

Eclipse Tutorials

Table of Contents

Starting Eclipse.....	1
<u>Arguments for the eclipse command.....</u>	1
<u>Troubleshooting.....</u>	3
<u>A Tour of the Eclipse Workbench.....</u>	4
<u>The Workbench.....</u>	4
<u>Editors and views.....</u>	5
<u>Editors.....</u>	7
<u>Views.....</u>	9
<u>Your first project.....</u>	11
<u>Using the File menu.....</u>	12
<u>Using the pop-up.....</u>	14
<u>Using the New button.....</u>	15
<u>Closing an editor.....</u>	17
<u>Navigating resources.....</u>	18
<u>Opening resources in the Navigator.....</u>	18
<u>Go To.....</u>	19
<u>Go Into.....</u>	19
<u>Where are my files?.....</u>	20
<u>Exporting files.....</u>	20
<u>Importing files.....</u>	22
<u>Deleting resources.....</u>	25
<u>Working with other editors.....</u>	26
<u>External editors.....</u>	26
<u>Editing files outside the Workbench.....</u>	26
<u>Copying, renaming and moving.....</u>	27
<u>Copying.....</u>	28
<u>Renaming.....</u>	28
<u>Moving.....</u>	29
<u>Searching.....</u>	31
<u>Starting a search.....</u>	31
<u>The Search view.....</u>	33
<u>Tasks and markers.....</u>	34
<u>Unassociated tasks.....</u>	35
<u>Associated Tasks.....</u>	36
<u>Opening files.....</u>	38
<u>Bookmarks.....</u>	38
<u>Adding and viewing bookmarks.....</u>	38
<u>Using bookmarks.....</u>	40
<u>Removing bookmarks.....</u>	40
<u>Rearranging views and editors.....</u>	41
<u>Set up.....</u>	41
<u>Drop cursors.....</u>	42
<u>Rearranging Views.....</u>	42
<u>Tiling Editors.....</u>	43
<u>Rearranging tabbed views.....</u>	44
<u>Maximizing.....</u>	44

Table of Contents

A Tour of the Eclipse Workbench

<u>Fast views</u>	45
<u>Creating fast views</u>	45
<u>Working with fast views</u>	46
<u>Perspectives</u>	47
<u>New perspectives</u>	48
<u>New windows</u>	50
<u>Saving perspectives</u>	50
<u>Configuring perspectives</u>	52
<u>Comparing</u>	53
<u>Simple compare</u>	54
<u>Understanding the comparison</u>	55
<u>Working with the comparison</u>	56
<u>Local history</u>	58
<u>Responsive UI</u>	59
<u>Exiting the Workbench</u>	61

Team CVS tutorial.....62

<u>Setting up a CVS repository</u>	62
<u>Starting offline</u>	62
<u>Sharing your project</u>	63
<u>Specifying a repository location</u>	63
<u>What is a repository location?</u>	65
<u>Synchronizing your project</u>	65
<u>Working with another user</u>	68
<u>CVS terminology</u>	69
<u>Checking out a project</u>	69
<u>Another user making changes</u>	72
<u>Making your own changes</u>	74
<u>Working with conflicting changes</u>	76
<u>Replacing</u>	78
<u>Versioning your project</u>	79
<u>A quick review</u>	79

Tutorial for Ant and external tools.....80

<u>The basics of working with Ant buildfiles in Eclipse</u>	80
<u>Creating Ant buildfiles</u>	80
<u>Editing Ant buildfiles</u>	80
<u>Running Ant buildfiles</u>	82
<u>Saving and reusing Ant options</u>	83
<u>Using the Ant view</u>	85
<u>Use cases for Ant in Eclipse</u>	86
<u>Deploying Eclipse plug-ins</u>	86
<u>Ant buildfiles as project builders</u>	91
<u>External tools</u>	96
<u>Non-Ant project builders</u>	96
<u>Stand-alone external tools</u>	98

Table of Contents

<u>CDT "Managed Make" Tutorial</u>	100
<u>Open the C/C++ Perspective</u>	101
<u>C/C++ Perspective</u>	102
<u>Create a project</u>	103
<u>New Project Wizard</u>	104
<u>New Project</u>	105
<u>Select a Target</u>	106
<u>Error Parser Settings</u>	107
<u>Additional Project Settings</u>	108
<u>New Project</u>	109
<u>Create a file</u>	110
<u>Name the new file</u>	111
<u>New project file</u>	112
<u>Enter code</u>	113
<u>Step through the code</u>	114
<u>Save the file</u>	115
<u>View Executable</u>	116
<u>Run the application</u>	117
<u>Run Configuration</u>	118
<u>Program Selection</u>	119
<u>Run Configuration</u>	120
<u>Console View</u>	121
<u>Run complete</u>	122
<u>CDT "Standard Make" Tutorial</u>	123
<u>Open the C/C++ Perspective</u>	124
<u>C/C++ Perspective</u>	125
<u>Create a project</u>	126
<u>New Project Wizard</u>	127
<u>New Project</u>	128
<u>Project References</u>	129
<u>Make Builder Settings</u>	130
<u>Error Parsers</u>	131
<u>Binary Parser Settings</u>	132
<u>Discovery Options</u>	133
<u>C/C++ Indexer</u>	134
<u>New Project</u>	135
<u>Create a makefile</u>	136
<u>Create a makefile</u>	137
<u>New Makefile</u>	138
<u>Enter the make script</u>	139
<u>Create a CPP file</u>	140
<u>Create a CPP file</u>	141
<u>New Project Files</u>	142
<u>Enter code</u>	143
<u>Build the project</u>	144
<u>Run the application</u>	145

Table of Contents

CDT "Standard Make" Tutorial

<u>Run Configuration</u>	146
<u>Run Configuration</u>	147
<u>Program Selection</u>	148
<u>Console View</u>	149
<u>Run complete</u>	150

A Tour of the Java Development Toolkit (JDT).....151

<u>Preparing the workbench</u>	151
<u>Verifying JRE installation and classpath variables</u>	151
<u>Creating your first Java project</u>	152
<u>Getting the Sample Code (JUnit)</u>	152
<u>Creating the project</u>	152
<u>Browsing Java elements using the Package Explorer</u>	155
<u>Opening a Java editor</u>	157
<u>Adding new methods</u>	159
<u>Using content assist</u>	161
<u>Identifying problems in your code</u>	162
<u>Using source code templates</u>	165
<u>Organizing import statements</u>	167
<u>Using the local history</u>	168
<u>Extract a new method</u>	170
<u>Creating a Java class</u>	174
<u>Renaming Java elements</u>	181
<u>Moving Java elements</u>	184
<u>Navigate to a Java element's declaration</u>	185
<u>Viewing the type hierarchy</u>	188
<u>Searching the workbench</u>	194
<u>Performing a JDT search</u>	194
<u>Searching from a Java view</u>	196
<u>Searching from an editor</u>	197
<u>Continuing a search from the Search view</u>	198
<u>Performing a file search</u>	200
<u>Viewing previous search results</u>	201
<u>Running your programs</u>	202
<u>Debugging your programs</u>	205
<u>Evaluate expressions</u>	210
<u>Evaluating snippets</u>	211
<u>Using the Java browsing perspective</u>	214
<u>Writing and running JUnit tests</u>	216
<u>Writing Tests</u>	216
<u>Running Tests</u>	217
<u>Customizing a Test Configuration</u>	219
<u>Debugging a Test Failure</u>	220
<u>Creating a Test Suite</u>	220

Table of Contents

<u>Project configuration tutorial</u>	222
<u>Detecting existing layout</u>	222
<u>Layout on file system</u>	222
<u>Steps for defining a corresponding project</u>	222
<u>Organizing sources</u>	225
<u>Layout on file system</u>	225
<u>Steps for defining a corresponding project</u>	226
<u>Sibling products in a common source tree</u>	230
<u>Layout on the file system</u>	230
<u>Steps for defining corresponding projects</u>	231
<u>Overlapping products in a common source tree</u>	235
<u>Layout on file system</u>	235
<u>Steps for defining corresponding "Product1" and "Product2" projects</u>	235
<u>Product with nested tests</u>	239
<u>Layout on file system</u>	239
<u>Steps for defining a corresponding project</u>	240
<u>Products sharing a common source framework</u>	243
<u>Layout on file system</u>	243
<u>Steps for defining corresponding projects</u>	244
<u>Product nesting resources in an output directory</u>	248
<u>Layout of the file system</u>	248
<u>Defining a corresponding project</u>	249

Starting Eclipse

The `eclipse` command launches the Eclipse Integrated Development Environment. The full command is:

```
eclipse [ -vm [platform options] [-vmargs [Java VM arguments]] ]
```

For example:

```
eclipse -vm /opt/IBMJava2-141/bin/java
```

You can use the `-vm` argument to explicitly specify the Java Virtual Machine (JVM) that you want to use; if you do not use the `-vm` argument, Eclipse uses the first Java VM found on your `$PATH`.

Note: You may want to explicitly specify the JVM because the installation of other products may change your path, resulting in a different Java VM being used when you next launch Eclipse.

Arguments for the eclipse command

All arguments following (but not including) the `-vm args` entry are passed directly through to the indicated Java VM as virtual machine arguments (that is, before the class to run).

`-application applicationId`

The application to run. Applications are declared by plug-ins supplying extensions to the `org.eclipse.core.runtime.applications` extension point. This argument is typically not needed. If specified, the value overrides the value supplied by the configuration. If not specified, the Eclipse Workbench is run.

`-arch architecture`

Defines the processor architecture on which the Eclipse platform is running. The Eclipse platform ordinarily computes the optimal setting using the prevailing value of `Java os.arch` property. If specified here, this is the value that the Eclipse platform uses. The value specified here is available to plug-ins as `BootLoader.getOSArch()`. Example values: "x86", "ppc".

`-classloaderproperties [file]`

Activates platform class loader enhancements using the class loader's properties file at the given location, if specified. The file argument can be either a file path or a URL. Note that relative URLs are not allowed.

`-configuration configurationFileURL`

The location for the Eclipse platform configuration file, expressed as a URL. The configuration file determines the location of the Eclipse platform, the set of available plug-ins, and the primary feature. Note that relative URLs are not allowed. The configuration file is written to this location when the Eclipse platform is installed or updated.

`-consolelog`

Mirrors the Eclipse platform's error log to the console used to run Eclipse. Handy when combined with `-debug`.

`-data workspacePath`

The path of the workspace on which to run the Eclipse platform. The workspace location is also the default location for projects. Relative paths are interpreted relative to the directory that Eclipse was started from.

`-debug [optionsFileURL]`

- Puts the platform in debug mode and loads the debug options from the file at the given URL, if specified. This file indicates which debug points are available for a plug-in and whether or not they are enabled. If a file path is not given, the platform looks in the directory that eclipse was started from for a file called `.options`. Note that relative URLs are not allowed.
- `-dev [classpathEntries]`
Puts the platform in development mode. The optional classpath entries (a comma-separated list) are added to the runtime classpath of each plug-in. For example, when the workspace contains plug-ins being developed, specifying `-dev bin` adds a classpath entry for each plug-in project's directory named `bin`, allowing freshly-generated class files to be found there. Redundant or non-existent classpath entries are eliminated.
- `-endsplash params`
Internal option for taking down the splash screen when the Eclipse platform is up and running. This option has different syntax and semantics at various points along the splash screen processing chain.
- `-feature featureId`
The ID of the primary feature. The primary feature gives the launched instance of Eclipse its product personality, and determines the product customization information used.
- `-keyring keyringFilePath`
The location of the authorization database (or "key ring" file) on disk. This argument must be used in conjunction with the `-password` option. Relative paths are interpreted relative to the directory that Eclipse was started from.
- `-nl locale`
Defines the name of the locale on which the Eclipse platform is running. The Eclipse platform ordinarily computes the optimal setting automatically. If specified here, this is the value that the Eclipse platform uses. The value specified here is available to plug-ins as `BootLoader.getNL()`. Example values: "en_US" and "fr_FR_EURO".
- `-nolazyregistrycacheloading`
Deactivates the cache-loading optimization of the platform plug-in registry. By default, extensions' configuration elements will be loaded from the registry cache (when available) only on demand, reducing memory footprint. This option forces the registry cache to be fully loaded at startup.
- `-nosplash`
Runs the platform without putting up the splash screen.
- `-noregistrycache`
Bypasses the reading and writing of an internal plug-in registry cache file.
- `-os operatingSystem`
Defines the operating system on which the Eclipse platform is running. The Eclipse platform ordinarily computes the optimal setting using the prevailing value of Java `os.name` property. If specified here, this is the value that the Eclipse platform uses. The value specified here is available to plug-ins as `BootLoader.getOS()`, and used to resolve occurrences of the `os` variable in paths mentioned in the plug-in manifest file. Example values: "win32", "linux", "hpux", "solaris", "aix".
- `-password password`
The password for the authorization database. Used in conjunction with the `-keyring` option.
- `-plugincustomization propertiesFile`
The location of a properties file containing default settings for plug-in preferences. These default settings override default settings specified in the primary feature. Relative paths are interpreted relative to the directory that eclipse was started from.
- `-showsplash params`
Internal option for showing the splash screen (done by the executable Eclipse platform launcher). This option has different syntax and semantics at various points along the splash screen processing chain.
- `-vm vmPath`

Eclipse Tutorials

The location of Java Runtime Environment (JRE) to use to run the Eclipse platform. If not specified, the JRE is at `jre`, sibling of the Eclipse executable. Relative paths are interpreted relative to the directory that eclipse was started from.

The default VM settings for IBM Developer Kit, Java(TM) Technology Edition 1.3 Linux work well for initial exploration, but are not sufficient for larger scale development. For large-scale development you should modify your VM arguments to make more heap available. For example, the following setting will allow the Java heap to grow to 256MB:

```
-vmargs -Xmx256M  
-ws windowSystem
```

Defines the window system on which the Eclipse platform is running. The Eclipse platform ordinarily computes the optimal setting using the prevailing value of Java `os.name` property. If specified here, this is the value that the Eclipse platform uses. The value specified here is available to plug-ins as `BootLoader.getWS()`, used to configure SWT, and used to resolve occurrences of the `$ws` variable in paths mentioned in the plug-in manifest file. Example values: "win32", "motif", "gtk".

Troubleshooting

Could not restore workbench layout.

This error can appear when Eclipse is terminated abruptly (as during a loss of power). Typically you need to manually open the perspectives and views that you want.

A Tour of the Eclipse Workbench

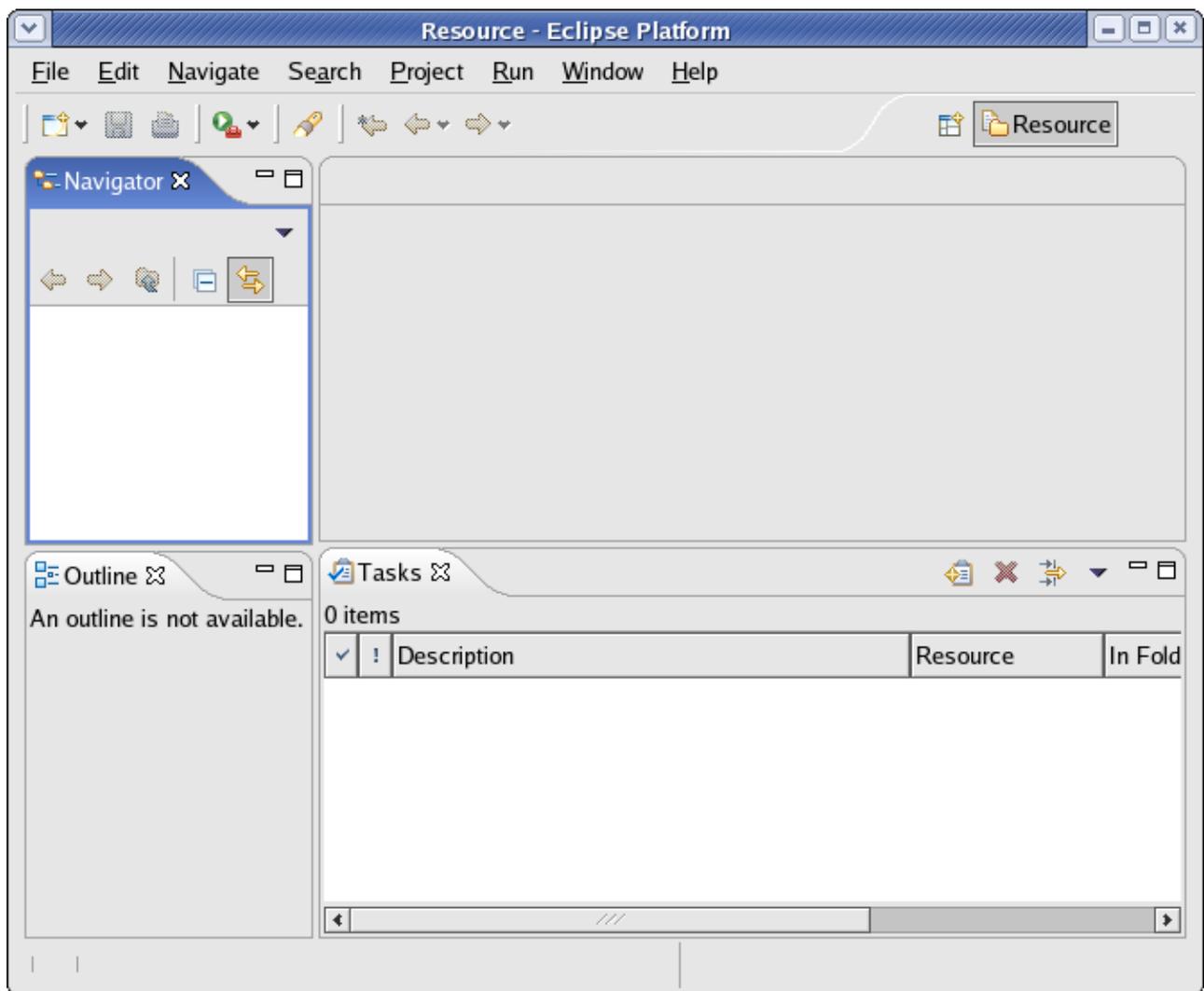
This tutorial provides a step-by-step walk-through of the features in the Eclipse Workbench. You will learn how to use these features while creating a project, then creating, running, and debugging a simple program.

