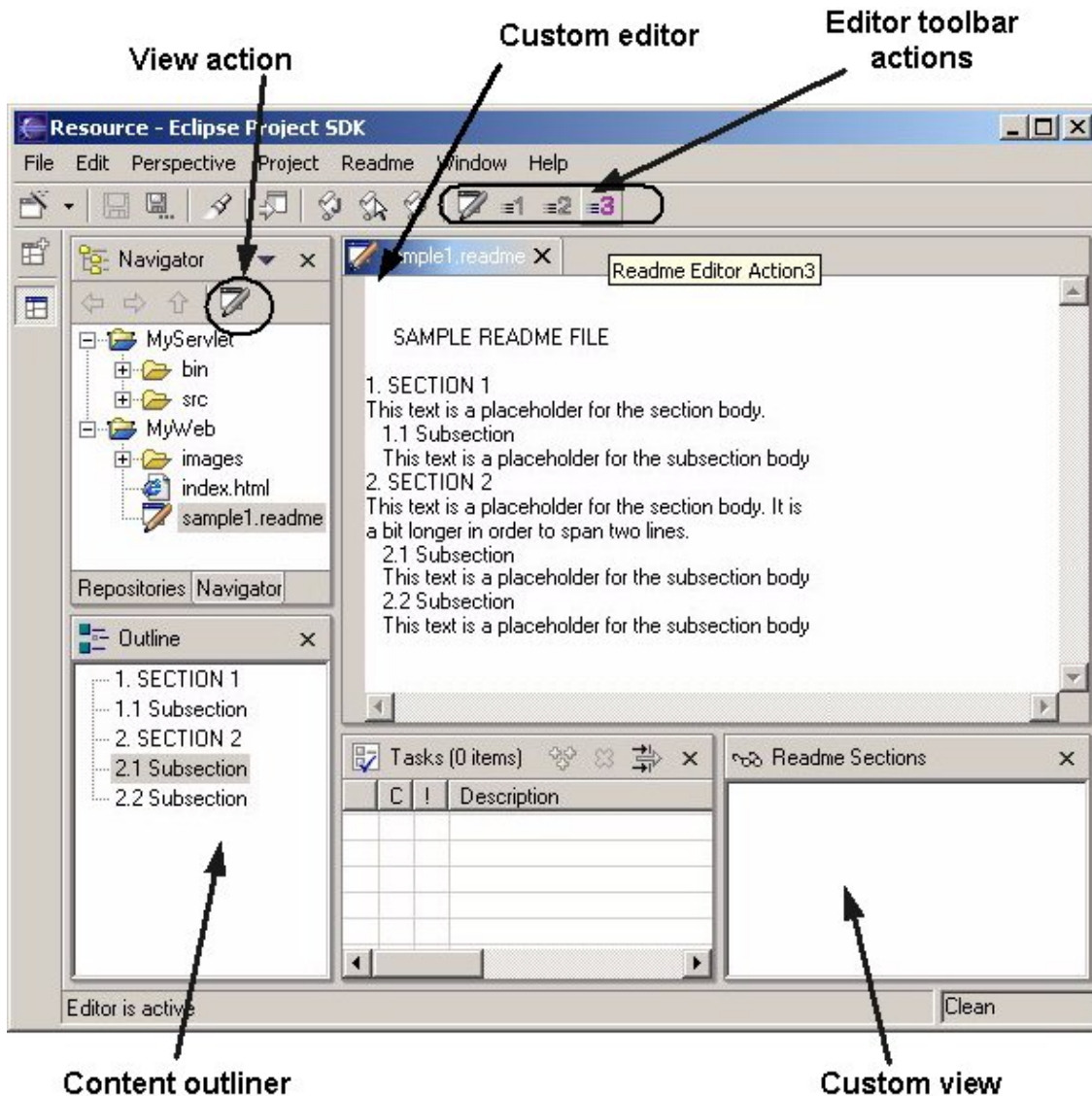
Notice that we didn't have to implement a **dispose()** method since we didn't do anything but create a label in the **createPartControl(parent)** method. If we had allocated any UI resources, such as images or fonts, we would have disposed of them here. Since we extended the [ViewPart](#) class, we inherit the "do nothing" implementation of **dispose()**.

Basic workbench extension points

The workbench defines extension points for plug-ins to provide implementations of views and editors as well as contribute behavior to existing views and editors. We are going to take a look at the contributions to these extension points from one of the workbench sample applications, the readme tool.

The readme tool is a plug-in that provides custom editing and navigation for a specific resource, a **.readme** file. The example shows many typical (but simplified) ways that extensions can be used to provide specialized tools.

The readme tool contributes to the menus of the navigator view, adds editor related actions to the workbench menus and tool bar, and defines a custom view and content outliner. The figure below shows some of the customized features added to the workbench by the readme tool.



The readme tool also contributes properties, preferences, and wizards to the workbench. We will look at these in [Dialogs and wizards](#) and [Preferences and properties](#). Here we will look at some of the basic contributions of the readme tool.

The readme tool is located in the `org.eclipse.ui.examples.readmetool` package. The `readmetool.jar` and `plugin.xml` can be found in the `org.eclipse.ui.examples.readmetool` directory underneath the `plugins` directory. To follow along, you will need to make sure that you have installed the platform examples. (See the [Examples Guide](#) for more information.)

The [readme tool](#) implements many different workbench extensions. We will start with one of the simplest workbench extension points, a view. We'll continue by looking at the readme tool extension points in the order in which you are likely to encounter them.

org.eclipse.ui.views

A view is a workbench part that can navigate a hierarchy of information or display properties for an object. Views are often provided to support a corresponding editor. For example, an **outline** view shows a structured view of the information in an editor. A **properties** view shows the properties of an object that is currently being edited.

When the user makes selections or otherwise makes changes in a view, the changes are reflected immediately in other related parts of the user interface. Only one instance of any given view is open in a workbench page.

The extension point [org.eclipse.ui.views](#) allows plug-ins to add views to the workbench. Plug-ins that contribute a view must register the view in their **plugin.xml** file, along with configuration information about the view, such as its implementation class, the category (or group) of views to which it belongs, and the name and icon that should be used to describe the view in menus and labels.

The interface for views is defined in [IViewPart](#), but plug-ins can choose to extend the [ViewPart](#) class rather than implement an [IViewPart](#) from scratch.

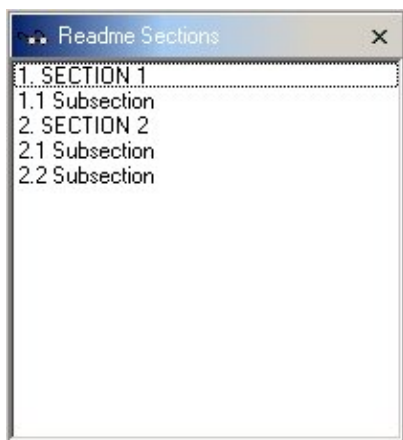
We implemented a minimal view extension in the hello world example. Now we'll look at one that is aware of other workbench views and responds to user navigation and selection in the workbench. First, let's take a look at the declaration of the extension in the **plugin.xml**.

```
<extension
  point="org.eclipse.ui.views">
  <category
    id="org.eclipse.ui.examples.readmetool"
    name="&Readme">
  </category>
  <view
    id="org.eclipse.ui.examples.readmetool.views.SectionsView"
    name="Readme Sections"
    icon="icons/basic/view16/sections.gif"
    category="org.eclipse.ui.examples.readmetool"
    class="org.eclipse.ui.examples.readmetool.ReadmeSectionsView">
  </view>
</extension>
```

This should look pretty familiar. We see that a new view, **ReadmeSectionsView**, is contributed to the workbench. The **view id**, **name**, and **category** are specified as we've seen before. An **icon** is also provided for the view, using a path relative to the plug-in's installation directory.

Let's look at the **ReadmeSectionsView**. You can show any view in the workbench by choosing **Perspective**→**Show View**→**Other...** and selecting the view from the **Show View** list.

When we show the **ReadmeSectionsView**, a view with a list in it pops up. The list is empty unless we click on a file with extension **.readme**, in which case it populates the list with sections from the readme file.



How does the plug-in know about selection changes and how did it recognize the readme file? If we can track

this down in the code, we are well on our way to understanding how to build integrated workbench plug-ins.

We'll start with the familiar **createPartControl** method. As we saw in the Hello World example, this is where the widgets that represent a view are created. We'll ignore some of the code to get started.

```
public void createPartControl(Composite parent) {
    viewer = new ListViewer(parent);
    ...
    // add myself as a global selection listener
    getSite().getPage().addSelectionListener(this);

    // prime the selection
    selectionChanged(null, getSite().getPage().getSelection());
}
```

The view creates and stores a `ListViewer` and registers itself as a selection listener on its page. It obtains the page from an [IViewSite](#), which contains information about the view's context, such as its workbench window, containing page, and plug-in. When we are notified of a selection change, what happens? The following code is executed:

```
public void selectionChanged(IWorkbenchPart part, ISelection sel) {
    //if the selection is a readme file, get its sections.
    AdaptableList input = ReadmeModelFactory.getInstance().getSections(sel);
    viewer.setInput(input);
}
```

It looks like the **ReadmeModelFactory** class is involved in turning the selection into readme sections, which are understood to be input for the viewer that we created in the **createPartControl** method.

We know that the view registered its interest in selection changes. Somehow the selection got converted into appropriate input for our viewer. And the viewer did some magic to populate a list widget.

If you must know right now what this viewer is all about, go to [Viewers](#). For now we'll just say that once the viewer was told its input element, it knew how to populate a list widget with the information. (It is a `ListViewer`, after all.) But how did we know when we had a readme file and where did the section information for the viewer come from? A quick look at the **ReadmeModelFactory** sheds some light.

```
public AdaptableList getSections(ISelection sel) {
    // If sel is not a structured selection just return.
    if (!(sel instanceof IStructuredSelection))
        return null;
    IStructuredSelection structured = (IStructuredSelection)sel;

    //if the selection is a readme file, get its sections.
    Object object = structured.getFirstElement();
    if (object instanceof IFile) {
        IFile file = (IFile) object;
        String extension = file.getFileExtension();
        if (extension != null && extension.equals(IReadmeConstants.EXTENSION)) {
            return getSections(file);
        }
    }

    //the selected object is not a readme file
    return null;
}
```

We check the selection to see if it is a structured (multiple) selection. (The concept of a structured selection comes from JFace [viewers](#).) For the first object in the selection, we check to see whether it is a file ([IFile](#)) resource. If it is, we check its extension to see if it matches the ".**readme**" extension. Once we know we have a readme file, we can use other methods to parse the sections. You can browse the rest of **ReadmeModelFactory**, **MarkElement**, and **DefaultSectionsParser** for the details about the file parsing.

We've covered a lot of common workbench concepts by studying this extension. Now we'll move on to some other workbench extensions and examine how your plug-in can contribute further to the workbench UI.

org.eclipse.ui.viewActions

It is common for plug-ins to contribute behavior to views that already exist in the workbench. This is done through the [org.eclipse.ui.viewActions](#) extension point. This extension point allows plug-ins to contribute menu items, submenus and tool bar entries to an existing view's local pull-down menu and local tool bar.

You may have noticed an item in the navigator view's local tool bar that becomes enabled whenever a readme file is selected. This item also appears in the view's local pull-down menu. These actions appear because the readme tool plug-in contributes them using the **viewActions** extension.



The relevant **plugin.xml** contribution is below.

```
<extension
  point = "org.eclipse.ui.viewActions">
  <viewContribution
    id="org.eclipse.ui.examples.readmetool.vcl"
    targetID="org.eclipse.ui.views.ResourceNavigator">
    <action
      id="org.eclipse.ui.examples.readmetool.val"
      label="&Readme View Extension"
      menubarPath="additions"
      toolbarPath="additions"
      icon="icons/basic/obj16/editor.gif"
      tooltip="Run Readme View Extension"
      helpContextId="org.eclipse.ui.examples.readmetool.view_action_context"
      class="org.eclipse.ui.examples.readmetool.ViewActionDelegate"
      enablesFor="1">
      <selection class="org.eclipse.core.resources.IFile" name="*.readme"/>
    </action>
  </viewContribution>
</extension>
```

A view contribution with a unique id is specified. The view to which we are adding the action is specified in the **targetID**. We are contributing to the resource navigator view's menu. We specify the label along with the menu bar and tool bar locations for the new action. (For a complete discussion of menu and toolbar locations, see [Menu and toolbar paths](#)).

We also specify the conditions under which the action should be enabled. You can see that this action will be enabled when there is one selection of type [IFile](#), whose name has ".readme" in the file extension. Sure enough, that's exactly what happens when you click around in the resource navigator.

The information in the **plugin.xml** is all that's needed to populate the appropriate menus and tool bars. No plug-in code will run until the action is actually selected from the menu or toolbar. The implementation class specified in the **plugin.xml** must implement the [IViewActionDelegate](#) interface.

In this example, the readme plug-in supplies **ViewActionDelegate** to implement the action. If you browse this class you will see that it includes methods for handling selection changes, invoking the action, and remembering what view it was created for.

The action itself simply launches a dialog that announces that the view action was executed.

```
public void run(org.eclipse.jface.action.IAction action) {
    MessageDialog.openInformation(view.getSite().getShell(),
        "Readme Editor",
        "View Action executed");
}
```

Although this action is simple, we can imagine how using selections and more functional dialogs could make this action do something more interesting.

org.eclipse.ui.editors

An editor is a workbench part that allows a user to edit an object (often a file). Editors operate in a similar manner to file system editing tools, except that they are tightly integrated into the platform workbench UI. An editor is always associated with an input object ([IEditorInput](#)). You can think of the input object as a document or file that is edited. Changes made in an editor are not committed until the user saves them.

Only one editor can be open for any particular editor input in a workbench page. For example, if the user is editing **readme.txt** in the workbench, opening it again in the same perspective will activate the same editor. (You can open another editor on the same file from a different workbench window or perspective). Unlike views, however, the same editor type (such as a text editor) may be open many times within one workbench page for different inputs.

The workbench extension point [org.eclipse.ui.editors](#) is used by plug-ins to add editors to the workbench. Plug-ins that contribute an editor must register the editor extension in their **plugin.xml** file, along with configuration information for the editor. Some of the editor information, such as the implementation **class** and the **name** and **icon** used in the workbench menus and labels, is similar to the view information. In addition, editor extensions specify the file extensions or file name patterns of the file types that the editor understands. Editors can also define a **contributorClass**, a class that adds actions to workbench menus and tool bars when the editor is active.

The interface for editors is defined in [IEditorPart](#), but plug-ins can choose to extend the [EditorPart](#) class rather than implement an [IEditorPart](#) from scratch.

Note: Editor extensions can also be configured to launch an external program to edit a file, or to provide a launcher class that is used to call pre-existing java code as the editor. In this discussion, we are focusing on those editors that are actually tightly integrated with the workbench and implemented using [IEditorPart](#).

The readme tool provides a custom editor primarily for the purpose of contributing its own content outliner page to the workbench outline view.

The configuration for the editor extension is defined as follows.

```
<extension
  point = "org.eclipse.ui.editors">
  <editor
    id = "org.eclipse.ui.examples.readmetool.ReadmeEditor"
    name="Readme File Editor"
    icon="icons/basic/obj16/editor.gif"
    class="org.eclipse.ui.examples.readmetool.ReadmeEditor"
    extensions="readme"
    contributorClass="org.eclipse.ui.examples.readmetool.ReadmeEditorActionBarContributor"
  </editor>
</extension>
```

We see the familiar configuration markup for **id**, **name**, **icon**, and **class**. There is also something new – a **contributorClass**.

Editor action contributors

The contributor class adds editor related actions to the workbench menu and toolbar. It must implement the [IEditorActionBarContributor](#) interface. The contributor is separated from the editor itself since any given workbench page can have multiple editors of the same type. Rather than have each instance of an editor type create actions and images, a single contributor is shared by all the editors of a specific type.

In **ReadmeEditorActionBarContributor**, we contribute three actions, "Editor Action1," "Editor Action2," and "Editor Action3." These are set up in the constructor.

```
public ReadmeEditorActionBarContributor() {
    ...
    action1 = new EditorAction("&Editor Action1");
    action1.setToolTipText("Readme Editor Action1");
    action1.setImageDescriptor(ReadmeImages.EDITOR_ACTION1_IMAGE);
    ...
    action2 = new EditorAction("&Editor Action2");
    action2.setToolTipText("Readme Editor Action2");
    action2.setImageDescriptor(ReadmeImages.EDITOR_ACTION2_IMAGE);
    ...
    action3 = new EditorAction("&Editor Action3");
    action3.setToolTipText("Readme Editor Action3");
    action3.setImageDescriptor(ReadmeImages.EDITOR_ACTION3_IMAGE);
    ...
}
```

The names and icons for the actions are set up in the code rather than in the **plugin.xml**. Note how similar this information is to the **viewActions** information we saw in the markup for the view action. The actions are set up in code since we have to manage the sharing of the actions among different instances of the same editor.

As they are created in the constructor, the actions are independent of any particular instance of the editor. When an editor becomes active and the actions need to be installed in the workbench menus and tool bar, the **setActiveEditor** message is sent to the contributor. The contributor connects the editor actions to a specific editor.

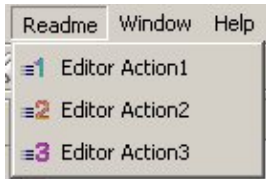
```
public void setActiveEditor(IEditorPart editor) {
```

```

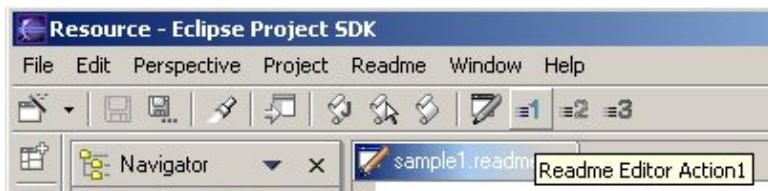
...
action1.setActiveEditor(editor);
action2.setActiveEditor(editor);
action3.setActiveEditor(editor);
...
}

```

As you can see, the actions show up in the workbench menu and tool bar when a readme editor is active.



Readme editor contributions to workbench menu bar

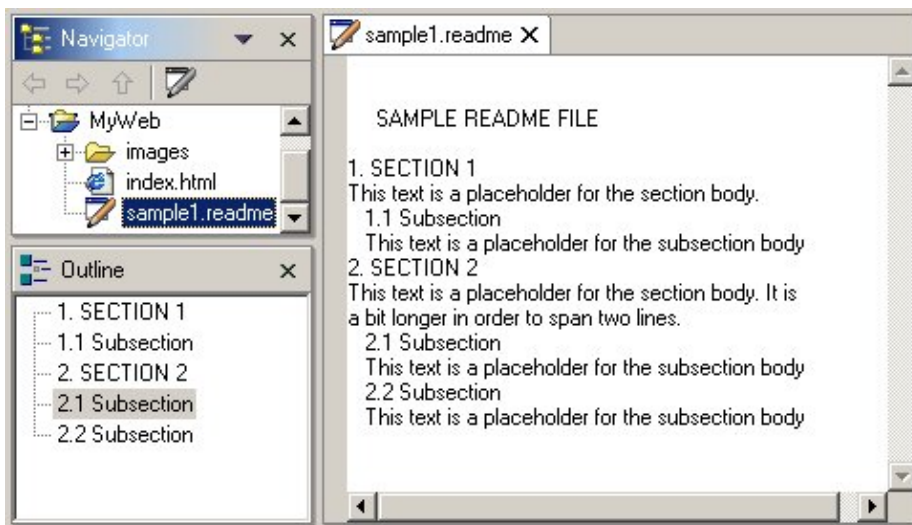


Readme editor contributions to workbench tool bar

These menu and tool bar items are only shown when the editor is active. The location for the menu and tool bar items can be specified as described in [Menu and toolbar paths](#).

Editors and content outliners

The readme editor itself, **ReadmeEditor**, is not very complicated. It extends the [TextEditor](#) class so that it can contribute a customized content outliner page to the outline view when a readme file is being edited. It does not change any behavior inside the text editor.



Editors often have corresponding content outliners that provide a structured view of the editor's contents and assist the user in navigating through the contents of the editor. See [Content outliners](#) for more detail.

We'll look at the implementation of text editors in [Text editors and JFace text](#).

org.eclipse.ui.editorActions

We've just seen how editors can contribute their own actions to the workbench menus and tool bar when they are active. The [org.eclipse.ui.editorActions](#) extension point allows a plug-in to add to the workbench menus and tool bar when another plug-in's editor becomes active.

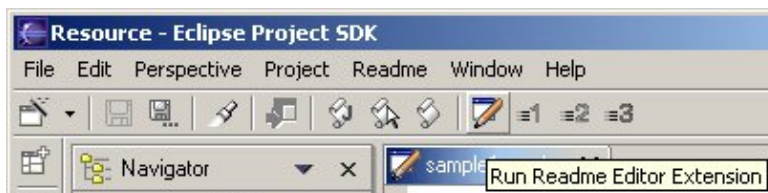
In the readme example, the plug-in uses the **editorActions** extension point to contribute additional actions to the menu contributed by the readme editor. The definition in our **plugin.xml** should look pretty familiar by now.

```
<extension
  point = "org.eclipse.ui.editorActions">
  <editorContribution
    id="org.eclipse.ui.examples.readmetool.ec1"
    targetID="org.eclipse.ui.examples.readmetool.ReadmeEditor">
    <action
      id="org.eclipse.ui.examples.readmetool.ea1"
      label="&Readme Editor Extension"
      toolbarPath="ReadmeEditor"
      icon="icons/basic/obj16/editor.gif"
      tooltip="Run Readme Editor Extension"
      class="org.eclipse.ui.examples.readmetool.EditorActionDelegate"
    />
  </editorContribution>
</extension>
```

Similar to a view action, the extension must specify the **targetID** of the editor to which it is contributing actions. The action itself is very similar to a view action (**id**, **label**, and **toolbarPath**), except that the specified class must implement [IEditorActionDelegate](#).

Note that a menu bar path is not specified in this markup. This means that the action will appear in the workbench tool bar when the editor is active, but not in the workbench menu bar. (See [Menu and toolbar paths](#) for a discussion of toolbar and menu paths.)

Sure enough, when the editor is active, we see our editor action on the tool bar next to the actions that were contributed by the editor itself.



The readme tool supplies **EditorActionDelegate** to implement the action. This is very similar to the view action delegate we saw earlier.

```
public void run(IAction action) {
    MessageDialog.openInformation(editor.getSite().getShell(),
        "Readme Editor",
        "Editor Action executed");
}
```

org.eclipse.ui.popupMenus

The [org.eclipse.ui.popupMenus](#) extension point allows a plug-in to contribute to the popup menus of other views and editors.

You can contribute an action to a specific popup menu by its id (**viewerContribution**), or you can contribute an action for an object type (**objectContribution**).

- A **viewerContribution** will cause the menu item to appear in the popup menu of a view or editor specified by id in the markup.
- An **objectContribution** will cause the menu item to appear in all popup menus for views or editors that have objects of the specified type selected.

The readme tool defines an **objectContribution**. The markup looks like this.

```
<extension point = "org.eclipse.ui.popupMenus">
  <objectContribution
    id="org.eclipse.ui.examples.readmetool"
    objectClass="org.eclipse.core.resources.IFile"
    nameFilter="*.readme">
    <action id="org.eclipse.ui.examples.readmetool.action1"
      label="Show Readme Action"
      icon="icons/basic/ctool16/openbrwsr.gif"
      menuBarPath="additions"
      helpContextId="org.eclipse.ui.examples.readmetool.open_browser_action_context"
      class="org.eclipse.ui.examples.readmetool.PopupMenuActionDelegate"
      enablesFor="1">
    </action>
  </objectContribution>
</extension>
```

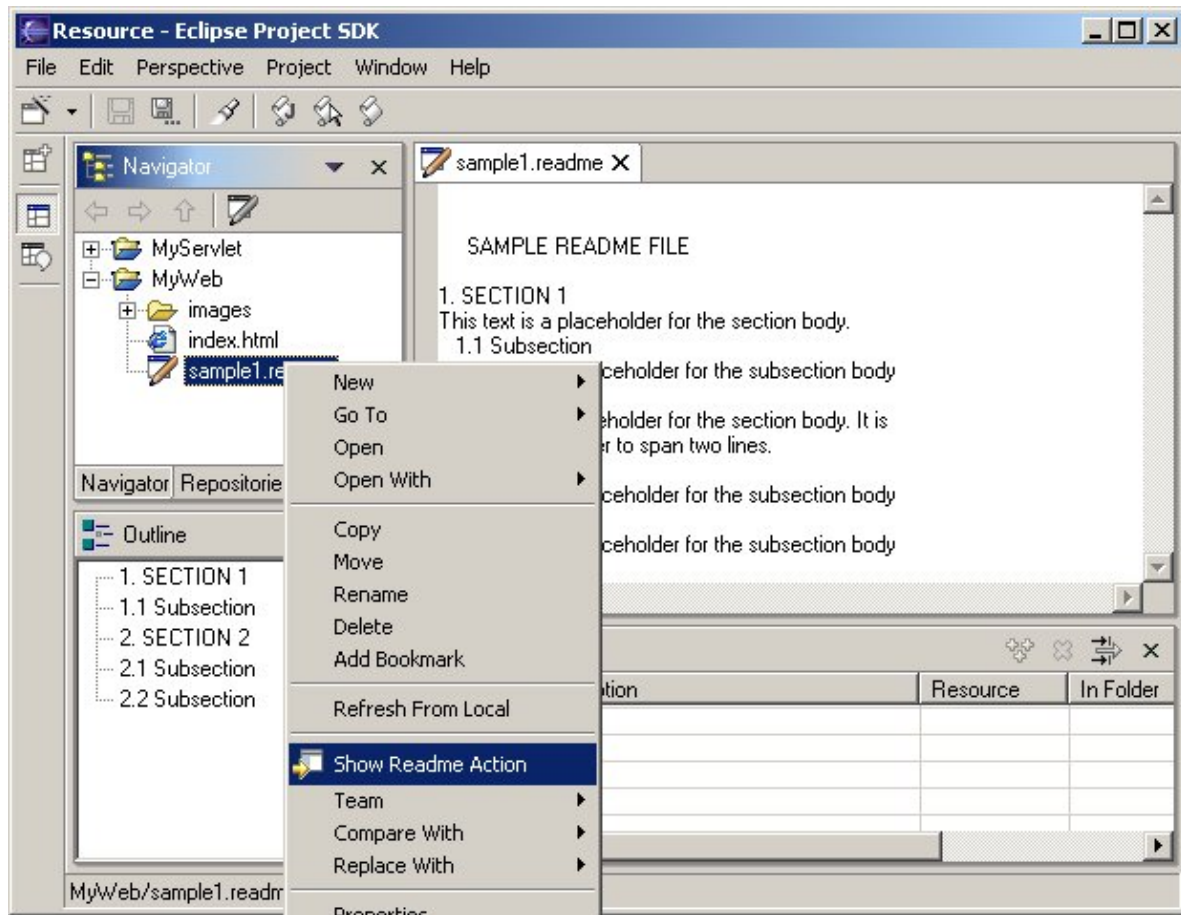
The action "Show Readme Action" is contributed for the object class [IFile](#). This means that any view containing [IFile](#) objects will show the contribution if [IFile](#) objects are selected. We see that the selection criteria is restricted further with a name filter (**nameFilter="*.readme"**) and for single selections (**enablesFor="1"**). As we've discussed before, the registration of this menu does not run any code from our plug-in until the menu item is actually selected.

When the menu item is selected, the workbench will run the specified class. Since the popup is declared as an **objectContribution**, the supplied class must implement [IObjectActionDelegate](#).

The action is implemented in **PopupMenuActionDelegate**.

```
public void run(IAction action) {
    MessageDialog.openInformation(
        this.part.getSite().getShell(),
        "Readme Example",
        "Popup Menu Action executed");
}
```

We can see the popup menu contribution when we select a readme file from the resource navigator.



The other type of pop-up menu contribution is called a **viewerContribution**.

*Note: The name **viewerContribution** is somewhat misleading, as it does not relate to JFace viewers. A better name would be **popupMenuContribution**.*

A viewer contribution is used to contribute to a specific view or editor's popup menu using its id. The following markup shows how a plug-in could register a specific action in the workbench task list popup menu.

```
<extension point="org.eclipse.ui.popupMenus">
  <viewerContribution
    id="com.example.C2"
    targetID="org.eclipse.ui.views.TaskList">
    <action
      id="com.example.showExample"
      label="& Show Example"
      menubarPath="additions"
      icon="icons/showExample.gif"
      helpContextId="com.example.show_action_context"
      class="com.example.actions.ExampleShowActionDelegate">
    </action>
  </viewerContribution>
</extension>
```

If the extension is a **viewerContribution**, the supplied class must implement the [IEditorActionDelegate](#) or [IViewActionDelegate](#) interface, depending on whether the action is contributed to an editor's or view's popup menu.

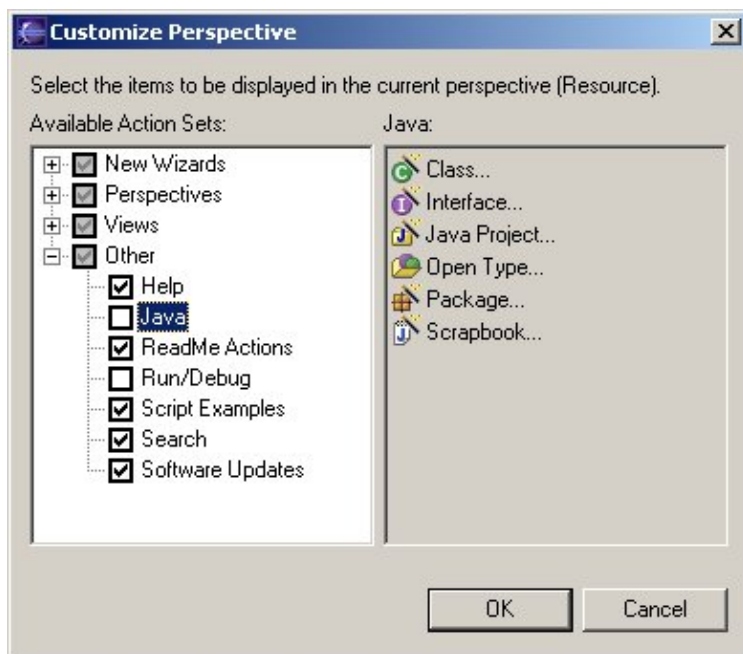
In this example, we specify the **id** of the popup and the **path** within the popup for our contribution.

*Note: The interfaces required for a **viewerContribution** to the **popupMenus** extension point are the same as those required by the **viewActions** and **editorActions** extension points. If you want to contribute the same action to the popup menu and the local menu of a view or editor, you can use the same class for both extensions.*

org.eclipse.ui.actionSets

Your plug-in can contribute menus, menu items, and tool bar items to the workbench menus and toolbar using the [org.eclipse.ui.actionSets](#) extension point. In order to reduce the clutter of having every plug-in's menu contributions shown at once, the contributions are grouped into action sets which can be made visible by user preference.

You can see which action sets have been contributed to your workbench by choosing **Perspective**→**Customize...** from the workbench menu. This will show you a dialog that lists all of the available action sets. A checkmark by the action set means that the menu and tool bar actions are visible in the workbench. You can select the name of the action set to see the list of available actions on the right. The figure below shows the list of action sets available in our workbench. (Your workbench may look different depending on which plug-ins you have installed and which perspective is active.)