The Workbench

When you launch the Workbench the first thing you see is a dialog box in which you to select where to locate your *workspace*. The workspace is the directory where your work will be stored. For now, just click OK to use the default location.

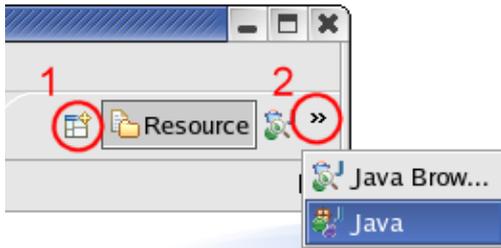
After the workspace location is chosen, the Workbench appears displaying a Welcome page. Click the arrow labeled Workbench to open the Eclipse integrated development environment.

Note: You can get the Welcome page back at any time by selecting **Help > Welcome**.



The Workbench window displays a *perspective*. A perspective contains views, such as the Navigator, and editors. Initially, the Resource perspective is displayed. More than one Workbench window can be open at one time.

A shortcut bar is in the top right corner of the window. The name of the active perspective is shown in the shortcut bar. To the left of the name is an icon (1) that enables you to open new perspectives; if other perspectives are open but inactive, a double arrow to the right of the active perspective's name (2) enables you to switch to any of those perspectives.



Looking at the titlebar of the Workbench window you can see that you are working with the Resource perspective. The Navigator, Tasks, and Outline views are open, along with an editor on the welcome page. You might want to take a moment to look over the Welcome page. It provides a quick starting point if you are eager to customize your Workbench or learn about other features.

Editors and views

Before using the Workbench, start by familiarizing yourself with the elements of the Workbench. A Workbench consists of:

- Perspectives
- Views
- Editors.

A *perspective* is a group of views and editors in the Workbench window. One or more perspectives can exist in a single Workbench window. Each perspective contains one or more views and editors. Within a window, each perspective may have a different set of views but all perspectives share the same set of editors.

A *view* is a visual component within the Workbench. It is typically used to display a hierarchy of information (such as the resources in your Workbench), an editor, or the properties of a resource. Modifications made in a view are saved immediately. Only one instance of a particular view type can exist within a Workbench window.

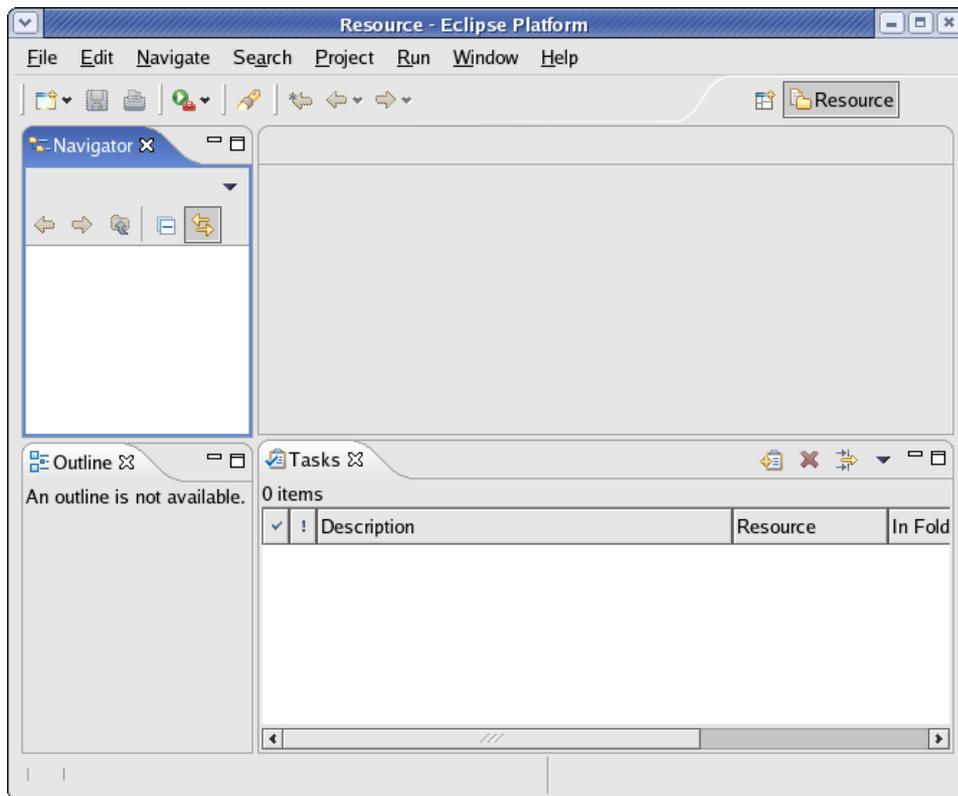
An *editor* is a component within the Workbench that you use to edit or browse a resource. Modifications made in an editor follow an open–save–close lifecycle model. Multiple instances of editors can run concurrently within a Workbench window.

Editors and views can be active or inactive, but only one view or editor can be active at any one time. The active editor or view is the one whose title bar is highlighted. The active part is the target for common operations like cut, copy and paste. The active part also determines the contents of the status line. If an editor tab is white, it indicates the editor is not active. However, views may show information based on the last

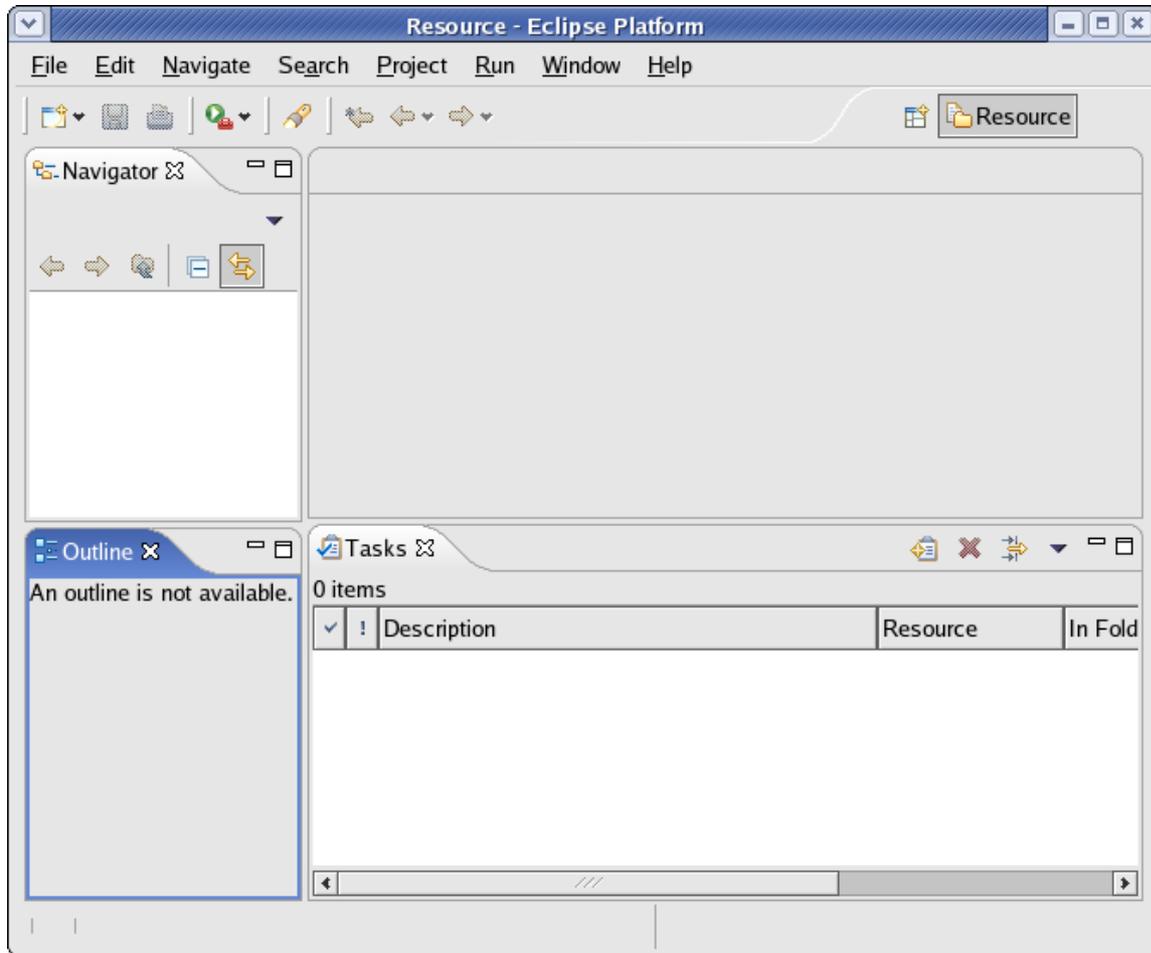
Eclipse Tutorials

active editor.

If you click on the Navigator view, you will notice it becomes active.

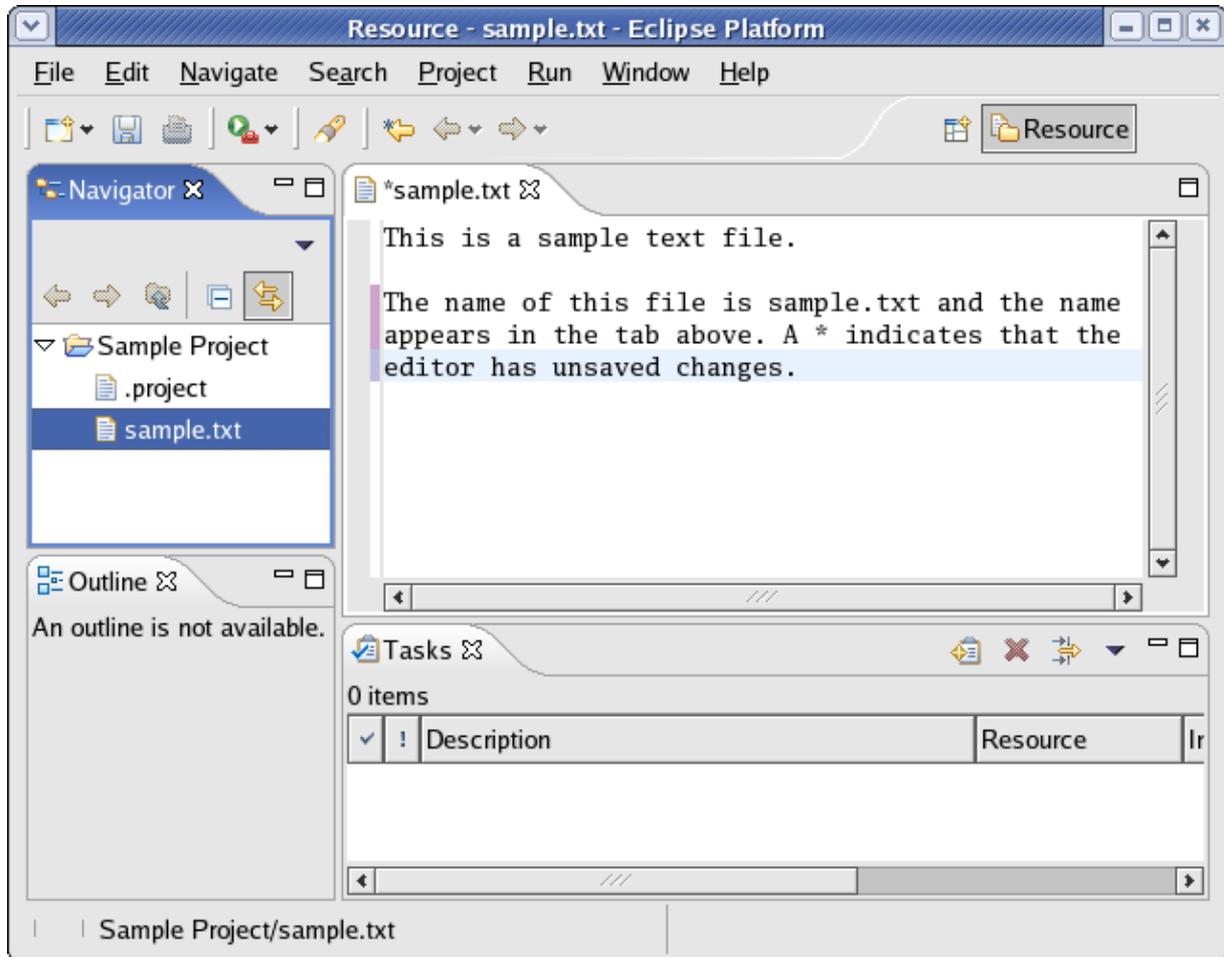


Clicking on the Outline view causes the its title bar to turn blue and the Navigator title bar to no longer be blue, as shown below. The Outline view is now active.



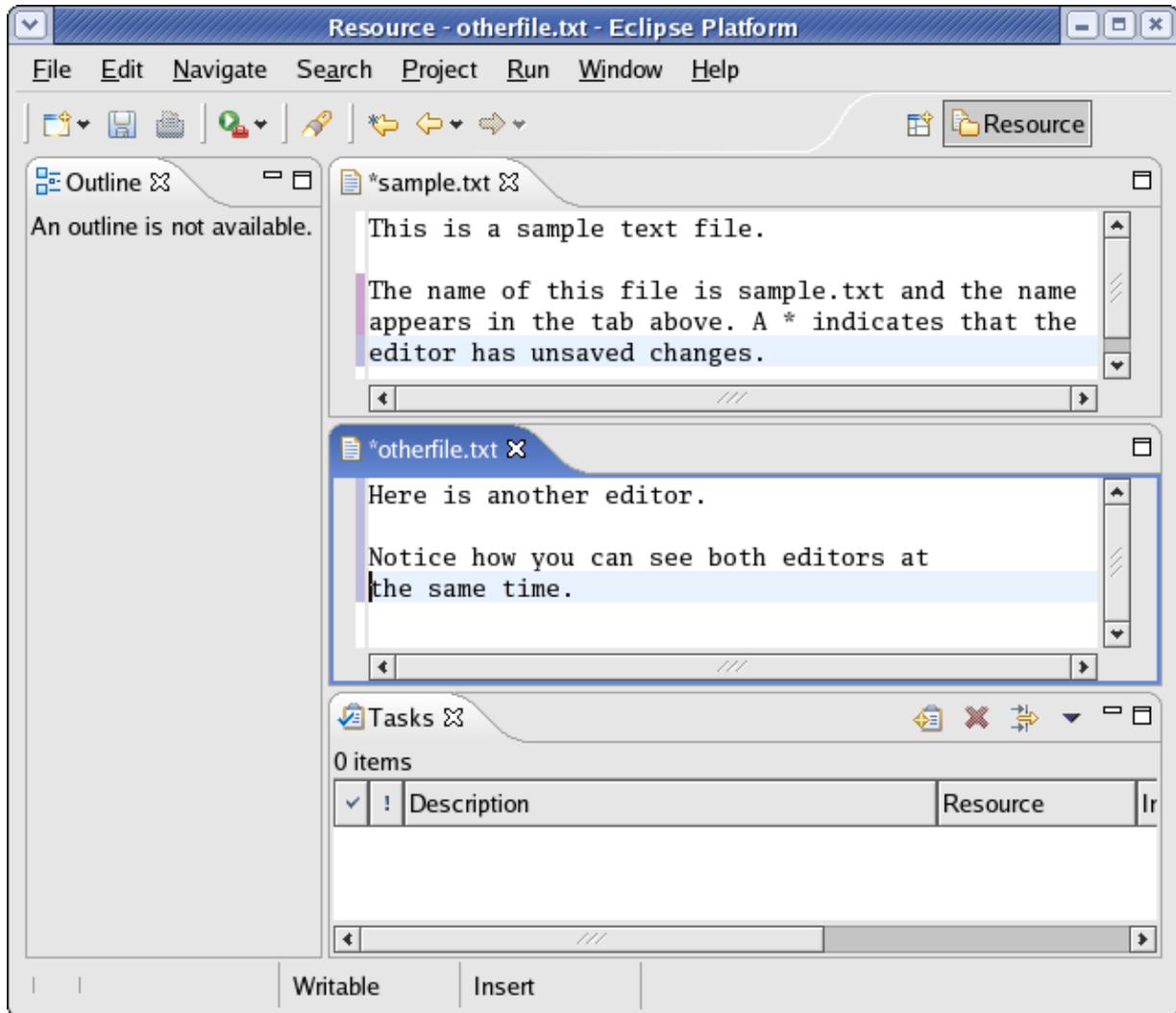
Editors

Depending on the type of file you are editing, the appropriate editor is displayed in the editor area. For example, if you are editing a .txt file, a text editor displays in the editor area. The figure below shows an editor open on the file sample.txt. The name of the file appears in the tab of the editor. An asterisk (*) at the left side of the tab indicates that the editor has unsaved changes. If you attempt to close the editor or exit the Workbench with unsaved changes, you will be prompted to save your editor's changes.



When an editor is active, the Workbench menubar and toolbar contain operations applicable to the editor. When a view becomes active, the editor operations are disabled. However, in some cases, certain editor functions may be appropriate in the context of a view and will remain enabled.

You can stack editors in the editor area and activate individual editors by clicking the tab for the editor. You can also tile editors side-by-side in the editor area so their content can be viewed simultaneously. In the figure below, editors for sample.txt and otherFile.txt have been placed one above the other. You will learn how you can rearrange views and editors in [Rearranging views and editors](#).



If you open a resource that does not have an associated editor, the Workbench attempts to launch an external editor registered with your platform. These external editors are not tightly integrated with the Workbench and are not embedded into the Workbench's editor area.

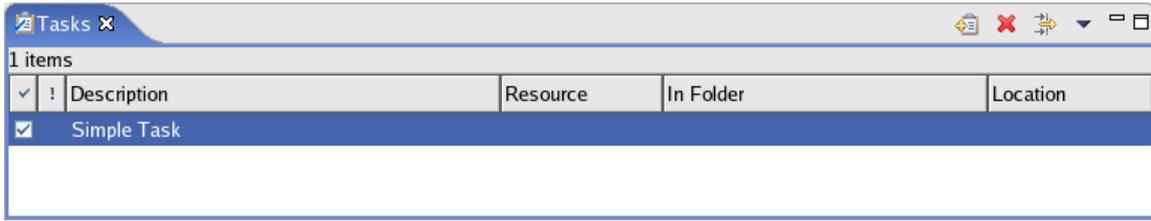
You can cycle through the active editors using the back and forward arrow buttons in the toolbar or by using [Ctrl]+[F6]. The arrow buttons move through the last mouse selection points and permit moving through several points in a file before moving to another one. [Ctrl]+[F6] pops up a list of currently selected editors; by default, the editor used before the current one is selected.

Views

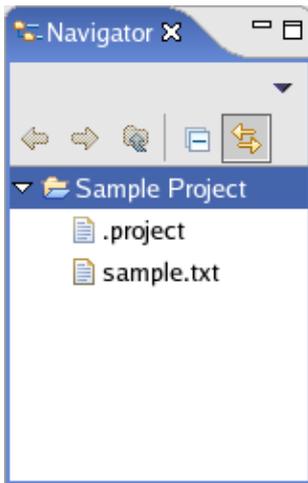
Views support editors and provide alternative presentations or ways to navigate the information in your Workbench. For example:

- The Bookmarks view displays all bookmarks in the Workbench along with the names of the files with which the bookmarks are associated.
- The Navigator view displays the projects and other resources that you are working with.

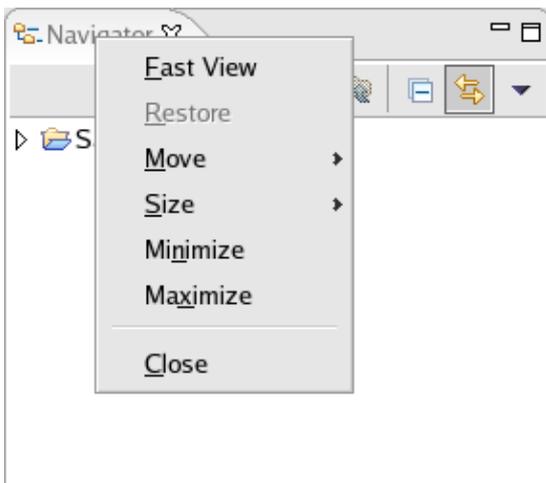
A view might appear by itself...



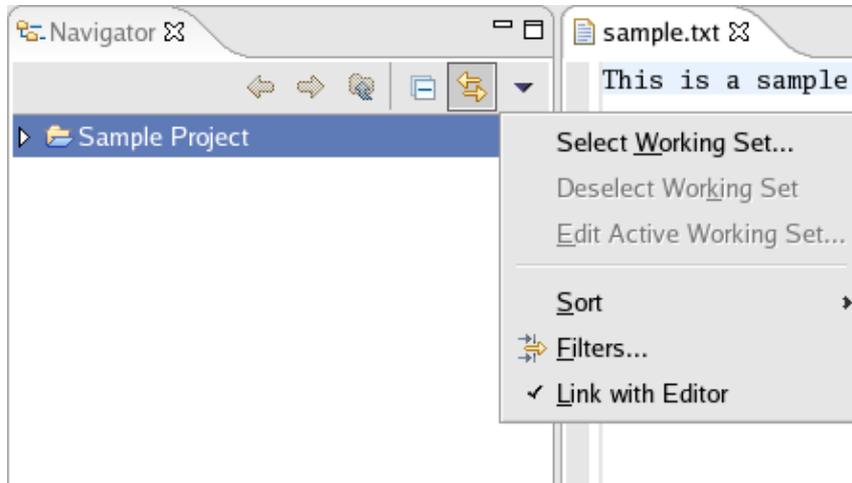
... or stacked with other views in a tabbed notebook. To activate a view that is part of a tabbed notebook simply click on its tab. As you will soon discover, the Workbench provides a number of quick and easy ways to configure your environment, including whether the tabs are at the bottom or top of your notebooks.



Views have two menus. The first menu is accessed by clicking on the icon on the left of the view's tool bar. This menu enables you to manipulate the view in much the same manner as the menu associated with the Workbench window.

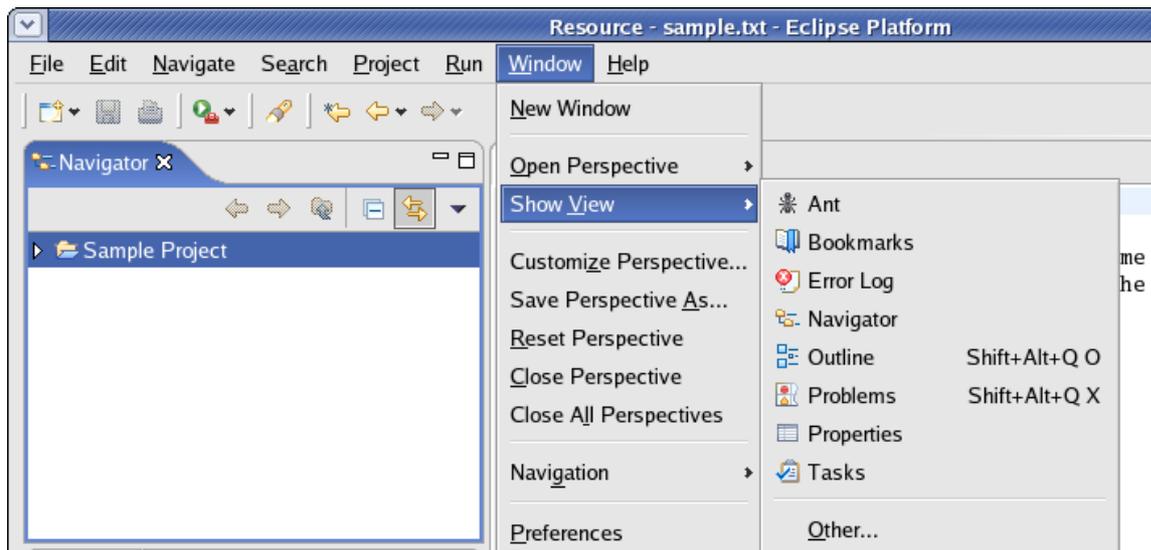


The second menu is accessed by clicking on the down arrow \blacktriangledown . The view pull-down menu typically contains operations that apply to the entire contents of the view, but not to a specific item shown in the view. Operations for sorting and filtering are commonly found on the view pull-down.



In the event you decided to experiment by rearranging your Workbench, this is probably a good opportunity to use the Window > Reset Perspective menu operation. The reset operation restores the layout to its original state.

You can display a view by selecting it from the Window > Show View menu. A perspective determines the views you are likely to need and shows these on the Show View submenu. Additional views are available by choosing Other at the bottom of the Show View submenu. As you will see later, this is just one of the many features that enables you to build your own custom work environment.



Your first project

Now that you have familiarized yourself with the basic elements of the Workbench, you can create your first project. In the Workbench, all resources reside in projects. Projects can contain folders and/or individual files.

New projects, folders, and files can be created using any one of several different approaches. In this section you will create resources using three different approaches:

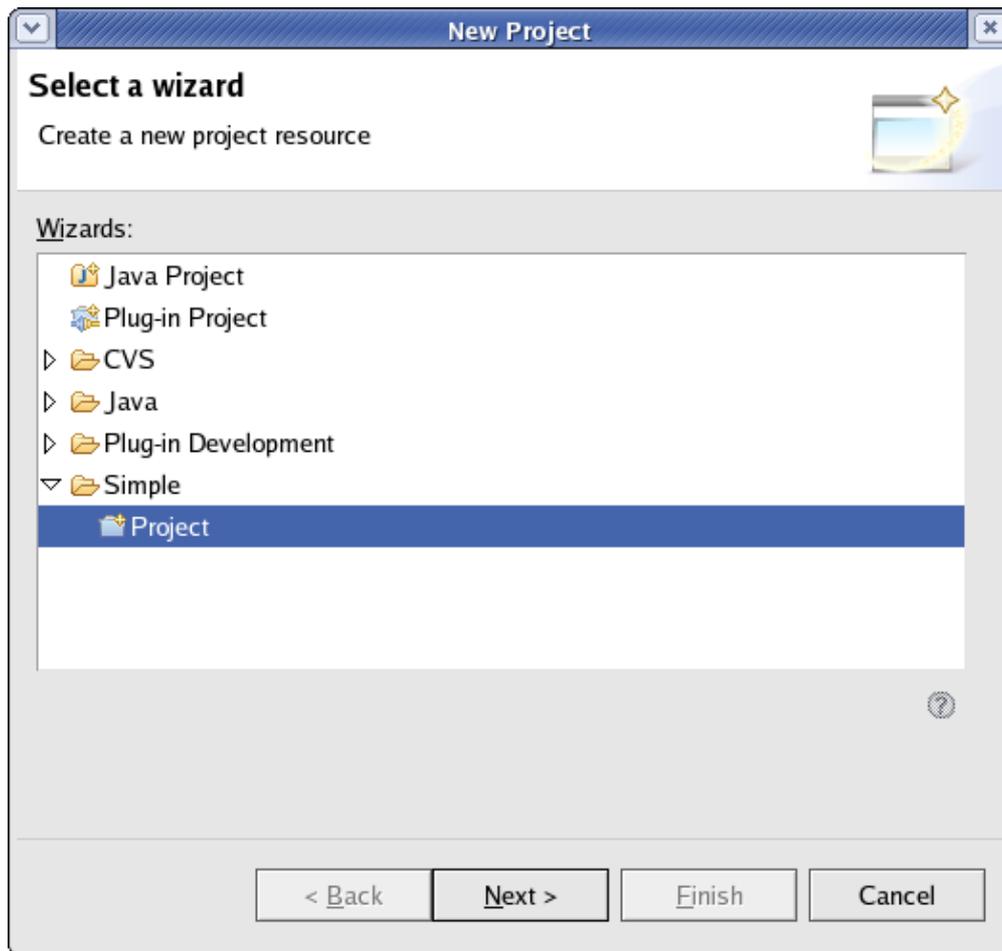
1. File menu
2. Navigator context menu
3. New Wizard button.

Start by creating a project using the File menu. Once you have a project, you can create folders and files.

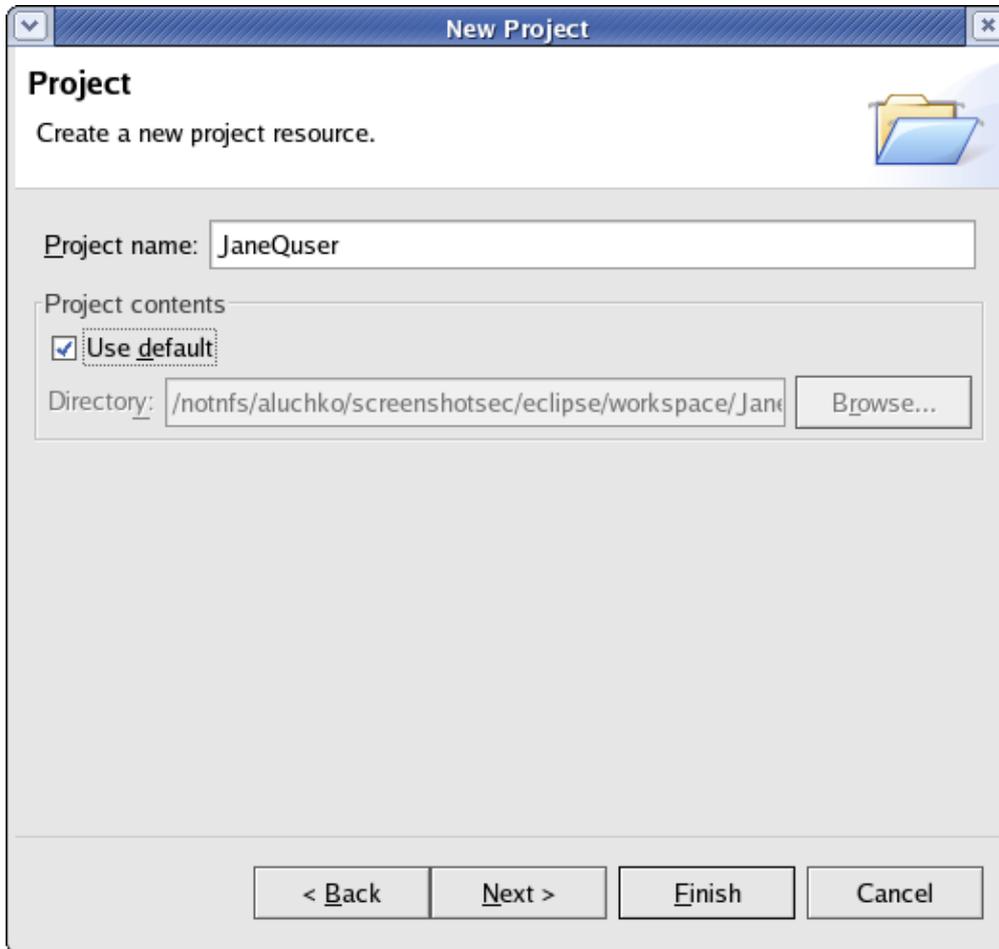
Using the File menu

You can create new resources by using the File > New menu on the Workbench menubar. Start by creating a simple project as follows:

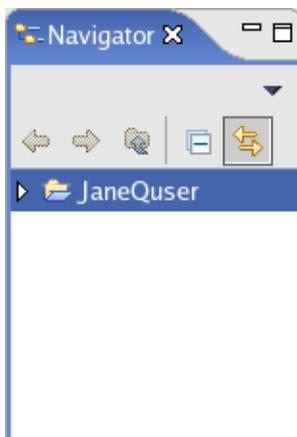
1. From the menu bar, select File > New > Project.