The readme tool uses an action set to contribute the "Open Readme Browser" action to the workbench menu. (We contributed a similar action to the popup menu of the resource navigator.) The markup follows:

```
<extension
  point = "org.eclipse.ui.actionSets">
  <actionSet
    id="org_eclipse_ui_examples_readmetool_actionSet"
    label="ReadMe Actions"
    visible="true">
    <menu
      id="org_eclipse_ui_examples_readmetool"
      label="Readme &File Editor"
      path="window/additions">
      <separator name="slot1"/>
```

```

        <separator name="slot2"/>
        <separator name="slot3"/>
    </menu>
    <action
        id="org_eclipse_ui_examples_readmetool_readmeAction"
        menubarPath="window/org_eclipse_ui_examples_readmetool/slot1"
        toolbarPath="readme"
        label="&Open Readme Browser@Ctrl+R"
        tooltip="Open Readme Browser"
        helpContextId="org.eclipse.ui.examples.readmetool.open_browser_action_context"
        icon="icons/basic/ctool16/openbrwsr.gif"
        class="org.eclipse.ui.examples.readmetool.WindowActionDelegate"
        enablesFor="1">
        <selection
            class="org.eclipse.core.resources.IFile"
            name="*.readme">
        </selection>
    </action>
</actionSet>
</extension>

```

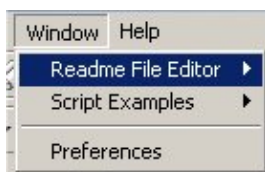
Wow, there's a lot going on here! Let's take it a step at a time.

First, the action set is declared and given a **label**. The label "ReadMe Actions" is used to display the action set in the dialog shown above. Since we set **visible** to true, the workbench will initially have the action set check marked in the action set list and the actions will be visible.

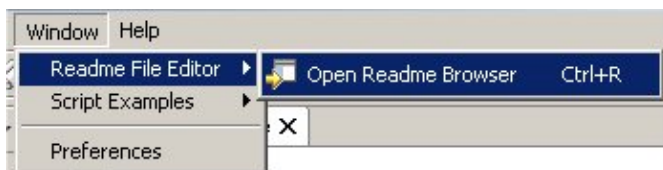
The rest of the action set declaration is concerned with defining the menu in which the action appears and the action itself.

We define a menu whose **label** ("Readme &File Editor") appears in the workbench menus. The menu's **path** tells the workbench to place the new menu in the **additions** slot of the **window** menu. (For a discussion of menu paths and slots, see [Menu and toolbar paths.](#)) We define some slots in our new menu that can be used to insert actions in specific locations in our menu.

This alone is enough to cause the menu to appear in the workbench **Window** menu.



Next, we define the action itself. The action definition (**id**, **label**, **icon**, **class**) is similar to the other actions we've seen in views, editors, and popups. We'll focus here on what's different: where does the action go? We use **menubarPath** and **toolbarPath** to indicate the location. First, we add the action to a slot in the menu that we just defined.



Then, we define a new toolbarPath to insert our action in the workbench tool bar. Since it's a new tool path, the workbench will decide where it goes relative to other plug-in's toolbar contributions.



Note that enabling conditions are also supplied for this action. Our new menu item and tool bar item will only be enabled when a single (`enablesFor="1"`) readme file (`selectionClass="org.eclipse.core.resources.IFile" name="*.readme"`) is selected.

These menu and tool bar items appear and enable based solely on the markup in the `plugin.xml` file. None of the plug-in code will execute until the user chooses the action and the workbench runs the action `class`.

The action `class` must implement [IWorkbenchWindowActionDelegate](#), or [IWorkbenchWindowPulldownDelegate](#) if the action set is shown as a pull-down in a tool bar. Since we are not creating a tool bar pull-down, we provide `WindowActionDelegate`. This class is similar to `ObjectActionDelegate`. It launches the readme sections dialog when the user chooses the action. We'll see the sections dialog in [Application dialogs](#).

The plug-in class

So far, we've been looking at the different extensions that are provided by the readme tool. Let's look at the general definition of the readme tool plug-in.

Plug-in definition

The readme tool plug-in is defined at the top of the `plugin.xml` file.

```
<plugin
  name = "Readme File Editing Tool"
  id = "org.eclipse.ui.examples.readmetool"
  version = "0.9"
  provider-name = "Object Technology International, Inc."
  class="org.eclipse.ui.examples.readmetool.ReadmePlugin">

  <requires>
    <import plugin="org.eclipse.ui"/>
    <import plugin="org.eclipse.core.resources"/>
  </requires>

  <runtime>
    <library name="readmetool.jar"/>
  </runtime>
</plugin>
```

The plug-in definition includes the name, id, version, and vendor name of the plug-in. We saw these parameters before in our hello world plug-in. The readme tool also defines a specialized plug-in class, `ReadmePlugin`.

The workbench UI and resources plug-ins are listed as required plug-ins so that the platform will know that the readme tool is dependent on them.

Finally, the name of the jar file is provided. File names specified in a `plugin.xml` file are relative to the plug-in's directory.

AbstractUIPlugin

The **ReadmePlugin** class represents the readme tool plug-in and manages the life cycle of the plug-in. As we saw in the Hello World example, you don't have to specify a plug-in class. The platform will provide one for you. In this case, our plug-in needs to initialize UI related data when it starts up. The platform class [AbstractUIPlugin](#) provides a structure for managing UI resources and is extended by **ReadmePlugin**.

[AbstractUIPlugin](#) uses the generic startup and shutdown methods to manage images, dialog settings, and a preference store during the lifetime of the plug-in. We'll look at the specifics of the **ReadmePlugin** class when we work with dialogs and preferences.

Workbench menu contributions

We've seen several different extension points that contribute to various menus and toolbars in the workbench. How do you know which one to use? The following table summarizes the various menu contributions and their use.

Extension point name	Location of Actions	Details
viewActions	Actions appear in a specific view's local toolbar and local pulldown menu.	Contribute an action class that implements IViewActionDelegate . Specify the id of the contribution and the id of the target view that should show the action. The label and image dictate the appearance of the action in the UI. The path specifies the location relative to the view's menu and toolbar items.
editorActions	Actions are associated with an editor and appear in the workbench menu and/or tool bar.	Contribute an action class that implements IEditorActionDelegate . Specify the id of the contribution and the id of the target editor that causes the action to be shown. The label and image specify the appearance of the action in the UI. Separate menu and toolbar paths specify the existence and location of the contribution in the workbench menu and toolbar.
popupMenu	Actions appear in the popup menu of an editor or view. Actions associated with an object type show up in all popups of views and editors that show the object type. Actions associated with a specific popup menu appear only in that popup menu.	Object contributions specify the type of object for which the action should appear in a popup menu. The action will be shown in all view and editor popups that contain the object type. Provide an action class that implements IObjectActionDelegate . Viewer contributions specify the id of the target popup menu in which the menu item should appear. Provide an action class that implements IEditorActionDelegate or IViewActionDelegate .
actionSets	Actions appear in the workbench main menus	Contribute an action class that implements

and toolbar. Actions are grouped into action sets. All actions in an action set will show up in the workbench menus and toolbars according to the user's selection of action sets and the current perspective shown in the workbench.

[IWorkbenchWindowActionDelegate](#) or [IWorkbenchWindowPulldownDelegate](#). Specify the **name** and **id** of the action set. Enumerate all of the actions that are defined for that action set. For each action, separate menu and toolbar paths specify the existence and location of the contribution in the workbench menu and toolbar.

Menu and toolbar paths

We've seen many action contributions that specify the path for the location of their action. Let's take a close look at what these paths mean. We'll look at paths by looking at the workbench **Help** menu.

Named groups

The locations for inserting new menus, menu items, or tool bar items are defined using named groups. You can think of a named group as a slot or placeholder that allows you to insert your menu and toolbar contributions at certain points in a menu or toolbar.

The workbench defines all of its group slot names in the class [IWorkbenchActionConstants](#). For each workbench menu, named groups are placed in the menu at locations where it is expected that plug-ins will insert new actions.

The following description of the help menu is adapted from the [IWorkbenchActionConstants](#) class definition.

```
Standard Help menu actions
Start group HELP_START "start"
End group HELP_END "end"
About action ABOUT "About"
```

The standard workbench help menu consists of a named group called "**start**," followed by a named group called "**end**," followed by the "**About**" action. Why two groups? To provide some control for plug-ins for how far down they appear in the help menu. When you define a menu, you can define as many slots as you like. Adding more slots gives other plug-ins more control over where their contributions appear relative to existing contributions.

But wait! We know that there are other menu items on the Help menu. These are added by plug-ins. For example, the help plug-in adds an action set containing the "Help Contents" menu to the workbench. Here's the markup from the **org.eclipse.help.ui** plug-in's **plugin.xml**.

```
<extension
  point="org.eclipse.ui.actionSets">
  <actionSet
    id="org.eclipse.help.internal.ui.HelpActionSet"
    label="%help"
    visible="true">
    <action
      id="org.eclipse.help.internal.ui.HelpAction"
      menubarPath="help/helpEnd"
      label="%helpcontents"
      class="org.eclipse.help.internal.ui.ShowHelp"/>
    </actionSet>
  </extension>
```

The new help action will be placed in the help menu, inside the **helpEnd** group. If no other plug-in has contributed to the help menu, this means the "Help Contents" menu item will appear as the first item in the menu above the "About" item. If another plug-in wanted to contribute an item that always appeared above the "Help Contents" item, then it could specify the **helpStart** group on its path.

Toolbar paths work similarly. Whenever a path is specified, it must end with the name of a group in the toolbar.

Fully qualified menu and tool paths

A complete menu or toolbar path is simply a list of menu name and named group pairs. Menu names for the workbench are also defined in [IWorkbenchActionConstants](#). This is how we knew that the fully qualified path name for our help action was "**help/helpEnd**."

Some menus have nested submenus. This is where longer paths come into play. If the help menu had defined a submenu called "**submenu**" with a named group called "**submenuStart**," then the fully qualified menu path for an action in the new submenu would be "**help/submenu/submenuStart**."

Externalizing UI labels

The example above also demonstrates a technique for externalizing strings that appear in the UI. This is useful for translating the plug-in's UI to other languages. We can externalize the strings in our **plugin.xml** files by replacing the string with a key (e.g. **%help**, **%helpcontents**) and creating entries in the **plugin.properties** file of the form:

```
help = "Help"
helpContents = "Help Contents"
```

This way, the **plugin.properties** file can be translated for different languages and the **plugin.xml** will not need to be modified.

Adding new menus and groups

In many of the examples we've seen so far, the actions contributed by the sample plug-ins have been added to existing named groups within menus and toolbars.

The [actionSets](#), [viewActions](#), [editorActions](#), and [popupMenus](#) extension points also allow you to define new menus and groups within your contribution. This means that you can define new submenus or new pull-down menus and contribute your actions to these new menus. In this case, the path for your new action will contain the name of your newly defined menu.

We saw this technique when the readme tool defined a new menu for its action set. Let's look at the markup one more time now that we've looked at menu paths in more detail.

```
<extension
  point = "org.eclipse.ui.actionSets">
  <actionSet
    id="org_eclipse_ui_examples_readmetool_actionSet"
    label="ReadMe Actions"
    visible="true">
    <menu
      id="org_eclipse_ui_examples_readmetool"
      label="Readme &File Editor"
      path="window/additions">
      <separator name="slot1"/>
```

```

        <separator name="slot2"/>
        <separator name="slot3"/>
    </menu>
    <action
        id="org_eclipse_ui_examples_readmetool_readmeAction"
        menubarPath="window/org_eclipse_ui_examples_readmetool/slot1"
        toolbarPath="readme"
        label="&Open Readme Browser@Ctrl+R"
        tooltip="Open Readme Browser"
        helpContextId="org.eclipse.ui.examples.readmetool.open_browser_action_context"
        icon="icons/basic/ctool16/openbrwsr.gif"
        class="org.eclipse.ui.examples.readmetool.WindowActionDelegate"
        enablesFor="1">
        <selection
            class="org.eclipse.core.resources.IFile"
            name="*.readme">
        </selection>
    </action>
</actionSet>
</extension>

```

We added a new menu called "**org_eclipse_ui_examples_readmetool.**" Its label is the string "**Readme &File Editor.**" Within this menu, we define three named groups: "**slot1,**" "**slot2,**" and "**slot3.**" We add this new menu to the path "**window/additions.**"

If we go back to [IWorkbenchActionConstants](#), we see this definition of the window menu in the javadoc:

```

* <h3>Standard Window menu actions</h3>
* <ul>
* <li>Extra Window-like action group (<code>WINDOW_EXT</code>)</li>

```

If we look further at the class definition, we will see these related definitions:

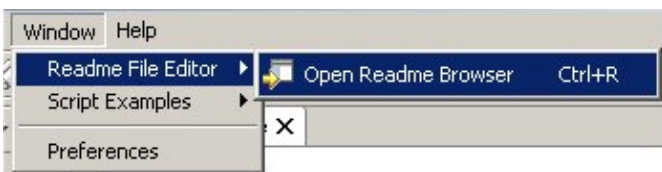
```

public static final String MENU_PREFIX = "";
...
public static final String M_WINDOW = MENU_PREFIX+"window";
...
public static final String MB_ADDITIONS = "additions"; // Group.
...
public static final String WINDOW_EXT = MB_ADDITIONS; // Group.

```

From this information, we can piece together the path for adding something to the workbench "Window" menu. The menu itself is called "**window**" and it defines one slot called "**additions.**" We use the path "**window/additions**" to add our new menu.

In the action set declaration, we add an action to our newly defined menu, using the path "**window/org_eclipse_ui_examples_readmetool/slot1.**"



Other plug-ins could add to our menu by using this same path (or perhaps one of the other slots) to add one of their own menus.

In general, it's not good practice to contribute to another plug-in's menu or tool bar by deriving the path name from the **plugin.xml**. It's possible that a future version of the plug-in could change the names of the paths. The recommended practice is to define a public interface (much like [IWorkbenchActionConstants](#)) which specifies exactly which menus, tool bar groups, and slots are considered fair game for use by other plug-ins.

More workbench extensions

We'll see the readme tool extensions associated with wizards, preferences, properties, and dialogs in [Dialogs and wizards](#) and [Preferences and properties](#).

Next, we will look at some additional workbench extensions points that you may want to use. Since the readme tool does not contribute to these extension points, we will look at example extensions that are implemented by the platform workbench, the platform help system, and Java tooling (JDT).

org.eclipse.ui.perspectives

We've already seen some ways the workbench allows the user to control the appearance of plug-in functionality. Views can be hidden or shown using the **Perspective**→**Show View** menu. Action sets can be hidden or shown using the **Perspective**→**Customize...** menu. These features help the user organize the clutter in the workbench at a fine grained level.

Perspectives

Perspectives provide an additional layer of organization. Users can switch between perspectives as they move across tasks. A perspective defines an initial collection and layout of views that should be used when the user first switches to it. It also defines the initial visible action sets.

The platform itself defines one perspective, the **Resource** perspective. Other platform plug-ins, such as the help system and the Java tooling, define additional perspectives. Your plug-in can define its own perspective by contributing to the [org.eclipse.ui.perspectives](#) extension point.

The specification of the perspective in the **plugin.xml** is straightforward. The following markup is used by the workbench in defining its own resource perspective.

```
<extension
  point="org.eclipse.ui.perspectives">
  <perspective
    id="org.eclipse.ui.resourcePerspective"
    name="Resource"
    class="org.eclipse.ui.internal.ResourcePerspective">
  </perspective>
</extension>
```

A plug-in must supply an **id** and **name** for the perspective, along with the name of the **class** that implements the perspective. An **icon** can also be specified. The perspective class should implement [IPerspectiveFactory](#).

We can see from the markup that the real work happens in the code. The interface for the perspective factory is straightforward. Implementors of [IPerspectiveFactory](#) are expected to configure an [IPageLayout](#) with information that describes the perspective and its perspective page layout.

Workbench part layout

One of the main jobs of an [IPageLayout](#) is to describe the placement of the editor and views in the workbench window. Note that these layouts are different than the [Layout](#) class in SWT. Although [IPageLayout](#) and [Layout](#) solve a similar problem (sizing and positioning widgets within a larger area), you do not have to understand SWT layouts in order to supply a perspective page layout.

A perspective page layout is initialized with one area for displaying an editor. The perspective factory is responsible for adding additional views relative to the editor. Views are added to the layout in relationship (top, bottom, left, right) to another part. Placeholders (empty space) can also be added for a view that is not initially shown.

[IFolderLayout](#) can be used to group views into tabbed folders. For example, the Resource perspective places the resource navigator inside a folder at the top left corner of the workbench. Placeholders are commonly used with folder layouts. The Resource perspective defines a placeholder for the bookmarks view in the same folder as the resource navigator. If the user shows the bookmarks view, it will appear in the same folder with the navigator, with a tab for each view.

[IPageLayout](#) also allows you to add action sets to a perspective. You can also add a number of shortcuts to perspective related menus. A new wizard shortcut will add a new entry to the **File**→**New** menu for a perspective and invoke the appropriate wizard. View shortcuts add the names of views that should appear in the **Perspective**→**Show View** menu when the perspective is active. Perspective shortcuts add the names of perspectives that should appear in the **Perspective**→**Open** menu when the perspective is active.

org.eclipse.ui.perspectiveExtensions

Plug-ins can add action sets, views, and various shortcuts to existing perspectives by contributing to the [org.eclipse.ui.perspectiveExtensions](#) extension point.

The concepts discussed above for action sets, wizard entries, view layout, view shortcuts, and perspective shortcuts also apply when plug-ins contribute to an existing perspective. One important difference is that these items are specified in the **plugin.xml** markup instead of configuring them into an [IPageLayout](#).

The following markup shows how the JDT extends the platform's debug perspective.

```
<extension
  point="org.eclipse.ui.perspectiveExtensions">
  <perspectiveExtension
    targetID="org.eclipse.debug.ui.DebugPerspective">
    <actionSet id="org.eclipse.jdt.ui.JavaActionSet"/>
    <view
      id="org.eclipse.jdt.debug.ui.DisplayView"
      relative="org.eclipse.debug.ui.InspectorView"
      relationship="stack"/>
    </perspectiveExtension>
  </extension>
```

The **targetID** is the id of the perspective that the extension is contributing to. Specifying a **perspectiveShortcut** indicates that another perspective (specified by **id**) should be added to the **Perspective**→**Open** menu of the target perspective. This is analogous to calling [IPageLayout.addPerspectiveShortcut](#) in the original perspective definition in the [IPerspectiveFactory](#). The **actionSet** parameter identifies the **id** of a previously declared action set that should be added to the target perspective. This is analogous to using [IPageLayout.addActionSet](#) in the [IPerspectiveFactory](#).

Contributing a view to a perspective is a little more involved, since the perspective page layout information must be declared. In addition to supplying the **id** of the contributed view, the id of a view that already exists in the perspective (a **relative** view) must be specified as a reference point for placing the new view. The **relationship** parameter specifies the layout relationship between the new view and the **relative** view. The value **stack** indicates that the view will be stacked with the relative view in a folder.

You can also specify **left**, **right**, **top**, or **bottom**, which indicates that the new view will be placed beside the **relative** view. In this case, a ratio between 0.0 and 1.0 must be defined, which indicates the percentage of area in the **relative** view that will be allocated to the new view.

Plug-ins can also add view shortcuts and new wizard shortcuts in a similar manner. See org.eclipse.ui.perspectiveExtensions for a complete definition of the extension point.

org.eclipse.ui.elementFactories

Element factories are used to recreate workbench model objects from data that was saved during workbench shutdown.

Before we look closely at the element factory extension, we need to review a general technique used throughout the platform to add plug-in specific behavior to common platform model objects.

IAdeptables and workbench adapters

When browsing the various workbench classes, you will notice that many of the workbench interfaces extend the [IAdeptable](#) interface.

Plug-ins use adapters to add specific behavior to pre-existing types in the system. For example, the workbench may want to add behavior to resources so that they answer a label and image to represent themselves in the UI. We know that it's not good design to add UI specific behavior to low level objects, so how can we add this behavior to the resource types?

The platform defines a technique for dynamically querying an object for its implementation of a particular interface. Plug-ins can register adapters which add behavior to pre-existing types. This way, application code can later query for a particular adapter for an object when using it in a specific context. If there is one registered for it, they can obtain the adapter and use the new behaviors defined in the adapter.

By providing a facility to dynamically query an adapter for an object, we can also improve the flexibility of the system as it evolves. New adapters can be registered for platform types by new plug-ins without having to change the definitions of the original types. The pattern is to ask an object for a particular adapter.

```
//given an object o, we want to do "workbench" things with it.
if (!(o instanceof IAdeptable)) {
    return null;
}
IWorkbenchAdapter adapter = (IWorkbenchAdapter)o.getAdapter(IWorkbenchAdapter.class);
if (adapter == null)
    return null;
// now I can treat o as an IWorkbenchAdapter
...
```

If there is no adapter registered for the object in hand, null will be returned as the adapter. Clients must be prepared to handle this case. This allows clients to gracefully handle the case where an expected adapter has not been registered.

The workbench uses adapters to obtain UI information from the base platform types, such as [IResource](#). This shields the base types from particular knowledge of the UI and allows the workbench to evolve its interfaces without changing the definitions of the base.

Without adapters, any class that might be passed around in the workbench API would have to implement the UI interfaces. This clutters the class definitions and introduces tight coupling. It introduces circular dependencies between the core and UI classes. With adapters, each class implements [IAdaptable](#) and uses the adapter registry to allow plug-ins to extend the behavior of the base types.

Throughout the workbench code, you'll see cases where a type is queried for an adapter. This is all happening in order to obtain an object that knows how to answer UI oriented information about a platform core type.

Element factories

When the workbench is shut down by the user, it must save the current state of the [IAdaptable](#) objects shown in the workbench. This is done by saving primitive data parameters of the object in a special format, an [IMemento](#), that is easily saved in the file system. The id of a factory that can recreate the object from an [IMemento](#) is also stored.

When the platform is restarted, the workbench finds the element factory associated with the factory id known in the memento. It finds the factory by checking the plug-in registry for contributions to the [org.eclipse.ui.elementFactories](#) extension.

The markup is pretty simple. We just have to specify the id of the factory and the corresponding class that implements the factory.

The following code snippet is from the workbench **plugin.xml**.

```
<extension
  point = "org.eclipse.ui.elementFactories">
  <factory
    id = "org.eclipse.ui.internal.model.ResourceFactory"
    class = "org.eclipse.ui.internal.model.ResourceFactory">
  </factory>
  <factory
    id = "org.eclipse.ui.internal.model.WorkspaceFactory"
    class = "org.eclipse.ui.internal.model.WorkspaceFactory">
  </factory>
  <factory
    id = "org.eclipse.ui.part.FileEditorInputFactory"
    class = "org.eclipse.ui.part.FileEditorInputFactory">
  </factory>
  <factory
    id = "org.eclipse.ui.internal.dialogs.WelcomeEditorInputFactory"
    class = "org.eclipse.ui.internal.dialogs.WelcomeEditorInputFactory">
  </factory>
</extension>
```

org.eclipse.ui.resourceFilters

The resource filters extension allows plug-ins to define filters that are useful for filtering out file types in the resource navigator view. This is useful when special file types are used to represent internal plug-in information that should not be shown in the workbench or manipulated by the user.

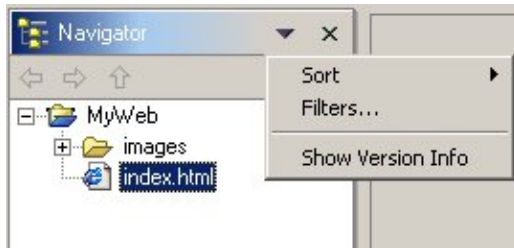
The workbench filters out the pattern `".*"` to exclude internal files such as `.metadata` from the resource

navigator. Likewise, the JDT plug-in filters out `*.class` files to hide compiled classes.

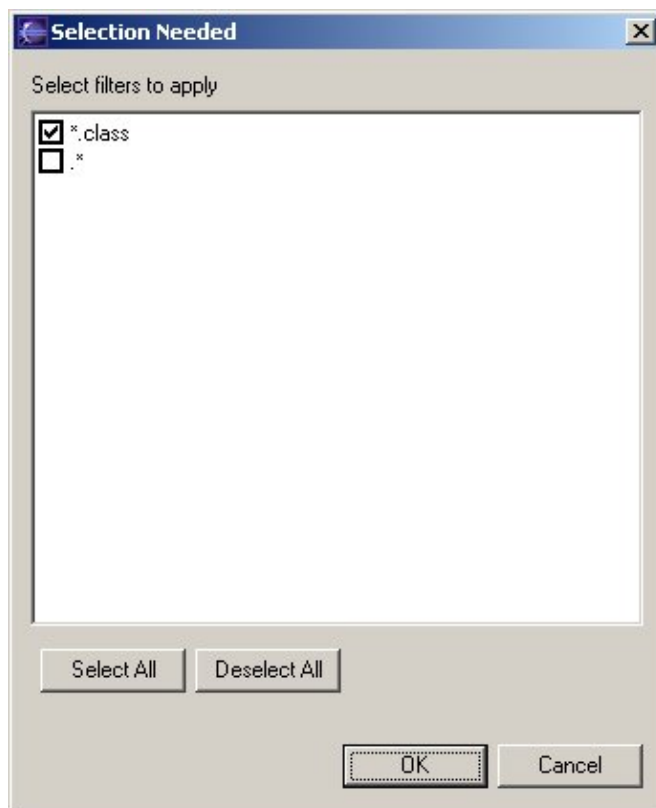
The markup for the resource filters extension is simple. The following is from the workbench `plugin.xml`.

```
<extension
  point= "org.eclipse.ui.resourceFilters">
  <filter pattern = ".*" selected = "false"/>
</extension>
```

The filters can be enabled by the user using the resource navigator's local pull-down menu.



In addition to declaring the **filter pattern**, the plug-in can use the **selected** attribute to specify whether the filter should be enabled in the resource navigator. This only determines the initial state of the filter pattern. The user can control which filter patterns are active.



Beyond the basics

You should know enough by now to browse the workbench code and the rest of this documentation as you see fit. The next few chapters take a look at the underlying UI architecture that supports the platform workbench.

- [Dialogs and wizards](#) introduces more workbench extensions for contributing preferences, properties, and wizards to the workbench. It also introduces the JFace UI framework and its support for dialogs and wizards.
- [JFace: UI framework for plug-ins](#) will cover many of the basic concepts in the JFace UI framework, such as viewers and actions.
- [Standard Widget Toolkit](#) introduces the lowest level of the UI architecture, SWT.

Dialogs and wizards

We've already seen how to extend the workbench UI by adding views, editors, and actions to the workbench. Now we can tie it all together by launching our own dialogs in response to these actions.

The JFace UI framework provides several standard dialogs and a framework for building your own dialogs and wizards. We'll look at the different kinds of dialogs and wizards and how to build them.

We'll also cover some simple workbench extensions for contributing wizards.

Standard dialogs

The package [org.eclipse.jface.dialogs](#) defines the basic support for dialogs. This package provides standard dialogs for displaying user messages and obtaining simple input from the user.

- [MessageDialog](#) displays a message to the user. You can set the dialog title, image, button text, and message in the constructor for this dialog.
- [ErrorDialog](#) displays information about an error. You can set the dialog title and message for the dialog. You can also supply an IStatus object which the dialog will use to obtain an error message.
- [InputDialog](#) allows the user to enter text. You can set the dialog title, default text value, and supply an object that will validate the text input.
- [ProgressMonitorDialog](#) shows progress to the user during the running of a long operation.

The standard dialogs are designed so that you can completely specify the dialog in its constructor. We saw a [MessageDialog](#) in action in the readme tool's view action:

```
MessageDialog.openInformation(
    view.getSite().getShell(), "Readme Editor", "View Action executed");
```

Application dialogs

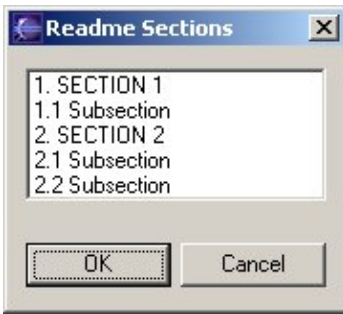
When a standard dialog is too simple for your plug-in, you can build your own dialog using the [Dialog](#) class. Earlier, we saw how the readme tool contributed an "Open Readme Browser" action in an action set. This action set is shown in the workbench tool bar and **Window->Readme File Editor** menu.

Now we are ready to look at the implementation of this action in the readme tool's **WindowActionDelegate**.

```
public void run(IAction action) {
    SectionsDialog dialog = new SectionsDialog(window.getShell(),
        ReadmeModelFactory.getInstance().getSections(selection));
    dialog.open();
}
```

The window action delegate for the action set uses the current selection in the resource navigator view (the **.readme** file) to get a list of sections in the readme file. This list and the workbench window's shell are passed to the **SectionsDialog**.

When the user selects the action, the **SectionsDialog** is opened.



The **SectionsDialog** is implemented in the readme tool plug-in by subclassing the [Dialog](#) class in the [org.eclipse.jface.dialogs](#) package.

The [Dialog](#) class provides basic support for building a dialog shell window, creating the common dialog buttons, and launching the dialog. The subclasses are responsible for handling the content of the dialog itself:

- **createDialogArea** creates the SWT controls that represent the dialog contents. This is similar to creating the controls for a view or editor.

The **SectionsDialog** creates an SWT list to display the list of sections. It uses a JFace viewer to populate the list. (We'll look at JFace viewers in [Viewers](#).) Note that our dialog does not have to create any of the buttons for the dialog since this is done by our superclass.

```
protected Control createDialogArea(Composite parent) {
    Composite composite = (Composite)super.createDialogArea(parent);
    List list = new List(composite, SWT.BORDER);
    ...
    ListViewer viewer = new ListViewer(list);
    ...
    return composite;
}
```

- **configureShell** is overridden to set an appropriate title for the shell window.

```
protected void configureShell(Shell newShell) {
    super.configureShell(newShell);
    newShell.setText("Readme Sections");
    ...
}
```

- **okButtonPressed** is overridden to perform whatever action is necessary when the user presses the OK button. (You can also override **cancelButtonPressed** or **buttonPressed(int)** depending on the design of your dialog.)

SectionsDialog does not implement an **okButtonPressed** method. It inherits the "do-nothing" implementation from [Dialog](#). This is not typical. Your dialog usually performs some processing in response to one of the dialog buttons being pressed.

Dialogs can be as simple or as complicated as necessary. When you implement a dialog, most of your dialog code is concerned with creating the SWT controls that represent its content area and handling any events necessary while the dialog is up. Once a button is pressed by the user, the dialog can query the state of the various controls (or viewers) that make up the dialog to determine what to do.

Dialog settings

The [org.eclipse.jface.dialogs](#) package provides a utility class, [DialogSettings](#), for storing and retrieving keyed values. You can use this class to save and retrieve primitive data types and string values that you associate with key names. The settings are loaded and saved using an XML file.

[AbstractUIPlugin](#) provides support for plug-in wide dialog settings stored in an XML file in your plug-in's directory. If a dialog settings file is not found in your plug-in directory, an empty [DialogSettings](#) will be created for you. When the plug-in is shut down, any settings that were added to it will be saved in an XML file and retrieved the next time the plug-in is started up.