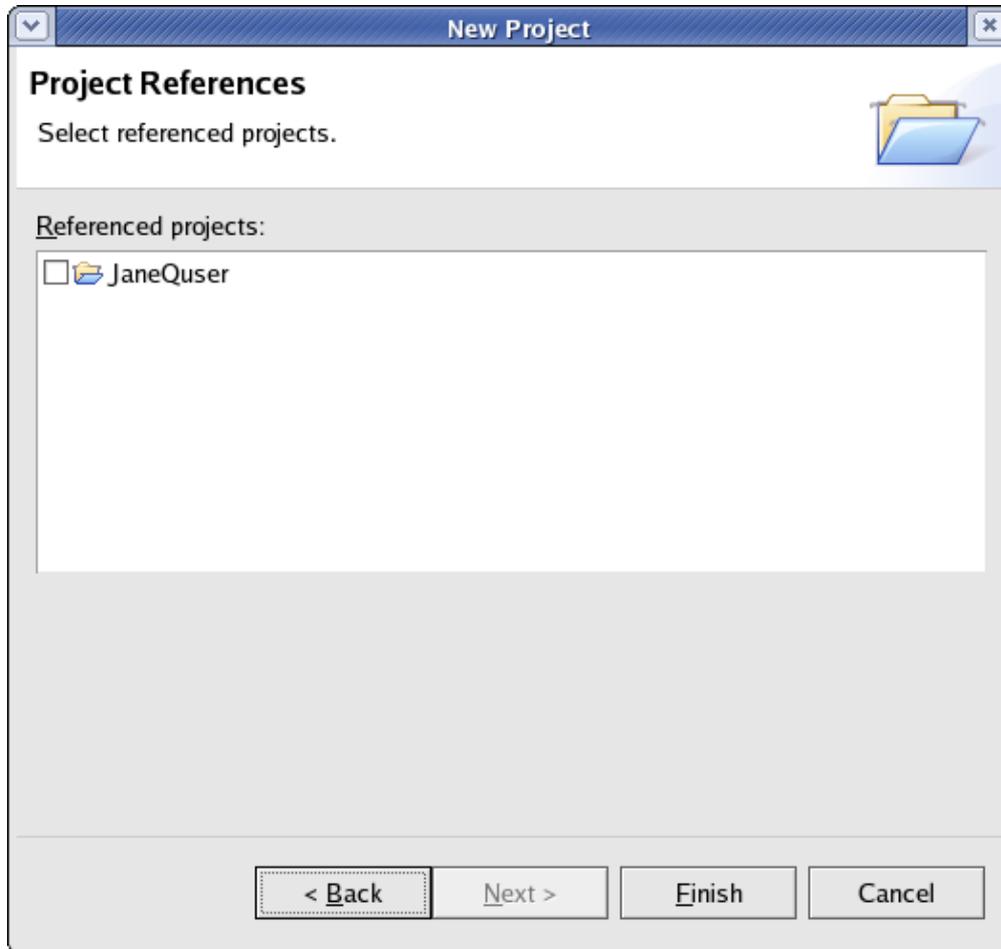
2. In the left pane, select Simple, and on the right, select Project; then click Next.
3. In the Project name field, type your name as the name of your new project (for example, **JaneQuser**). Do not use spaces or special characters in the project name.
4. Leave the box checked to use the default location for your new project. Click Finish when you are done.



In the Navigator view you will see the simple project you just created.



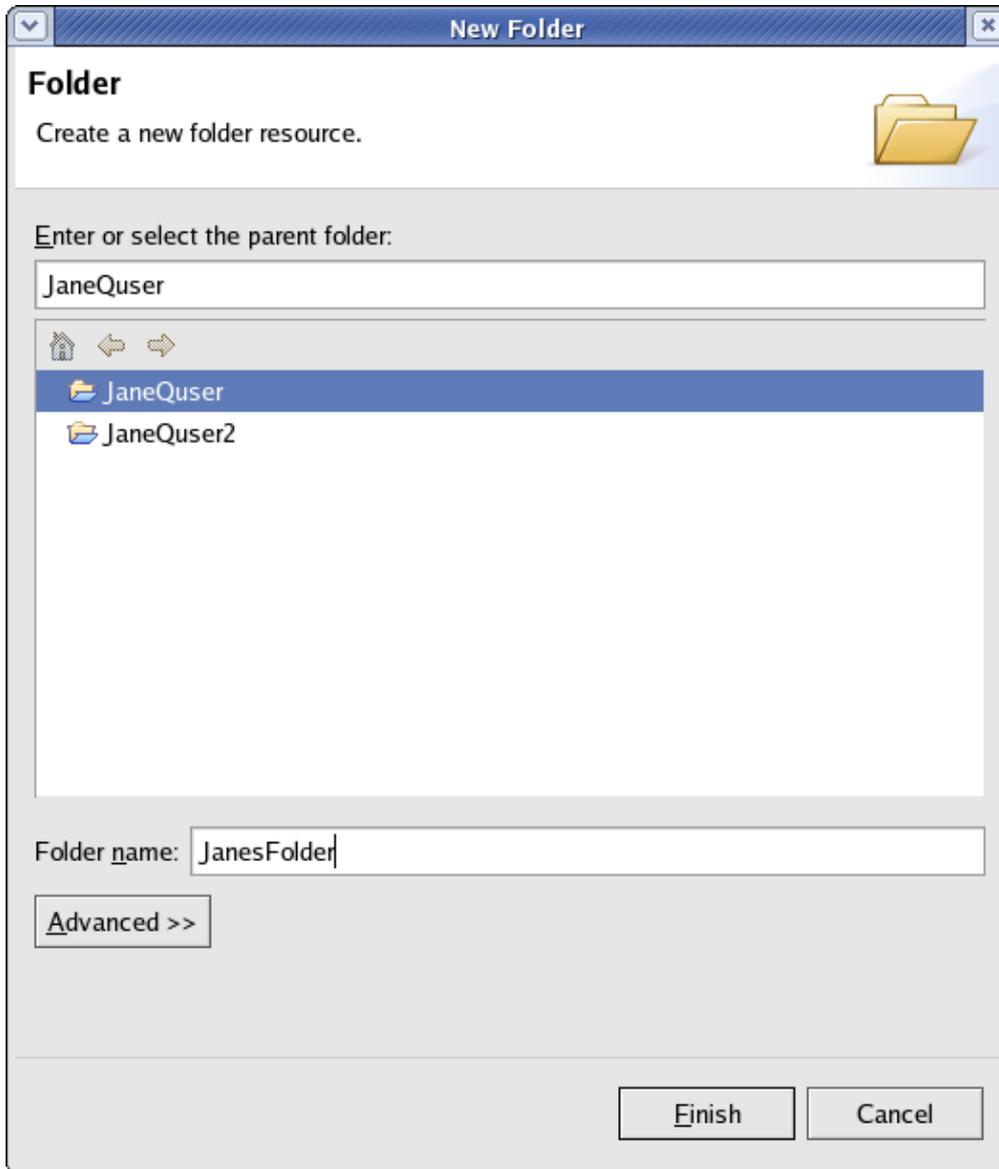
Create a second project called JaneQuser2 using the same steps, but instead of clicking Finish click Next. At this point you can specify other projects that project JaneQuser2 depends on. Because you want to create two independent projects, you will not select any of the projects in the Referenced projects table. Click Finish to create your second simple project.



Using the pop-up

Now that you have your project, you will create a folder using the Navigator view's pop-up menu.

1. Activate the Navigator view and select the project JaneQuser (the first project you created in the Navigator view). Right-click on the project and from the view's pop-up menu, choose New > Folder.
2. In the New Folder wizard, your project name appears by default in the Enter or select the folder field. (This is because you chose to create the new folder from the project's context menu.)
3. In the Folder name field, type a unique name for your new folder (for example, using your first and last names in the title). Do not use spaces or special characters in the folder name (for example, "JanesFolder").



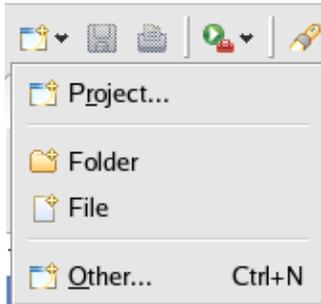
4. Click Finish when you are done. The Navigator view updates to show your newly created folder.

Note: Click the Advanced button to enter the location of a project's folder outside of the project's hierarchy. This is called a *linked folder*.

Using the New button

You have seen how to create resources using File > New and New from the Navigator view's context menu. You will now create a file for your project using the third alternative, the New button.

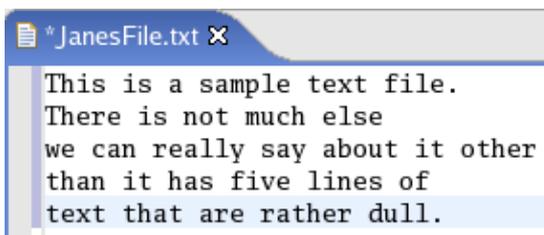
1. Select the folder JanesFolder in the Navigator view.
2. In the Workbench window's toolbar, activate the drop-down menu on the New Wizard button  and select File. To activate the drop-down menu, click on the down arrow .



3. In the New File wizard, your folder's name and path already appear by default in the Enter or select the folder field. This is because you chose to create the new file while this folder was selected in the Navigator view and the view was active.
4. In the File name field, type a unique name for a new text file, including the .txt file extension (for example, **JanesFile.txt**). Do not use spaces or special characters in this file name.
5. Click Finish when you are done.
6. The Workbench has an editor capable of editing text files. A text editor is automatically opened on the newly created file.
7. In the text editor, type in the following five lines:

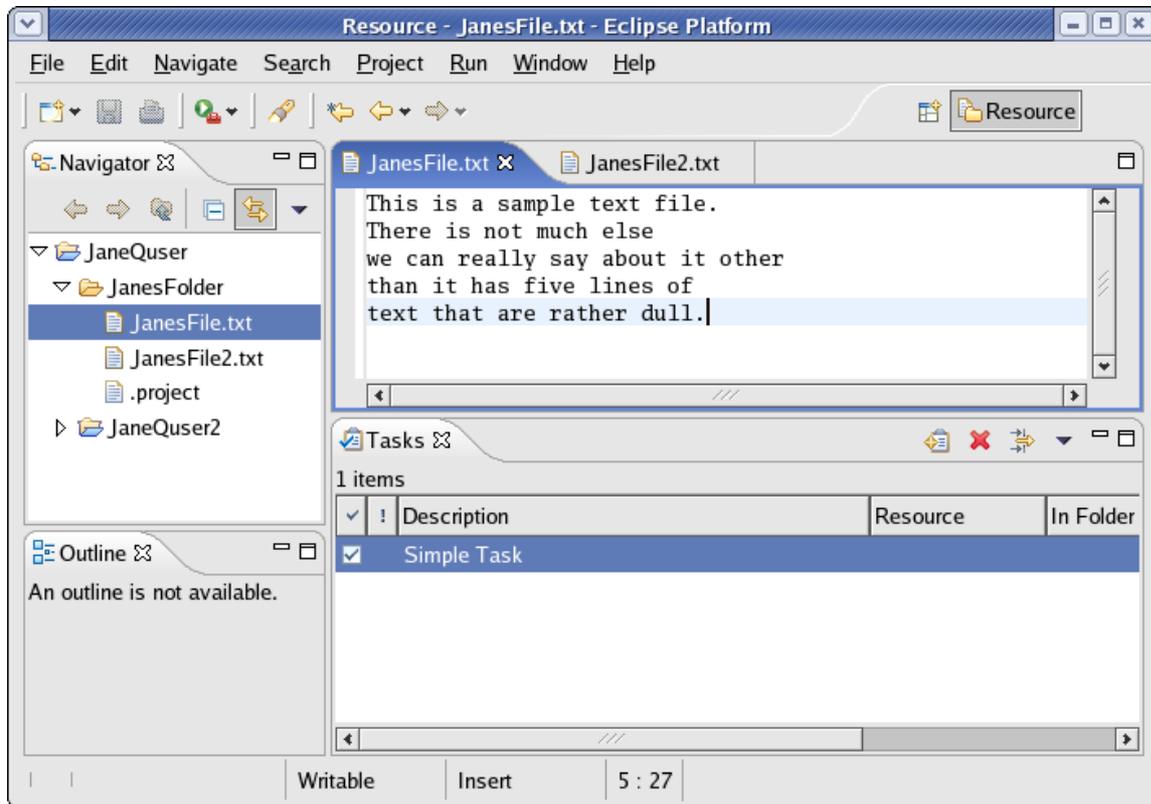
```
This is a sample text file.
There is not much else
we can really say about it other
than it has five lines of
text that are rather dull.
```

Notice that the editor tab has an asterisk (*) at the left of the filename. The asterisk indicates that the editor has unsaved changes.



8. In the Workbench window's toolbar, click the Save button  to save your work.
9. In the Navigator view ensure your folder is still selected.
10. Click the New Wizard button in the Workbench toolbar. Previously you clicked on the drop-down arrow of the New button. Here you click on the button itself, which has the same effect as choosing File > New > Other.
11. In the New wizard, click Simple in the left pane, and in the right pane, click File. Then click Next.
12. Once again, your folder's name and path appears by default in the Enter or select the folder field.
13. In the File name field, type a unique name for a new file (for example, "JanesFile2.txt"). Do not use any spaces or special characters in the file name. Click Finish when you are done.

Now that you have created your resources, the Navigator view shows your two projects, the folder, and its two files. To the right of the Navigator view a text editor is open (JanesFile.txt). You can also see a second, inactive editor tab for JanesFile2.txt



Click on the JanesFile.txt editor tab. Now select the file JanesFile2.txt in the Navigator view. Then select the Link with Editor button  on the Navigator view. Lastly, click on the editor tab for JanesFile.txt. Notice how the Navigator updated itself to select the file you are currently editing (JanesFile2.txt). If you do not like to have the file selected in the Navigator take the focus in the editor, you can easily turn this function off by deselecting the Link with Editor button.

Closing an editor

To close an editor:

1. Select the JanesFile.txt editor tab.
2. In the text area add a 6th line of text:

This is a 6th line
3. To close the editor, do one of the following:
 - ◆ Click the close button ("X")  in the tab of any open editor.
 - ◆ Choose File > Close from the menu bar.

Note that you are prompted to save the file before the editor is closed.
4. Click OK to save your changes and close the editor.

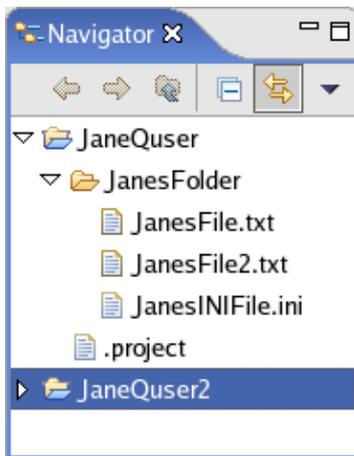
If you closed the editor using File > Close, you may have noticed the option File > Close All. This is a quick way to close all of your open editors. If you choose **File > Close All**, you will be prompted to choose which editors with unsaved changes you want to save.

Navigating resources

In this section you will work with the Navigator and Tasks views. These views are initially part of the resource perspective. If you want to experiment with other views, you can show them by using the Window > Show View menu.

One important view to become familiar with is the Navigator view, which displays information about the contents of the Workbench and how the resources all relate to one another in a hierarchy.

In the Workbench, all resources reside in projects. Projects can contain folders and/or individual files.



Opening resources in the Navigator

Using the Navigator there are different ways to edit a file. For one of these options, you need to make a second editor active in the Workbench.

To make a second editor active in the Workbench:

1. Click Window > Preferences > Workbench > File Associations.
2. Beside the Associated editors area click on Add. The Editor Selection window appears.
3. Select a new editor (for example, Text Editor).
4. Click OK, then click OK again.

You now have two available editors.

5. In the Navigator select the file: JanesFile.txt
6. To open the file, choose one of the following approaches:

- ◆ Double-click on the file in the Navigator.
- ◆ Right-click on the file and choose Open from its pop-up menu.
- ◆ Right-click on the file and choose Open With from its pop-up menu.

The Workbench remembers the last editor that you used for editing a specific file, which makes it easy to use the same editor later. The Workbench is also pre-configured to open particular file types with the appropriate editor. For example, C and C++ files automatically open in a C/C++ editor.

You can configure the default editors for given file types by using Window > Preferences > Workbench > File Associations.

Go To

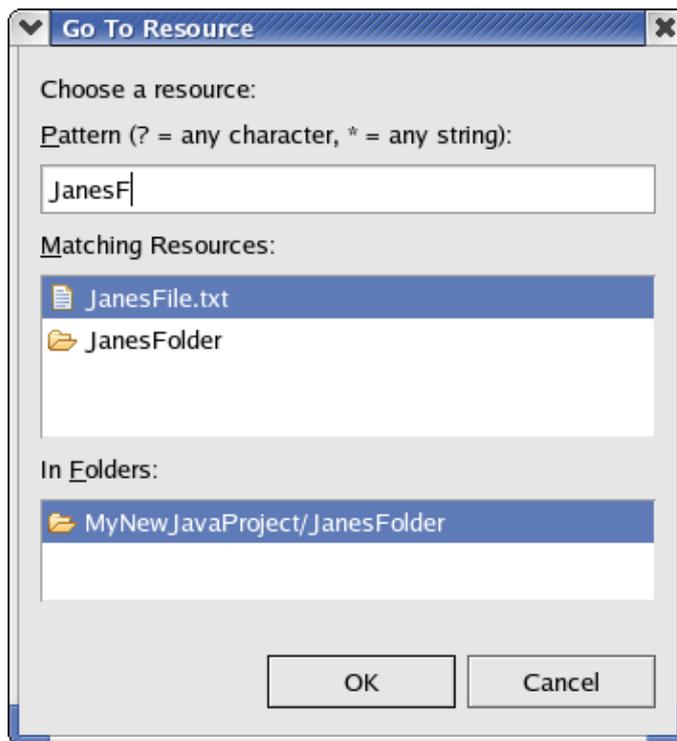
The Go To function makes it easy to jump to a specific resource in the Navigator.

1. Select the Navigator view. Its titlebar becomes highlighted.
2. Click Navigate > Go To > Resource.
3. In the Go To Resource dialog, type **JanesF** into the Pattern field at the top of the dialog.

As you type the filename the dialog filters the set of possible matches based on what you have entered so far.

4. Select JanesFile.txt from the Matching Resources field and click OK.

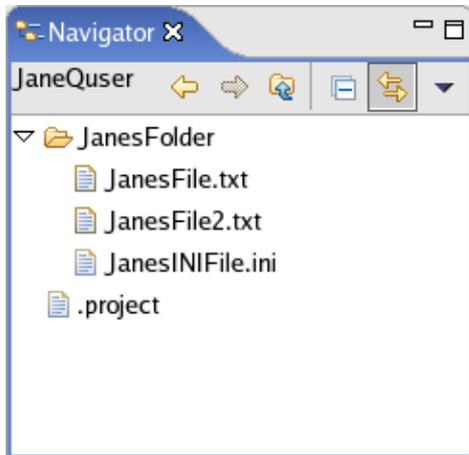
The Navigator selects the file JanesFile.txt.



Go Into

The Navigator shows all of the resources in your Workbench. You can pick a project or folder, "Go Into it" and hide all other resources. Try this with your JaneQuser project:

1. Select the Navigator view. Its title bar becomes highlighted.
2. Right-click on the JaneQuser project and choose Go Into from the pop-up menu.
3. The Navigator now shows the contents of only JaneQuserProject. The title of the Navigator shows the name of the resource you are currently looking at.



4. You can use the back, forward and up buttons  to move between showing all resources and showing only the JaneQuser project.

Click the back button to show all of the resources.

Where are my files?

The projects, folders, and files that you create with the Workbench are stored under a single directory that represents your workspace. The location of the workspace was set in the dialog that first opened when you started the Workbench.

If you have forgotten where that location is, you can find it by clicking File > Switch Workspace. The workspace directory is displayed in the dialog that appears.

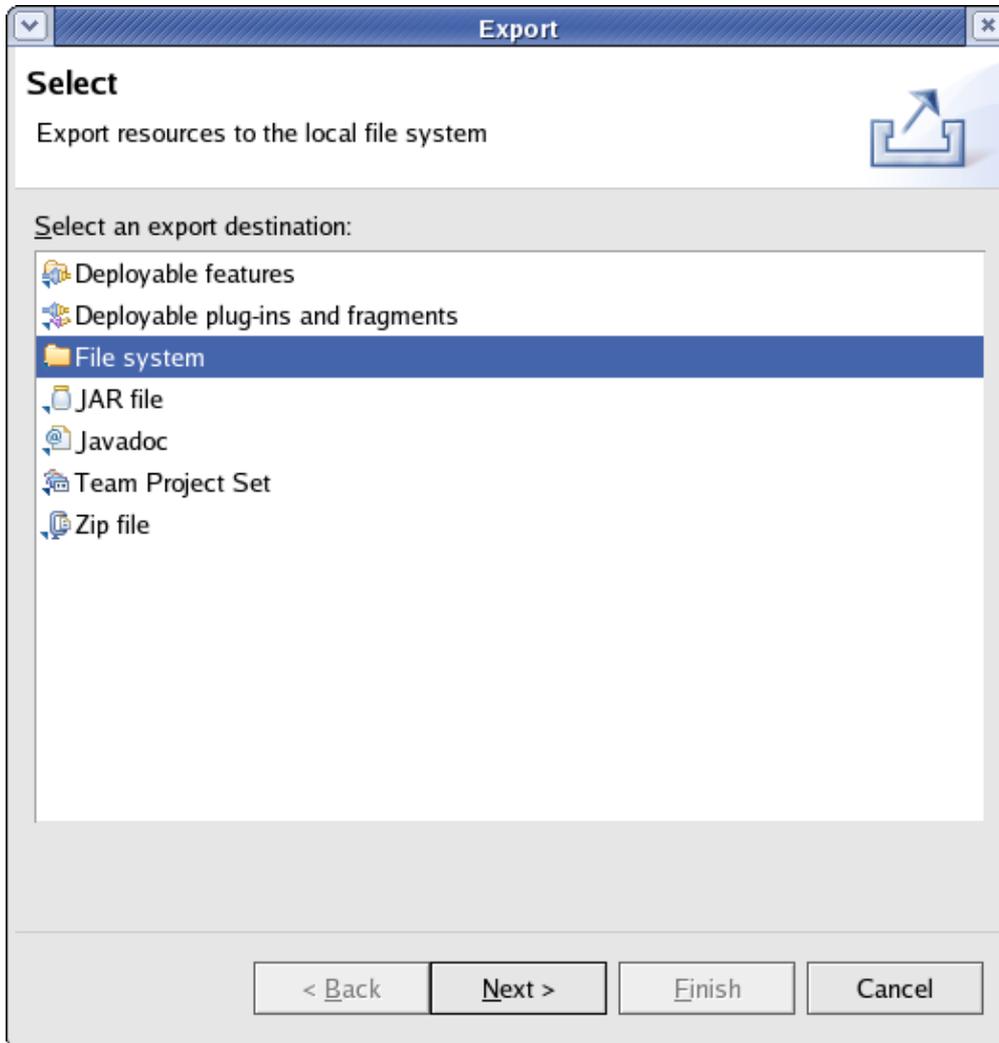
Important: After recording this location, click Cancel to close the dialog, or the Workbench will exit and re-open on whatever workspace was selected.

All of the projects, folders, and files that you create with the Workbench are stored as normal directories and files on the machine. This enables you to use of other tools when working with the files. Those tools can be completely unassociated with the Workbench. A later section will show how to work with external editors that are not integrated into the Workbench.

Exporting files

You can use the export wizard to export from the Workbench to the file system.

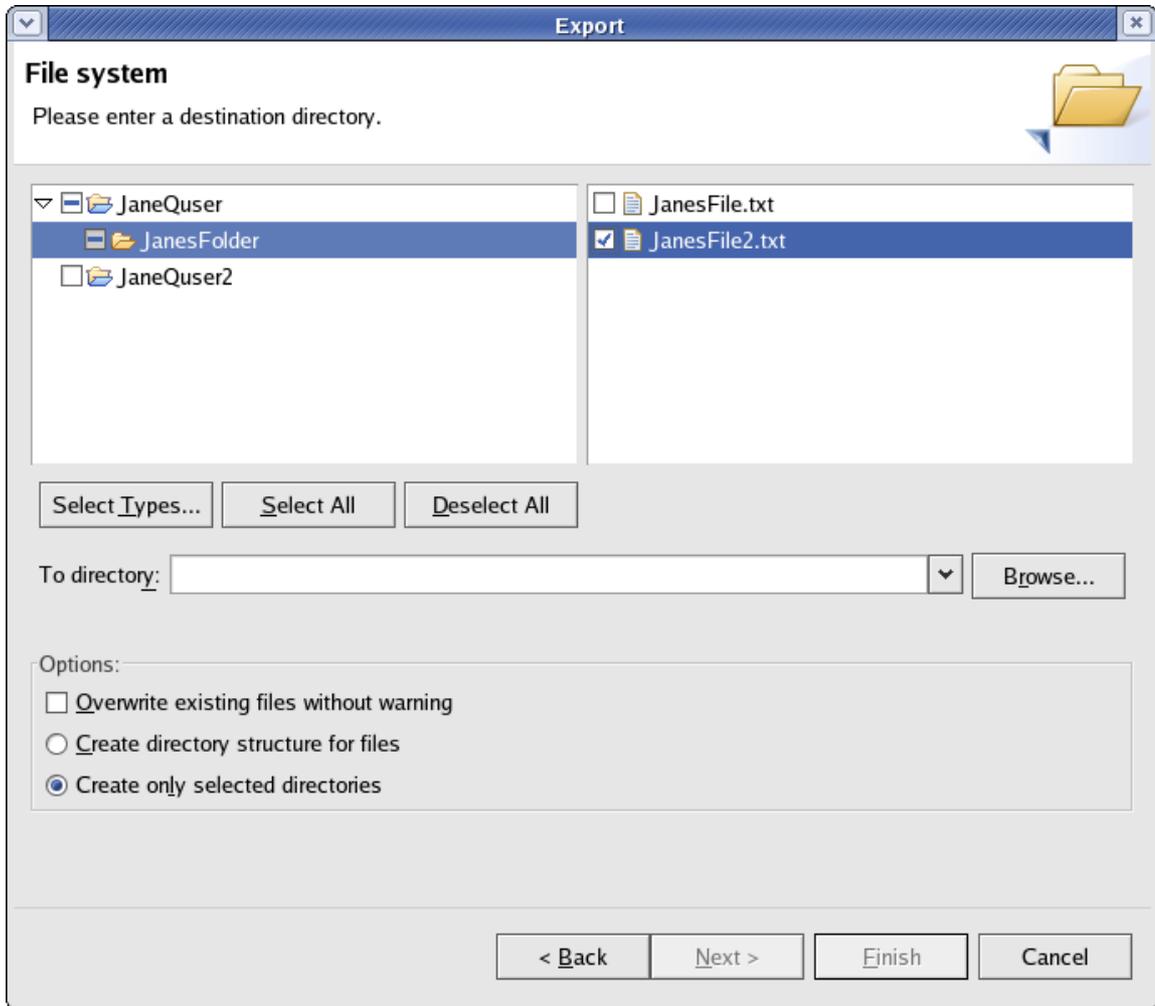
1. In the Navigator, select the project JaneQuser.
2. Right-click in the Navigator and choose Export. (Alternatively, from the menu bar select File > Export.)
3. In the export wizard, select File system, then click Next.



- Expand JaneQuser project and click on JanesFolder. In the right pane, ensure that only JanesFile2.txt is checked. Notice the folder and project in the left pane now have a grayed checkbox indicating that some, but not all, of their contents will be exported.

You can use the Select Types button to filter the types of resources you want to export.

If you had known ahead of time that you wanted to export only JanesFile2.txt, you could have simply selected it in the Navigator and chosen File > Export. The export wizard would have automatically ensured it was the only file checked for export.



5. In the Directory field, type or browse to select a location in the file system where you want the exported resources to reside.

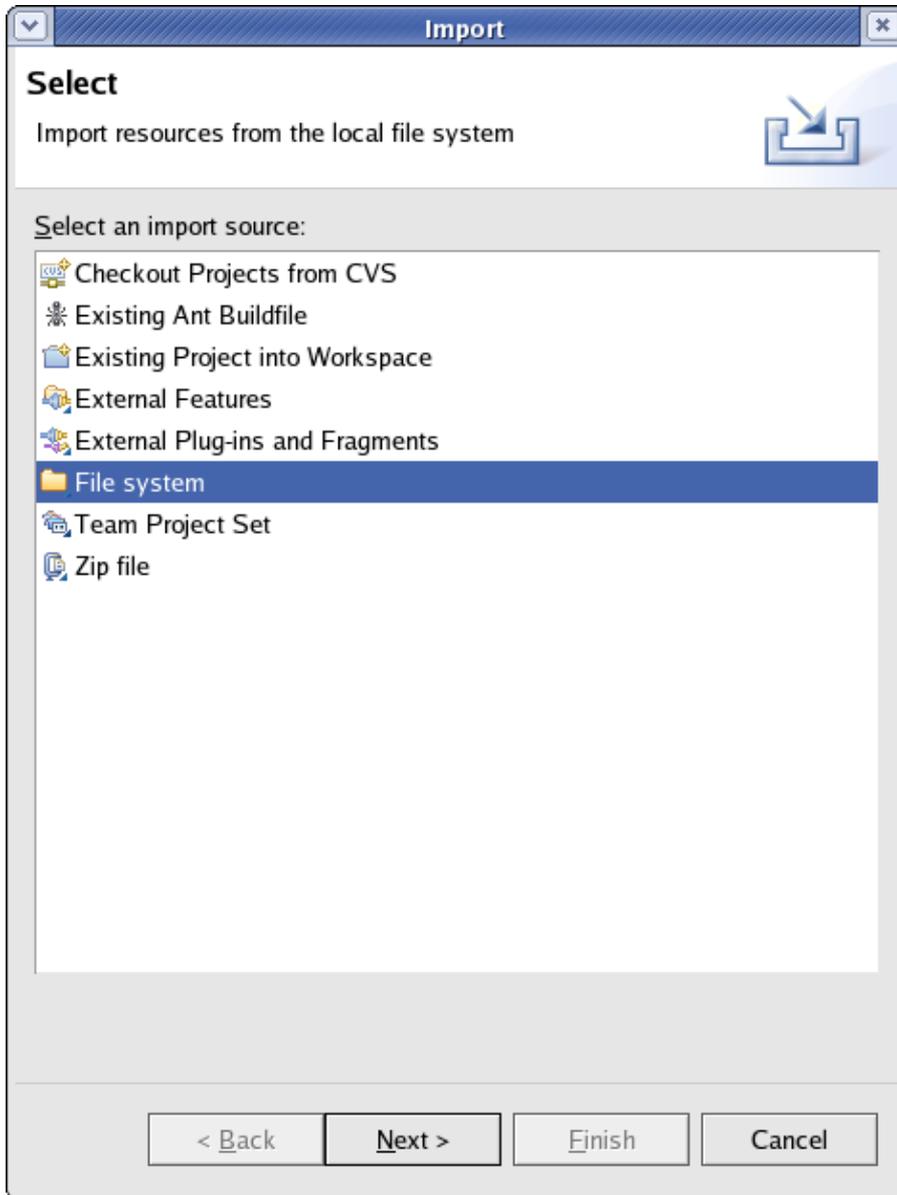
If you enter the name of a directory that does not exist the export wizard will offer to create it for you once you press Finish.