You can access your dialog settings anywhere in your plug-in code. The following snippet shows how you could obtain the dialog settings for the readme tool.

```
IDialogSettings settings = ReadmePlugin.getDefault().getDialogSettings();
```

Values are stored and retrieved using get and put methods. The get methods are named after the type of primitive that is being accessed. You can store and retrieve boolean, long, double, float, int, array, and string values. The following snippet shows how we could use dialog settings to initialize control values in a dialog.

```
protected Control createDialogArea(Composite parent) {
    IDialogSettings settings = ReadmePlugin.getDefault().getDialogSettings();
    checkbox = new Button(parent, SWT.CHECK);
    checkbox.setText("Generate sample section titles");
    // initialize the checkbox according to the dialog settings
    checkbox.setSelection(settings.getBoolean("GenSections"));
}
```

The value of the setting can be stored later when the ok button is pressed.

```
protected void okPressed() {
    IDialogSettings settings = ReadmePlugin.getDefault().getDialogSettings();
    // store the value of the generate sections checkbox
    settings.put("GenSections", checkbox.getSelection());
    super.okPressed();
}
```

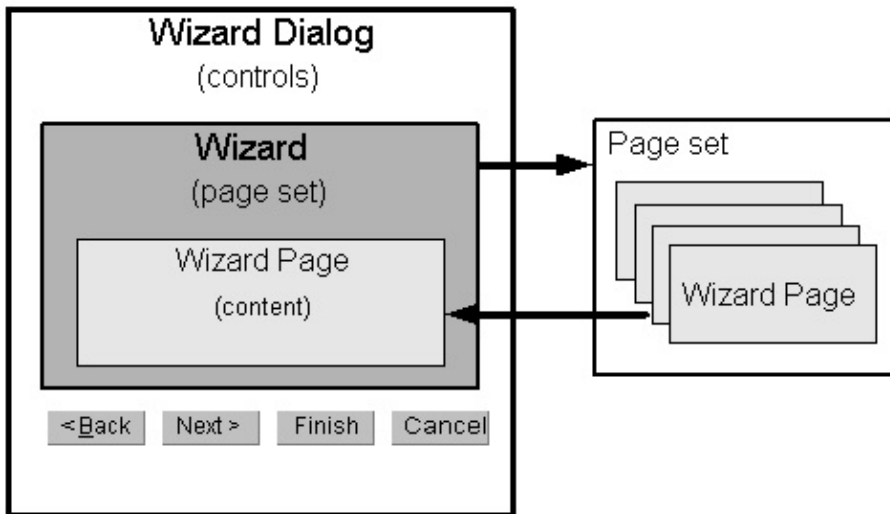
Wizards

Wizards are used to guide the user through a sequenced set of tasks. Your plug-in can contribute wizards at predefined extension points in the workbench. It can also create and launch its own wizards.

When you contribute to a workbench wizard extension point, the actions that launch the wizard are already set up by the workbench. You need only supply the wizard that will be used.

If you need to launch other wizards that are not already defined in workbench wizard extension points, you must launch them yourself. You can launch your own wizards by adding an action to a view, editor, popup, or an action set.

A wizard is composed of several different underlying parts.



Wizard dialog

The wizard dialog ([WizardDialog](#)) is the top level dialog in a wizard. It defines the standard wizard buttons and manages a set of pages that are provided to it.

When you contribute to a workbench wizard extension, you do not have to create a wizard dialog. One is created on your behalf by the workbench, and your wizard is set into it.

The wizard dialog performs the enabling and disabling of the **Next**, **Back**, and **Finish** buttons based on information it obtains from the wizard and the current wizard page.

Wizard

The wizard ([IWizard](#)) controls the overall appearance and behavior of the wizard, such as title bar text, image, and the availability of a help button. Wizards often use a corresponding [DialogSettings](#) to obtain (and store) the default values for the settings of controls on the wizard pages.

The [Wizard](#) class implements many of the details for standard wizard behavior. You typically extend this class to implement behavior specific to your wizard. The primary responsibilities of your wizard will include:

- creating and adding your pages to your wizard
- implementing the behavior that should occur when the user presses the **Finish** button.

Wizard page

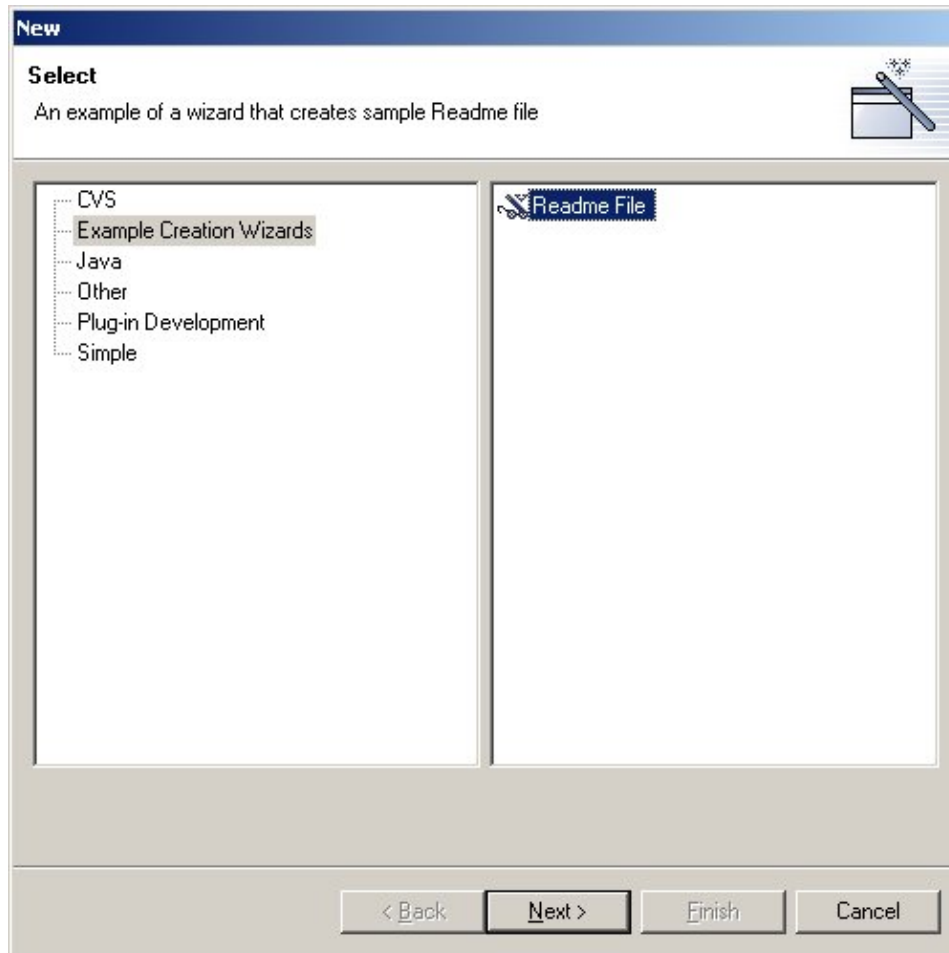
The wizard page ([IWizardPage](#)) defines the controls that are used to show the content of the wizard page. It responds to events in its content areas and determines when the page is completed.

Your wizard page typically extends the [WizardPage](#) class. The primary responsibilities of your wizard page will include:

- creating the SWT controls that represent the page
- determining when the user has supplied enough information to complete the page (that is, when the user can move to the next page.)

Workbench wizard extension points

The workbench defines extension points for wizards that create new resources, import resources, or export resources. When you select the new, import, or export menu, the workbench uses a wizard selection dialog to display all the wizards that have been contributed for that particular extension point. The new wizard dialog is shown below.



Your wizard takes control once it is selected in the list and the **Next** button is pressed. This is when your first page becomes visible.

org.eclipse.ui.newWizards

You can add a wizard to the **File**→**New** menu option in the workbench using the [org.eclipse.ui.newWizards](#) extension point. The readme tool example uses this extension point definition to add the Readme File wizard:

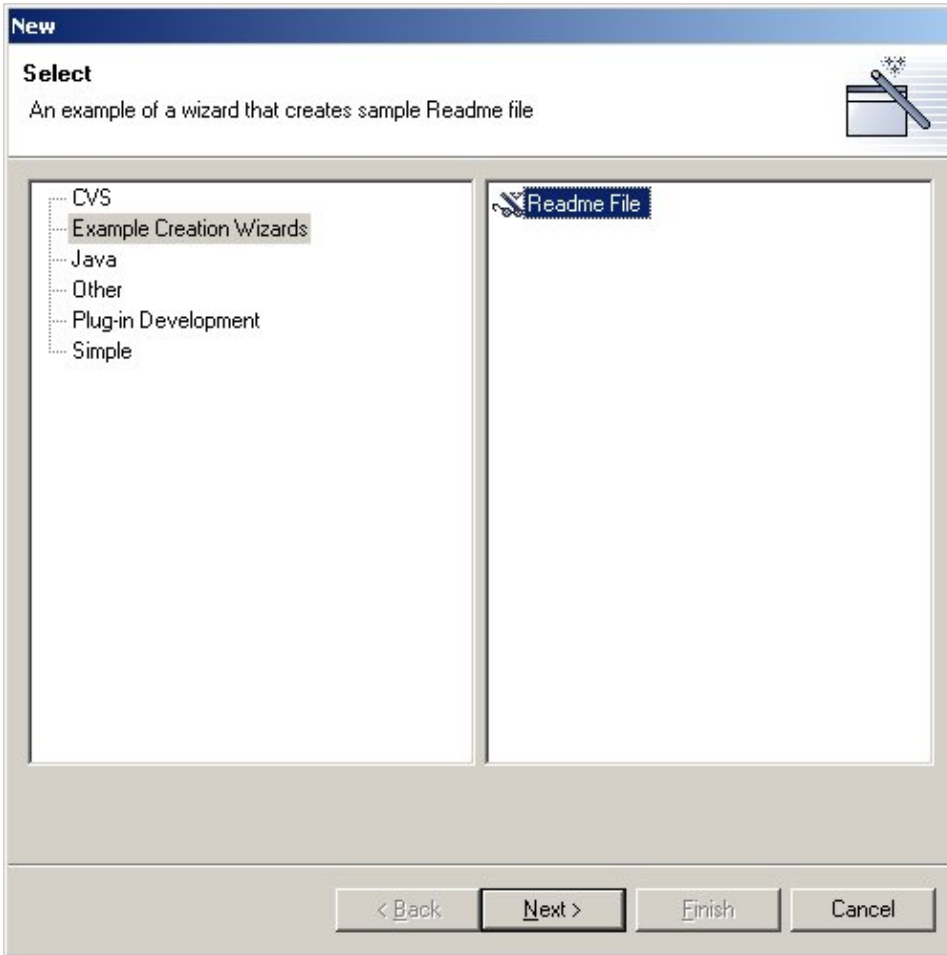
```
<extension
  point = "org.eclipse.ui.newWizards">
  <category
    id = "org.eclipse.ui.examples.readmetool.new"
    name="Example Creation Wizards">
  </category>
  <wizard
    id = "org.eclipse.ui.examples.readmetool.wizards.new.file"
    name = "Readme File"
    class="org.eclipse.ui.examples.readmetool.ReadmeCreationWizard"
```

```

category="org.eclipse.ui.examples.readmetool.new"
icon="icons/basic/obj16/newreadme_wiz.gif">
<description>
    An example of a wizard that creates sample Readme file
</description>
<selection class="org.eclipse.core.resources.IResource"/>
</wizard>
</extension>

```

The **category** describes where the wizard will be grouped when the wizard selection dialog is used to find and launch a wizard. The category **name** ("Example Creation Wizards") defines the label that is used in the wizard. The wizard itself is assigned to the category. Its **name**, **icon**, and **description** all show up in the new wizard selection dialog shown below.



All of this appears based on the markup in the **plugin.xml** file. None of the plug-in code runs until the user chooses the **Next** button. Once this happens, the workbench will instantiate the wizard **class** specified in the markup and pass it an expected selection **class**.

The class identified in this extension (**ReadmeCreationWizard**) must implement the [INewWizard](#) interface. Most wizards do so by extending the platform [Wizard](#) class although this is an implementation mechanism and not required by the extension point.

The wizard itself does little but create the pages inside of it. Let's look at the implementation of the page first, and then come back to the wizard.

Pages

The workbench provides base wizard page classes that support the type of processing performed for each wizard extension point. You can use these pages, or extend them to add additional processing.

The goal of the **ReadmeCreationWizard** is to create a new file, add the required content to the file, and as an option, open an editor on the file. Our page needs to define the controls that let the user specify what content goes in the file and whether an editor should be launched.

We create the wizard page, **CreateReadmePage1**, by extending [WizardNewFileCreationPage](#). The controls for a wizard page are defined in a fashion similar to the definition of the controls for a view or an editor. The page implements a **createControl** method, creating the necessary SWT widgets as children of the supplied [Composite](#). Since the superclass already adds widgets that support new file processing, we need only extend the **createControl** method in our wizard page to add the additional checkboxes that control generation of sections and opening of the editor.

```
public void createControl(Composite parent) {
    // inherit default container and name specification widgets
    super.createControl(parent);
    Composite composite = (Composite)getControl();
    ...
    // sample section generation group
    Group group = new Group(composite, SWT.NONE);
    group.setLayout(new GridLayout());
    group.setText("Automatic sample section generation");
    group.setLayoutData(new GridData(GridData.GRAB_HORIZONTAL |
        GridData.HORIZONTAL_ALIGN_FILL));
    ...
    // sample section generation checkboxes
    sectionCheckbox = new Button(group, SWT.CHECK);
    sectionCheckbox.setText("Generate sample section titles");
    sectionCheckbox.setSelection(true);
    sectionCheckbox.addListener(SWT.Selection, this);

    subsectionCheckbox = new Button(group, SWT.CHECK);
    subsectionCheckbox.setText("Generate sample subsection titles");
    subsectionCheckbox.setSelection(true);
    subsectionCheckbox.addListener(SWT.Selection, this);
    ...
    // open file for editing checkbox
    openFileCheckbox = new Button(composite, SWT.CHECK);
    openFileCheckbox.setText("Open file for editing when done");
    openFileCheckbox.setSelection(true);
    ...
}
```

You should be able to follow this code if you understand the concepts in [Standard Widget Toolkit](#).

The basic patterns for implementing a page include:

- Add listeners to any controls that affect dynamic behavior of the page. For example, if selecting an item in a list or checking a box affects the state of other controls of the page, add a listener so you can change the state of the page.
- Populate the controls with data based on the current selection when the wizard was launched. Some of the data may depend on the values in other controls. Some of the controls may use dialog settings to initialize their values.
- Use **setPageComplete(true)** when enough information is provided by the user to exit the page (and

move to the next page or finish the wizard.)

The **ReadmeCreationPage** class inherits a lot of this behavior from the [WizardNewFileCreationPage](#). Browse the implementation of these classes for further information.

Now that we understand what a page does, let's look again at the wizard.

Wizard

The wizard is responsible for creating the pages and providing the "finish" logic.

The basic patterns for implementing a wizard include:

- Implement the **init** method to set up local variables for context information such as the workbench and the current selection.

```
public void init(IWorkbench workbench, IStructuredSelection selection) {
    this.workbench = workbench;
    this.selection = selection;
    setWindowTitle("New Readme File");
    setDefaultPageImageDescriptor(ReadmeImages.README_WIZARD_BANNER);
}
```

- Implement **addPages** by creating instances of the pages.

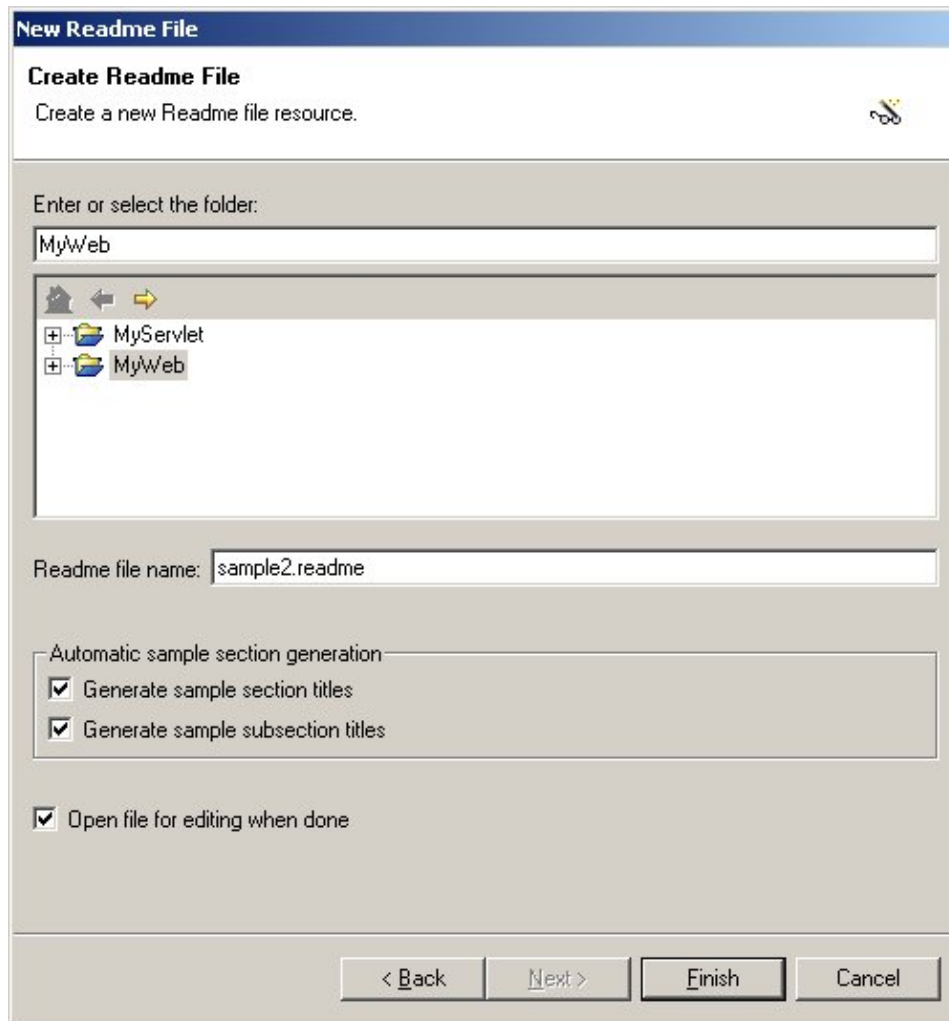
```
public void addPages() {
    mainPage = new ReadmeCreationPage(workbench, selection);
    addPage(mainPage);
}
```

- Implement **performFinish** to finish the task.

Multi-page wizards typically handle the finish logic in the wizard itself, since each page will contribute information that determines how the task is implemented. Single page wizards can implement the logic in the wizard or ask the page to finish the job. The approach you take largely depends on where your important state is kept. In the case of the readme wizard, we are going to ask our page to handle the finish processing.

```
public boolean performFinish() {
    return mainPage.finish();
}
```

The completed wizard looks like this:



org.eclipse.ui.importWizards

You can add a wizard to the **File**→**Import** menu option in the workbench using the [org.eclipse.ui.importWizards](#) extension point. The process for defining the extension and implementing the wizard is similar to [org.eclipse.ui.newWizards](#). The primary difference in the markup is that import wizards do not define or assign categories for the wizards themselves. The wizards appear uncategorized in a wizard dialog.



The wizard supplied in the class parameter of the markup must implement [IImportWizard](#). Its pages are typically extended from [WizardImportPage](#).

org.eclipse.ui.exportWizards

You can add a wizard to the **File**→**Export** menu option in the workbench using the [org.eclipse.ui.exportWizards](#) extension point. The process for defining the extension and implementing the wizard is similar to [org.eclipse.ui.newWizards](#). The primary difference in the markup is that export wizards do not define or assign categories for the wizards themselves. The wizards appear uncategorized in a wizard dialog.



The wizard supplied in the class parameter of the markup must implement [IExportWizard](#). Its pages are typically extended from [WizardExportPage](#).

Using wizard dialogs

The previous example supplied a wizard for a specified extension point. Another, perhaps more common, case is that you want to launch your own plug-in's wizard from some action that you have defined. (In [Workbench menu contributions](#), we discuss the ways you can contribute actions to the workbench.)

When you are launching your own wizard, you need to wrap the wizard in a [WizardDialog](#). This detail will not be handled for you by the workbench like it is when you contribute a wizard extension.

For example, the **ReadmeCreationWizard** could be launched independently by creating a wizard dialog and associating it with the **ReadmeCreationWizard**. The following code snippet shows how this could be done from some action delegate. (The method assumes that we know the workbench and the selection.)

```
public void run(IAction action) {
    // Create the wizard
    ReadmeCreationWizard wizard = new ReadmeCreationWizard();
    wizard.init(getWorkbench(), selection);

    // Create the wizard dialog
```

```

WizardDialog dialog = new WizardDialog
    (getWorkbench().getActiveWorkbenchWindow().getShell(), wizard);
// Open the wizard dialog
dialog.open();
}

```

Multi-page wizards

If your wizard implements a complex task, you may want to use more than one page to obtain information from the user.

In general, the implementation pattern is the same as for a single page wizard.

- Create a [WizardPage](#) subclass for each page in your wizard. Each wizard page should use **setPageComplete(true)** when it has enough information.
- Create a [Wizard](#) subclass which adds each page to the wizard.
- Implement a **performFinish** method to perform the finish logic.

When you design a wizard, it's good practice to put all the required information on the first page if possible. This way, the user does not have to traverse the entire set of pages in order to finish the task. Optional information can go on subsequent pages.

When a page requires input from the user before it can be considered complete, use **setPageComplete(false)** to signify that it is not complete. As the page receives events from its controls, it rechecks to see if the page is complete. Once the required input is provided, **setPageComplete(true)** signals completion.

The [Wizard](#) class handles the logic required to enable and disable the **Finish** button according to the completion state of the pages. The **Finish** button is only enabled for a wizard when each of its pages have set its completion state to true.

Validation and page control

The classes [WizardNewFileCreationPage](#) and [CreateReadme1](#) show a common pattern for implementing page validation.

[WizardNewFileCreationPage](#) defines a common event handler for all SWT events which validates the page. This means the page will be validated whenever an event is received from a widget to which the page added a listener.

```

public void handleEvent(Event event) {
    setPageComplete(validatePage());
}

```

Once the [CreateReadme1](#) page creates its controls, it sets the state of the page using **validatePage**.

```

public void createControl(Composite parent) {
    super.createControl(parent);
    // create controls, add listeners, and layout the page
    ...
    // sample section generation checkboxes
    sectionCheckbox = new Button(group, SWT.CHECK);
    sectionCheckbox.setText("Generate sample section titles");
    sectionCheckbox.setSelection(true);

    // we add a listener to this checkbox
}

```

```
        sectionCheckbox.addListener(SWT.Selection, this);
        ...
        setPageComplete(validatePage());
    }
}
```

Using this pattern, a wizard page can put all of its page validation code in one method, **validatePage()**. This method determines the initial state of the page and recalculates the state any time it receives an event from a widget on its page.

Since we added a listener to the section checkbox, we will recompute the valid state of the page whenever that checkbox receives a selection event. Note that the page's **handleEvent** method must call **super** to ensure that the inherited page validation behavior occurs in addition to any specific event handling for this page.

```
public void handleEvent(Event e) {
    Widget source = e.widget;
    if (source == sectionCheckbox) {
        if (!sectionCheckbox.getSelection())
            subsectionCheckbox.setSelection(false);
        subsectionCheckbox.setEnabled(sectionCheckbox.getSelection());
    }
    super.handleEvent(e);
}
```


Preferences and properties

The workbench provides additional extension points for contributing specialized dialog pages. Preference pages allow the user to customize a plug-in by selecting values for a plug-in defined set of preferences.

Properties pages allow users to see, and possibly edit, the plug-in specific properties of a resource.

Preferences

The workbench implements a generic preferences architecture that allows plug-ins to store user preference values and contribute a preference page to the workbench preferences dialog. We will look again at the readme tool example to see how this is done, and then look at some of the underlying support for building preference pages.

`org.eclipse.ui.preferencePages`

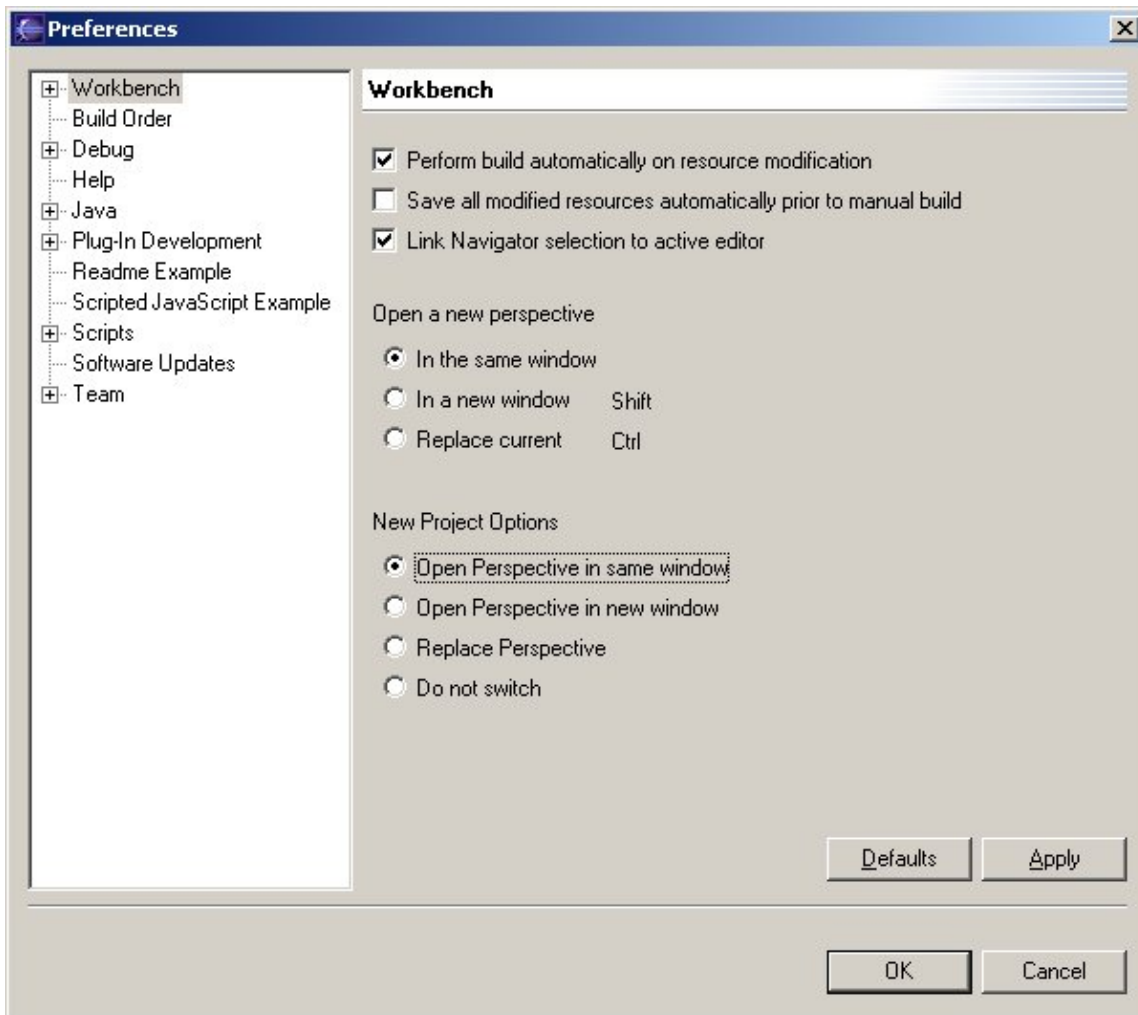
The [org.eclipse.ui.preferencePages](#) extension point allows you to contribute pages to the workbench preferences (**Window**→**Preferences**) dialog. The preferences dialog presents a hierarchical list of user preference entries. Each entry displays a corresponding preference page when selected.

The readme tool uses this extension point to add the Readme Example preferences page.

```
<extension
  point = "org.eclipse.ui.preferencePages">
  <page
    id="org.eclipse.ui.examples.readmetool.Page1"
    class="org.eclipse.ui.examples.readmetool.ReadmePreferencePage"
    name="Readme Example">
  </page>
</extension>
```

This markup defines a preference page named "Readme Example" which is implemented by the class **ReadmePreferencePage**. The class must implement the [IWorkbenchPreferencePage](#) interface.

The workbench uses a [PreferenceManager](#) to keep a list of all nodes in the preference tree and their corresponding pages. This list can be initialized from information in the plug-in registry without running any plug-in code. Your plug-in's contribution to the preferences dialog (the "Readme Example" entry on the left) is shown before any of your code is run.



The "Readme Example" preference is added to the top level of the preference tree on the left. Why? Because a preference page contribution will be added as a root of the tree unless a **category** attribute is specified. (The name **category** is somewhat misleading. Perhaps a better name is **path**.) The **category** attribute specifies the id (or a sequence of ids from the root) of the parent page. For example, the following markup would create a second readme tool preference page, "Readme Example Child Page," as a child of the original page.

```
<extension
  point = "org.eclipse.ui.preferencePages">
  <page
    id="org.eclipse.ui.examples.readmetool.Page1"
    class="org.eclipse.ui.examples.readmetool.ReadmePreferencePage"
    name="Readme Example">
  </page>
  <page
    id="org.eclipse.ui.examples.readmetool.Page2"
    class="org.eclipse.ui.examples.readmetool.ReadmePreferencePage2"
    name="Readme Example Child Page"
    category="org.eclipse.ui.examples.readmetool.Page1">
  </page>
</extension>
```

Once the user selects the entry for a preference page in the tree on the left, the workbench will create and display a preference page using the **class** specified in the extension definition. This action is what activates the plug-in (if it wasn't already activated due to another user operation).

Preference Page

Defining the page

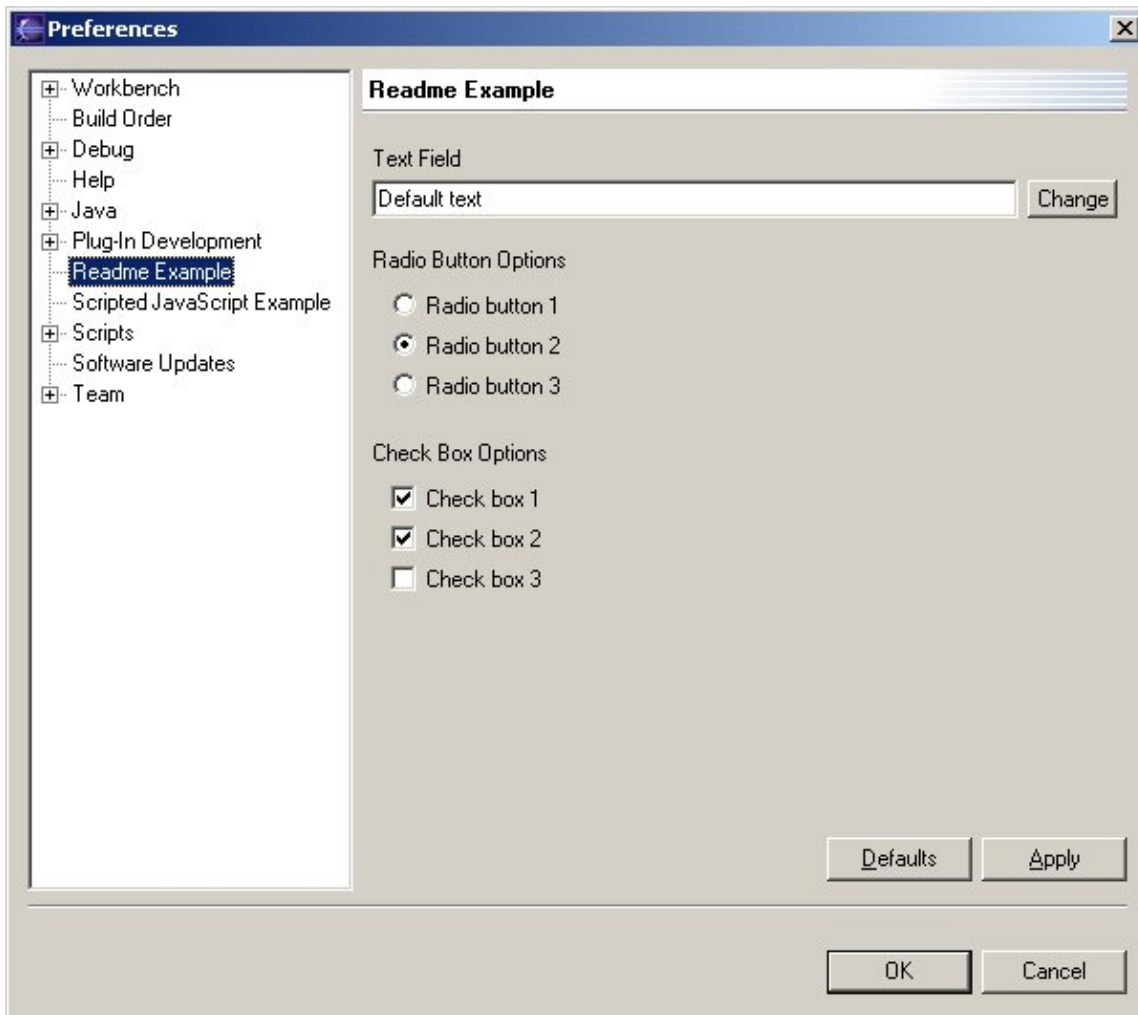
Implementing a preference page is similar to creating a page for a wizard. The preference page supplies a **createContents** method that creates the SWT controls representing the page content and adds listeners for any events of interest. The page is responsible for creating and returning the composite that will parent all of the controls in the page. The following snippet shows the highlights:

```
protected Control createContents(Composite parent) {
    ...
    //composite_textField << parent
    Composite composite_textField = createComposite(parent, 2);
    Label label_textField = createLabel(composite_textField, "Text Field");
    textField = createTextField(composite_textField);
    pushButton_textField = createPushButton(composite_textField, "Change");

    //composite_tab << parent
    Composite composite_tab = createComposite(parent, 2);
    Label label1 = createLabel(composite_tab, "Radio Button Options");

    tabForward(composite_tab);
    //radio button composite << tab composite
    Composite composite_radioButton = createComposite(composite_tab, 1);
    radioButton1 = createRadioButton(composite_radioButton, "Radio button 1");
    radioButton2 = createRadioButton(composite_radioButton, "Radio button 2");
    radioButton3 = createRadioButton(composite_radioButton, "Radio button 3");
    ...
    initializeValues();
    ...
    return new Composite(parent, SWT.NULL);
}
```

Most of the code in this method is concerned with creating and laying out the controls, so we won't dissect it here. Here is what the corresponding page looks like:



The other primary responsibility of a preference page is to react to the **performOk** message. Typically, this method updates and stores the user preferences and, if necessary, updates any other plug-in objects to reflect the change in preferences.

Preference pages should override the **doGetPreferenceStore()** method to return a preference store for storing their values.

Plug-in preference store

Preference stores are similar in nature to dialog settings. In [Dialog settings](#), we saw how the **AbstractUIPlugin** class maintains dialog settings during the lifetime of a plug-in. The same strategy is employed for user preferences. Your plug-in can add entries to a preference store and update the values as the user changes the settings in your preferences page. The platform will take care of saving these values in your plug-in's working directory and initializing the preference store from the saved settings.

The following code in the **ReadmePreferencePage** obtains the preference store for the **ReadmePlugin**.

```
protected IPreferenceStore doGetPreferenceStore() {
    return ReadmePlugin.getDefault().getPreferenceStore();
}
```

Because **ReadmePlugin** extends the **AbstractUIPlugin** class, it automatically inherits a preference store. This preference store is initialized from a preferences file stored in the plug-in's directory. The only thing

the **ReadmePlugin** has to do is implement a method that initializes the preferences to their default values. These values are used the first time the preference page is shown or when the user presses the **Defaults** button in the preferences page.

```
protected void initializeDefaultPreferences(IPreferenceStore store) {
    // These settings will show up when Preference dialog
    // opens up for the first time.
    store.setDefault(IReadmeConstants.PRE_CHECK1, true);
    store.setDefault(IReadmeConstants.PRE_CHECK2, true);
    store.setDefault(IReadmeConstants.PRE_CHECK3, false);
    store.setDefault(IReadmeConstants.PRE_RADIO_CHOICE, 2);
    store.setDefault(IReadmeConstants.PRE_TEXT, "Default text");
}
```

Note: If there are no preferences saved for a plug-in, the plug-in will get an empty preference store.

Retrieving and saving preferences

Once you've associated your plug-in's preference store with your preference page, you can implement the logic for retrieving and saving the preferences.

Preference pages are responsible for initializing the values of their controls using the preferences settings from the preference store. This process is similar to initializing dialog control values from dialog settings. The **ReadmePreferencePage** initializes all of its controls in a single method, **initializeValues**, which is called from its **createContents** method.

```
private void initializeValues() {
    IPReferenceStore store = getPreferenceStore();
    checkBox1.setSelection(store.getBoolean(IReadmeConstants.PRE_CHECK1));
    checkBox2.setSelection(store.getBoolean(IReadmeConstants.PRE_CHECK2));
    checkBox3.setSelection(store.getBoolean(IReadmeConstants.PRE_CHECK3));
    ...
}
```

When the **OK** (or **Apply**) button is pressed, the current values of the controls on the preference page should be stored back into the preference store. The **ReadmePreferencePage** implements this logic in a separate method, **storeValues**.

```
private void storeValues() {
    IPReferenceStore store = getPreferenceStore();
    store.setValue(IReadmeConstants.PRE_CHECK1, checkBox1.getSelection());
    store.setValue(IReadmeConstants.PRE_CHECK2, checkBox2.getSelection());
    store.setValue(IReadmeConstants.PRE_CHECK3, checkBox3.getSelection());
    ...
}
```

When the user presses the **Defaults** button, the platform will restore all preference store values to the default values specified in the plug-in class. However, your preference page is responsible for reflecting these default values in the controls on the preference page. The **ReadmePreferencePage** implements this in **initializeDefaults**.

```
private void initializeDefaults() {
    IPReferenceStore store = getPreferenceStore();
    checkBox1.setSelection(store.getDefaultBoolean(IReadmeConstants.PRE_CHECK1));
    checkBox2.setSelection(store.getDefaultBoolean(IReadmeConstants.PRE_CHECK2));
    checkBox3.setSelection(store.getDefaultBoolean(IReadmeConstants.PRE_CHECK3));
}
```

```
    ...  
}
```

Field editors

The implementation of a preference page is primarily SWT code. SWT code is used to create the preference page controls, set the values of the controls, and retrieve the values of the controls. The [org.eclipse.jface.preference](#) package provides helper classes, called **field editors**, that create the widgets and implement the value setting and retrieval code for the most common preference types. The platform provides field editors for displaying and updating many value types, including booleans, colors, strings, integers, fonts, and file names.

[FieldEditorPreferencePage](#) implements a page that uses these field editors to display and store the preference values on the page.

Instead of creating SWT controls to fill its contents, a [FieldEditorPreferencePage](#) creates field editors to display the contents.

```
public void createFieldEditors() {  
    // The first string is the preference key name  
    // The second string is the label shown next to the widget  
    addField(new BooleanFieldEditor(USE_OLD_MODE, "Use old mode",  
        getFieldEditorParent()));  
    addField(new StringFieldEditor(APPLICATION_NAME, "Application Name",  
        getFieldEditorParent()));  
    addField(new ColorFieldEditor(COLOR, "Text Color", getFieldEditorParent()));  
    ...  
}
```

Each field editor is assigned the name of its corresponding preference key and the text label for the SWT control that it will create. The kind of control created depends on the type of field editor. For example, a boolean field editor creates a checkbox.

Since the preference page is associated with a preference store (specified in the **doGetPreferenceStore** method), the code for storing the current values, for initializing the control values from the preference store, and for restoring the controls to their default values can all be implemented in the [FieldEditorPreferencePage](#).

The [FieldEditorPreferencePage](#) will use a grid layout with one column as the default layout for field editor widgets. For special layout requirements, you can override the **createContents** method.

Property pages

Property pages are very similar to preference pages. The primary difference is that property pages are associated with a particular resource, while preferences are associated with the plug-in itself.

org.eclipse.ui.propertyPages

You can contribute a property page for a resource by using the [org.eclipse.ui.propertyPages](#) extension point. A resource's property page is invoked using the **Properties** menu in the resource navigator view. This menu is available when a single resource is selected.

The readme tool contributes two property pages.

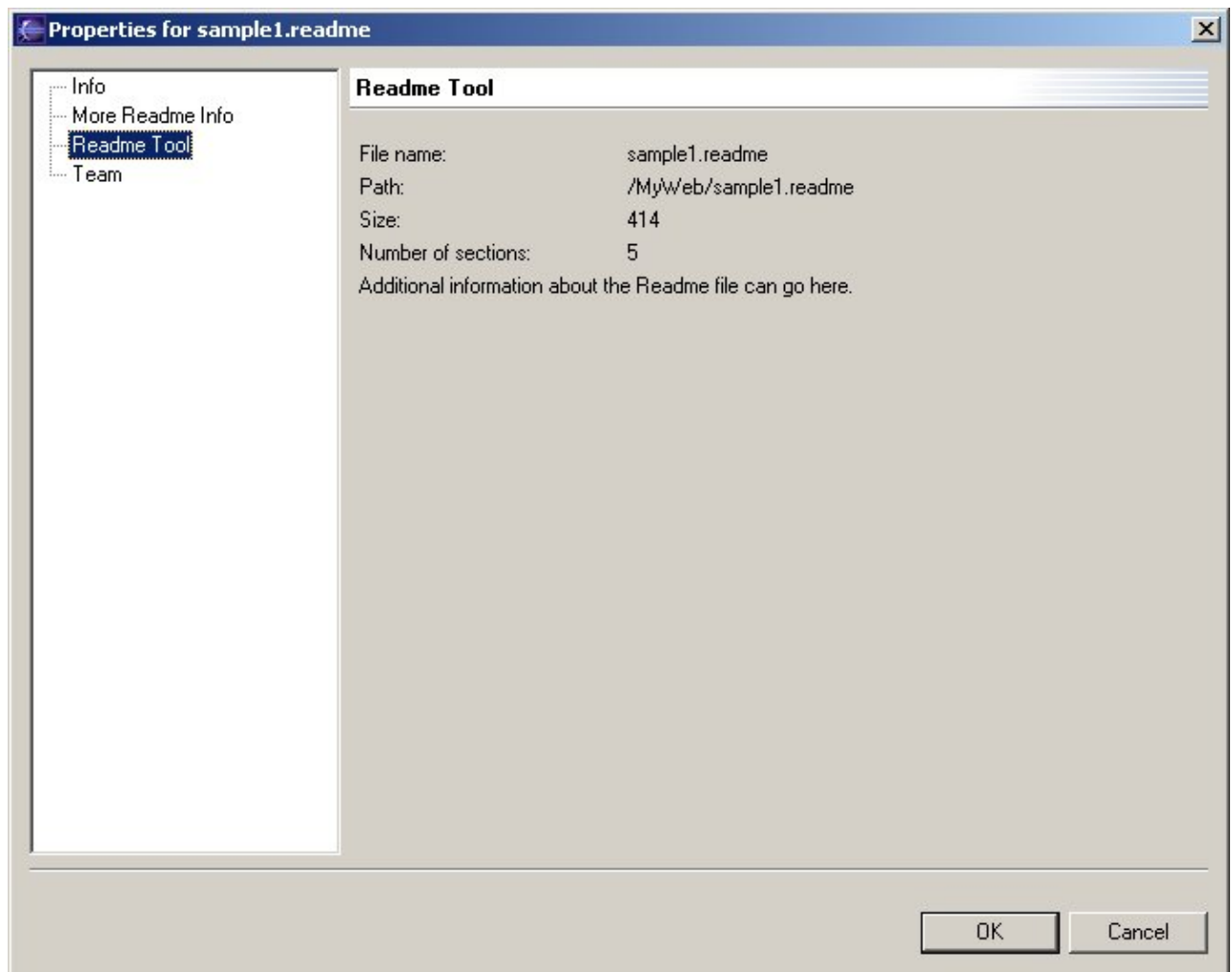
```

<extension
  point = "org.eclipse.ui.propertyPages">
  <page
    id="org.eclipse.ui.examples.readmetool.FilePage"
    name="Readme Tool"
    objectClass="org.eclipse.core.resources.IFile"
    class="org.eclipse.ui.examples.readmetool.ReadmeFilePropertyPage"
    nameFilter="*.readme">
  </page>
  <page
    id="org.eclipse.ui.examples.readmetool.FilePage2"
    name="More Readme Info"
    objectClass="org.eclipse.core.resources.IFile"
    class="org.eclipse.ui.examples.readmetool.ReadmeFilePropertyPage2"
    nameFilter="*.readme">
  </page>
</extension>

```

Both pages are contributed for objects of type **IFile** with a **.readme** file extension.

Property pages look a lot like preference pages, except there is no hierarchy or categorization of property pages. In the dialog below, both readme property pages appear in the main list of pages.



Properties page

When the workbench creates and launches a properties page, it sets the selected resource into the page. The page can use the `getElement()` method to obtain its element, an [IAdaptable](#).

The pattern for creating property pages is similar to that of preference pages, so we will only focus on what is different. Property pages show information about their element. This information can be obtained by accessing the element in order to query or compute the relevant information. The information can also be stored and retrieved from the resource's properties.

The **ReadmeFilePropertyPage** computes most of its information using its element. The following snippet shows how the number of sections is computed and displayed in a label.

```
...
IResource resource = (IResource) getElement();
...
IAdaptable sections = getSections(resource);
if (sections instanceof AdaptableList) {
    AdaptableList list = (AdaptableList)sections;
    label = createLabel(panel, String.valueOf(list.size()));
}
...
```

When a property is computed, there is no need for corresponding logic to save the value, since the user cannot update this value.

Properties pages are commonly used for viewing and for setting the application-specific properties of a resource. (See [Resource properties](#) for a discussion of session and persistent properties.) Since the property page knows its resource, the resources API can be used in the page to initialize control values or to set new property values based on user selections in the properties page.

The following snippet shows a checkbox value being initialized from a property on a property page's element.

```
private void initializeValues() {
    ...
    IResource resource = (IResource) getElement();
    label.setText(resource.getPersistentProperty("MyProperty"));
    ...
}
```

The corresponding code for saving the checkbox value back into the property looks like this:

```
private void storeValues() {
    ...
    IResource resource = (IResource) getElement();
    resource.setPersistentProperty("MyProperty", label.getText());
    ...
}
```


JFace: UI framework for plug-ins

The workbench defines extension points for plug-ins to contribute UI function to the platform. Many of these extension points, particularly wizard extensions, are implemented using classes in the **org.eclipse.jface.*** packages. What's the distinction?

JFace is a UI toolkit that provides helper classes for developing UI features that can be tedious to implement. JFace operates above the level of a raw widget system. It provides classes for handling common UI programming tasks:

- **Viewers** handle the drudgery of populating, sorting, filtering, and updating widgets.
- **Actions** and **contributions** introduce semantics for defining user actions and specifying where to make them available.
- **Image** and **font registries** provide common patterns for handling UI resources.
- **Dialogs** and **wizards** define a framework for building complex interactions with the user.

JFace frees you up to focus on the implementation of your specific plug-in's function, rather than focusing on the underlying widget system or solving problems that are common in almost any UI application.

JFace and the workbench

Where does JFace end and the workbench begin? Sometimes the lines aren't so obvious. In general, the JFace APIs (from the packages **org.eclipse.jface.***) are independent of the workbench extension points and APIs. Conceivably, a JFace program could be written without using any workbench code at all.

The workbench makes use of JFace but attempts to reduce dependencies where possible. For example, the workbench part model ([IWorkbenchPart](#)) is designed to be independent of JFace. We saw earlier that views and editors can be implemented using SWT widgets directly without using any JFace classes. The workbench attempts to remain "JFace neutral" wherever possible, allowing programmers to use the parts of JFace they find useful. In practice, the workbench uses JFace for much of its implementation and there are references to JFace types in API definitions. (For example, the JFace interfaces for [IMenuManager](#), [IToolBarManager](#), and [IStatusLineManager](#) show up as types in the workbench [IActionBar](#) methods.)

JFace and SWT

The lines between SWT and JFace are much cleaner. SWT does not depend on any JFace or platform code at all. Many of the SWT examples show how you can build a standalone application.

JFace is designed to provide common application UI function on top of the SWT library. JFace does not try to "hide" SWT or replace its function. It provides classes and interfaces that handle many of the common tasks associated with programming a dynamic UI using SWT.

The relationship between JFace and SWT is most clearly demonstrated by looking at viewers and their relationship to SWT widgets.

Viewers

Why would you ever want to use a viewer when we have already seen that workbench UI contributions like views, editors, wizards, and dialogs can be implemented directly with SWT widgets?

Viewers allow you to create widgets while still using your model objects. If you use an SWT widget directly, you have to convert your objects into the strings and images expected by SWT. Viewers act as adapters on SWT widgets, handling the common code for handling widget events that you would otherwise have to implement yourself.

We first saw a viewer in the readme tool's view contribution, inside the **ReadmeSectionsView**.

```
public void createPartControl(Composite parent) {
    viewer = new ListViewer(parent);
    ...
}
```

*Note: Viewers can be used to provide the implementation for both workbench views and editors. The term viewer does not imply that they are only useful for implementing views. For example, the **TextViewer** is used in the implementation in many of the workbench and plug-in editors.*

Standard viewers

JFace provides viewers for most of the non-trivial widgets in SWT. Viewers are most commonly used for list, tree, table, and text widgets.

Each viewer has an associated SWT widget. This widget can be created implicitly by supplying the parent **Composite** in a convenience viewer constructor, or explicitly by creating it first and supplying it to the viewer in its constructor.

List-oriented viewers

Lists, trees, and tables share many common capabilities from a user's point of view, such as population with objects, selection, sorting, and filtering.

These viewers keep a list of domain objects (called **elements**) and display them in their corresponding SWT widget. A list viewer knows how to get a text label from any element in the list. It obtains the label from an **ILabelProvider** which can be set on the viewer. List viewers know how to map from the widget callbacks back into the world of elements known by the viewer client.

Clients that use a plain SWT widget have to operate at the SWT level – where items are strings and events often relate to an index within the list of strings. Viewers provide higher level semantics. Clients are notified of selections and changes to the list using the elements they provided to the viewer. The viewer handles all the grunt work for mapping indexes back to elements, adjusting for a filtered view of the objects, and re-sorting when necessary.

Filtering and sorting capability is handled by designating a viewer sorter (**ViewerSorter**) and/or viewer filter (**ViewerFilter**) for the viewer. (These can be specified for tree and table viewers in addition to list viewers.) The client need only provide a class that can compare or filter the objects in the list. The viewer handles the details of populating the list according to the specified order and filter, and maintaining the order and filter as elements are added and removed.

Viewers are not intended to be extended by clients. To customize a viewer, you can configure it with your own content and label providers.

A **ListViewer** maps elements in a list to an SWT **List** control.

A [TreeViewer](#) displays hierarchical objects in an SWT [Tree](#) widget. It handles the details for expanding and collapsing items. There are several different kinds of tree viewers for different SWT tree controls (plain tree, table tree, checkbox tree).

A [TableViewer](#) is very similar to a list viewer, but adds the ability to view multiple columns of information for each element in the table. Table viewers significantly extend the function of the SWT table widget by introducing the concept of editing a cell. Special cell editors can be used to allow the user to edit a table cell using a combo box, dialog, or text widget. The table viewer handles the creation and placement of these widgets when needed for user editing. This is done using the [CellEditor](#) classes, such as [TextCellEditor](#) and [CheckboxCellEditor](#).

Text viewer

Text widgets have many common semantics such as double click behavior, undo, coloring, and navigating by index or line. A [TextViewer](#) is an adapter for an SWT [StyledText](#) widget. Text viewers provide a document model to the client and manage the conversion of the document to the styled text information provided by the text widget.

Text viewers are covered in more detail in [Workbench Editors](#).

Viewer architecture

To understand a viewer, you must become familiar with the relationship between a viewer's input element, its contents, its selection, and the information actually displayed in the widget that it is manipulating.

Input elements

An **input element** is the main object that the viewer is displaying (or editing). From the viewer's point of view, an input element can be any object at all. It does not assume any particular interface is implemented by the input element. (We'll see why in a moment when we look at content providers.)

A viewer must be able to handle a change of input element. If a new input element is set into a viewer, it must repopulate its widget according to the new element, and disassociate itself from the previous input element. The semantics for registering as a listener on an input element and populating the widget based on the element are different for each kind of viewer.

Content viewers

A **content viewer** is a viewer that has a well defined protocol for obtaining information from its input element. Content viewers use two specialized helper classes, the [IContentProvider](#) and [ILabelProvider](#), to populate their widget and display information about the input element.

[IContentProvider](#) provides basic lifecycle protocol for associating a content provider with an input element and handling a change of input element. More specialized content providers are implemented for different kinds of viewers. The most common content provider is [IStructuredContentProvider](#), which can provide a list of objects given an input element. It is used in list-like viewers, such as lists, tables, or trees. In general, the content provider knows how to map between the input element and the expected viewer content.

[ILabelProvider](#) goes a step further. Given the content of a viewer (derived from the input element and content provider), it can produce the specific UI elements, such as names and icons, that are needed to display the content in the viewer. Label providers can aid in saving icon resources since they can ensure the same instance of the icon is used for all like types in a viewer.

Note: Instances of particular content and label providers are not intended to be shared across multiple viewers. Even if all your viewers use the same type of content or label provider, each viewer should be initialized with its own instance of the provider class. The provider life cycle protocol is designed for a 1-to-1 relationship between a provider and its viewer.

Input elements, content providers, and label providers allow viewers to hide most of the implementation details for populating widgets. Clients of a viewer need only worry about populating a viewer with the right kind of input and content provider. The label provider must know how to derive the UI information from the viewer content.

Viewers and the workbench

The flexibility provided by viewers, content providers, and label providers can be demonstrated by looking at how the workbench uses them.

The [WorkbenchContentProvider](#) is a structured content provider that obtains contents from an input element by asking for its children. The concept of adapters is used again in order to implement generic function. When asked for the list of elements from its input element, the [WorkbenchContentProvider](#) obtains an [IWorkbenchAdapter](#) for the input element. If an [IWorkbenchAdapter](#) has been registered for the input element, then the content provider can safely assume that the element can be queried for its children. [WorkbenchContentProvider](#) also does the work needed to keep its viewer up to date when the workspace changes.

The [WorkbenchLabelProvider](#) is a label provider that obtains an [IWorkbenchAdapter](#) from an object in order to find its text and image. The concept of a label provider is particularly helpful for workbench objects because it allows a single label provider to cache images that are commonly used in a viewer. For example, once the [WorkbenchLabelProvider](#) obtains an image to use for an [IProject](#), it can cache that image and use it for all [IProject](#) objects shown in the viewer.

By defining a common adapter, [IWorkbenchAdapter](#), and registering it for many of the platform types, we make it possible for these types to be represented correctly in many of the common viewers and the workbench views that contain them.

Actions and contributions

The action classes allow you to define user commands independently from their presentation in the UI. This gives you the flexibility to change the presentation of an action in your plug-in without changing the code that actually performs the command once it has been chosen. The contribution classes are used to manage the actual UI items representing the commands. You don't program to the contribution classes, but you will see them in some of the workbench and JFace API.

Actions

An action ([IAction](#)) represents a command that can be triggered by the end user. Actions are typically associated with buttons, menu items, and items in tool bars.

Although actions do not place themselves in the UI, they do have UI oriented properties, such as tool tip text, label text, and an image. This allows other classes to construct widgets for the presentation of the action.

When the user triggers the action in the UI, the action's run method is invoked to do the actual work. A common pattern in the run method is to query the workbench selections and manipulate the objects that are

selected. Another common pattern is to launch a wizard or dialog when an action is chosen.

You should not directly implement the [IAction](#) interface. Instead, you should subclass the [Action](#) class. Browse the subclasses of this class to see many of the common patterns for actions. The code below implements the "About" action. It is one of the simpler actions in the workbench.

```
public void run() {  
    new AboutDialog(workbenchWindow.getShell()).open();  
}
```

Earlier we saw the workbench interfaces [IViewActionDelegate](#) and [IEditorActionDelegate](#). These interfaces are used when contributing view actions or editor actions to the workbench. The workbench action delegates are initialized with a reference to their associated view or editor. With this knowledge, they can navigate to the workbench page or window, accessing selections or any other information needed to perform the action.

You will implement your own action classes whenever you want to define a command in your plug-in. If you are contributing actions to other views and editors, you will implement action delegates.

Contribution items

A contribution item ([IContributionItem](#)) represents the UI portion of an action. More specifically, it represents an item that is contributed to a shared UI resource such as a menu or tool bar.

Contribution items know how to fill a specific SWT widget with the appropriate SWT item that represents the contribution.

You don't have to worry about creating a contribution item when you are contributing actions to the workbench UI. This is done on your behalf when the workbench creates UI items for the actions that you have defined.

Contribution managers

A contribution manager ([IContributionManager](#)) represents a collection of contribution items that will be presented in the UI. You can add and insert contribution items using named contribution ids to place the items in the appropriate order. You can also find items by id and remove individual items.

Each implementation of [IContributionManager](#) knows how to fill a specific SWT widget with its items. JFace provides contribution managers for menus ([IMenuManager](#)), tool bars ([IToolBarManager](#)), and status lines ([IStatusLineManager](#)).

As a plug-in developer, you do not need to implement these interfaces, but you will see references to some of these managers in API methods.

User interface resources

The [org.eclipse.jface.resource](#) package defines classes that help plug-ins manage UI resources such as fonts and icons.

Many of the workbench extension points allow plug-ins to supply icons that can be used to show their contributions in the workbench. Since GUI operating systems support a limited number of images or fonts in memory at once, a plug-in's UI resources must be carefully managed and sometimes shared between widgets.

We've already seen several references to icons in the readme tool plug-in. Some of its icons are specified in the **plugin.xml** markup.

```
<extension
  point="org.eclipse.ui.views">
  <category
    id="org.eclipse.ui.examples.readmetool"
    name="&Readme">
  </category>
  <view
    id="org.eclipse.ui.examples.readmetool.views.SectionsView"
    name="Readme Sections"
    icon="icons/basic/view16/sections.gif"
    category="org.eclipse.ui.examples.readmetool"
    class="org.eclipse.ui.examples.readmetool.ReadmeSectionsView">
  </view>
</extension>
```

We've also seen code that describes images on the fly. The following is from the readme tool's **ReadmeEditorActionBarContributor**.

```
public ReadmeEditorActionBarContributor() {
  ...
  action1 = new EditorAction("&Editor Action1");
  action1.setToolTipText("Readme Editor Action1");
  action1.setImageDescriptor(ReadmeImages.EDITOR_ACTION1_IMAGE);
  ...
}
```

JFace provides the basic support classes that allow plug-ins to manage their icons and fonts without worrying about when the corresponding platform graphics objects are created and destroyed. These support classes are used directly by plug-ins as shown above, or indirectly when the workbench uses these classes to obtain images that are described in extension point markup.

Image descriptors and the registry

The SWT [Image](#) class represents an image from the operating system's perspective. Because most GUI operating systems have a limit on the number of images that can be open at once, plug-ins should be very careful when creating them, and ensure that they also dispose of them properly when finished using them. By using the JFace [ImageDescriptor](#) and [ImageRegistry](#) classes instead of the SWT image, plug-ins can generally avoid creating, managing, and disposing these images directly.

Image descriptor

The [ImageDescriptor](#) class can be used as a lightweight description of an image. It specifies everything that is needed to create an image, such as the URL or filename where the image can be obtained.

[ImageDescriptors](#) do not allocate an actual platform image unless specifically requested using the **createImage()** method.

Image descriptors are the best strategy when your code is structured such that it defines all the icons in one place and allocates them as they are needed. Image descriptors can be created at any time without concern for OS resources, making it convenient to create them all in initialization code.

Image registry

The [ImageRegistry](#) class is used to keep a list of named images. Clients can add image descriptors or SWT images directly to the list. When an image is requested by name from the registry, the registry will return the image if it has been created, or create one from the descriptor. This allows clients of the registry to share images.

Images that are added to or retrieved from the registry must not be disposed by any client. The registry is responsible for disposing of the image since the images are shared by multiple clients. The registry will dispose of the images when the platform GUI system shuts down.

Plug-in patterns for using images

Specifying the image in the plugin.xml

Where possible, specify the icon for your plug-in's UI objects in the **plugin.xml** file. Many of the workbench extension points include configuration parameters for an icon file. By defining your icons in your extension contribution in the plugin.xml, you leave the image management strategy up to the platform. Since the icons are typically kept in your plug-in's directory, this allows you to specify the icons and manage the files all in one place.

The other patterns should only be considered when you can't specify the icon as part of your extension contribution.

Explicit creation

Explicitly creating an image is the best strategy when the image is infrequently used and not shared. The image can be created directly in SWT and disposed after it is used.

Images can also be created explicitly using an [ImageDescriptor](#) and invoking the **createImage()** method. As in the first case, the **dispose()** method for the image must be invoked after the image is no longer needed. For example, if a dialog creates an image when it is opened, it should dispose the image when it is closed.

Image registry

When an image is used frequently in a plug-in and shared across many different objects in the UI, it is useful to register the image descriptor with an [ImageRegistry](#). The images in the registry will be shared with any object that queries an image by the same name. You must not dispose any images in the registry since they are shared by other objects.

Adding an image to the image registry is the best strategy when the image is used frequently, perhaps through the lifetime of the plug-in, and is shared by many objects. The disadvantage of using the registry is that images in the registry are not disposed until the GUI system shuts down. Since there is a limit on the number of platform (SWT) images that can be open at one time, plug-ins should be careful not to register too many icons in a registry.

The class [AbstractUIPlugin](#) includes protocol for creating a plug-in wide image registry.

Label providers

When an icon is used frequently to display items in a particular viewer, it can be shared among similar items in the viewer using a label provider. Since a label provider is responsible for returning an image for any object

in a viewer, it can control the creation of the image and any sharing of images across objects in the viewer.

The label provider can use any of the previously discussed techniques to produce an image. If you browse the various implementations of `getImage()` in the [LabelProvider](#) subclasses, you will see a variety of approaches including caching a single icon for objects and maintaining a table of images by type. Images created by a label provider must be disposed in the provider's `dispose()` method, which is called when the viewer is disposed.

Using a label provider is a good compromise between explicit creation and the image registry. It promotes sharing of icons like the image registry, yet still maintains control over the creation and disposal of the actual image.

Plug-in wide image class

When fine-tuning a plug-in, it is common to experiment with all of these different image creation patterns. It can be useful to isolate the decision making regarding image creation in a separate class and instruct all clients to use the class to obtain all images. This way, the creation sequence can be tuned to reflect the actual performance characteristics of the plug-in.

Font registry

Fonts are another limited resource in platform operating systems. The creation and disposal issues are the same for fonts as for images, requiring similar speed/space tradeoffs. In general, fonts are allocated in SWT by requesting a font with a platform dependent font name.

The [FontRegistry](#) class keeps a table of fonts by their name. It manages the allocation and disposal of the font.

In general, plug-ins should avoid allocating any fonts or describing fonts with platform specific names. Although the font registry is used internally in JFace, it is typically not used by plug-ins. The [JFaceResources](#) class should be used to access common fonts.

It is very common to allow users to specify their preferences for the application's fonts in a preference page. In these cases, the [FontFieldEditor](#) should be used to obtain the font name from the user, and a [FontRegistry](#) may be used to keep the font. The [FontFieldEditor](#) is only used in preference pages.

JFaceResources

The class [JFaceResources](#) controls access to common platform fonts and images. It maintains an internal font and image registry so that clients can share named fonts and images.

There are many techniques used in the workbench and other plug-ins to share images where required. The [JFaceResources](#) image registry is not widely used across the workbench and plug-in code.

Use of fonts is much simpler. The workbench and most plug-ins use the [JFaceResources](#) class to request fonts by logical name. Methods such as `getDialogFont()` and `getDefaultFont()` are provided so that plug-ins can use the expected fonts in their UI.

Long-running operations

The [org.eclipse.jface.operations](#) package defines interfaces for long-running operations that require progress indicators or allow user cancellation of the operation.

Runnables and progress

The platform core runtime defines a common interface, [IProgressMonitor](#), which is used to report progress to the user while long running operations are in progress. The client can provide a monitor as a parameter in many platform API methods when it is important to show progress to the user.

JFace defines more specific interfaces for objects that implement the user interface for a progress monitor.