6. In the Options area, you can choose to:
 - ◆ Overwrite existing resources without warning
 - ◆ Create directory structure for files
 - ◆ Create only selected directories.
7. Click Finish when you are done.

Importing files

You can use the Import wizard to copy resources into the Workbench.

1. In the Navigator, select the project JaneQuser.
2. Right-click in the Navigator and choose Import. (Alternatively, from the menu bar select File > Import).
3. In the Import wizard, select File System, then click Next.



4. In the From directory field, type or browse to select the directory containing the file JanesFile2.txt that you recently exported.

Recent directories that you have imported from are shown on the Directory field's combo box.

The directory's name appears in the left pane.

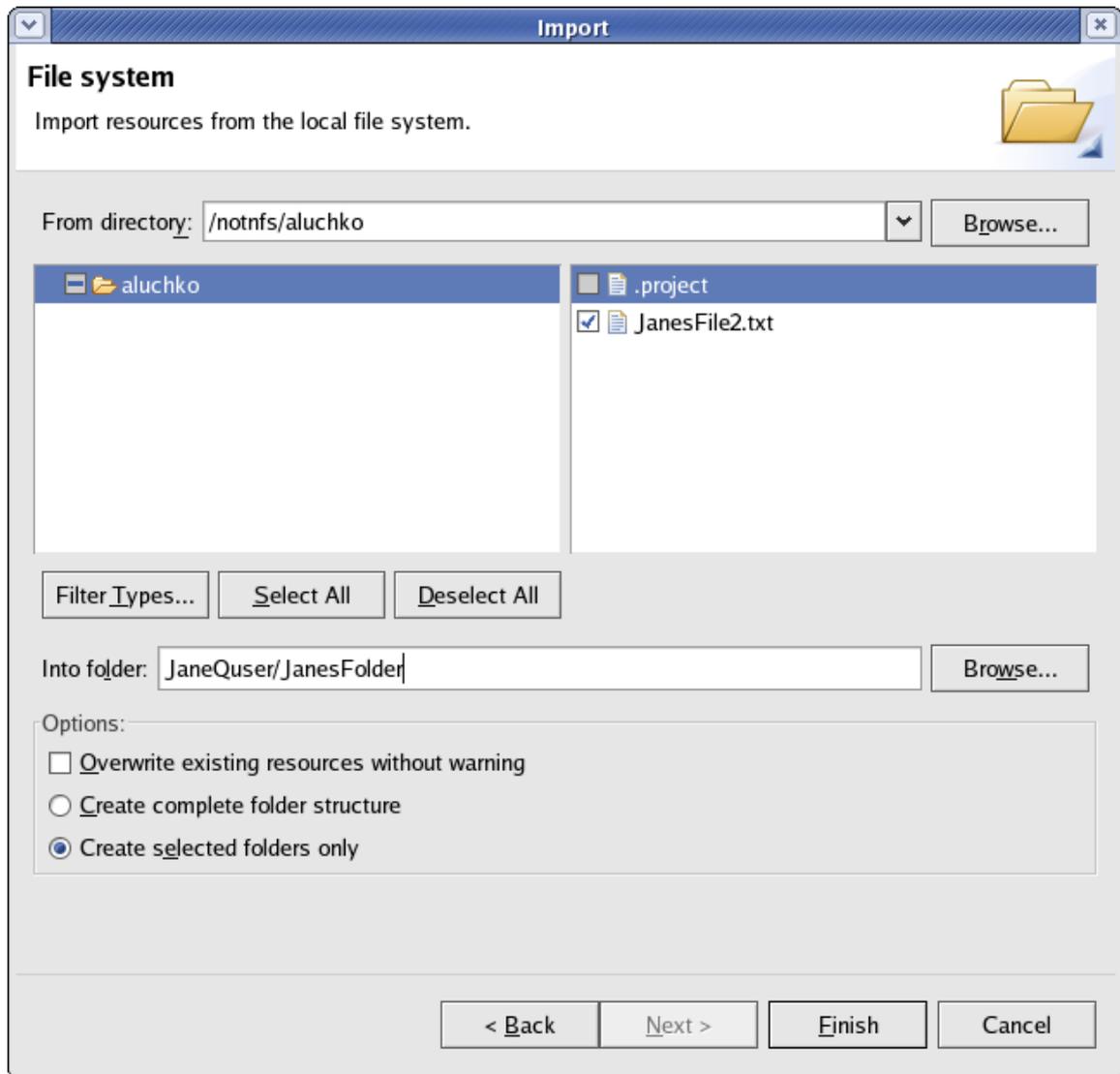
5. Click on the directory's name in the left pane.



6. In the right pane check the file JanesFile2.txt

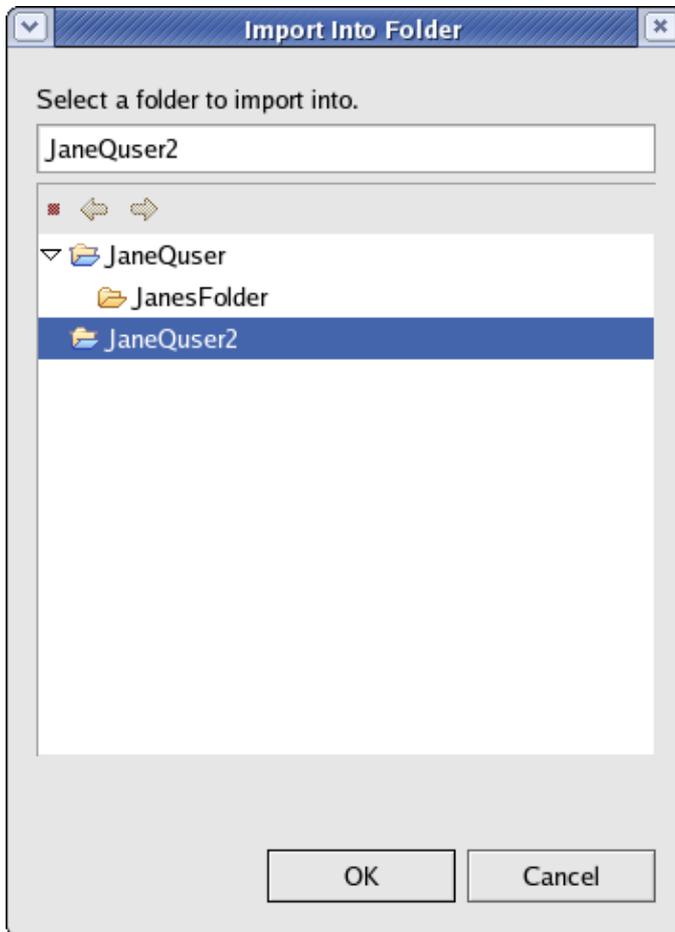
Checking a folder in the left pane will import its entire contents into the Workbench. A grayed checkbox (as shown below) indicates that only some of the files in the folder will be imported into the Workbench.

You can use the Filter Types button to filter the types of files you want to import.



7. The Into folder field should already be filled in with the name of the project (JaneQuser).
8. You can change the destination project or folder by clicking Browse.

Click the Browse button and choose JaneQuser2.



9. Click OK.
10. In the Options area, you can choose to:
 - ◆ Overwrite existing resources without warning
 - ◆ Create complete folder structure
 - ◆ Create selected folders only
11. Click Finish when you are done. The file `JanesFile2.txt` is now shown in the Navigator in the project `JaneQuser2`.

Deleting resources

Now that you have imported a few files into your second project (`JaneQuser2`), you will now see how easy it is to delete the project.

1. Select project `JaneQuser2` in the Navigator view.
2. To delete the project do one of the following:
 - ◆ From the project's pop-up menu choose `Delete`.
 - ◆ Press the `[Delete]` key.
 - ◆ Choose `Edit > Delete` from the pull-down menu.
3. You are asked to confirm the deletion and also choose the type of deletion you want to perform. You can:
 - ◆ Delete the contents of the project from the file system.
 - ◆ Delete the project from the workspace but do not delete its contents in the file system.

Just accept the default (which is "Do not delete contents") and click Yes.

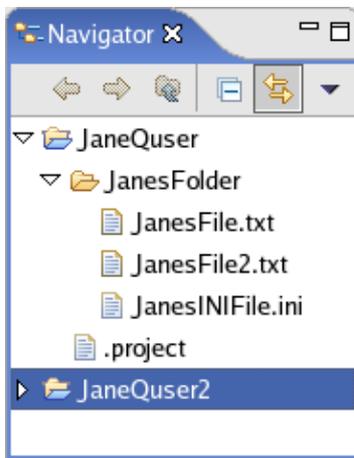
Note: The same steps work for any resource shown in the Navigator. However, the contents of files and folders are always deleted from the file system during delete operations.

Working with other editors

You have seen how to import and export resources from the Workbench. In this section you will look at how to edit Workbench resources using the following approaches:

- External editors launched by the Workbench
- External editors launched without the Workbench's knowledge.

Before you continue, take a moment and confirm that the Navigator contains the following resources:



External editors

When you open a resource the Workbench first consults its list of registered editors. If no registered editors are found for the resource, the Workbench checks with the underlying operating system to determine if it has any editors registered for the particular file type. If an editor is located, the Workbench will automatically launch that editor. This type of editor is referred to as an external editor because it does not show up as an editor tab in the Workbench.

1. Select the file `JanesFile2.txt`.
 2. Double-click the file in the Navigator view to launch the external editor.
-

Editing files outside the Workbench

In a previous section you launched an external editor on the file `JanesFile2.txt` by double-clicking on it in the Navigator. You can also use the same external editor by launching it outside of the Workbench.

1. Close any editors that are open on `JanesFile2.txt`.
2. In a terminal emulator, navigate to the directory where you installed the Workbench and go into the workspace subdirectory.

3. Edit the file JanesFile2.txt and save it. Do not use the Workbench's Navigator to open the editor.
4. Return to the Workbench and in the Navigator view, select the project JaneQuser.
5. Right-click on the project and choose Refresh from the project's context menu. This instructs the Workbench to look for any changes to the project that have been made in the local file system by external tools.
6. In the Navigator, select the file JanesFile2.txt.
7. Right-click on the file and choose Open With > Text Editor from the file's pop-up menu.
8. Observe that the Workbench's own default text editor is opened.

In the default text editor verify the changes you made externally are reflected in the Workbench.

The Workbench stores all of its resources in the local file system. This means you can use your system file explorer to copy files from the Workbench's workspace area even when the Workbench is not running. You can also copy resources into the workspace directory. Remember to use Refresh to update the Workbench with any changes you have made outside the Workbench.

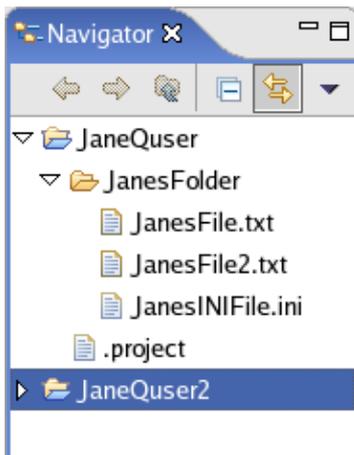
Copying, renaming and moving

You can copy, move and rename Workbench resources using pop-up menu operations in the Navigator. In this section you will copy and rename several files you have created.

Before copying files, some setup is required:

Set up

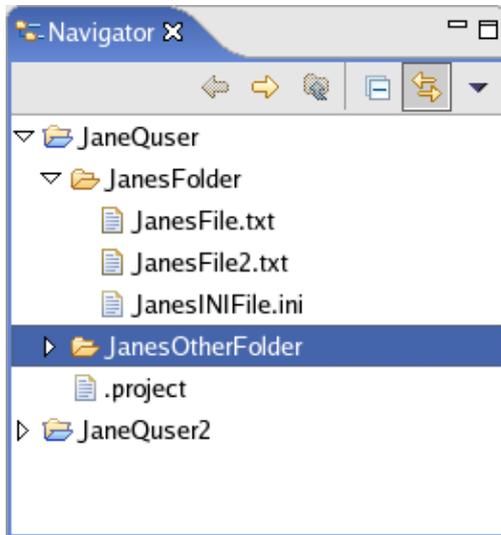
1. Your Navigator should look like this:



2. Double-click on JanesFile.txt and ensure that it contains the following text.

```
This is a sample text file
There is not much else
we can really say about it other
than it has five lines of
text that are rather dull.
```

3. Close the editor on JanesFile.txt.
4. Right-click on the project JaneQuser and select New > Folder. In the New Folder window's Folder name field, type **JanesOtherFolder**, then click Finish.



Copying

Start by copying `JanesFile.txt` to your new folder (`JanesOtherFolder`). Once you have copied your file you will rename it.

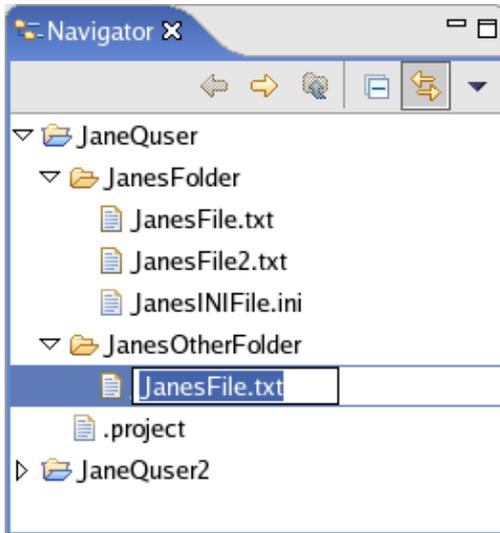
1. Ensure that you have performed the set up described in the [introduction to this section](#).
2. In the Navigator view, select `JanesFile.txt`.
3. Right-click on the file and select Copy (or press [Ctrl]+[C]).
4. In the Navigator view, select `JanesOtherFolder` as the destination.
5. Right-click on `JanesOtherFolder` and select Paste (or press [Ctrl]+[V]).

You have seen how to copy files using the copy operation. It is also possible to copy files by holding down the [Ctrl] key while dragging a file from one folder to another folder.

Renaming

Now that you have copied `JanesFile.txt` from `JanesFolder` to `JanesOtherFolder`, you are ready to rename something.

1. In the Navigator view, right-click on `JanesFile.txt` in `JanesOtherFolder`.
2. From the file's context menu, select Rename.
3. The Navigator overlays the file's name with a text field. Type in **`JanesText.txt`** and press [Enter].



Note: If you decide you do not want to rename a resource, you can press [Esc] to dismiss the text field.

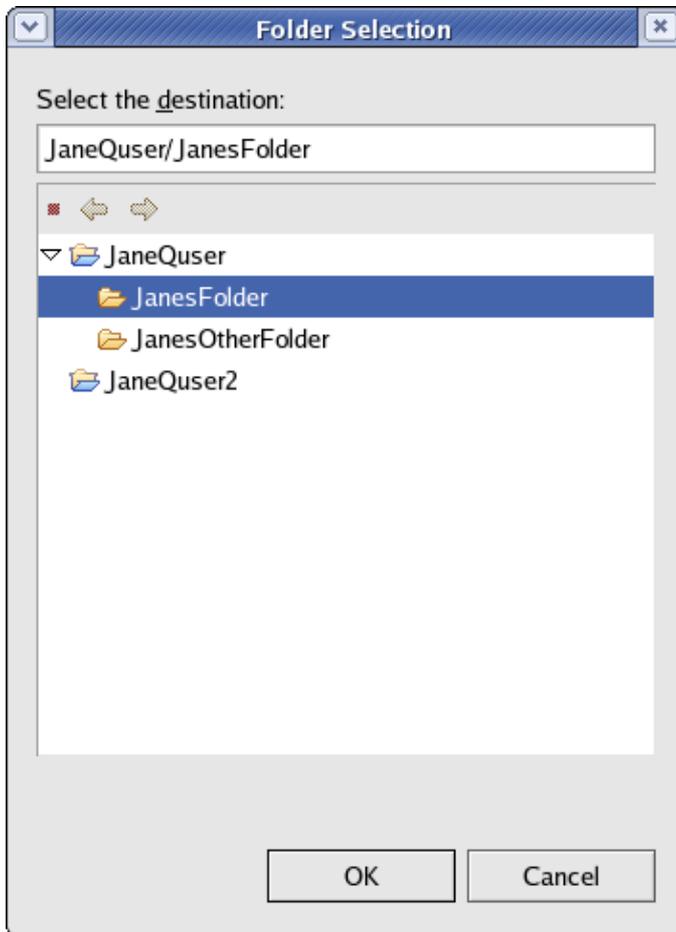
Copy and rename works on folders as well.