[IRunnableWithProgress](#) is the interface for a long-running operation. The **run** method for this interface has an [IProgressMonitor](#) parameter that is used to report progress and check for user cancellation.

[IRunnableContext](#) is the interface for the different places in the UI where progress can be reported. Classes that implement this interface may choose to use different techniques for showing progress and running the operation. For example, [ProgressMonitorDialog](#) implements this interface by showing a progress dialog. [IWorkbenchWindow](#) implements this interface by showing progress in the workbench window's status line. [WizardDialog](#) implements this interface to show long running operations inside the wizard status line.

Note: The workbench UI provides additional support for operations in [WorkspaceModifyOperation](#). This class simplifies the implementation of long-running operations that modify the workspace. It maps between [IRunnableWithProgress](#) and [IWorkspaceRunnable](#). See the javadoc for further detail.

Modal operations

The [ModalContext](#) class is provided to run an operation that is modal from the client code's perspective. It is used inside the different implementations of [IRunnableContext](#). If your plug-in needs to wait on the completion of a long-running operation before continuing execution, [ModalContext](#) can be used to accomplish this while still keeping the user interface responsive.

When you run an operation in a modal context, you can choose to fork the operation in a different thread. If **fork** is false, the operation will be run in the calling thread. If **fork** is true, the operation will be run in a new thread, the calling thread will be blocked, and the UI event loop will be run until the operation terminates.

For more information on the UI event loop, see [Threading issues for clients](#).

Standard Widget Toolkit

The Standard Widget Toolkit (SWT) is a widget toolkit for Java developers that provides a portable API and tight integration with the underlying native OS GUI platform.

Many low level UI programming tasks are handled in higher layers of the Eclipse platform. The **plugin.xml** markup for UI contributions specifies menu and toolbar content without requiring any SWT programming. JFace viewers and actions provide implementations for the common interactions between applications and widgets. However, knowledge of the underlying SWT architecture and design philosophy is important for understanding how the rest of the platform works.

Portability and platform integration

A common issue in widget toolkit design is the tension between portable toolkits and platform integration. The Java AWT (Advanced Widget Toolkit) provides platform integrated widgets for lower level widgets such as lists, text, and buttons, but does not provide access to higher level platform components such as trees or rich text. This forces application developers into a "least common denominator" situation: they can only use widgets that are available on all platforms.

The Swing toolkit attempts to address this problem by providing non-native implementations of high level widgets like trees, tables, and text. This provides a great deal of functionality, but applications developed in Swing stand out as being different. Platform look and feel emulation layers help the applications look more like the platform, but the user interaction is different enough to be noticed. This makes it difficult to use emulated toolkits to build applications that compete with shrink-wrapped applications developed specifically for a particular OS platform.

SWT addresses this problem by defining a common portable API that is provided on all supported platforms, and implementing the API on each platform using native widgets where possible. This allows the toolkit to immediately reflect any changes in the underlying OS GUI look and feel, while maintaining a consistent programming model on all platforms.

The "least common denominator" problem is solved by SWT in several ways.

- Features that are not available on all platforms but generally useful for the workbench and tooling plug-ins can be emulated on platforms that provide no native support. For example, the OSF/Motif 1.2 widget toolkit does not contain a tree widget. SWT provides an emulated tree widget on Motif 1.2 that is API compatible with the Windows native implementation.
- Features that are not available on all platforms but not widely used can be omitted from SWT. For example, Windows provides a widget that implements a calendar, but this is not provided in SWT.
- Features that are specific to a platform, such as ActiveX integration, are only provided on the relevant platform. Platform specific features are separated into separate packages that clearly denote the platform name in the package.

Consistency with the platform

Platform integration is not strictly a matter of look and feel. Tight integration includes the ability to interact with native desktop features such as drag and drop, integrate with OS desktop applications, and use components developed with OS component models like Win32 ActiveX.

 SWT ActiveX support is discussed in the article [ActiveX Support in SWT](#).

Consistency is also achieved in the code itself by providing an implementation that looks familiar to the native OS developer. Rather than hide OS differences in native C code or attempt to build portable and non-portable layers in the Java implementation, SWT provides separate and distinct implementations in Java for each platform.

One important implementation rule is that natives in C map one to one with calls to the OS. A Windows programmer will immediately recognize the implementation of the SWT toolkit on Windows, because it uses natives that directly map to the system calls used in C. None of the "platform magic" is hidden in C code. A platform developer can eyeball the code and know exactly which platform calls are executed by the toolkit. This greatly simplifies debugging. If a failure occurs when calling a native methods, calling the platform API with the same parameters from C code will exhibit the same failure. (A complete discussion of this issue can be found in [SWT Implementation Strategy for Java Natives.](#))

Widgets

SWT includes many rich features, but a basic knowledge of the system's core – **widgets, layouts, and events** – is all that is needed to implement useful and robust applications.

Widget application structure

When you are contributing UI elements using platform workbench extensions, the mechanics of starting up SWT are handled for you by the workbench.

If you are writing an SWT application from scratch (outside of the workbench), you must understand more about SWT's application structure.

A typical stand-alone SWT application has the following structure:

- Create a [Display](#) which represents an SWT session.
- Creates one or more [Shells](#) which serve as the main window(s) for the application.
- Create any other widgets needed inside the shell.
- Initialize the sizes and other necessary state for the widgets. Register listeners for widget events that need to be handled.
- Open the shell window.
- Run the event dispatching loop until an exit condition occurs (typically when the main shell window is closed by the user).
- Dispose the display.

The following code snippet is adapted from the `org.eclipse.swt.examples.helloworld.HelloWorld2` application. Since the application only displays the string "Hello World," it does not need to register for any widget events.

```
public static void main (String [] args) {
    Display display = new Display ();
    Shell shell = new Shell (display);
    Label label = new Label (shell, SWT.CENTER);
    label.setText ("Hello_world");
    label.setBounds (shell.getClientArea ());
    shell.open ();
    while (!shell.isDisposed ()) {
        if (!display.readAndDispatch ()) display.sleep ();
    }
    display.dispose ();
}
```

Display

The [Display](#) represents the connection between SWT and the underlying platform's GUI system. Displays are primarily used to manage the platform event loop and control communication between the UI thread and other threads. (See [Threading issues for clients](#) for a complete discussion of UI threading issues.)

For most applications, you can follow the pattern used above. You must create a display before creating any windows, and you must dispose of the display when your shell is closed. You don't need to think much more about the display unless you are designing a multi-threaded application.

Shell

A [Shell](#) is a "window" managed by the OS platform window manager. Top level shells are those that are created as a child of the display. These windows are the windows that users move, resize, minimize, and maximize while using the application. Secondary shells are those that are created as a child of another shell. These windows are typically used as dialog windows or other transient windows that only exist in the context of another window.

Parents and children

All widgets that are not top level shells have a parent. Top level shells do not have a parent, but they are all created in association with a particular [Display](#). You can access this display using `getDisplay()`. All other widgets are created as descendants (direct or indirect) of top level shells.

[Composite](#) widgets are widgets that can have children.

When you see an application window, you can think of it as a widget tree, or hierarchy, whose root is the shell. Depending on the complexity of the application, there may be a single child of the shell, several children, or nested layers of composites with children.

Widget life cycle

When your application creates a widget, SWT immediately creates the underlying platform widget. This eliminates the need for code that operates differently depending on whether the underlying OS widget exists. It also allows a majority of the widget's data to be kept in the platform layer rather than replicated in the toolkit. This means that the toolkit's concept of a widget lifecycle must conform to the rules of the underlying GUI system.

Widget creation

Most GUI platforms require you to specify a parent when you create a widget. Since SWT creates a platform widget as soon as you create a toolkit widget, the parent widget must be specified in the constructor for the widget.

Style bits

Some widget properties must be set by the OS at the time a widget is created and cannot be changed afterward. For example, a list may be single or multi-selection, and may or may not have scroll bars.

These properties, called **styles**, must be set in the constructor. All widget constructors take an **int** argument that specifies the bitwise **OR** of all desired styles. In some cases, a particular style is considered a hint, which means that it may not be available on all platforms, but will be gracefully ignored on platforms that do not

support it.

The style constants are located in the [SWT](#) class as public static fields. A list of applicable constants for each widget class is contained in the API Reference for [SWT](#).

Resource disposal

The OS platforms underneath SWT require explicit allocation and freeing of OS resources. In keeping with the SWT design philosophy of reflecting the platform application structure in the widget toolkit, SWT requires that you explicitly free any OS resources that you have allocated. In SWT, the **dispose()** method is used to free resources associated with a particular toolkit object.

The rule of thumb is that if you create the object, you must dispose of it. Here are some specific ground rules that further explain this philosophy:

- If you create a widget or graphic object using a constructor, you must dispose of it manually when you are finished using it.
- If you get a widget or graphic object without using a constructor, you must not dispose of it manually since you did not allocate it.
- If you pass a reference to your widget or graphic object to another object, you must take care not to dispose of it while it is still being used. (We saw this rule in practice earlier in [Plug-in patterns for using images](#).)
- When the user closes a [Shell](#), the shell and all of its child widgets are recursively disposed. In this case, you do not need to dispose of the widgets themselves. However, you must free any graphics resources allocated in conjunction with those widgets.
- If you create a graphic object for use during the lifetime of one of your widgets, you must dispose of the graphic object when the widget is disposed. This can be done by registering a dispose listener for your widget and freeing the graphic object when the **dispose** event is received.

There is one exception to these rules. Simple data objects, such as [Rectangle](#) and [Point](#), do not use operating system resources. They do not have a **dispose()** method and you do not have to free them. If in doubt, check the javadoc for a particular class.

Controls

So far, we've been using the term **widget** without a formal definition. In the SWT class hierarchy, a [Widget](#) is the abstract class for any UI object that can be placed inside another widget. A [Control](#) is a widget that typically has a counterpart representation (denoted by an OS window handle) in the underlying platform.

We tend to use the terms widget and control interchangeably. Although the distinction matters in the SWT implementation, we don't focus on this difference from an application's point of view. If you review the SWT widget hierarchy, you will see that a **Control** is something you can create and place anywhere you want in your widget parent/child tree. Widgets that are not controls are typically more specialized UI objects that can be created only for certain types of parents.

The [SWT API Reference](#) and examples are full of information about the different kinds of controls and their usage. The [org.eclipse.swt.widgets](#) package defines the core set of widgets in SWT. The following table summarizes the concrete types of controls provided in this package and their purpose. (Abstract classes are eliminated from this list.)

Widget	Purpose	Styles	Events
<u>Button</u>	Selectable control that issues notification when pressed and/or released.	BORDER, ARROW, CHECK, PUSH, RADIO, TOGGLE, FLAT, LEFT, RIGHT, CENTER	Dispose, Control*, Selection
<u>Canvas</u>	Composite control that provides a surface for drawing arbitrary graphics. Often used to implement custom controls.	BORDER, H_SCROLL, V_SCROLL, NO_BACKGROUND, NO_FOCUS, NO_MERGE_PAINTS, NO_REDRAW_RESIZE	Dispose, Control*
<u>Caret</u>	An i-beam that is typically used as the insertion point for text.		Dispose
<u>Combo</u>	Selectable control that allows the user to choose a string from a list of strings, or optionally type a new value into an editable text field. Often used when limited space requires a pop-down presentation of the available strings rather than using a single selection list box.	BORDER, DROP_DOWN, READ_ONLY, SIMPLE	Dispose, Control*, DefaultSelection, Modify, Selection
<u>Composite</u>	Control that is capable of containing other widgets.	BORDER, H_SCROLL, V_SCROLL	Dispose, Control*
<u>CoolBar</u>	Composite control that allows users to dynamically reposition the cool items contained in the bar.	BORDER	Dispose, Control*
<u>CoolItem</u>	Selectable user interface object that represents a dynamically positionable area of a cool bar.		Dispose
<u>Group</u>	Composite control that groups other widgets and surrounds them with an etched border and/or label.	BORDER, SHADOW_ETCHED_IN, SHADOW_ETCHED_OUT, SHADOW_IN, SHADOW_OUT, SHADOW_NONE	Dispose, Control*
<u>Label</u>	Non-selectable control that displays a string or an image.	BORDER, CENTER, LEFT, RIGHT, WRAP, SEPARATOR (with HORIZONTAL, SHADOW_IN, SHADOW_OUT, VERTICAL)	Dispose, Control*
<u>List</u>	Selectable control that allows the user to choose a string or strings from a list of strings.	BORDER, H_SCROLL, V_SCROLL, SINGLE, MULTI	Dispose, Control*, Selection, DefaultSelection
<u>Menu</u>	User interface object that contains menu items.	BAR, DROP_DOWN, POP_UP	Dispose, Help, Hide, Show
<u>MenuItem</u>	Selectable user interface object that represents an item in a menu.	CHECK, CASCADE, PUSH, RADIO, SEPARATOR	Dispose, Arm, Help, Selection
<u>ProgressBar</u>			Dispose, Control*

	Non-selectable control that displays progress to the user, typically in the form of a bar graph.	BORDER, SMOOTH, HORIZONTAL, VERTICAL	
Sash	Selectable control that allows the user to drag a rubber banded outline of the sash within the parent window. Used to allow users to resize child widgets by repositioning their dividing line.	BORDER, HORIZONTAL, VERTICAL	Dispose, Control*, Selection
Scale	Selectable control that represents a range of numeric values.	BORDER, HORIZONTAL, VERTICAL	Dispose, Control*, Selection
ScrollBar	Selectable control that represents a range of positive numeric values. Used in a Composite that has V_SCROLL and/or H_SCROLL styles.	HORIZONTAL, VERTICAL	Dispose, Selection
Shell	Window that is managed by the OS window manager. Shells can be parented by a Display (top level shells) or by another shell (secondary shells).	BORDER, H_SCROLL, V_SCROLL, CLOSE, MIN, MAX, NO_TRIM, RESIZE, TITLE (see also SHELL_TRIM, DIALOG_TRIM)	Dispose, Control*, Activate, Close, Deactivate, Deiconify, Iconify
Slider	Selectable control that represents a range of numeric values. A slider is distinguished from a scale by providing a draggable thumb that can adjust the current value along the range.	BORDER, HORIZONTAL, VERTICAL	Dispose, Control*, Selection
TabFolder	Composite control that groups pages that can be selected by the user using labeled tabs.	BORDER	Dispose, Control*, Selection
TabItem	Selectable user interface object corresponding to a tab for a page in a tab folder.		Dispose
Table	Selectable control that displays a list of table items that can be selected by the user. Items are presented in rows that display multiple columns representing different aspects of the items.	BORDER, H_SCROLL, V_SCROLL, SINGLE, MULTI, CHECK, FULL_SELECTION, HIDE_SELECTION	Dispose, Control*, Selection, DefaultSelection
TableColumn	Selectable user interface object that represents a column in a table.	LEFT, RIGHT, CENTER	Dispose, Move, Resize, Selection
TableItem	Selectable user interface object that represents an item in a table.		Dispose
Text	Editable control that allows the user to type text into it.	BORDER, H_SCROLL, V_SCROLL, MULTI, SINGLE, READ_ONLY, WRAP	Dispose, Control*, DefaultSelection, Modify, Verify

ToolBar	Composite control that supports the layout of selectable tool bar items.	BORDER, FLAT, WRAP, RIGHT, HORIZONTAL, VERTICAL	Dispose, Control*,
ToolItem	Selectable user interface object that represents an item in a tool bar.	PUSH, CHECK, RADIO, SEPARATOR, DROP_DOWN	Dispose, Selection
Tracker	User interface object that implements rubber banding rectangles.		Dispose, Move
Tree	Selectable control that displays a hierarchical list of tree items that can be selected by the user.	BORDER, H_SCROLL, V_SCROLL, SINGLE, MULTI, CHECK	Dispose, Control*, Selection, DefaultSelection, Collapse, Expand
TreeItem	Selectable user interface object that represents a hierarchy of tree items in a tree.		Dispose

Control* = Events inherited from Control: FocusIn, FocusOut, Help, KeyDown, KeyUp, MouseDoubleClick, MouseDown, MouseEnter, MouseExit, MouseHover, MouseUp, MouseMove, Move, Paint, Resize

Events

We've seen how to create a display and some widgets and run our application's message loop. Where does the real work happen? It happens every time an event is read from the queue and dispatched to a widget. Most of the application logic is implemented as responses to user events.

The basic pattern is that you add a listener to some widget that you have created. When the appropriate event occurs, the listener code is executed. This simple example is adapted from **org.eclipse.swt.examples.helloworld.HelloWorld3**.

```
Display display = new Display ();
Shell shell = new Shell (display);
Label label = new Label (shell, SWT.CENTER);
...
shell.addControlListener(new ControlAdapter() {
    public void controlResized(ControlEvent e) {
        label.setBounds (shell.getClientArea ());
    }
});
```

For each type of listener, there is an interface that defines the listener (**XYZListener**), a class that provides event information (**XYZEvent**), and an API method to add the listener (**addXYZListener**). If there is more than one method defined in the listener interface, an adapter (**XYZAdapter**) that implements the listener interface and provides empty methods is provided. All of the events, listeners, and adapters are defined in the package [org.eclipse.swt.events](#).

The following table summarizes the events that are available and the widgets that support each event.

Event Type	Description	Widgets
Arm	Generated when a widget, such as a menu item, is armed.	MenuItem

Control	Generated when a control is moved or resized.	Control, TableColumn, Tracker
Dispose	Generated when a widget is disposed, either programmatically or by the user.	Widget
Focus	Generated when a control gains or loses focus.	Control
Help	Generated when the user requests help for a widget, such as pressing the F1 key.	Control, Menu, MenuItem
Key	Generated when the user presses or releases a keyboard key when the control has keyboard focus.	Control
Menu	Generated when a menu is hidden or shown.	Menu
Modify	Generated when a widget's text is modified.	CCombo, Combo, Text, StyledText
Mouse	Generated when the user presses, releases, or double clicks the mouse over the control.	Control
MouseMove	Generated as the user moves the mouse across the control.	Control
MouseTrack	Generated when the mouse enters, exits, or hovers over the control.	Control
Paint	Generated when the control needs to be repainted.	Control
Selection	Generated when the user selects an item in the control.	Button, CCombo, Combo, CTabFolder, List, MenuItem, Sash, Scale, ScrollBar, Slider, StyledText, TabFolder, Table, TableColumn, TableTree, Text, ToolItem, Tree
Shell	Generated when a shell is minimized, maximized, activated, deactivated, or closed.	Shell
Traverse	Generated when a control is traversed by the user using keystrokes.	Control
Tree	Generated when the user expands or collapses items in the tree.	Tree, TableTree
Verify	Generated when a widget's text is about to be modified. Gives the application a chance to alter the text or prevent the modification.	Text, StyledText

Untyped events

The typed event system described above is implemented with a low level, untyped widget event mechanism. This mechanism is not intended to be used by applications, but you will see it used inside of the SWT implementation. It is also used in many of the workbench wizard page implementations.

The untyped mechanism relies on a constant to identify the type of event and defines a generic listener that is supplied with this constant. This allows the listener to implement a "case style" listener. In the following snippet, we define a generic event handler and add several listeners to a shell.

```
Shell shell = new Shell();
Listener listener = new Listener() {
    public void handleEvent(Event e) {
        switch (e.type) {
            case SWT.Resize:
                System.out.println("Resize received.");
                break;
            case SWT.Paint:
                System.out.println("Paint received.");
                break;
            default:
                System.out.println("Unknown event received");
        }
    }
};
shell.addListener(SWT.Resize, listener);
shell.addListener(SWT.Paint, listener);
```

Custom Widgets

Occasionally, you may find that none of the controls provided in SWT meet the need of your application. In these cases, you may want to extend SWT by implementing your own custom widget. SWT itself provides a package, org.eclipse.swt.custom, which contains custom controls that are not in the core set of SWT controls but are needed to implement the platform workbench.

Control	Purpose	Styles
AnimatedProgress	Static control that shows animation during the running of a long operation.	BORDER, HORIZONTAL, VERTICAL
CCombo	Similar to Combo, but custom drawn to allow for using a combo without a border. This class was developed for using combos inside table cells.	BORDER, FLAT, READ_ONLY
CLabel	Similar to Label, but supports clipping of text with ellipsis. Also supports a gradient effect for the background color as seen in the active workbench view. Does not support wrapping.	CENTER, LEFT, RIGHT, SHADOW_IN, SHADOW_OUT, SHADOW_NONE
CTabFolder	Similar to TabFolder, but supports additional configuration of the visual appearance of tabs (top or bottom) and borders.	BORDER, BOTTOM, TOP
SashForm	Composite control that lays out its	BORDER, HORIZONTAL,

	children in a row or column arrangement and uses a Sash to separate them so that the user can resize them.	VERTICAL
<u>ScrolledComposite</u>	Composite control that scrolls its contents and optionally stretches its contents to fill the available space.	BORDER, H_SCROLL, V_SCROLL
<u>StyledText</u>	Editable control that allows the user to type text. Ranges of text inside the control can have distinct fonts and colors (foreground and background.)	BORDER, FULL_SELECTION, MULTI, SINGLE, READ_ONLY
<u>TableTree</u>	Selectable control that displays a hierarchical list of items that can be selected by the user. Items are presented in rows that display multiple columns representing different aspects of the items.	BORDER, SINGLE, MULTI, CHECK, FULL_SELECTION
<u>ViewForm</u>	Composite control that lays out three children horizontally and allows programmatic control of layout and border parameters. Used in the workbench to implement a view's label/toolbar/menu local bar.	BORDER, FLAT

Before implementing a custom widget, you should consider several important issues:

- Can an existing SWT control be used as a simpler or plainer version of your widget? If so, consider using the existing control and finishing your application. You can always write a custom widget later that is API compatible with the simpler control.
- Does the widget have to be portable? If so, does a native widget exist on all platforms? If you are comfortable writing native platform code, you can provide a native implementation for each platform that your application supports. If it does not exist on any (or all) platforms, you will also have to provide a portable implementation.
- Can the function be achieved by extending the behavior of an existing control? If so, consider using an existing control and wrapping it with additional behavior.
- Will your widget contain other controls from the user's point of view? It's useful to think about this issue in advance because it influences your implementation and API.

Once you've determined that you need a custom widget and have decided which platforms must be supported, you can consider several implementation techniques for your widget. These techniques can be mixed and matched depending on what is available in the underlying OS platform.

Native implementation

If your application requires a native widget that is not provided by SWT, you will need to implement it natively. This may be a platform widget, a third party widget, or any other widget in a platform shared library.

Each SWT platform is shipped with both a shared library (for example, a DLL on Windows) and a JAR (for the Java class files). The shared library contains all of the native function required for SWT, but it was not meant to be a complete set of the functions available on the platform. To expose native function or native widgets that were not exposed by SWT, you need to write your own shared library. If you are using a combination of native code on one platform and portable code on another, make sure you call your shared

library on the platform with the native widget, and your jar on the platform with the portable widget.

To implement a native widget, you must understand the Java Native Interface (JNI), the API of the widget in the shared library, and the underlying OS platform APIs in C.

The basic process for implementation is to decide which part of the API of the native widget will be exposed in the Java API and writing the Java code that calls the natives to implement the behavior. JNI C code must be written to call the shared library.

It is a good idea to follow the design principles used to implement SWT when building your own native widget implementation. For example, your JNI natives should map one to one with the API calls being made into the shared library.

A complete example of a native custom widget implementation can be found in [Creating Your Own Widgets using SWT.](#)

Extending an existing widget

If your new widget is similar in concept or implementation to an existing widget, you may want to wrap an existing SWT widget. This technique is used for the implementation of [TableTree](#).

To wrap a widget, you create a subclass of the [Composite](#) or [Canvas](#) widget (depending on whether your control will have children). In the constructor for the custom widget, create the wrapped widget. The resulting widget will be 100% Java portable since you are calling the wrapped widget's API for your implementation.

Wrapping a widget is often a simpler way to implement custom widgets than starting from scratch. However, you must be careful in designing the API of your new widget. Here are some important tips:

Consider whether your widget is a "kind of" wrapped widget or whether it just uses one for its implementation. For example, a table tree is not a kind of table. It doesn't refer to items by row number index. The [TableTree](#) just uses a table to implement the presentation and adds tree behavior. If you are wrapping a widget purely for implementation reasons, then your API may not look similar to the underlying widget's API.

Forward as few methods and events as possible. Don't reimplement the entire API of the wrapped widget or you'll be constantly playing catch-up when the wrapped API changes in a future release. Methods that are common to most widgets, such as **setFont**, **setForeground**, **setBackground**, should be forwarded.

If you find yourself implementing most of the wrapped widget's API, consider exposing the wrapped widget at the API level and letting the application code use the wrapped widget directly. In this case, you may want to reconsider whether providing a new widget makes sense at all. It may be better to implement your feature as an "adapter" which adds behavior to a widget but does not pretend to be a widget. (JFace viewers follow this pattern.)

Note: This discussion has focused solely on extending the behavior of a widget by wrapping it. Extending a widget by subclassing it is highly discouraged, since it will make your widget dependent on the implementation of the superclass.

Custom drawn implementation

In some cases, you don't have any native code or existing widgets that help you in the implementation of your new widget. This means you must draw the widget yourself using SWT graphics calls. Although this technique can become quite complicated, it has the advantage of producing a completely portable

implementation.

Custom drawn controls are implemented by subclassing the [Canvas](#) or [Composite](#) class using these rules:

- Subclass [Canvas](#) if your widget will not have any children. This means you do not intend to allow applications to create children of your widget, and that you do not intend to create any children in order to implement your widget. [Canvas](#) is used for both simple controls, such as stylized labels, and for more complex controls, such as the styled text editor. In both cases, the widget is implemented completely internally using graphics calls and no children are added by applications.
- Subclass [Composite](#) if your widget will have children from the application's perspective, or if you will be creating children to implement your widget. [Composite](#) is used when you are combining widgets to create a new widget, such as using a text and list to implement a combo control. It is also used when you implement a widget without children that allows clients to add children. This is the case with the [ViewForm](#) control. The [SashForm](#) widget represents both cases: it uses widgets (sashes) internally for its implementation and allows clients to add their own children.

In a custom drawn control, your internal state is kept in Java instance variables. You define your API and styles according to the requirements of your widget.

The internal implementation of a custom drawn widget usually involves these major tasks:

- Create any graphics objects needed in your constructor and store them in an instance variable. Register a listener for the **dispose** event on your canvas or composite so that you can free these objects when the widget is destroyed.
- Add a **paintListener** to your canvas or composite and paint the widget according to your design. For complex widgets, a lot of work goes into optimizing this process by calculating and repainting only what's absolutely necessary.
- Ensure that any API calls that affect the appearance of your widget trigger a repaint of the widget. In general, you should use **redraw** to damage your widget when you know you must repaint, rather than call your internal painting code directly. This gives the platform a chance to collapse the paint you want to generate with any other pending paints and helps streamline your code by funneling all painting through one place.
- If your widget defines events in its API, determine what low level [Canvas](#) or [Composite](#) events will trigger your widget's events. For example, if you have a clicked event, you will want to register a mouse event on your canvas and perform calculations (such as hit testing) to determine whether the mouse event in your canvas should trigger your widget event.

Many of the widgets implemented in the [org.eclipse.swt.custom](#) use this approach. A simple example can be found in [CLabel](#).

Further information on custom widgets can be found in [Creating Your Own Widgets using SWT](#).

Layouts

We've seen some simple examples that show how to size or position child widgets based on the size of the parent. So far, this kind of computation has occurred in response to a resize listener. This is often the best way to handle simple widget positioning. However, there are common patterns used by applications when placing widgets. These patterns can be structured as configurable layout algorithms that can be reused by many different applications.

SWT defines **layouts** that provide general purpose positioning and sizing of child widgets in a composite. Layouts are subclasses of the abstract class [Layout](#). The SWT standard layouts can be found in the

[org.eclipse.swt.layout](#) package.

Widget layout concepts

You should understand some general definitions when resizing and positioning widgets.

- The **location** of a widget is its x,y coordinate location within its parent widget.
- The **preferred size** of a widget is the minimum size needed to show its content. This is computed differently for each kind of widget. In the case of a composite, the preferred size is the minimum size that contains the composite and all of its children at their preferred size.
- The **clientArea** is the size of a widget's content area.
- The **trim** is the distance between a widget's client Area and its actual border. Trim is occupied by the widget's borders or extra space at the edge of a widget. The size and appearance of the trim is widget and platform dependent.

These concepts are relevant for applications regardless of whether a layout is used. You can think of a layout as a convenient way to package resize functionality for reuse.

Some additional concepts are introduced by layouts.

- Some layouts support **spacing** between widgets in the layout.
- Some layouts support a **margin** between the edge of the layout and the widget adjacent to the edge.

See [Understanding Layouts in SWT](#) for further discussion and pictures demonstrating these concepts.

The following code snippet shows the simple case of an application using a resize callback to size a label to the size of its parent shell.

```
Display display = new Display ();
Shell shell = new Shell (display);
Label label = new Label (shell, SWT.CENTER);
shell.addControlListener(new ControlAdapter() {
    public void controlResized(ControlEvent e) {
        label.setBounds (shell.getClientArea ());
    }
});
```

The next snippet uses a layout to achieve the same effect:

```
Display display = new Display ();
Shell shell = new Shell (display);
Label label = new Label (shell, SWT.CENTER);
shell.setLayout(new FillLayout());
```

Even for this simple example, using a layout reduces the application code. For more complex layouts, the simplification is much greater.

SWT provides three default layout classes that can be used for many situations.

Fill Layout

[FillLayout](#) is the simplest layout class. It lays out widgets in a single row or column, forcing them to be the same size. Initially, the widgets will all be as tall as the tallest widget, and as wide as the widest. [FillLayout](#) does not wrap, and you cannot specify margins or spacing.

You might use a [FillLayout](#) to lay out buttons in a task bar or tool bar, or to stack checkboxes in a [Group](#). [FillLayout](#) can also be used when a [Composite](#) only has one child. In the example above, the [FillLayout](#) causes the label to completely fill its parent shell.

RowLayout

[RowLayout](#) also lays out widgets in rows, but is more flexible than [FillLayout](#). It can wrap the widgets, creating as many rows as needed to display them. It also provides configurable margins on each edge of the layout, and configurable spacing between widgets in the layout. You can pack a [RowLayout](#), which will force all widgets to be the same size. If you justify a [RowLayout](#), extra space remaining in the [Composite](#) will be allocated as margins between the widgets.

The height and width of each widget in a [RowLayout](#) can be specified in a [RowData](#) object which should be set in the widget using `setLayoutData`.

Grid Layout

[GridLayout](#) is the most powerful (and most complex) layout. [GridLayout](#) lays out widgets in a grid, providing many configurable parameters that control the behavior of the grid rows and columns when the composite is resized.

The [GridLayout](#) defines API that controls the overall strategy of the layout. The most important attribute is `numColumns`, which determines the horizontal size of the grid. Typically you decide on this value when you first design your window's appearance. The order of the widgets in the grid is the same as the order in which you create them. To change the order of the widgets in the grid, you can use the [Control](#) methods `moveAbove(Control)` and `moveBelow(Control)`. These methods allow widgets to be inserted before or after each other in the layout. (The "above" and "below" refer to the widget Z ordering, not the location in the grid itself.)

The following table summarizes the configurable parameters for a [GridLayout](#).

Attribute	Description
<code>horizontalSpacing</code>	Number of pixels between the right edge of one cell and the left edge of its neighboring cell.
<code>makeColumnsEqualWidth</code>	Specifies whether all columns should be forced to the same width.
<code>marginWidth</code>	Number of pixels used for margin on the right and left edge of the grid.
<code>marginHeight</code>	Number of pixels used for margin on the top and bottom edge of the grid.
<code>numColumns</code>	Number of columns that should be used to make the grid.
<code>verticalSpacing</code>	Number of pixels between the bottom edge of one cell and the top edge of its neighboring cell.

[GridLayout](#) supports many other layout parameters for each widget in the grid. These properties are specified in a [GridData](#) object. You must set a [GridData](#) as the `layoutData` for each widget in the grid.

The [GridData](#) class defines style constants that let you specify commonly used combinations of layout parameters in the [GridData](#) constructor. You can also set these attributes individually using the public methods in [GridData](#).

You can achieve some highly dynamic and complex layouts using the [GridData](#). Tweaking these values for different widgets in the grid can produce many different combinations of layouts. Even the most complex

dialog layouts can be specified using the [GridData](#). The ability to allow widgets to span across cells produces many layouts that don't even look like a grid.

The following table summarizes the configurable parameters for [GridData](#). See [Understanding Layouts in SWT](#) for further description and example screen captures using the various grid parameters.

Attribute	Description
grabExcessHorizontalSpace	Specifies whether a cell should grow to use any extra horizontal space available in the grid. After the cell sizes in the grid are calculated based on the widgets and their grid data, any extra space remaining in the Composite will be allocated to those cells that grab excess space.
grabExcessVerticalSpace	Specifies whether a cell should grow to use any extra vertical space available in the grid.
heightHint	Specifies a minimum height for the widget (and therefore for the row that contains it).
horizontalAlignment	Can be one of BEGINNING , CENTER , END , FILL . FILL means that the widget will be sized to take up the entire width of its grid cell.
horizontalIndent	Number of pixels between the widget and the left edge of its grid cell.
horizontalSpan	Specifies the number of columns in the grid that the widget should span. By default, a widget takes up one cell in the grid. It can take additional cells horizontally by increasing this value.
verticalAlignment	Can be one of BEGINNING , CENTER , END , FILL . FILL means that the widget will be sized to take up the entire height of its grid cell.
verticalSpan	Specifies the number of rows in the grid that the widget should span. By default, a widget takes up one cell in the grid. It can take additional cells vertically by increasing this value.
widthHint	Specifies a minimum width for the widget (and therefore the column that contains it).

Custom layouts

Occasionally, you may need to write your own custom [Layout](#) class. This is most appropriate when you have a complex layout that is used in many different places in your application. It may be appropriate when you can optimize layout using application specific knowledge. Before building a custom layout, consider the following:

- Can the layout be achieved with an existing layout such as [GridLayout](#) or by nesting several different layouts?
- Can the layout be isolated into a common resize listener?
- Are you defining a general layout algorithm with programmer supplied configuration parameters or just positioning widgets specifically for your application?

Unless you are writing a very generic layout that will be used by several [Composite](#) widgets, it is often simpler and easier to calculate sizes and position children in a resize listener. Many of the SWT custom widgets were written this way. Although a new widget can be implemented as a **Composite/Layout** pair, implementing it as a [Composite](#) that does its layout in a resize listener and computes its preferred size in **computeSize** is clearer, and does not involve writing an extra class.

If you still believe you need a custom layout class, it is a good idea to first implement the layout algorithm in a resize listener. This makes for simpler debugging of the algorithm itself. Be sure to test the various cases for layout: resizing smaller, larger, wrapping, and clipping. Once you have the algorithm working, the code can be refactored into a subclass of [Layout](#).

Layouts are responsible for implementing two methods:

- **computeSize** calculates the width and height of a rectangle that encloses all of the composite's children once they have been sized and placed according to the layout algorithm. The hint parameters allow the width and/or height to be constrained. For example, a layout may choose to grow in one dimension if constrained in another.
- **layout** positions and sizes the composite's children. A layout can choose to cache layout-related information, such as the preferred extent of each of the children. The **flushCache** parameter tells the [Layout](#) to flush cached data, which is necessary when other factors besides the size of the composite have changed (such as the creation or removal of children, or a change in the widget's font, etc.)

Further discussion of custom layouts can be found in [Understanding Layouts in SWT](#).

Threading issues for clients

When working with a widget toolkit, it's important to understand the underlying thread model used for reading and dispatching platform GUI events. The implementation of the UI thread affects the rules that applications must follow when using Java threads in their code.

Native event dispatching

Underneath any GUI application, regardless of language or UI toolkit, the OS platform is detecting GUI events and placing them in application event queues. Although the mechanics are slightly different on different OS platforms, the basics are similar. As the user clicks the mouse, types characters, or surfaces windows, the OS generates application GUI events, such as mouse clicks, keystrokes, or window paint events. It determines which window (and application) should receive the event and places it in the application's event queue.

The underlying structure for any windowed GUI application is an event loop. Applications initialize and then start a loop which simply reads the GUI events from the queue and reacts appropriately. Any work done while handling one of these events must happen quickly in order to keep the GUI system responsive to the user.

Long operations triggered by UI events should be performed in a separate thread in order to allow the event loop thread to return quickly and fetch the next event from the application's queue. However, access to the widgets and platform API from other threads must be controlled with explicit locking and serialization. An application that fails to follow the rules can cause an OS call to fail, or worse, lock up the entire GUI system.

Toolkit UI threads

Native GUI programmers using C are quite familiar with the design considerations for working with the platform event loop. However, many higher level widget toolkits in Java often attempt to shield application developers from UI threading issues by hiding the platform event loop.

A common way to achieve this is to set up a dedicated toolkit UI thread for reading and dispatching from the event loop, and posting the events to an internal queue that is serviced by applications running in separate threads. This allows the toolkit to respond in sufficient time to the operating system, while not placing any restrictions on the application's timing in handling the event. Applications must still use special locking

techniques to access UI code from their application thread, but it is done consistently throughout the code since all application code is running in a non–UI thread.

Although it sounds tempting to "protect" applications from UI threading issues, it causes many problems in practice.

It becomes difficult to debug and diagnose problems when the timing of GUI events is dependent on the Java threading implementation and application performance.

Modern GUI platforms perform many optimizations with the event queue. A common optimization is to collapse multiple paint events in the queue. Every time part of a window must be repainted, the queue can be checked for overlapping or redundant paint events that have not been dispatched yet. These events can be merged into one paint event, causing less flicker and less frequent execution of the application's paint code. This optimization is defeated if the widget toolkit is pulling the events off the queue quickly and posting them to an internal queue.

Changing the developer's perception of the threading model causes confusion for programmers with experience programming the native GUI system in other languages and toolkits.

SWT UI thread

SWT follows the threading model supported directly by the platforms. The application program runs the event loop in its main thread and dispatches events directly from this thread. This is the application's "UI thread."

*Note: Technically, the UI thread is the thread that creates the **Display**. In practice, this is also the thread that runs the event loop and creates the widgets.*

Since all event code is triggered from the application's UI thread, application code that handles events can freely access the widgets and make graphics calls without any special techniques. However, the application is responsible for forking computational threads when performing long operations in response to an event.

*Note: SWT will trigger an **SWTException** for any calls made from a non–UI thread that must be made from the UI thread.*

The main thread, including the event loop, for an SWT application looks like this:

```
public static void main(String [] args) {
    Display display = new Display();
    Shell shell = new Shell(display);
    shell.open();
    // start the event loop. We stop when the user has done
    // something to dispose our window.
    while (!shell.isDisposed()) {
        if (!display.readAndDispatch())
            display.sleep();
    }
    display.dispose();
}
```

Once the widgets are created and the shell is opened, the application reads and dispatches events from the OS queue until the shell window is disposed. If there are no events available for us in the queue, we tell the display to sleep to give other applications a chance to run.

Note: The most common threading model for an SWT application is to run a single UI

thread and perform long operations in computational threads. However, SWT does not restrict developers to this model. An application could run multiple UI-threads, with a separate event loop in each thread.

SWT provides special access methods for calling widget and graphics code from a background thread.

Executing code from a non-UI thread

Applications that wish to call UI code from a non-UI thread must provide a **Runnable** that calls the UI code. The methods **syncExec(Runnable)** and **asyncExec(Runnable)** in the [Display](#) class are used to execute these runnables in the UI thread at an appropriate time.

- **syncExec(Runnable)** should be used when the application code in the non-UI thread depends on the return value from the UI code or otherwise needs to ensure that the runnable is run to completion before returning to the thread. SWT will block the calling thread until the runnable has been run from the application's UI thread. For example, a background thread that is computing something based on a window's current size would want to synchronously run the code to get the window's size and then continue with its computations.
- **asyncExec(Runnable)** should be used when the application needs to perform some UI operations, but does not depend on the operations being complete before continuing. For example, a background thread that updates a progress indicator or redraws a window could request the update asynchronously and continue with its processing. In this case, there is no guaranteed relationship between the timing of the background thread and the execution of the runnable.

The following code snippet demonstrates the pattern for using these methods.

```
// do time intensive computations
...
// now update the UI. We don't depend on the result,
// so use async.
Display.getCurrent().asyncExec(new Runnable() {
    public void run() {
        myWindow.redraw();
    }
});
// now do more computations
...
```

The workbench and threads

The threading rules are very clear when you are implementing an SWT application from the ground up, because you control the creation of the event loop and the decision to fork computational threads in your application.

What are the rules if you are contributing plug-in code to the workbench? Fortunately, there is no threading "magic" hidden in the JFace or workbench code. The rules are straightforward.

- Your workbench plug-in code executes in the workbench's UI thread.
- If you receive an event from the workbench, it is always executing in the UI thread of the workbench.
- If your plug-in forks a computational thread, it must use the [Display](#) **asyncExec** or **syncExec** methods when calling any API for the workbench, JFace, or SWT.
- Workbench and JFace API calls do not check that the caller is executing in the UI thread. However, SWT triggers an [SWTException](#) for all API calls made from a non-UI thread.

- If your plug-in uses the JFace [IRunnableContext](#) interface to invoke a progress monitor and run an operation, it supplies an argument to specify whether a computational thread is forked for running the operation.

Error handling

SWT has a well defined strategy for triggering errors and exceptions. Where possible, exceptions are triggered consistently across platforms. However, some errors are specific to an SWT implementation on a particular platform.

SWT can trigger three types of exceptions: **IllegalArgumentException**, [SWTException](#), and [SWTError](#). Applications shouldn't have to catch any other kind of exception or error when calling SWT.

Note: If any other exception besides these three is thrown from SWT, it should be considered a bug in the SWT implementation.

IllegalArgumentException

The arguments passed in SWT API methods are checked for appropriate state and range before any other work is done. An **IllegalArgumentException** will be thrown when it is determined that an argument is invalid.

SWT throws this exception consistently across all platforms. Code that causes an **IllegalArgumentException** on one platform will cause the same exception on a different platform.

SWTException

[SWTException](#) is thrown when a recoverable error occurs internally in SWT. The error code and message text provide a further description of the problem.

SWT throws this exception consistently across all platforms. On all platforms, SWT remains in a known stable state after throwing the exception. For example, this exception is thrown when an SWT call is made from a non-UI thread.

SWTError

[SWTError](#) is thrown when an unrecoverable error occurs inside SWT.

SWT will throw this error when an underlying platform call fails, leaving SWT in an unknown state, or when SWT is known to have an unrecoverable error, such as running out of platform graphics resources.

Once an SWT error has occurred, there is little that an application can do to correct the problem. These errors should not be encountered during normal course of operation in an application, but high reliability applications should still catch and report the errors.

Graphics

SWT provides a robust graphics engine for drawing graphics and displaying images in widgets. You can get pretty far without ever programming to the graphics interface, since widgets handle the painting of icons, text, and other data for you. If your application displays custom graphics, or if you are implementing a custom drawn widget, you will need to understand some basic drawing objects in SWT.

Graphics context

The graphics context, [GC](#), is the focal point for SWT graphics support. Its API describes all of the drawing capabilities in SWT.

A [GC](#) can be used for drawing on a control (the most common case) or for drawing on an image, display, or printer. When drawing on a control, you use the [GC](#) supplied to you in the control's paint event. When drawing on an image, display, or printer, you must create a [GC](#) configured for it (and dispose of it when you are finished using it).

Once you've got a [GC](#), you can set its attributes, such as color, line width, and font, which control the appearance of the graphics drawn in the [GC](#).

The API Reference for [GC](#) describes the complete set of graphics functions.

Fonts

The [Font](#) and [FontData](#) classes are used when manipulating fonts in SWT.

[FontData](#) describes the characteristics of a font. You can create a [FontData](#) by specifying a font name, style, and size. [FontData](#) includes API for querying these attributes. Since [FontData](#) does not allocate any OS resources, you do not need to dispose of it.

The [Font](#) is the actual graphic object representing a font that is used in the drawing API. You create a [Font](#) for a [Display](#) by specifying the [Display](#) and the [FontData](#) of the font that you want. You can also query a [Font](#) for its [FontData](#).

You must dispose of a [Font](#) when you are finished using it.

Colors

Colors are similar to fonts. You create a [Color](#) for a [Display](#) by specifying the RGB values for the desired color. You must dispose of a color when you are finished using it.

The [Display](#) method `getSystemColor` allows you to query the predefined system colors for the OS platform. You should not free colors obtained using this technique.

The color model is discussed in detail in the article, [SWT Color Model](#).

Images

The [Image](#), [ImageData](#), and [ImageLoader](#) classes are used when manipulating Images in SWT.

[ImageData](#) describes the actual pixels in the image, using the [PaletteData](#) class to describe the color values used in the image. [ImageData](#) is a device and platform independent description for an image.

[ImageLoader](#) loads and saves [ImageData](#) in different file formats. SWT currently supports loading and saving of **BMP** (Windows Bitmap), **ICO** (Windows Icon), **JPEG**, **GIF**, and **PNG** image formats.

The [Image](#) is the actual graphic object representing the image that is used in the drawing API. You create an image for a particular [Display](#). Images can be created in several ways:

- use an [ImageData](#) to initialize the image's contents
- copy an existing [Image](#)
- load an [Image](#) from a file.

No matter how you create the [Image](#), you are responsible for disposing it.

Graphics object lifecycle

Most of the graphics objects used for drawing in SWT allocate resources in the underlying OS and must be explicitly freed. The same rule of thumb discussed earlier applies here. If you create it using a constructor, you should free it. If you get access to it from somewhere else, do not free it.

Creation

Graphics objects such as graphics contexts, fonts, colors, and images are allocated in the OS as soon as the object is created. How you plan to use your graphics objects determines when you should create them.

For graphics objects used heavily throughout the application, you can create them at the time that you create your widgets. This is commonly done for colors and fonts. In other cases, it is more appropriate to create your graphics objects on the fly. For example, you might create a graphics context in one of your widget event handlers to perform some calculations.

If you are implementing a custom widget, you typically allocate graphics objects in the constructor if you always use them. You might allocate them on the fly if you don't always use them or if they are dependent on the state of some attribute.

Painting

Once you've allocated your graphics objects, you are ready to paint. **Always do your painting inside a paint listener.** There are rare cases (particularly when implementing custom widgets) when you paint while responding to some other event. This is generally discouraged. If you think you need to paint while handling some other event, you should first try using the **redraw** method, which will generate another paint event in the OS. Drawing outside of the paint method defeats platform optimizations and can even cause bugs depending on the number of pending paints in the event queue.

When you receive a paint event, you will be supplied with a [GC](#) pre-configured for drawing in the widget. **Do not free this GC!** You did not create it.

Any other graphics objects must be allocated while handling the event (or beforehand). Below is a snippet based on the [org.eclipse.swt.examples.HelloWorld5](#) sample. The color red was previously allocated when creating the widget, so it can be used here.

```
shell.addPaintListener(new PaintListener () {
    public void paintControl(PaintEvent event){
        GC gc = event.gc;
        gc.setForeground(red);
        Rectangle rect = event.widget.getClientArea();
        gc.drawRectangle(rect.x + 10, rect.y + 10, rect.width - 20, rect.height - 20);
        gc.drawString(resHello.getString("Hello_world"), rect.x + 20, rect.y + 20);
    }
});
```

Disposal

Every graphics object that you allocate must be freed when you are finished using it.

The timing of the disposal depends on when you created the object. If you create a graphics object while creating your widget, you should generally add a dispose listener onto the widget and dispose of the graphics when the widget is disposed. If you create an object on the fly while painting, you should dispose of it when finished painting.

The next code snippet shows a slightly modified version of our paint listener. In this example, it allocates and frees the color red while painting.

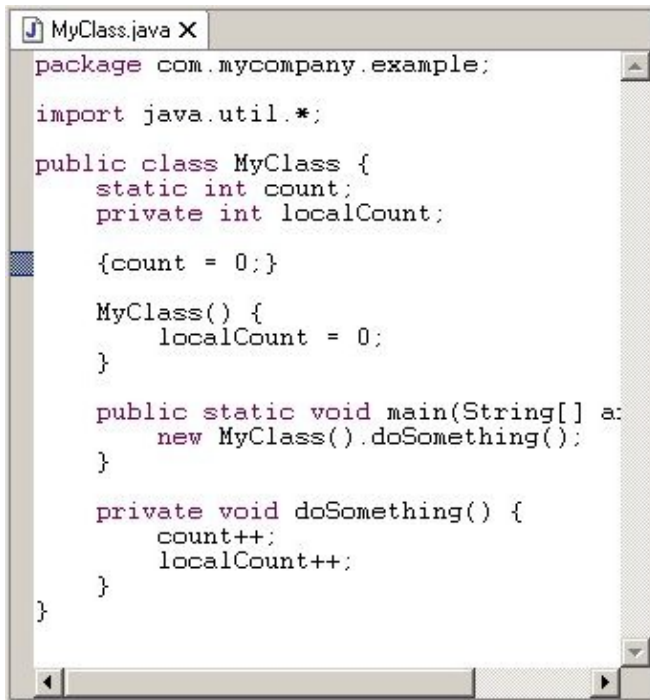
```
shell.addPaintListener(new PaintListener () {
    public void paintControl(PaintEvent event){
        GC gc = event.gc;
        Color red = new Color(event.widget.getDisplay(), 0xFF, 0, 0);
        gc.setForeground(red);
        Rectangle rect = event.widget.getClientArea();
        gc.drawRectangle(rect.x + 10, rect.y + 10, rect.width - 20, rect.height - 20);
        gc.drawString(resHello.getString("Hello_world"), rect.x + 20, rect.y + 20);
        red.dispose();
    }
});
```


Editors

We have seen how plug-ins can contribute an editor to the workbench, but we haven't yet looked at the implementation of an editor.

There is no "typical" implementation pattern for an editor, because editors usually provide application-specific semantics. For example, a tool that edits and manages a particular resource type will provide customized behavior for manipulating the data represented by the resource.

Editors can come in all shapes and sizes. If a plug-in's editor is text-based, then the editor can either use the existing default text editor, or create a customized text editor by using the facilities provided in JFace. The latter approach is used by the Java development tooling (JDT) editors.



```
MyClass.java X
package com.mycompany.example;

import java.util.*;

public class MyClass {
    static int count;
    private int localCount;

    {count = 0;}

    MyClass() {
        localCount = 0;
    }

    public static void main(String[] a:
        new MyClass().doSomething();
    }

    private void doSomething() {
        count++;
        localCount++;
    }
}
```

If a plug-in's editor is not text based, then a custom editor must be implemented by the plug-in. There are several approaches for building custom editors, all of which depend on the look and behavior of the editor.

- Form-based editors can layout controls in a fashion similar to a dialog or wizard. The Plug-in Development Environment (PDE) uses this approach in building its manifest editors.
- Graphics intensive editors can be written using SWT level code. For example, an editor could create its own SWT window for displaying the information, or it could use a custom SWT control that is optimized for the application.
- List-oriented editors can use JFace list, tree, and table viewers to manipulate their data.

Once the implementation model for the editor has been determined, programming the editor behavior is much like programming a stand-alone JFace or SWT application.

Workbench editors

Although the implementation of a workbench editor will be specific to your plug-in and the resources that you want to edit, the workbench provides a general structure for building an editor. The following concepts

apply to all workbench editors.

An editor must implement [IEditorPart](#) and is often built by extending the [EditorPart](#) class. An editor implements its user interface in the `createPartControl` method. This method is used to assemble the SWT widgets or JFace viewers that present the editor contents.

An **editor input** is a description of something to be edited. You can think of an editor input as a file name, though it is more general. [IEditorInput](#) defines the protocol for an editor input, including the name of the input and the image that should be used to represent it in the labels at the top of the editor.



Two generic editor inputs are provided in the platform. [IFileEditorInput](#) represents an input that is a file in the file system. [IStorageEditorInput](#) represents an input that is a stream of bytes. These bytes may come from sources other than the file system.

The rest of your editor's implementation depends on the content that you are trying to present. We'll look next at the most common type of editor – the text editor.

Text editors and JFace text

The workbench package [org.eclipse.ui.editors.text](#) implements the default text editor for the platform. It uses the text editor framework in [org.eclipse.ui.texteditor](#) for its implementation.

The text editor framework provides a domain–model independent editor that supports the following features:

- Standard text editing operations such as cut/copy/paste, find/replace
- Visual presentation of resource markers adjacent to the text editing area
- Automatic update of resource markers as the user edits text
- Context menu management
- Responses to user actions in the workbench, such as refreshing resources from the file system, closing projects, or removal of the editor's input element resource

[ITextEditor](#) is defined as a text specific extension of [IEditorPart](#). The default implementation of this interface is provided by [AbstractTextEditor](#).

[IDocumentProvider](#) is used to establish the link between a domain model and an [ITextEditor](#). The document provider manages the text presentation of the domain model and can be shared between multiple editors.

The workbench text editing framework is built on top of JFace text. The Java editor example in [org.eclipse.ui.examples.javaeditor](#) is a good place to start learning about the text editor framework and JFace text. It shows how complex features like text coloring, hover help, and automatic indenting can be implemented.

JFace text

The package [org.eclipse.jface.text](#) and its sub–packages support the implementation of robust text editors such as the workbench text editor and the JDT Java editor.

The following roadmap gives an overview of the support in JFace text.

- [org.eclipse.jface.text](#) defines a generic **document** model for text and provides a viewer that displays text using this model. Documents can be divided into non-overlapping **partitions**, which can be useful when the text represents multiple elements with different meanings (such as methods and comments inside a Java file). Partitions have **content types** which are used to identify places where different behavior should be assigned for different kinds of content. Document **positions** can be used to define text regions that remain updated as the user edits text. [IDocument](#) and [TextViewer](#) are good places to begin learning about this package.
- [org.eclipse.jface.text.formatter](#) defines a text viewer add-on which can be configured with different **formatting** behavior per partition content type. Formatting is achieved by manipulating white spaces and delimiters in order to present the text in a structured fashion. Formatting is most commonly used when editing code and is often driven by user preference. The JDT source code editor uses this support to provide user-driven Java code formatting.
- [org.eclipse.jface.text.contentassist](#) defines a text viewer add-on that provides user-driven **text completion** support. Popup windows are used to propose possible text choices to complete a phrase. The user can select these choices for automatic insertion in the text. Content assist also supports contextual popups for providing the user with information that is related to the current position in the document. [IContentAssistant](#) is a good place to begin learning about this package. It can be configured with different phrase completion strategies for different partition content types.
- [org.eclipse.jface.text.presentation](#) defines a text viewer add-on which can control the **visual presentation** (font, font style, colors) of the text shown in the text viewer. For each change applied to a document, the presentation reconciler determines which region of the visual presentation should be invalidated and how to repair it. Different strategies can be used for different partition content types.
- [org.eclipse.jface.text.reconciler](#) defines a text viewer add-on that supports the **synchronization** of a document with some external structure that may also be manipulating the text.
- [org.eclipse.jface.text.rules](#) provides **rule-based document scanning**. Plug-ins can use rules to distinguish tokens such as line delimiters, white space, and generic patterns when scanning a document. This package also provides support for rule-driven presentation reconciling and document partitioning. The Java editor example uses this package to parse Java code.
- [org.eclipse.jface.text.source](#) defines a **source viewer**. A source viewer extends a text viewer in order to support visual text annotations. These annotations are used in the JDT source code editor to annotate Java source code with problem descriptions and breakpoints.

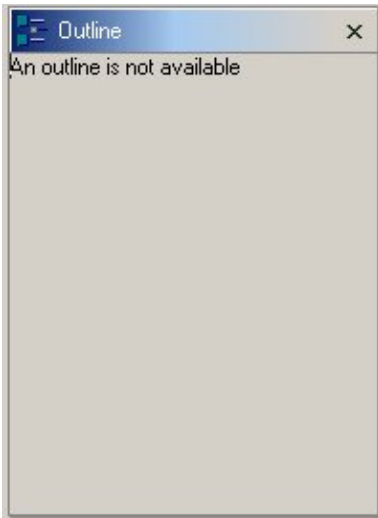
The SWT [StyledText](#) widget is used by the JFace text support.

Content outliners

Editors often have corresponding **content outliners** that provide a structured view of the editor contents and assist the user in navigating through the contents of the editor.

The workbench provides a standard **Outline** view for this purpose. The workbench user controls when this view is visible using the **Perspective->Show View** menu.

Since the generic [TextEditor](#) doesn't know anything about the structure of its text, it cannot provide behavior for an interesting outline view. Therefore, the default **Outline** view, shown below, doesn't do much.

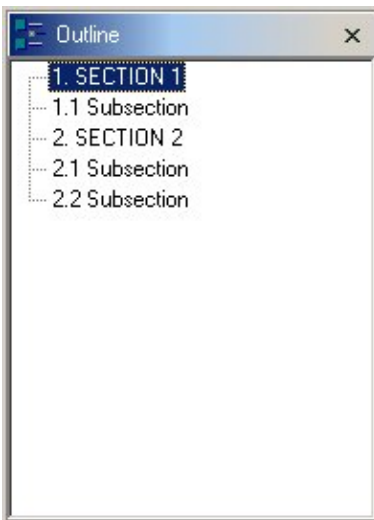


Plug-ins can extend [TextEditor](#) for the sole purpose of adding a custom content outliner page to the outline view. This approach is used in the workbench readme tool example. The **ReadmeEditor** overrides a few methods in [TextEditor](#) to supply its own outliner.

The outliner for an editor is specified when the workbench requests an adapter of type [IContentOutlinePage](#).

```
public Object getAdapter(Class key) {
    if (key.equals(IContentOutlinePage.class)) {
        IEditorInput input = getEditorInput();
        if (input instanceof IFileEditorInput) {
            page = new ReadmeContentOutlinePage(
                ((IFileEditorInput)input).getFile());
            return page;
        }
    }
    return super.getAdapter(key);
}
```

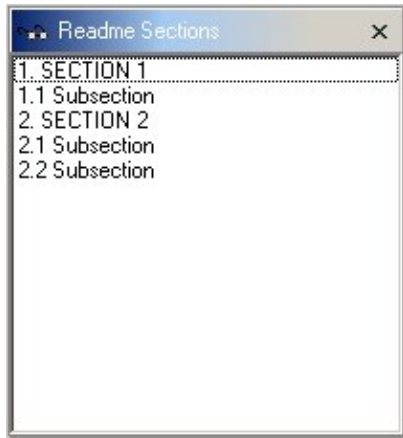
When a **ReadmeEditor** is opened (on a **.readme** file), the corresponding readme outliner is displayed (if the workbench user is showing the **Outline** view.)



A content outliner page must implement [IContentOutlinePage](#). This interface combines the ability to notify selection change listeners ([ISelectionProvider](#)) with the behavior of being a page in a view ([IPage](#)).

Content outliners are typically implemented using JFace viewers. The default implementation of a content outliner ([ContentOutlinePage](#)) uses a JFace tree viewer to display a hierarchical representation of the outline. This representation is suitable for many structured outliners, including **ReadmeContentOutlinePage**.

The **ReadmeContentOutlinePage** is similar in appearance to the readme sections view, shown below, that we saw when we implemented the [readme views extension](#).



In fact, the only real difference is that the outliner displays a hierarchical view of the sections, while the readme sections view displays a flat list of the sections. It should be no great surprise that the implementation of the outliner is very similar to that of the view. The only difference is that the outliner uses a tree viewer instead of a list viewer.

When the outline page was created by the editor, it was passed the editor's input element in the constructor. This input can often be passed directly to the outline page's viewer, as is done below.

```
public void createControl(Composite parent) {  
    ...  
    TreeViewer viewer = getTreeViewer();  
    viewer.setContentProvider(new WorkbenchContentProvider());  
    viewer.setLabelProvider(new WorkbenchLabelProvider());  
    viewer.setInput(getContentOutline(input));  
    ...  
}
```

The tree viewer creation is inherited from [ContentOutlinePage](#). The same content and label providers that were used in the readme sections view are used here, and the outline for the content is constructed using the same **ReadmeModelFactory** that constructed the sections for the view.

```
private IAdaptable getContentOutline(IAdaptable input) {  
    return ReadmeModelFactory.getInstance().getContentOutline(input);  
}
```

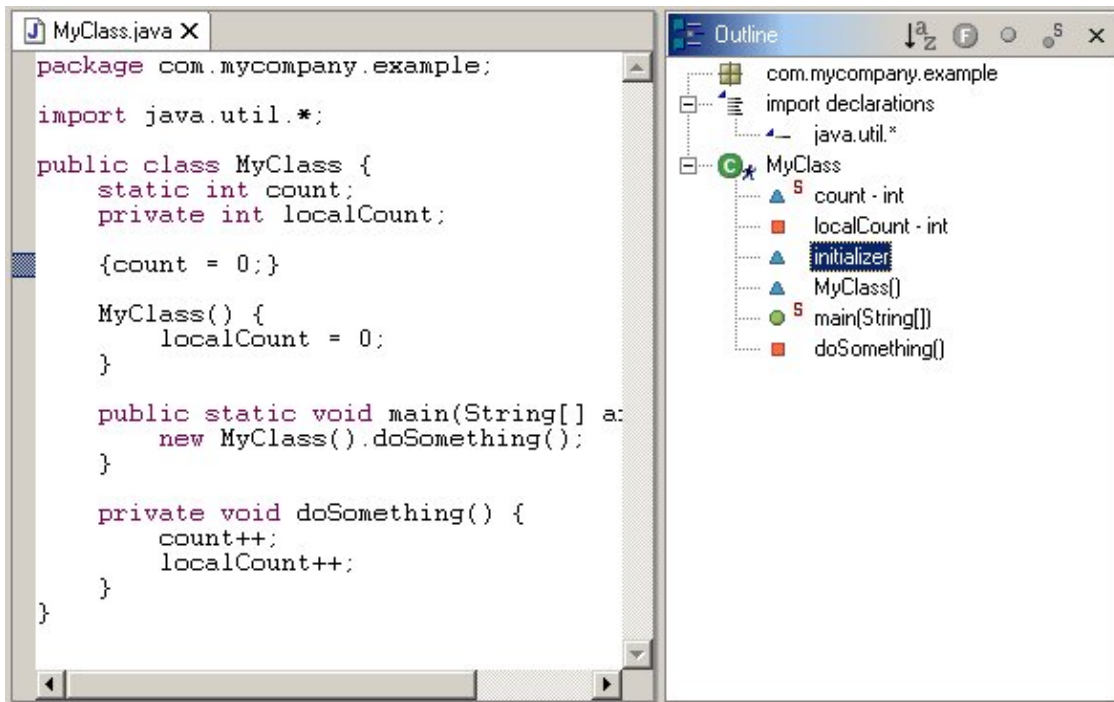
That's all there is to it!

Of course, the outliner itself doesn't provide any interesting behavior. Selecting the sections does not navigate the text in the editor! What good is this content outliner anyway? You could argue that the content outliner doesn't provide any behavior (besides the hierarchical representation) that we didn't already see in the sections view. Couldn't we have just used a tree viewer in the sections view instead of providing an outliner that doesn't do anything?

It's just an example! Actually, the **ReadmeContentOutlinePage** is provided to demonstrate how a content outliner can be customized for a text editor. It is not a good example of a content outliner itself. Users expect content outliners to assist them in the navigation of the editor content, so it would be a better decision to use the sections view if the only purpose is to display the structure of the content.

Where can you find more interesting content outliners? Look at the subclasses of [ContentOutlinePage](#) and their corresponding editors. The more typical pattern is that an editor supplies an outline page and registers selection events on it. As items are selected in the content outline, the editor updates itself appropriately.