1. In the Navigator view, right-click the folder `JanesOtherFolder`.
 2. From the folder's context menu choose `Rename`.
 3. Once again the Navigator overlays the folder name with an entry field to allow you to type in the new name. Change the folder name to be `JanesSecondFolder`.
 4. Rename the folder back to its original name (`JanesOtherFolder`).
-

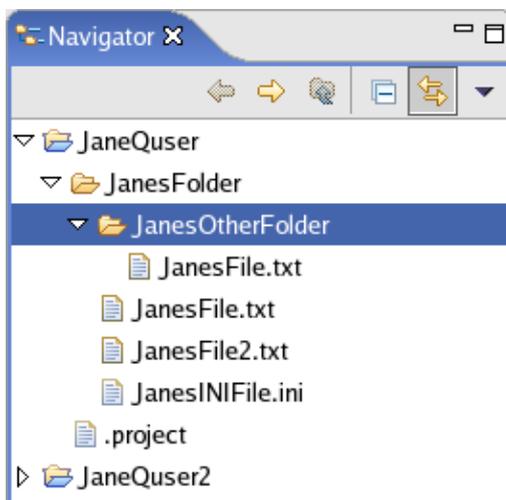
Moving

You have copied and renamed several of your resources. Now it is time to move some resources around. You will move your `JanesOtherFolder` and its file to be a subfolder of your original folder `JanesFolder`.

1. In the Navigator view, right-click on `JanesOtherFolder`.
2. Right-click on the file and from its context menu, select `Move`.
3. In the Folder Selection dialog choose `JanesFolder` and click `OK`.



4. In the Navigator, JanesFolder now contains JanesOtherFolder. Expand JanesOtherFolder and confirm that it contains JanesText.txt.



Searching

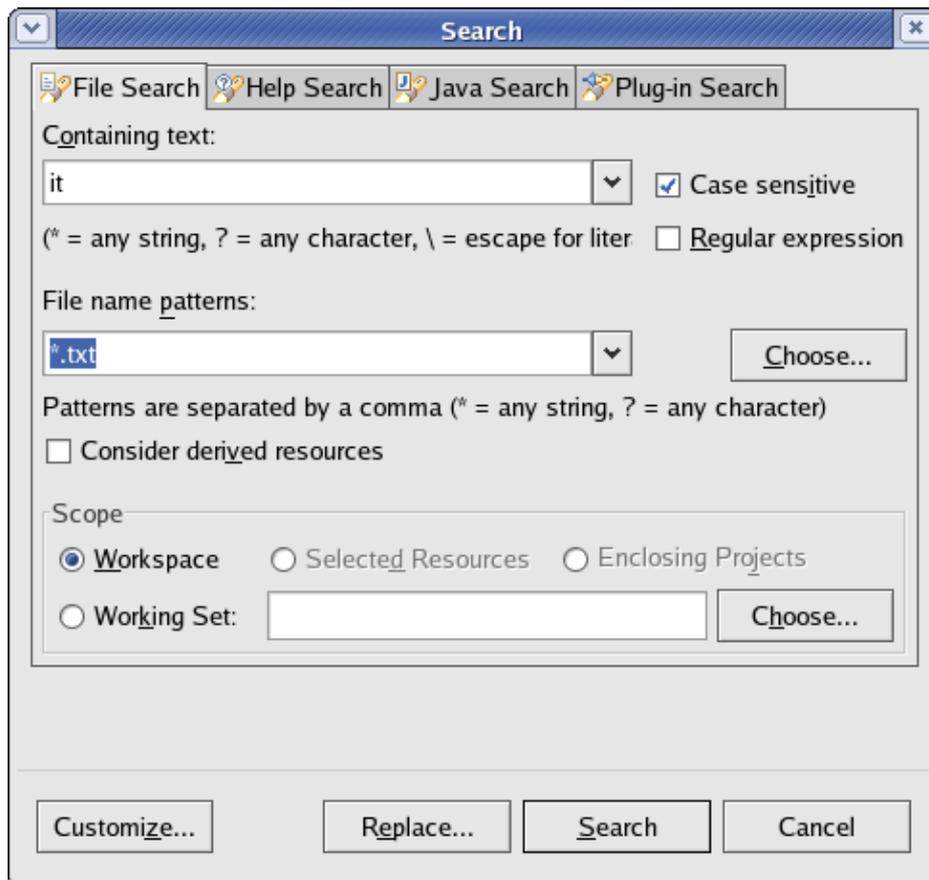
You can search for text strings and files in the Workbench. In this section you will use the Search button  to perform a text search across the resources shown in the Navigator view. Next you will learn how to use the Search view to work with the search results.

Starting a search

You can search for text strings in the Workbench as follows:

1. In the Workbench toolbar, click the Search button .
2. In the Containing text field, type: `it`

The combo box for the Search Expression field also lets you select from a list of recently performed searches.

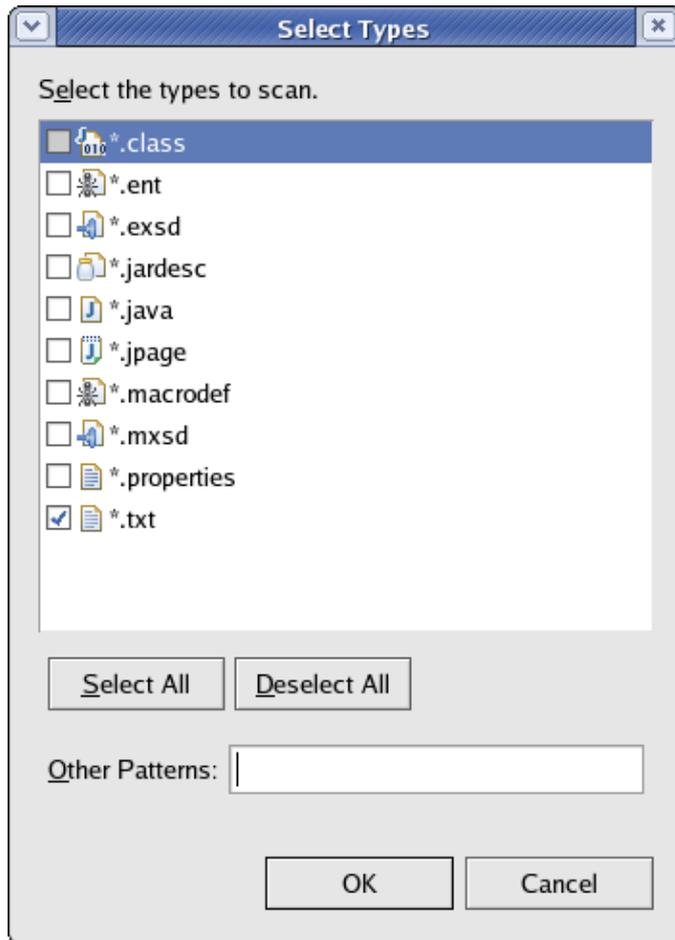


3. You can select or deselect the Case sensitive checkbox depending on whether or not you want to perform a case sensitive or insensitive search.

You want to search for lowercase "it" so check the Case sensitive box.

4. In the File name patterns field you can specify the kinds of files to include in your search.

Click Browse to open the Type Selection dialog. This dialog provides you with a quick way to select from a list of registered extensions.

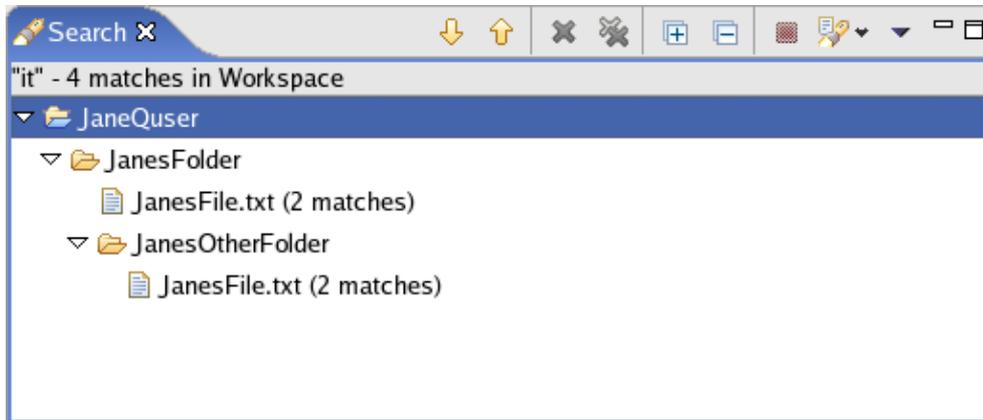


For the moment you will confine your search to .txt files. Ensure .txt is checked and click OK.

5. In the Scope field you can specify the files and folders to include in the search. The choices are: the entire workspace, the currently selected resources in the Workbench, or a working set (which is a named, customized group of files and folders). Leave the scope as Workspace.
6. The Customize button allows you to choose what kinds of searches are available in the dialog. This setting may be left unchanged.
7. Click Search.

The Search dialog shows the progress of the search. Note that you can click Cancel to cancel a search while it is in progress.

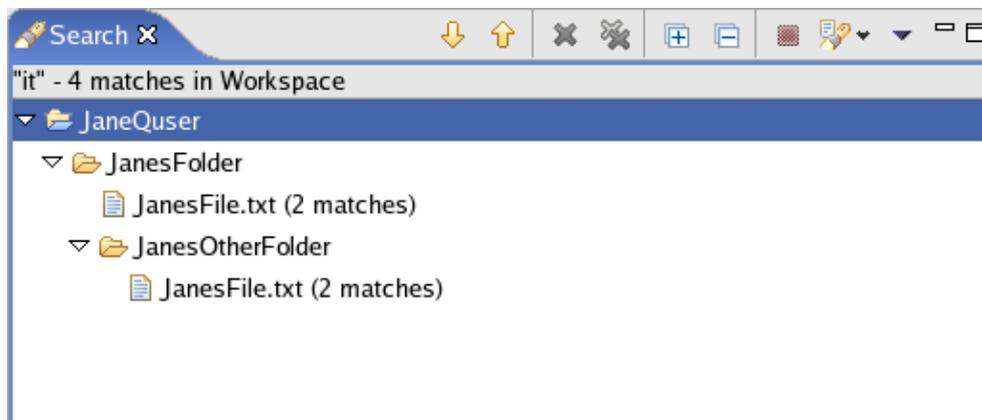
8. Observe that the Search view displays:



The next section describes how to work with the Search view.

The Search view

Now that you have completed your search for "it", the Search view is visible. The title of the Search view shows that four matches were found.



Within the Search view two files are shown and you can also see that within each file there were two matches found. Take a more detailed look at the Search view.

1. The title of the Search view provides a description of the search you just performed.
2. Click the Show Next Match button  to navigate to the first match of the search expression ("it").

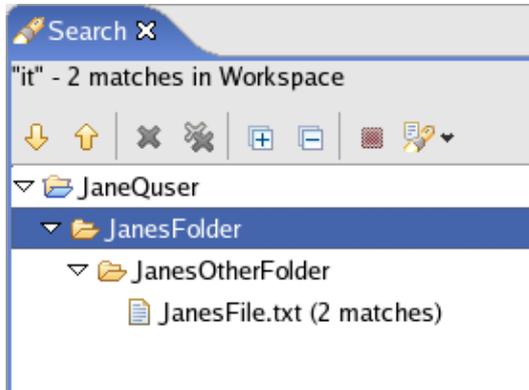
Notice that after two clicks the file JanesText.txt is automatically selected and opened in the editor area.

Click the Show Next Match button two more times. Once again the Search view automatically opens the file.

3. It is sometimes useful to remove uninteresting matches from the search results. Right-clicking on the Search view displays a pop-up menu that provides two ways to remove particular matches:
 - ◆ Remove Current Match – Removes the actual match that you are currently looking at. If a file contains multiple matches, as in this case, only the current match is removed.
 - ◆ Remove Selected Matches – Removes the file entry and all matches in it.

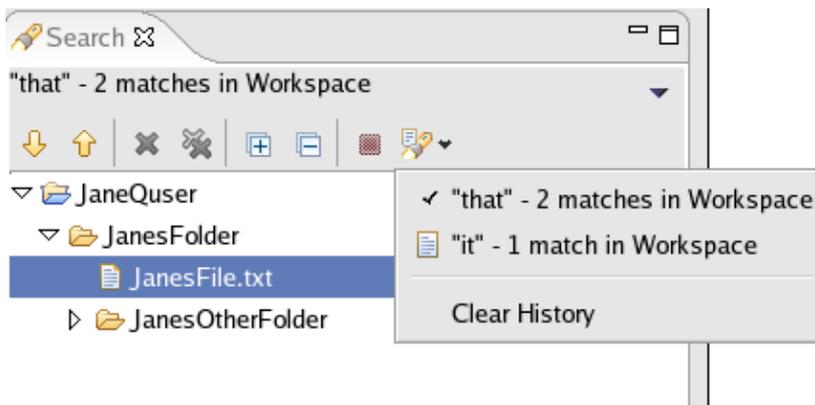
4. Choose Remove Current Match from the pop-up. Notice how JanesFile.txt has only one match left.
5. Select JanesText.txt in the Search view.
6. Choose Remove Selected Matches from the pop-up menu.

The Search view now shows only one match for JanesFile.txt.



7. Perform a second search for "that" by clicking on the Search button  in the Workbench's toolbar.
8. The Search view updates to show the results of the new search.

You can move back and forth between your two search results using the drop down button on the Search view's toolbar.



9. In the drop down button, choose "it" – 1 Occurrence. The Search view switches back to show you the original search. Right-click anywhere in the Search view and on the context menu choose Search Again to repeat the initial search. Notice that once again there are four matches.

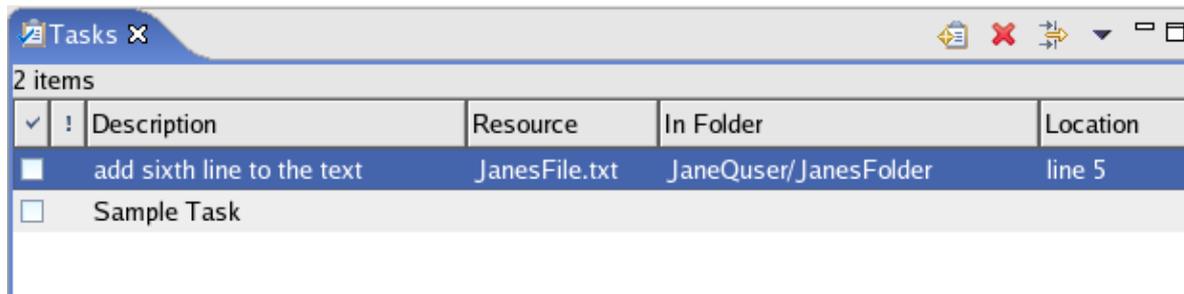
Tasks and markers

There are several types of markers including bookmarks, task markers, debugging breakpoints and problems. In this section you will focus on tasks and the Tasks view.

To access the Tasks view, from the menu bar select Window > Show View > Tasks.

The Tasks view displays all the tasks and problems in the Workbench, both those associated with specific files (and even with specific lines in specific files) as well as generic tasks that are not associated with any specific file.

In the figure below, "sample task" is a generic task not associated with any specific resource. The second task ("add sixth line to the text") is associated with the file JanesFile.txt.