The Java source code editor (provided in the JDT) demonstrates an interesting content outliner. The Java outliner presents a structured view of Java source code and allows the user to navigate through declarations, methods, and fields in the corresponding editor. As the outliner reports selection events, the Java editor updates its vertical ruler to show where the elements in the outliner are located in the source code.



Resource and workspace API

In [Introduction to resources](#), we saw how resources are represented in the platform and how they map to files in the file system. Now that we have built several plug-ins for the platform, we look again at the resources API and how you can use it for additional functionality in your plug-in.

The resources API allows you to tag resources with special information, find out about changes to resources, implement specialized builders that perform transformations on resources, and save plug-in specific information along with a resource.

Resource markers

We know that plug-ins can define specialized file extensions and contribute editors that provide specialized editing features for these file types. During the course of editing (or building) a resource, a plug-in may need to tag resources to communicate problems or other information to the user. The resource marker mechanism is used to manage this kind of information.

A **marker** is like a yellow sticky note stuck to a resource. On the marker you can record information about a problem (e.g., location, severity) or a task to be done. Or you can simply record a location for a marker as a bookmark.

Users can quickly jump to the marked location within a resource. The workbench UI supports presentation of bookmarks, breakpoints, tasks, and problems along the side of the editor. These markers can also be shown as items in views, such as the tasks or bookmarks view.

The platform resources API defines methods for creating markers, setting marker values, and extending the platform with new marker types. While the platform manages markers, it is the plug-ins that control their creation, removal and attribute values.

Markers are intended to be small, lightweight objects. There could be hundreds, even thousands of markers in a single project. For example, the Java compiler uses a marker to flag each problem it finds in source code.

The platform will throw away markers attached to resources that are deleted, but plug-ins are responsible for removing their stale markers when they no longer apply to a resource that still exists.

Marker operations

Manipulating a marker is similar to manipulating a resource. Markers are **handle** objects. You can obtain a marker handle from a resource, but you don't know if it actually exists until you use **exists()** protocol or otherwise try to manipulate it. Once you've established that a marker exists, you can find out the resource to which it belongs, query its **id**, or query named attributes that may have been assigned to it.

Markers are owned and managed by the platform, which takes care of making markers persistent and notifying listeners as markers are added, deleted, or changed. Plug-ins are responsible for creating any necessary markers, changing their attributes, and removing them when they are no longer needed.

Marker creation

Markers are not directly created using a constructor. They are created using a factory method (**IResource.createMarker()**) on the associated resource.

```
IMarker marker = file.createMarker(IMarker.TASK);
```

To create a marker that has global scope (not associated with any specific resource), you can use the workspace root (**IWorkspace.getRoot()**) as the resource.

Marker deletion

The code for deleting a marker is straightforward.

```
try {
    marker.delete();
} catch (CoreException e) {
    // Something went wrong
}
```

When a marker is deleted, its marker object (handle) becomes "stale." Plug-ins should use the **IMarker.exists()** protocol to make sure a marker object is still valid.

Markers can be deleted in batch by asking a resource to delete its markers. This method is useful when removing many markers at once or if individual marker references or ids are not available.

```
int depth = IResource.DEPTH_INFINITE;
try {
    resource.deleteMarkers(IMarker.PROBLEM, true, depth);
} catch (CoreException e) {
    // something went wrong
}
```

When deleting a group of markers, you specify a marker **type** to delete, such as **IMarker.PROBLEM**, or **null** to delete all markers. The second argument indicates whether you want to delete subtype markers. (We'll look at subtypes in a moment when we define new marker types.) The **depth** argument controls the depth of deletion.

You can also delete markers using **deleteMarkers(IMarker [])**.

Marker attributes

Given a marker, you can ask for its associated resource, its id (unique relative to that resource), and its type. You can also access additional information via generic attributes.

Each type of marker has a specific set of attributes that are defined by the creator of the marker type using naming conventions. The **IMarker** interface defines a set of constants containing the standard attribute names (and some of the expected values) for the platform marker types. The following method manipulates attributes using the platform constants.

```
IMarker marker = file.createMarker(IMarker.TASK);
if (marker.exists()) {
    try {
        marker.setAttribute(IMarker.MESSAGE, "A sample marker message");
        marker.setAttribute(IMarker.PRIORITY, IMarker.PRIORITY_HIGH);
    } catch (CoreException e) {
        // You need to handle the case where the resource doesn't exist
    }
}
```


Attributes are maintained generically as name/value pairs, where the names are strings and a value can be any one of the supported value types (boolean, integer, string). The limitation on value types allows the platform to persist the markers quickly and simply.

Querying markers

Resources can be queried for their markers and the markers of their children. For example, querying the workspace root with infinite depth considers all of the markers in the workspace.

```
IMarker[] problems = null;
int depth = IResource.DEPTH_INFINITE;
try {
    problems = resource.findMarkers(IMarker.PROBLEM, true, depth);
} catch (CoreException e) {
    // something went wrong
}
```

The result returned by **findMarkers** depends on the arguments passed. In the snippet above, we are looking for all markers of **type PROBLEM** that appear on the resource and all of its direct and indirect descendants.

If you pass **null** as the marker type, you will get all the marker types associated with the resource. The second argument specifies whether you want to look at the resource's children. The **depth** argument controls the depth of the search when you are looking at the resource's children. The depth can be **DEPTH_ZERO** (just the given resource), **DEPTH_ONE** (the resource and all of its direct children) or **DEPTH_INFINITE** (the resource and all of its direct and indirect descendants).

Marker persistence

The platform standard markers (task, problem, and bookmark) are **persistent**. This means that their state will be saved across workbench shutdown and startup.

New marker types declared by plug-ins are not persistent unless they are declared as such.

Extending the platform with new marker types

Plug-ins can declare their own marker types using the [org.eclipse.core.resources.markers](#) extension point. The standard marker types for problems, tasks and bookmarks are declared by the platform in the resources plug-in's markup.

```
<extension
  id="problemmarker"
  point="org.eclipse.core.resources.markers"
  name="%problemName">
  <super type="org.eclipse.core.resources.marker"/>
  <persistent value="true"/>
  <attribute name="severity"/>
  <attribute name="message"/>
  <attribute name="location"/>
</extension>
<extension
  id="taskmarker"
  point="org.eclipse.core.resources.markers"
  name="%taskName">
  <super type="org.eclipse.core.resources.marker"/>
  <persistent value="true"/>
  <attribute name="priority"/>
```

```

    <attribute name="message"/>
    <attribute name="done"/>
</extension>
<extension
    id="bookmark"
    point="org.eclipse.core.resources.markers"
    name="%bookmarkName">
    <super type="org.eclipse.core.resources.marker"/>
    <persistent value="true"/>
    <attribute name="message"/>
    <attribute name="location"/>
</extension>

```

New marker types are derived from existing ones using multiple inheritance. New marker types inherit all of the attributes from their super types and add any new attributes defined as part of the declaration. They also transitively inherit attributes from the super types of their super types. The following markup defines a new kind of marker in a hypothetical **com.example.markers** plug-in.

```

<extension
    id="mymarker"
    point="org.eclipse.core.resources.markers" />
<extension
    id="myproblem"
    point="org.eclipse.core.resources.markers">
    <super type="org.eclipse.core.resources.problemmarker"/>
    <super type="com.example.markers.mymarker"/>
    <attribute name="myAttribute" />
    <persistent value="true" />
</extension>

```

Note that the type **org.eclipse.core.resources.problemmarker** is actually one of the pre-defined types (aka **IMarker.PROBLEM**).

The only aspect of a marker super type that is not inherited is its **persistence** flag. The default value for persistence is false, so any marker type that should be persistent must specify **<persistent value="true"/>**.

After declaring the new marker type in your plug-in manifest file, you can create instances of **com.example.markers.myproblem** marker type and freely set or get the **myAttribute** attribute.

Declaring new attributes allows you to associate data with markers that you plan to use elsewhere (in your views and editors). Markers of a particular type do not have to have values for all of the declared attributes. The attribute declarations are more for solving naming convention problems (so everyone uses "message" to talk about a marker's description) than for constraining content.

```

public IMarker createMyMarker(IResource resource) {
    try {
        IMarker marker = resource.createMarker("com.example.markers.myproblem");
        marker.setAttribute("myAttribute", "MYVALUE");
        return marker;
    } catch (CoreException e) {
        // You need to handle the cases where attribute value is rejected
    }
}

```

You can query your own marker types in the same way you query the platform marker types. The method below finds all **mymarkers** associated with the given target resource and all of its descendents. Note that this will also find all **myproblems** since true is passed for the **includeSubtypes** argument.

```

public IMarker[] findMyMarkers(IResource target) {
    String type = "com.example.markers.mymarker";
    IMarker[] markers = target.findMarkers(type, true, IResource.DEPTH_INFINITE);
}

```

Tracking resource changes

The resources plug-in includes an event mechanism for notifying interested parties of changes to resources. If you need to keep track of changes to the resource tree while your plug-in is running, you can register an [IResourceChangeListener](#) with the workspace. Your listener will be notified of the changes via an [IResourceChangeEvent](#) object, which describes the changes.

Resource change processing

To track changes to resources, you must register a resource change listener with the workspace.

```

IResourceChangeListener listener = new MyResourceChangeReporter();
ResourcesPlugin.getWorkspace().addResourceChangeListener(
    listener, IResourceChangeEvent.POST_CHANGE);

```

Your listener will be notified after modifications to the workspace resources have been made. Resource API methods that modify resources trigger these events as part of their documented behavior. The method comment for a resource API method explicitly states whether or not it triggers a resource change event. For example, the following is included in the **IFile.createContents()** comment:

```

This method changes resources; these changes will be reported in a subsequent
resource change event, including an indication that this file's content have
been changed.

```

Methods that create, delete, or change a resource typically trigger these events. Methods that read, but do not write, resources typically do not trigger these events.

Batch changes and resource deltas

If you need to modify a number of resources at once, you can batch your resource change API calls so that only one resource change event will be sent for the entire set of changes. **IWorkspace.run(runnable, monitor)** is used to batch the operations. Resource change events are only sent to registered listeners when the specified **runnable** is completed. Any additional (nested) **Runnable**s created in the **runnable** will be considered part of the parent batch operation and the resource changes made in those **runnables** will appear in the parent's resource change notification.

You should batch your changes whenever possible to limit the overhead of broadcasting many granular changes. Failure to use batching will likely flood the system with resource change notifications and auto-builds.

The resource change event describes the specifics of the change (or set of changes) that have occurred in the workspace. The event contains a **resource delta** that describes the *net* effect of the changes. For example, if you add a resource and later delete it during one batch of changes, the resource will not appear in the delta.

The resource delta is structured as a tree rooted at the workspace root. The resource delta tree describes these types of changes:

- Resources that have been created, deleted, or changed. If you have deleted (or added) a folder, the

resource delta will include the folder and all files contained in the folder.

- Resources that have been moved or renamed using the **IResource.move()** API.
- Markers that have been added, removed, or changed. Marker modification is considered to be a workspace modification operation.
- Files that have been modified. Changed files are identified in the resource delta, but you do not have access to the previous content of the file in the resource delta.

To traverse a resource delta tree, you may implement the [IResourceDeltaVisitor](#) interface or traverse the tree explicitly using **IResource.getAffectedChildren**. Resource delta visitors implement a **visit** method that is called by the resource delta as it enumerates each change in the tree.

Note: Changes made to resource session properties or resource persistent properties are not identified in the resource delta.

Resource change events

Resource change events are sent whenever a change (or batched set of changes) is made to the workspace. In addition, resource change events are sent for certain specific workspace operations. The table below summarizes the types of resource change events and when they are reported.

Event type	Description
PRE_CLOSE	Notifies listeners that a project is about to be closed. This event can be used to extract and save necessary information from the in-memory representation (e.g., session properties) of a project before it is closed. (When a project is closed, the in-memory representation is disposed). The workspace is locked (no resources can be updated) during this event. The PRE_CLOSE event is also triggered during workspace shutdown since all open projects will be closed. The event contains the project that is being closed.
PRE_DELETE	Notifies listeners that a project is about to be deleted. This event can be used to perform clean-up operations, such as removing any saved state that is related to the project from your plug-in's directory. The workspace is locked (no resources can be updated) during this event. The event contains the project that is being deleted.
PRE_AUTOBUILD	Notifies listeners before any auto-building occurs. This event is broadcast when the platform detects an auto-build needs to occur, regardless of whether auto-building is actually enabled. The workspace is not locked during this event (resources can be updated). The event contains a resource delta describing the changes that have occurred since the last POST_CHANGE event was reported.
POST_AUTOBUILD	Notifies listeners after any auto-building has occurred. This event is broadcast after the platform would have performed an auto-build, regardless of whether auto-building is actually enabled. The workspace is not locked during this event (resources can be updated). The event contains a resource delta describing the changes that have occurred since the last POST_CHANGE event was reported.
POST_CHANGE	Describes a set of changes that have occurred to the workspace since the last POST_CHANGE event was reported. Triggered after a resource change API is used individually or in a batched set of workspace changes. Also triggered after any PRE_AUTOBUILD or POST_AUTOBUILD notification is complete. The event contains a resource delta describing the net changes since the last POST_CHANGE event. The workspace is locked (no resources can be updated) during this event.

Implementing a resource change listener

The following example implements a console-based resource change listener. A resource change listener is registered for specific types of events and information about these events is printed to the console:

```
IResourceChangeListener listener = new MyResourceChangeReporter();
ResourcesPlugin.getWorkspace().addResourceChangeListener(listener,
    IResourceChangeEvent.PRE_CLOSE
    | IResourceChangeEvent.PRE_DELETE
    | IResourceChangeEvent.PRE_AUTO_BUILD
    | IResourceChangeEvent.POST_AUTO_BUILD
    | IResourceChangeEvent.POST_CHANGE);
```

The listener checks for each event type and reports information about the resource that was changed and the kinds of changes that occurred. Although this example is designed to show a general listener that handles all the types of resource events, a typical listener would register for just one type of event.

The implementation for **POST_CHANGE** uses another class that can be used to visit the changes in the resource delta.

```
import org.eclipse.resources.*;
import org.eclipse.runtime.*;

public class MyResourceChangeReporter implements IResourceChangeListener {
    public void resourceChanged(IResourceChangeEvent event) {
        IResource res = event.getResource();
        switch (event.getType()) {
            case IResourceChangeEvent.PRE_CLOSE:
                System.out.print("Project ");
                System.out.print(res.getFullPath());
                System.out.println(" is about to close.");
                break;
            case IResourceChangeEvent.PRE_DELETE:
                System.out.print("Project ");
                System.out.print(res.getFullPath());
                System.out.println(" is about to be deleted.");
                break;
            case IResourceChangeEvent.POST_CHANGE:
                System.out.println("Resources have changed.");
                event.getDelta().accept(new DeltaPrinter());
                break;
            case IResourceChangeEvent.PRE_AUTO_BUILD:
                System.out.println("Auto build about to run.");
                event.getDelta().accept(new DeltaPrinter());
                break;
            case IResourceChangeEvent.POST_AUTO_BUILD:
                System.out.println("Auto build complete.");
                event.getDelta().accept(new DeltaPrinter());
                break;
        }
    }
}
```

The **DeltaPrinter** class implements the [IResourceDeltaVisitor](#) interface to interrogate the resource delta. The **visit()** method is called for each resource change in the resource delta. The visitor uses a return value to indicate whether deltas for child resources should be visited.

```
class DeltaPrinter implements IResourceDeltaVisitor {
    public boolean visit(IResourceDelta delta) {
```

```

IResource res = delta.getResource();
switch (delta.getKind()) {
    case IResourceDelta.ADDED:
        System.out.print("Resource ");
        System.out.print(res.getFullPath());
        System.out.println(" was added.");
        break;
    case IResourceDelta.REMOVED:
        System.out.print("Resource ");
        System.out.print(res.getFullPath());
        System.out.println(" was removed.");
        break;
    case IResourceDelta.CHANGED:
        System.out.print("Resource ");
        System.out.print(res.getFullPath());
        System.out.println(" has changed.");
        break;
}
return true; // visit the children
}
}

```

Further information can be obtained from the supplied resource delta. The following snippet shows how the **IResourceDelta.CHANGED** case could be implemented to further describe the resource changes.

```

...
case IResourceDelta.CHANGED:
    System.out.print("Resource ");
    System.out.print(delta.getFullPath());
    System.out.println(" has changed.");
    switch (delta.getFlags()) {
        case IResourceDelta.CONTENT:
            System.out.println("--> Content Change");
            break;
        case IResourceDelta.REPLACED:
            System.out.println("--> Content Replaced");
            break;
        case IResourceDelta.REMOVED:
            System.out.println("--> Removed");
            break;
        case IResourceDelta.MARKERS:
            System.out.println("--> Marker Change");
            IMarkerDelta[] markers = delta.getMarkerDeltas();
            // if interested in markers, check these deltas
            break;
    }
    break;
...

```

For a complete description of resource deltas, visitors, and marker deltas, consult the API specification for [IResourceDelta](#), [IResourceDeltaVisitor](#), and [IMarkerDelta](#).

Note: Resource change listeners are useful for tracking changes that occur to resources while your plug-in is activated. If your plug-in registers a resource change listener during its startup code, it's possible that many resource change events have been triggered before the activation of your plug-in. The resource delta contained in the first resource change event received by your plug-in will not contain all of the changes made since your plug-in was last activated. If you need to track changes made between activations of your plug-in, you should use the support provided for workspace saving. This is described in [Workspace save participation](#).

Incremental project builders

An **incremental project builder** is an object that manipulates the resources in a project in a fashion that is defined by the builder itself. Incremental project builders are often used to apply a transformation on a resource to produce a resource or artifact of another kind.

Plug-ins contribute incremental project builders to the platform in order to implement specialized resource transformations. For example, the Java development tooling (JDT) plug-in that is provided with the platform SDK defines an incremental project builder that compiles a Java source file into a class file any time a file is added or modified in a Java project and recompiles any other files affected by the change.

The platform defines two types of builds:

- A **full build** performs a build from scratch. It treats all resources in the projects as if they have never been seen by the builder.
- An **incremental build** uses a "last build state," maintained internally by the builder, to do an optimized build based on the changes since the last build.

Incremental builds are seeded with a resource change delta. The delta reflects the net effect of all resource changes since the builder last built the project. This delta is similar to the one used inside resource change events.

Builders are best understood by example. The Java development tooling (JDT) provides a Java compiler which is driven by a Java incremental project builder to recompile files that are affected by changes. When a full build is triggered, all of the **.java** files in the project are compiled. Any compile problems encountered are added as problem markers on the affected **.java** files. When an incremental build is triggered, the builder selectively recompiles the added, changed, or otherwise affected **.java** files that are described in the resource delta and updates the problem markers as necessary. Any **.class** files or markers that are no longer appropriate are removed.

Incremental building has obvious performance benefits for projects with hundreds or thousands of resources, most of which are unchanging at any given point in time.

The technical challenge for incremental building is to determine exactly what needs to be rebuilt. For example, the internal state maintained by the Java builder includes things like a dependency graph and a list of compilation problems reported. This information is used during an incremental build to identify which classes need to be recompiled in response to a change in a Java resource.

Although the basic structure for building is defined in the platform, the real work is done in your plug-in. Patterns for implementing complex incremental builders are beyond the scope of this discussion, since the implementation is dependent on the specific builder design.

Invoking a build

A builder can be invoked explicitly in one of the following ways:

- **IProject.build()** runs the build processing on the receiving project.
- **IWorkspace.build()** runs the build processing on all open projects in the workspace.

In practice, the workbench user triggers a build by selecting the corresponding commands in the resource navigator menu.

Incremental project builders are also invoked implicitly by the platform during an auto-build. If enabled, auto-builds run whenever the workspace is changed.

Defining an incremental project builder

The [org.eclipse.core.resources.builders](#) extension point is used to contribute an incremental project builder to the platform. The following markup shows how the hypothetical plug-in **com.example.builders** could contribute an incremental project builder.

```
<extension
  point="org.eclipse.core.resources.builders">
  <builder
    id="com.example.builders"
    name="MyBuilder">
    <run
      class="com.example.builders.BuilderExample">
      <parameter name="optimize" value="true" />
      <parameter name="comment" value="Builder comment" />
    </run>
  </builder>
</extension>
```

The class identified in the extension point must extend the platform class [IncrementalProjectBuilder](#).

```
public class BuilderExample extends IncrementalProjectBuilder {
    IProject[] build(int kind, Map args, IProgressMonitor monitor)
        throws CoreException {
        // add your build logic here
        return null;
    }
    protected void startupOnInitialize() {
        // add builder init logic here
    }
}
```

Build processing begins with the **build()** method, which includes information about the kind of build that has been requested, **FULL_BUILD**, **INCREMENTAL_BUILD**, or **AUTO_BUILD**. If an incremental build has been requested, a resource delta is provided to describe the changes in the project's resources since the last build. The following snippet further refines the **build()** method.

```
protected IProject[] build(int kind, Map args, IProgressMonitor monitor
    throws CoreException {
    if (kind == IncrementalProjectBuilder.FULL_BUILD) {
        fullBuild(monitor);
    } else {
        IResourceDelta delta = getDelta(getProject());
        if (delta == null) {
            fullBuild(monitor);
        } else {
            incrementalBuild(delta, monitor);
        }
    }
    return null;
}
```

It sometimes happens that when building project "X," a builder needs information about changes in some other project "Y." (For example, if a Java class in X implements an interface provided in Y.) While building X, a delta for Y is available by calling **getDelta(Y)**. To ensure that the platform can provide such

deltas, X's builder must have declared the dependency between X and Y by returning an array containing Y from a previous `build()` call. If a builder has no dependencies, it can simply return null. See [IncrementalProjectBuilder](#) for further information.

Full build

The logic required to process a full build request is specific to the plug-in. It may involve visiting every resource in the project (if the build was triggered for a project) or even examining other projects if there are dependencies between projects. The following snippet suggests how a full build might be implemented.

```
protected void fullBuild(final IProgressMonitor monitor) throws CoreException {
    try {
        getProject().accept(new MyBuildVisitor());
    } catch (CoreException e) { }
}
```

The build visitor would perform the build for the specific resource (and answer true to continue visiting all child resources).

```
class MyBuildVisitor implements IResourceVisitor {
    public boolean visit(IResource res) {
        //build the specified resource.
        //return true to continue visiting children.
        return true;
    }
}
```

The visit process continues until the full resource tree has been traveled.

Incremental build

When performing an incremental build, you work with a resource change delta instead of the whole project.

```
protected void incrementalBuild(IResourceDelta delta,
    IProgressMonitor monitor) throws CoreException {
    // the visitor does the work.
    delta.accept(new MyBuildDeltaVisitor());
}
```

Note that for an incremental build, the build visitor works with a resource delta tree instead of a complete resource tree.

The visit process continues until the complete resource delta tree has been traveled. The specific nature of changes is similar to that described in [Implementing a resource change listener](#). One important difference is that with incremental project builders, you are typically working with a resource delta based on a particular project, not the entire workspace.

Associating an incremental project builder with a project

To make a builder available for a given project, it must be included in the build spec for the project. A project's build spec is a list of commands to run, in sequence, when the project is built. Each command names a single incremental project builder.

The following snippet adds a new builder as the first builder in the existing list of builders.

```

IProjectDescription desc = project.getDescription();
ICommand[] commands = desc.getBuildSpec();
boolean found = false;

for (int i = 0; i < commands.length; ++i) {
    if (commands[i].getBuilderName().equals(BUILDER_ID)) {
        found = true;
        break;
    }
}
if (!found) {
    //add builder to project
    ICommand command = desc.newCommand();
    command.setBuilderName(BUILDER_ID);
    ICommand[] newCommands = new ICommand[commands.length + 1];

    // Add it before other builders.
    System.arraycopy(commands, 0, newCommands, 1, commands.length);
    newCommands[0] = command;
    desc.setBuildSpec(newCommands);
    project.setDescription(desc, null);
}

```

Configuring a project's builder is done just once, usually as the project is being created.

Workspace save participation

Workspace save processing is triggered when the workbench is shut down by the user and at other times periodically by the platform. Plug-ins can participate in the workspace save process so that critical plug-in data is saved to disk whenever the rest of the workspace's persistent data is saved.

The workspace save process can also be used to track changes that occur between activations of your plug-in.

Implementing a save participant

To participate in workspace saving, you must add a save participant to the workspace. This is typically done during your plug-in's startup method. Let's look at a simple plug-in which will demonstrate the save process.

```

package com.example.saveparticipant;

import org.eclipse.core.runtime.*;
import org.eclipse.core.resources.*;
import java.io.File;
import java.util.*;

public class MyPlugin extends Plugin {
    private static MyPlugin plugin;

    public MyPlugin(IPluginDescriptor descriptor) {
        super(descriptor);
        plugin = this;
    }

    public static MyPlugin getDefault() {
        return plugin;
    }

    protected void readStateFrom(File target) {

```

```

    }

    public void startup() throws CoreException {
        super.startup();
        ISaveParticipant saveParticipant = new MyWorkspaceSaveParticipant();
        ISavedState lastState =
            ResourcesPlugin.getWorkspace().addSaveParticipant(this, saveParticipant);
        if (lastState == null)
            return;
        IPath location = lastState.lookup(new Path("save"));
        if (location == null)
            return;
        // the plugin instance should read any important state from the file.
        File f = getStateLocation().append(location).ToFile();
        readStateFrom(f);
    }

    protected void writeImportantState(File target) {
    }
}

```

To participate in workspace saving, you must add a save participant to the workspace. This is typically done during your plug-ins **startup** method. This is also where you read any state that you might have saved when your plug-in was last shut down.

ISaveParticipant defines the protocol for a workspace save participant. Implementors of this interface can provide behavior for different stages of the save process. Let's look at the stages and how our class **WorkspaceSaveParticipant** implements each of these steps.

- **prepareToSave** notifies the participant that the workspace is about to be saved and that it should suspend normal operation until further notice. Our save participant does nothing here.

```

public void prepareToSave(ISaveContext context) throws CoreException {
}

```

- **saving** tells the participant to save its important state.

```

public void saving(ISaveContext context) throws CoreException {
    switch (context.getKind()) {
        case ISaveContext.FULL_SAVE:
            MyPlugin myPluginInstance = MyPlugin.getDefault();
            // save the plug-in state
            int saveNumber = context.getSaveNumber();
            String saveFileName = "save-" + Integer.toString(saveNumber);
            File f = myPluginInstance.getStateLocation().append(saveFileName).ToFile();
            // if we fail to write, an exception is thrown and we do not update the path
            myPluginInstance.writeImportantState(f);
            context.map(new Path("save"), new Path(saveFileName));
            context.needSaveNumber();
            break;
        case ISaveContext.PROJECT_SAVE:
            // get the project related to this save operation
            IProject project = context.getProject();
            // save its information, if necessary
            break;
        case ISaveContext.SNAPSHOT:
            // This operation needs to be really fast because
            // snapshots can be requested frequently by the
            // workspace.
            break;
    }
}

```

```
}  
}
```

The **ISaveContext** describes information about the save operation. There are three kinds of save operations: **FULL_SAVE**, **SNAPSHOT**, and **PROJECT_SAVE**. Save participants should be careful to perform the processing appropriate for the kind of save event they have received. For example, snapshot events may occur quite frequently and are intended to allow plug-ins to save their critical state. Taking a long time to save state which can be recomputed in the event of a crash will slow down the platform.

A save number is used to create data save files that are named using sequential numbers (**save-1**, **save-2**, etc.) Each save file is mapped to a logical file name (**save**) that is independent of the save number. Plug-in data is written to the corresponding file and can be retrieved later without knowing the specific save number of the last successful save operation. Recall that we saw this technique in our plug-in's startup code:

```
IPath location = lastState.lookup(new Path("save"));
```

After we have saved our data and mapped the file name, we call **needSaveNumber** to indicate that we have actively participated in a workspace save and want to assign a number to the save activity. The save numbers can be used to create data files as above.

- **doneSaving** notifies the participant that the workspace has been saved and the participant can continue normal operation.

```
public void doneSaving(ISaveContext context) {  
    MyPlugin myPluginInstance = MyPlugin.getDefault();  
  
    // delete the old saved state since it is not necessary anymore  
    int previousSaveNumber = context.getPreviousSaveNumber();  
    String oldFileName = "save-" + Integer.toString(previousSaveNumber);  
    File f = myPluginInstance.getStateLocation().append(oldFileName).toFile();  
    f.delete();  
}
```

Here, we clean up the save information from the previous save operation. We use **getPreviousSaveNumber** to get the save number that was assigned in the previous save operation (not the one we just completed). We use this number to construct the name of the file that we need to delete. Note that we do not use the save state's logical file map since we've already mapped our current save file number.

- **rollback** tells the participant to rollback the important state because the save operation has failed.

```
public void rollback(ISaveContext context) {  
    MyPlugin myPluginInstance = MyPlugin.getDefault();  
  
    // since the save operation has failed, delete the saved state we have just written  
    int saveNumber = context.getSaveNumber();  
    String saveFileName = "save-" + Integer.toString(saveNumber);  
    File f = myPluginInstance.getStateLocation().append(saveFileName).toFile();  
    f.delete();  
}
```

Here, we delete the state that we just saved. Note that we use the current save number to construct the file name of the file we just saved. We don't have to worry about the fact that we mapped this file name into the **ISaveContext**. The platform will discard the context when

a save operation fails.

If your plug-in throws an exception at any time during the save lifecycle, it will be removed from the current save operation and will not get any of the remaining lifecycle methods. For example, if you fail during your **saving** method, you will not receive a **rollback** or **doneSaving** message.

Using previously saved state

When you add a save participant to the workspace, it will return an [ISavedState](#) object, which describes what your plug-in saved during its last save operation (or **null** if your plug-in has not previously saved any state). This object can be used to access information from the previous save file (using the save number and file map) or to process changes that have occurred between activations of a plug-in.

Accessing the save files

If a file map was used to save logically named files according to the save number, this same map can be used to retrieve the data from the last known save state.

```
ISaveParticipant saveParticipant = new MyWorkspaceSaveParticipant();
ISavedState lastState =
    ResourcesPlugin.getWorkspace().addSaveParticipant(myPluginInstance, saveParticipant);

if (lastState != null) {
    String saveFileName = lastState.lookup(new Path("save")).toString();
    File f = myPluginInstance.getStateLocation().append(saveFileName).toFile();
    // the plugin instance should read any important state from the file.
    myPluginInstance.readStateFrom(f);
}
```

Processing resource deltas between activations

Recall that any number of resource change events could occur in the workspace before your plug-in is ever activated. If you want to know what changes have occurred since your plug-in was deactivated, you can use the save mechanism to do so, even if you don't need to save any other data.

The save participant must request that the platform keep a resource delta on its behalf. This is done as part of the save operation.

```
public void saving(ISaveContext context) throws CoreException {
    // no state to be saved by the plug-in, but request a
    // resource delta to be used on next activation.
    context.needDelta();
}
```

During plug-in startup, the previous saved state can be accessed and change events will be created for all changes that have occurred since the last save.

```
ISaveParticipant saveParticipant = new MyWorkspaceSaveParticipant();
ISavedState lastState =
    ResourcesPlugin.getWorkspace().addSaveParticipant(myPluginInstance, saveParticipant);
if (lastState != null) {
    lastState.processResourceChangeEvents(new MyResourceChangeReporter());
}
```

The provided class must implement [IResourceChangeListener](#), as described in [Tracking resource changes](#). The changes since the last save are reported as part of the **POST_AUTO_BUILD** resource change event.

*Note: Marker changes are not reported in the change events stored in an **ISavedState**. You must assume that any or all markers have changed since your last state was saved.*

Project natures

Project natures allow a plug-in to tag a project as a specific kind of project. For example, the Java development tools (JDT) use a "Java nature" to add Java-specific behavior to projects.

Natures are installed on a per-project basis and are configured automatically when added to a project and deconfigured when removed. A project can have more than one nature.

To implement your own nature, you need to define an extension and supply a class which implements [IProjectNature](#).

Defining a nature

The [org.eclipse.core.resources.natures](#) extension point is used to add a project nature definition. The following markup adds a nature for the hypothetical **com.example.natures** plug-in.

```
<extension
  point="org.eclipse.core.resources.natures"
  id="mynature"
  name="My Nature">
  <runtime>
    <run class="com.example.natures.MyNature">
    </run>
  </runtime>
</extension>
```

The class identified in the extension must implement the platform interface [IProjectNature](#). This class implements plug-in specific behavior for associating nature-specific information with a project when the nature is configured.

```
public class MyNature implements IProjectNature {

    private IProject project;

    public void configure() throws CoreException {
        // Add nature-specific information
        // for the project, such as adding a builder
        // to a project's build spec.
    }
    public void deconfigure() throws CoreException {
        // Remove the nature-specific information here.
    }
    public IProject getProject() {
        return project;
    }
    public void setProject(IProject value) {
        project = value;
    }
}
```

The **configure()** and **deconfigure()** methods are sent by the platform when natures are added and removed from a project. You can implement the **configure()** method to add a builder to a project as discussed in [Builders](#).

Associating the nature with a project

Once you've defined the nature and implemented its class, you must assign the nature to a project. The following snippet assigns our new nature to a given project.

```
try {
    IProjectDescription description = project.getDescription();
    String[] natures = description.getNatureIds();
    String[] newNatures = new String[natures.length + 1];
    System.arraycopy(natures, 0, newNatures, 0, natures.length);
    newNatures[natures.length] = "com.example.natures.mynature";
    description.setNatureIds(newNatures);
    project.setDescription(description, null);
} catch (CoreException e) {
    // Something went wrong
}
```

The identifier used for the nature is the fully qualified name (plug-in id + extension id) of the nature extension.

Natures are typically added to a project when it is created. You typically provide a customized new project wizard that captures your specialized information about a project and assigns your nature to it.

Scripting a user interface

The platform scripting plug-in allows you to develop various workbench extensions using JavaScript. This support is provided using the Rhino JavaScript engine. You can use JavaScript to contribute the following to the workbench:

- actions (actionSets)
- editors
- wizards (newWizards, exportWizards, importWizards)
- popup menus
- preference pages

The following sections assume you have a working knowledge of JavaScript, workbench extensions, and SWT.

Concepts

Scripts are managed within the workbench as regular project files. You typically create a separate project to contain your scripts. A specialized editor is provided for editing the scripts.

There are two types of scripts supported by the platform:

1. **Batch scripts** use JavaScript to call methods on various workbench objects and manipulate the results. These scripts are well-suited for creating new, repetitive actions that are not part of the basic workbench function.
2. **UI scripts** resemble a client-side HTML page with scripted UI events. These scripts are well-suited for creating scripts that require more elaborate interaction with the user.

Script writers gain access to the workbench environment through several platform objects that are exposed as local JavaScript variables. Scripts can also use the facilities of JavaScript to create Java objects and call their methods.

Once a file containing the script is created, it can be registered with the workbench through the preferences dialog (**Window**→**Preferences**→**Scripts**). Expanding this entry in the preferences page will reveal a list of the workbench extension points that can be scripted. Scripts are added to each extension point by selecting it in the preferences tree and pressing the corresponding **Add** button in the preference page.

The workbench must be restarted the first time each script is registered. Once registered, the script can be modified and immediately executed without restarting the workbench.

All console output from scripts, as well as any errors detected while running scripts are written to the script console. You can add the script console view to any other perspective, or you can always run your scripts from the Scripting perspective (includes the console view by default).

Limitations of scripts

Scripts that are added to the workbench may only access the API methods defined in the following plug-ins:

- **org.eclipse.core.runtime**
- **org.eclipse.core.boot**
- **org.eclipse.core.resources**

- [org.eclipse.ui](#)
- [org.eclipse.swt](#)
- [org.eclipse.scripting](#)

Future releases will lift this restriction so that arbitrary plug-ins can be scripted.

Writing a batch script

A "batch" script is simply a collection of JavaScript statements that work with platform objects by invoking their API methods and manipulating the results. There are three platform objects exposed to script writers:

1. **toolkit** – a utility object that implements [org.eclipse.scripting.IToolkit](#). It provides convenience methods for basic workspace resource manipulations, prompting, and loading of persisted script state.
2. **plugin** – a default instance of [org.eclipse.core.runtime.Plugin](#). It provides access to plug-in methods. All user scripts execute as part of a single plug-in managed by the platform scripting support.
3. **platform** – a convenience object with methods corresponding to [org.eclipse.core.runtime.Platform](#). It provides access to general platform functions.

The method `toolkit.print(String)` is used to write output to the script console.

In addition to the predefined local variables, the JavaScript language allows you to create Java objects and use their public API. The JavaScript property **Packages** can be used for this purpose. The following snippet uses the **Packages** property to obtain the install URL and the splash bitmap object.

```
var install = Packages.org.eclipse.core.boot.BootLoader.getInstallURL();
var splash = new Packages.java.net.URL(install, "splash.bmp");
```

Writing user interface scripts

SWT-based user interfaces can be created with the platform scripting support. The resulting scripts are very similar to client-side HTML pages with a few exceptions:

- only a subset of html elements and attributes is used
- the resulting html must be well formed **xhtml**
- several "non-html" elements and attributes are defined

A scripting editor is provided by the scripting plug-in. This editor supports a source view and a preview that shows how the source page will be rendered when executed. The editor makes use of the **Outline** view and the **Properties** view, so you should show these views when editing UI scripts.

The UI script editor is registered as the default editor for files of type **xhtml**.

UI scripts are best understood by example. See the [Scripting Examples](#) for specific techniques. The following roadmap describes the elements that can be used in scripts and their supported attributes and event blocks.

Body

- **body** – **id**, **onload**, **onunload**. Defines the body of the UI definition.

Form

- **form** – **id**, **onreset**, **onsubmit**, **title**. Defines the major UI group (typically rendered as a separate page).

UI elements

- **label** – **id**, **value**. The value is the string label.
- **entry field** (<input type="text">) – **id**, **onblur**, **onchange**, **onfocus**, **onkeypress**, **value**. Defines a text entry field.
- **textarea** – **id**, **onblur**, **onchange**, **onfocus**, **onkeypress**. Defines a text area.
- **push button** (<input type="button">) – **id**, **onpush**, **value**. Defines a push button.
- **radio button** (<input type="radio">) – **id**, **checked**, **onclick**, **value**. Defines a radio button.
- **check-box** (<input type="checkbox">) – **id**, **checked**, **onclick**, **value**. Defines a check box.
- **combo-box** (<select size="1">) – **id**, **onchange**. Defines a combo-box.
- **list-box** (<select size="3">) – **id**, **onchange**. Defines a list-box.
- **image** – **id**, **source**. Allows placement of an image as a user interface element.

Layout controls

- **group box** (<fieldset>) – **id**, **title**. Allows grouping of individual UI elements
- **table** – **id**. Creates a table

table row

table data – **colspan**, **rowspan**. Defines a table cell.

- **line break**
- **horizontal rule**

Scripting

- **script**

In general, all UI elements can specify an **id** attribute. This will define a local script variable of the same name. The script can manipulate the UI elements through these local variables. The available methods are typically those of the equivalent SWT widget.

Simple UI layout can be most effectively performed by inserting line breaks
 between groups of UI elements to start the new layout row. The script rendering support performs default layout which will be sufficient in most simple cases.

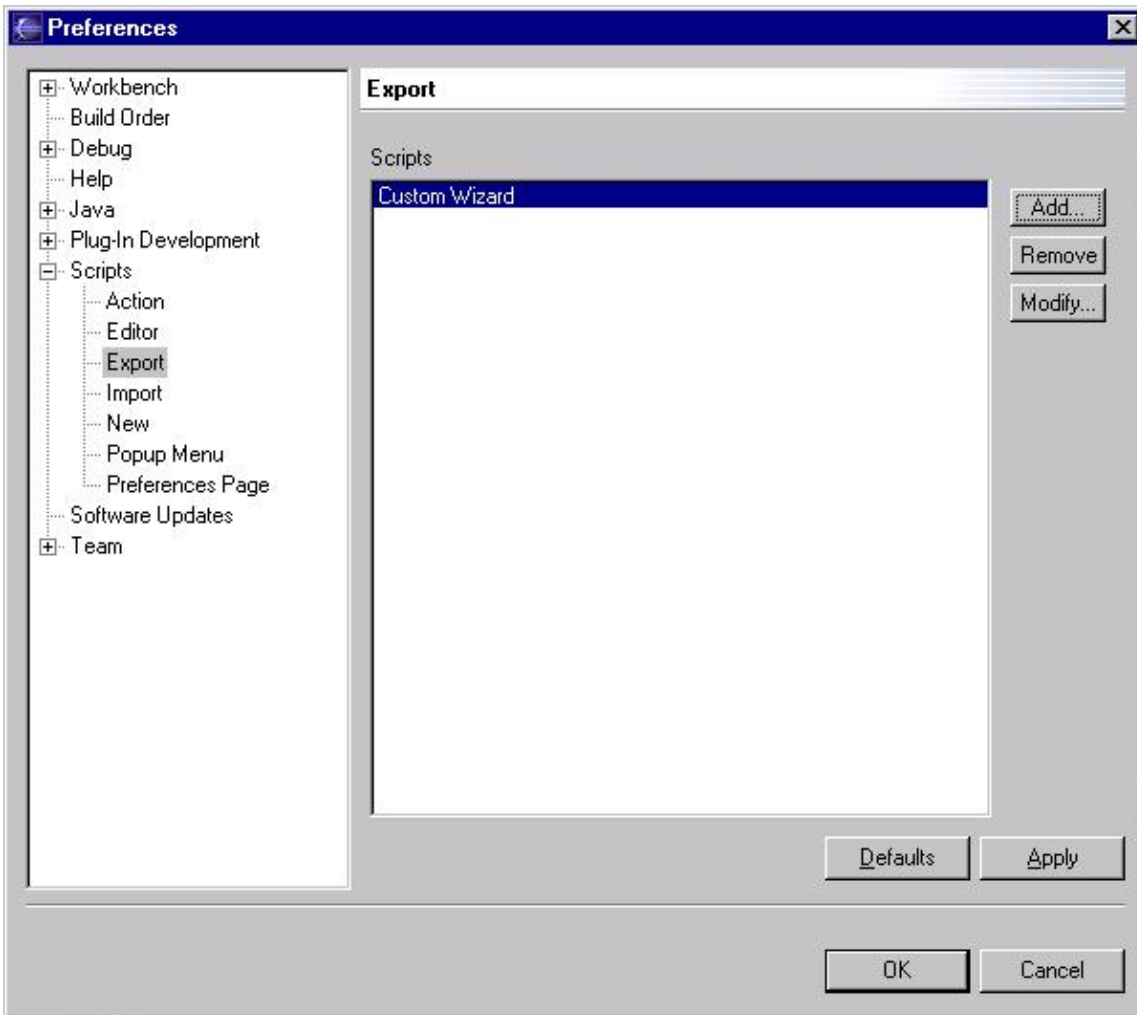
More complex layout can be achieved by using tables to control placement of UI elements. In addition, separate UI elements can be grouped using the <fieldset> element (valid HTML but not frequently used in web page design).

The actual script function bodies are defined using the <script> element(s). The functions are triggered as a result of calls specified within the "**onXXX**" attributes (e.g., **onload**, **onreset**) of the corresponding UI elements (just as in html pages).

For UI scripts, the **toolkit** local variable is an object instance of the org.eclipse.scripting.IToolkitUI interface. It contains additional methods that are not available in the **toolkit** object exposed to batch scripts.

Registering scripts with the workbench

Once created as workbench resources, the scripts can be registered into the appropriate workbench extension point using the workbench preferences dialog.



Scripts are added using the **Add** dialog. The dialog prompts for the following information:

- display name of the script
- script location (as a workbench resource)
- icon (optional)
- whether to register as batch script or UI script. Some extension points allow either.
- file extension for which to register the script. Some extension points require a file extension.

Using UI scripting with Java

The UI scripting facility will support Java as its scripting language instead of JavaScript. When Java is used as the scripting language, you must create an instance of [org.eclipse.scripting.Renderer](#). This object will be used to render the UI and can be associated with one or more event handler objects, also written in Java. Use the **addObject(String,Object)** method of the renderer to add an event handler.

The JavaScript event model is used even when scripting with Java. The various "**onXXX**" clauses are specified as before. In this case, they are implemented as reflective callbacks to methods on the registered

event handlers, rather than as calls to "inline" JavaScript functions.

Explicit use of script adapters

We've seen how scripts can be added and managed by the workbench user. Scripts can also be supplied as extensions by plug-ins. This is done by specifying script adapters in place of implementation classes as the **class** attribute in the extension definition. The adapter specification includes the name of the actual script to execute. For example, the following markup from the scripting examples **plugin.xml** contributes an action set and specifies a [ScriptAdapterWorkbenchAction](#) with the script name as the **class**. The relevant markup is in bold.

```
<extension point = "org.eclipse.ui.actionSets">
  <actionSet
    id="scriptActionSet"
    label="%S_Script_Examples"
    visible="true">
    ...
    <action
      id = "scriptaction1"
      menubarPath = "window/org_eclipse_scripting_examples/slot1"
      toolbarPath = "%S_Script_Examples"
      label = "%S_Scripted_Action_Example"
      tooltip = "%S_Perform_Scripted_JavaScript_Action"
      icon = "icons/full/ctool16/script_scp.gif"
      class = "org.eclipse.scripting/org.eclipse.scripting.ScriptAdapterWorkbenchAction"
      file = "platform:/plugin/org.eclipse.scripting.examples/excalibur.js"
      language = "javascript"
    </action>
    ...
```

Standard script adapters can be referenced directly as part of the **plugin.xml** specification of any plug-in. The list of supported adapters can be found in the [org.eclipse.scripting](#) package.

The [Scripting Examples](#) show additional uses of these adapters. The script example plug-in implements each of the script extensions by using the predefined adapters. Refer to the **plugin.xml** file for additional details.

References

The API Specification for [org.eclipse.scripting](#) contains additional information about scripting support.

The scripting examples also provide descriptions and usage information. See [Scripting Examples](#) for further information.

Plugging in help

The platform help system allows you to contribute your plug-in's online help using the [org.eclipse.help.contributions](#) extension point. You can either contribute the online help as part of your code plug-in or provide it separately in its own documentation plug-in.

Separating the documentation into a separate plug-in is beneficial in those situations where the code and documentation teams are different groups or where you want to reduce the coupling between the documentation and code.

The platform's help facilities provide you with the raw building blocks to structure and contribute your help. It does not dictate structure or granularity of documentation. The platform does, however, provide and control the integrated help viewers, thus ensuring seamless integration of your documentation.

The [org.eclipse.help.contributions](#) extension point provides four elements through which you can contribute your help, they are:

- topics
- infoset (also known as a book)
- infoview
- actions (also known as wiring)

The topics, infoset and actions contributions all specify an associated **.xml** file that contains the details of the contribution. The best way to demonstrate these elements is to create a documentation plug-in that uses them.

Building a help plug-in

In this example, we assume that a documentation author has already supplied you with the raw documentation in the form of HTML files. The granularity and structure of these files is completely up to the documentation team. Once the documentation is delivered, setting up the plug-in and topics can be done independently.

We start by assuming that the documentation has already been provided in the following tree.

```
doc/  
  concepts/  
    concept1.html  
    concept1_1.html  
    concept1_2.html  
  tasks/  
    task1.html  
    task2.html  
    task3_1.html  
    task3_2.html  
  ref/  
    ref1.html  
    ref2.html
```

We will assume that the plug-in name is **com.example.helpexample**.

The first step is to create a plug-in directory, **com.example.helpexample** underneath the platform **plugins** directory. The **doc** sub tree shown above should be copied into the directory.

Documentation plug-ins need a manifest just like code plug-ins. The following markup defines the

documentation plug-in.

```
<?xml version="1.0" ?>
<plugin name="Online Help Sample"
  id="com.example.helpexample"
  version="1.0"
  provider-name="MyExample" />
```

Defining the help topics

Now that we have our sample content files we can create a topics file. A topics file defines the key entry points into the HTML content files by mapping a topic id and label to a reference in one of the HTML files. A topics file acts like a table of contents for a set of html content.

Applications that are being migrated to the platform can reuse existing documentation by using the topics file to define entry points into that documentation.

A plug-in can have one or more topics files. Topics files are sometimes referred to as navigation files since they describe how to navigate the html content. Our example documentation is organized into three main categories: concepts, tasks and reference. How do we make topics files that represent this structure?

We could make one large topics file, or we could create a separate topics file for each main category of content. This decision should be made according to the way your documentation teams work together. If a different author owns each category, it might be preferable to keep separate topics files for each category. It is not dictated by the platform architecture.

In this example, we will create a topics file for each major content category. For such a small number of files, having separate topics files for each category may not be necessary. We will wire this example as if we had many more files or had separate authors who own each content category.

Our files look like this:

topics_Concepts.xml

```
<topics id="conceptsAll">
  <topic label="Concept1" href="doc/concepts/concept1.html">
    <topic label="Concept1_1" href="doc/concepts/concept1_1.html"/>
    <topic label="Concept1_2" href="doc/concepts/concept1_2.html"/>
  </topic>
</topics>
```

topics_Tasks.xml

```
<topics id="tasksAll">
  <topic id="plainTasks" label="Plain Stuff">
    <topic label="Task1" href="doc/tasks/task1.html"/>
    <topic label="Task2" href="doc/tasks/task2.html"/>
  </topic>
  <topic id="funTasks" label="Fun Stuff" >
    <topic label="Task3_1" href="doc/tasks/task3_1.html"/>
    <topic label="Task3_2" href="doc/tasks/task3_2.html"/>
  </topic>
</topics>
```


topics_Ref.xml

```
<topics id="refAll">
  <topic label="Ref1" href="doc/ref/ref1.html"/>
  <topic label="Ref2" href="doc/ref/ref2.html"/>
</topics>
```

Topics are contributed as part of the topics container element. A topic can be a simple link to content. For example, "Task1" provides a **label** and an **href** linking to the content. A topic can also be a hierarchical grouping of sub topics with no content of its own. For example, "Fun Stuff" has only a **label** and sub topics, but no **href**. Topics can do both, too. "Concept1" has an **href** and sub topics.

When used as a link, the argument in an **href** is assumed to be relative to the current plug-in.

When we start wiring these topics into the overall documentation web we will refer to them by their id. Only topics with an id can be manipulated. To wire in all of the sub topics for a particular topic, we can wire in the parent topic. For example, wiring in "Concept1" would also wire in "Concept1_1" and "Concept1_2."

Later we will modify the plugin.xml to add the actual contributions pointing to these files.

Creating the infoset

Now that we have defined topics from our content files, we will create the infoset. An information set (infoset) is a documentation web or book. The platform can display any number of infosets.

An infoset contains one or more infoviews. An infoview provides a high-level semantic grouping within the infoset. Infoviews can be used to create multiple views onto the document web. For example, we can use them to create an integrated view of documentation that is supplied by many components. We can also use infoviews to create separate views for our various help topics. Or, as we will shortly do, we can create one infoview to show all of our help topics. The term **infoview** is used to avoid collision/confusion with the term **view** in the platform's user interface.

Each infoview contains a collection of topics. Sometimes a higher-level component or product team is responsible for weaving together the documentation and topics supplied by a number of its component teams. For now we assume that our plug-in should supply both the topics and the book that integrates the topics.

The following infoset has the id **infoset_SampleGuide** and declares one infoview whose id is **view_Contents**. The id of the infoview will become important as we start wiring in the high level structure for the infoview, and ultimately wiring in the topics we defined earlier.

infoset_SampleGuide.xml

```
<infoset id="infoset_SampleGuide" label="Online Help Sample" href="doc/splash.html">
  <infoview label="Content" id="view_Contents" />
</infoset>
```

When the user selects the book called "Online Help Sample" he/she will see one possible infoview called "Content". The splash page contained in **splash.html** is also displayed.

Top level wiring

Now we need to define the top-level structure that a user will see within our "Contents" infoview. We start by creating the following topics file for the top-level topics:

topics_view_Contents.xml

```
<topics id="topics_view_Contents">
  <topic id="conceptsRoot" label="Concepts" />
  <topic id="tasksRoot" label="Tasks" />
  <topic id="funRoot" label="Fun Things" />
  <topic id="refRoot" label="Reference" />
</topics>
```

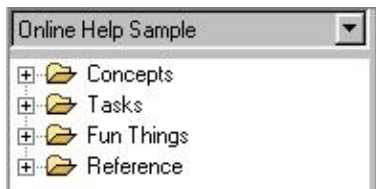
Now we need to "wire in" these topics into our "Contents" infoview. Once we've done that we can then proceed to wire in all of our other topics underneath the above top-level topics. We start wiring in the top-level topics using the following actions file.

actions_view_Contents.xml

```
<actions infoview="com.example.helpexample.view_Contents">
  <insert
    from="com.example.helpexample.topics_view_Contents"
    to="com.example.helpexample.view_Contents"
    as="child"/>
</actions>
```

The fully qualified topic id of the **topics** element "**topics_view_Contents**" in the topics file above is **com.example.helpexample.topics_view_Contents**. In the above actions file we take this **topics** element and wire it into the infoview with id **com.example.helpexample.view_Contents**.

The figure below shows what our infoset (Online Help Sample) will look like as a result of wiring in these top-level topics. The title (label) of the infoset is only displayed in the combo box if there is more than one infoset.



Wiring actions

The actions file contains scripting actions to be performed on topics and infoviews. Currently there is only one kind of action, the **insert** action, which is used to insert topics into infoviews at specified locations called **insertion points**. Infoviews and topics can both be used as an insertion point. Only topics that define an optional id can be used as an insertion point. (Infoviews must always have ids.)

When defining an insert action, the **to** attribute specifies the target insertion point. The topic specified in the **from** attribute is the topic being inserted.

Following are some possible ways to insert a topic:

- As a child of the insertion point (the most common)
- As the first child of the insertion point
- As the last child of the insertion point

- As the previous sibling of the insertion point (just before the insertion point at the same level in the navigation tree)
- As the next sibling of the insertion point (just after the insertion point at the same level in the navigation tree)

Insert actions can be nested to define "alternate" insert actions that should be performed if the parent insert action fails. This can be useful when insertion points are located in other plug-ins that may or may not be installed. You can insert your topic to another plug-in's topic, and if the insert action fails (because the other plug-in is not installed) you can install the topic somewhere else. Once an insertion point has been satisfied, the nested insert actions are ignored.

Actions files always apply their actions to a single infoview. If you want to wire your topics into different infoviews, you will need an actions file for each. A topic may only be used once within a particular infoview. If you need to reference a topic twice within one infoview, you will need to create another topic element (different **id**) that uses the same **label** and **href**.

Integrating the topics

The time has come to finally integrate our topics into the top-level topics of the "Contents" infoview. To do this we need another actions file that we will call **actions_All.xml** (since it integrates all of our topics).

actions_All.xml

```
<actions infoview="com.example.helpexample.view_Contents">
  <insert
    from="com.example.helpexample.conceptsAll"
    to="com.example.helpexample.conceptsRoot"
    as="child"/>
  <insert
    from="com.example.helpexample.refAll"
    to="com.example.helpexample.refRoot"
    as="child"/>
  <insert
    from="com.example.helpexample.plainTasks"
    to="com.example.helpexample.tasksRoot"
    as="child"/>
  <insert
    from="com.example.helpexample.funTasks"
    to="com.example.helpexample.funRoot"
    as="child"/>
</actions>
```

Recall that we had a number of task related html files, and the structure/navigation of these files was defined by **topics_tasks.xml** as follows:

topics_tasks.xml

```
<topics id="tasksAll">
  <topic id="plainTasks" label="Plain Stuff">
    <topic label="Task1" href="doc/tasks/task1.html"/>
    <topic label="Task2" href="doc/tasks/task2.html"/>
  </topic>
  <topic id="funTasks" label="Fun Stuff" >
    <topic label="Task3_1" href="doc/tasks/task3_1.html"/>
    <topic label="Task3_2" href="doc/tasks/task3_2.html"/>
  </topic>
</topics>
```

In the above actions file we use the insert as child action to take the topic with id **com.example.helpexample.plainTasks** and insert it and its subtopics under the topic with id **com.example.helpexample..tasksRoot**

```
<insert
  from="com.example.helpexample.plainTasks"
  to="com.example.helpexample.tasksRoot"
  as="child"/>
```

The actions file also takes some of our more entertaining tasks and inserts them into the "Fun Things" area of the web using the following action.

```
<insert
  from="com.example.helpexample.funTasks"
  to="com.example.helpexample.funRoot"
  as="child"/>
```

Completing the plug-in manifest

We started this example by creating our plug-in and document files. Next we created topic files to describe the structure and navigation of our content. We then moved up one level above content, to the wiring and integration of our book.

The one remaining piece of work is to update our **plugin.xml** to actually contribute the action, topics, and infoSet files that we created. We start by updating the **plugin.xml** to contribute our infoSet:

```
<extension point="org.eclipse.help.contributions">
  <infoSet name="infoSet_SampleGuide.xml" />
</extension>
```

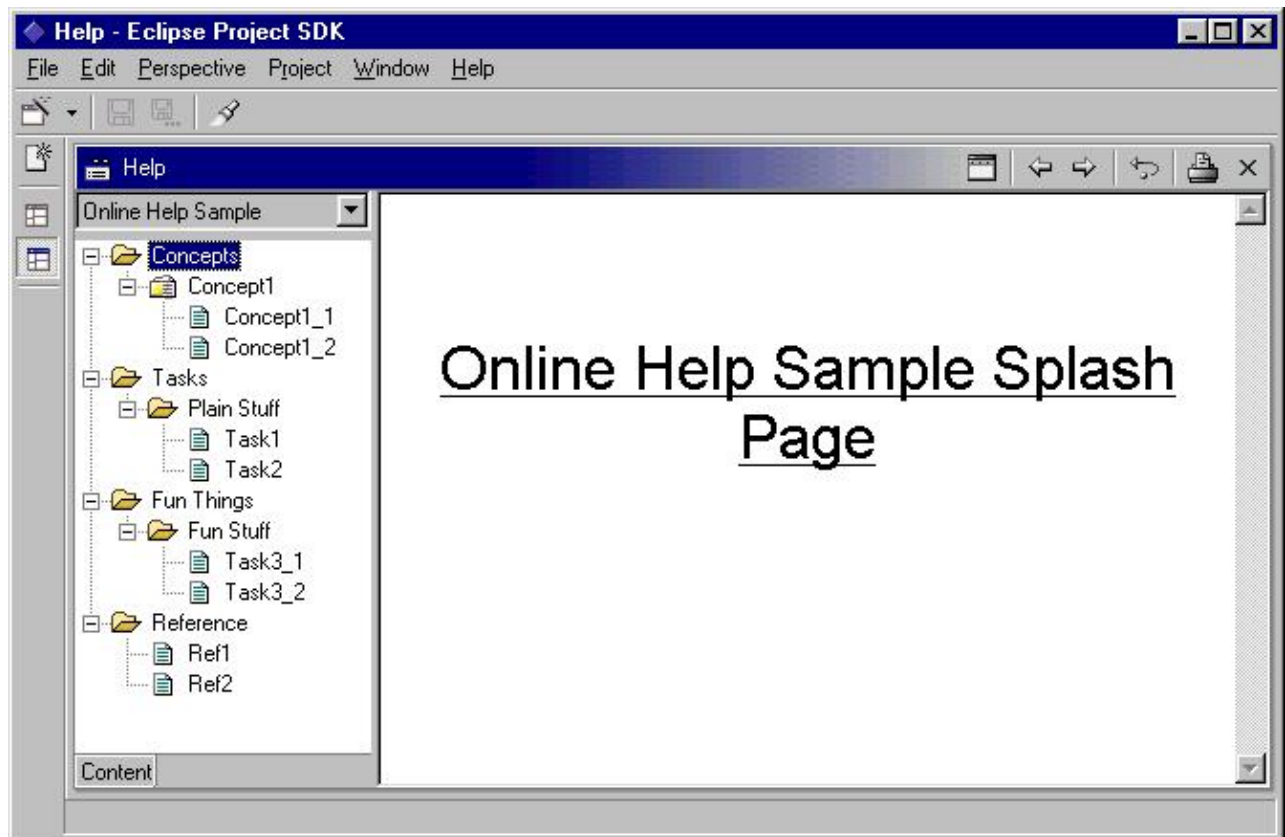
Next we contribute the top-level topics, and the actions file that wires them into our infoSet:

```
<extension point="org.eclipse.help.contributions">
  <topics name="topics_view_Contents.xml" />
  <actions name="actions_view_Contents.xml" />
</extension>
```

Lastly we contribute the bulk of our topics and wire them in:

```
<extension point="org.eclipse.help.contributions">
  <topics name="topics_Concepts.xml" />
  <topics name="topics_Tasks.xml" />
  <topics name="topics_Ref.xml" />
  <actions name="actions_All.xml" />
</extension>
```

That's it. If you copy your plug-in directory to the platform's **plugins** directory, start the platform, and choose **Help->Help Contents**, you should see the following in the "Contents" infoView:



Documentation integration

What if we expect that our plug-in will sometimes be installed by itself, and in other cases it will be installed as part of a larger component or product?

When we are a free floating plug-in we want to ensure that our document infoset is visible in the Help Contents view. When our topics are integrated into a larger web, it probably doesn't make sense for our standalone book to show up anymore. Instead, we may be part of the larger web's infoset.

To support non-integrated or loosely integrated documentation, a plug-in can define its infoset and actions with the "**standalone=true**" attribute. An insert operation will be executed for a standalone action only if the action's topic has not been contributed elsewhere. A standalone infoset will not appear in the help contents view if it is empty. An infoset will not appear in the help contents combo box if all of the following conditions are met:

- the infoset is standalone
- all of the infoset's actions are standalone
- all of the action's topics have been contributed to other info sets

Setting standalone attributes on actions and infosets is useful when providing a "Catch all" scenario. Documentation that doesn't successfully contribute to another infoset will still appear somewhere.

To support this mode of operation, we need to make the following additions to our infoset and action **.xml** files. The addition is marked in bold.

infoset_Guide.xml

```
<infoset id="infoset_SampleGuide" label="Online Help Sample"
  href="doc/splash.html" standalone="true">
```

actions_All.xml

```
<actions infoview="com.example.helpexample.view_Contents" standalone="true">
```

actions_View_Contents.xml

```
<actions infoview="com.example.helpexample.view_Contents" standalone="true">
```

If another plug-in's infoset includes all of our topics, then our actions will not be used, and consequently our infoset will be empty. The infoset will not show up in the help contents view combo box.

Externalizing Strings

We can externalize the strings in our **plugin.xml** files by replacing the string with a key (e.g. **%pluginName**) and creating an entry in the **plugin.properties** file of the form:

```
pluginName = "Online Help Sample Plugin"
```

The contribution XML files are externalized using a similar approach. To externalize

```
<topic id="plainTasks" label="Plain Stuff">
```

we replace its label with a key **%plainStuff**. Our topic now looks like:

```
<topic id="plainTasks" label="%plainStuff">
```

We create an entry in the **doc.properties** file:

```
plainStuff = Plain Stuff
```

The help system will use **document.properties** when looking up strings externalized by online help contributions.

Help server and zip files

The platform utilizes its own documentation server to provide the actual web pages for an infoset. A custom server allows the platform to handle HTML content in a browser independent manner and to provide plug-in aware support. The main difference to you as a plug-in developer is that you have a little more flexibility in the way you structure your files and specify your links.

Documentation can be delivered in a zip file, thus avoiding problems that may result when a large number of files are present. In our example plug-in, we created a subdirectory called **doc**. Alternatively, we could have placed our html files into a zip file called **doc.zip**. This zip file must mimic the file structure underneath the plug-in directory. In our case, it must contain the subdirectory **doc** and all the contents underneath **doc**.

When resolving file names, the help server looks in the **doc.zip** file for documents before it looks in the plug-in directory itself.