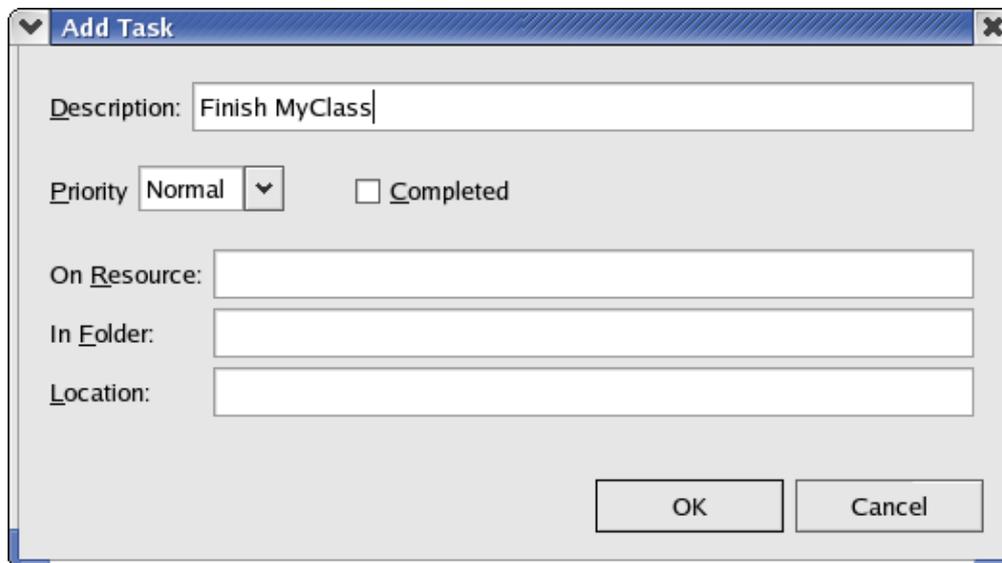


Note: How you create unassociated and associated tasks are covered in the next two topics.

Unassociated tasks

Unassociated tasks are not associated with any specific resource. To create an unassociated task:

1. In the Tasks view, click the New Task button . A New Task dialog appears.

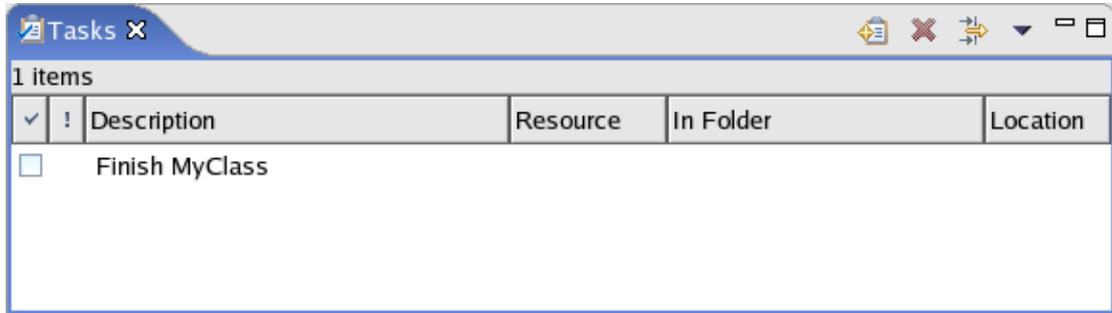


2. Type a brief description for the task and press [Enter].

The new task appears in the Tasks view.

Notes: If you change your mind while entering the description, you can press [Esc] to cancel the dialog.

The fields On Resource, In Folder, and Location are not writable when you create an unassociated task. However, they are filled in automatically when you create a task that is associated with a resource.

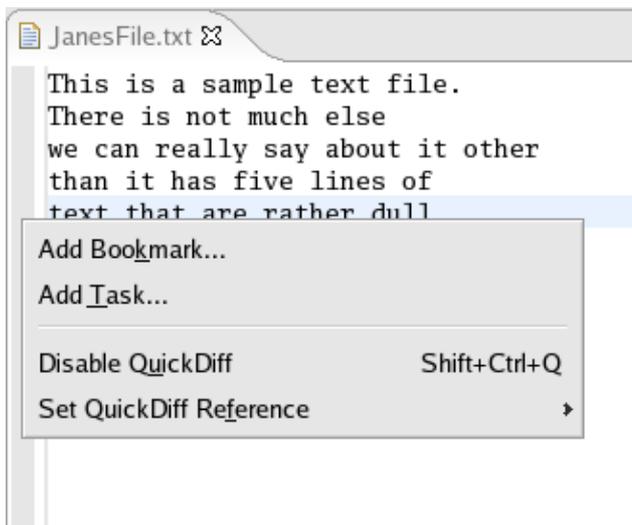


Associated Tasks

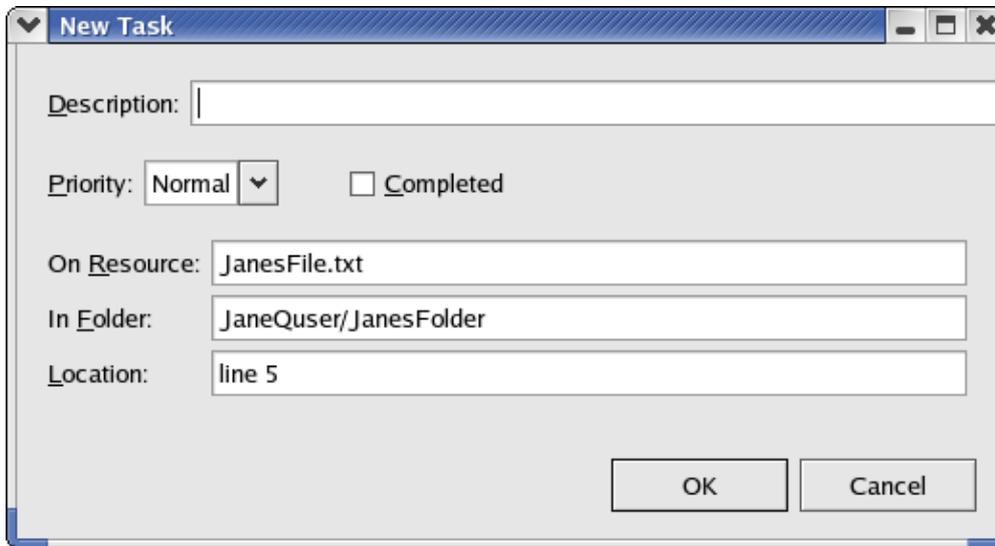
Associated tasks are associated with a specific location in a resource. To associate a task with your JanesFile.txt file:

1. In the Navigator view, double-click JanesFile.txt to open it.
2. In the editor area, access the context menu by right-clicking on the vertical bar directly to the left of any line in the text editor. (The vertical bar to the left of the main text area is called the *marker bar*.)
3. From the marker bar's context menu, select Add Task.

The marker bar displays any marker including bookmarks, task markers (for associated tasks), and debugging breakpoints. You can associate various markers with specific lines in a file by accessing the context menu from the marker bar directly to the left of that line.



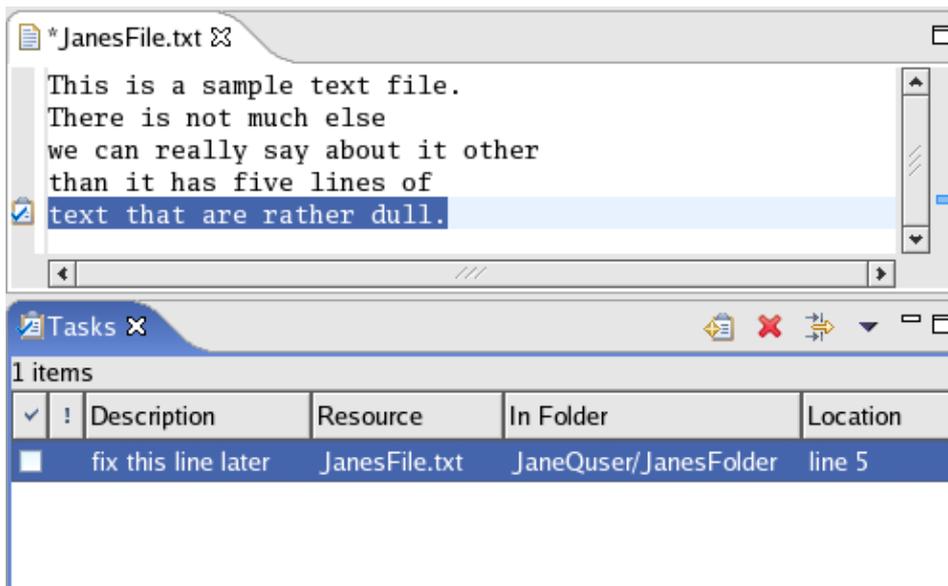
4. In the Description field, type a brief description for the task you want to associate with that line in the text file.



Note: Only the Description field can be edited when you create an associated task.

5. Click OK when you are done.

Notice that a new task marker appears in the marker bar, directly to the left of the line where you added the task. Also, notice that the new task appears in the Tasks view.



6. After you have added the new task, click in the editor on the first line or any other line above the line with which the new task is associated.

7. Add several lines of text to the file at this point.

Notice that as you add lines of text above it, the task marker moves down in the marker bar in order to remain with the associated line in the file. The line number in the Tasks view is updated when the file is saved.

8. There are different ways to remove an entry from the Tasks list:

- ◆ From the editor:

1. In the marker bar, right-click on the checkmark icon of the task you just created.

2. From the marker's context menu, select Remove Task.
- ◆ From the Tasks view:
 1. In the Tasks view, right-click on the task you just created.
 2. Select Mark Completed.
 3. Select Delete Completed Tasks from the marker's context menu.

Notice that the task marker disappears from the marker bar and the task is removed from the Tasks view.

Opening files

The Tasks view provides two approaches for opening the file associated with a task:

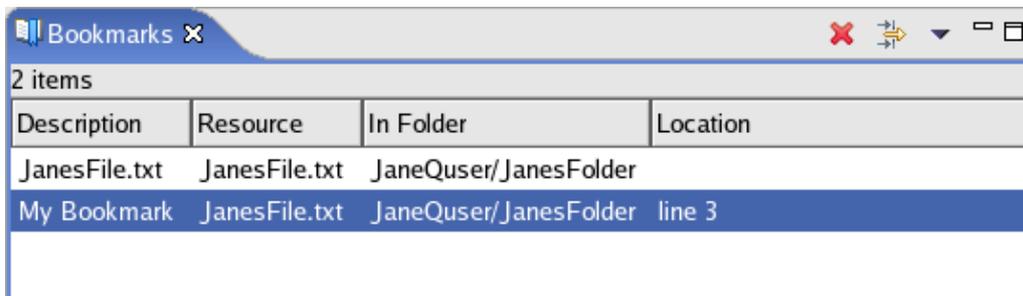
- Right-click on a task, and from the context menu choose Go To.
- Double-click on the task.

In both cases the file's editor is opened and the line with which the selected task is associated is highlighted.

Bookmarks

Bookmarks are a simple way to navigate to resources that you frequently use. In this section you will look at adding, removing, and viewing bookmarks in the Bookmarks view.

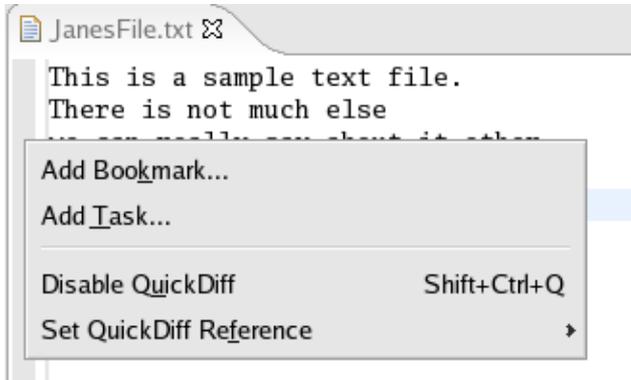
The Bookmarks view displays all bookmarks in the Workbench. To show the Bookmarks view, from the menu bar choose Window > Show View > Bookmarks.



Adding and viewing bookmarks

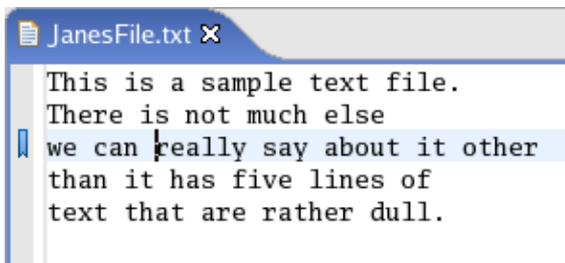
The Workbench allows you to bookmark individual files or specific locations within a file. In this section you will set both types of bookmarks and view them using the Bookmarks view.

1. Click Window > Show View > Bookmarks. The Bookmarks view appears in the Workbench.
2. Edit the file JanesFile.txt.
3. In the editor area, access the context menu by right-clicking on the marker bar, the vertical bar directly to the left of any line in the text editor. From the context menu on the marker bar, select Add Bookmark.

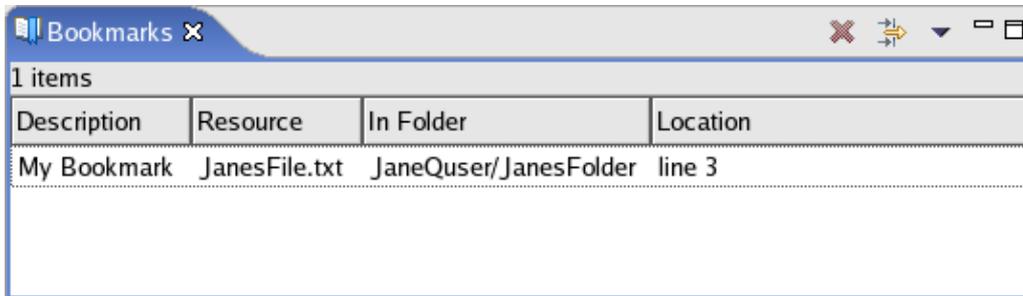


4. In the Add Bookmark dialog, type in a description for this bookmark: **My Bookmark**

A new bookmark icon appears in the marker bar.



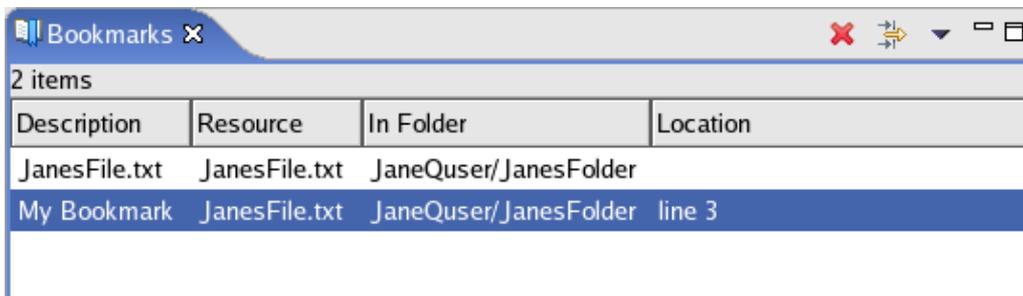
The new bookmark also appears in the Bookmarks view (click Window > Show View > Bookmarks).



5. In the Navigator view, right-click the file JanesText.txt and choose Add Bookmark.

This bookmarks the file; by default Eclipse uses the filename as the bookmark description.

Notice that the Bookmarks view now contains two bookmarks.



Summary: The two methods for adding a bookmark have quite different results. When you right-clicked in the file's marker bar, you were given a dialog to fill in. The resulting bookmark appeared in the editor and in the Bookmark view. The bookmark that you add from the Navigator view did not give you that dialog and no bookmark appears in editor. The only way to see the bookmark is to go to the Bookmark view, which may be hidden under the Navigator view.

Using bookmarks

Now that you have your bookmarks, you will see how to get to the files associated with the bookmarks.

1. Close all of the files in the editor area.
2. In the Bookmarks view, double-click on the first bookmark you created (My Bookmark).
3. Notice that a file editor opens displaying the file with which the bookmark is associated and that the line associated with the bookmark is highlighted.

The Bookmarks view supports two additional ways to open the file associated with a selected bookmark:

1. From the bookmark's context menu, choose Go to File.
2. From the view's toolbar, click on the Go to File toolbar button 

The Bookmarks view allows you to select the associated file in the Navigator.

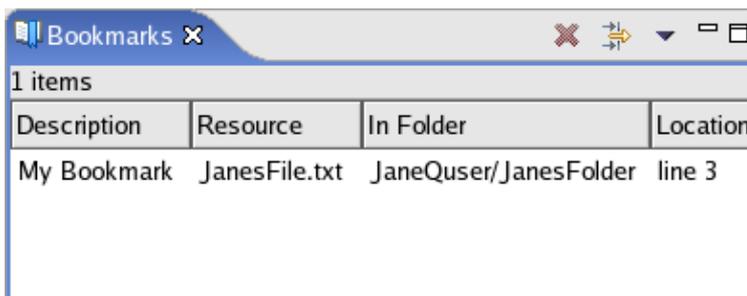
1. In the Bookmarks view, right-click on My Bookmark.
2. From the bookmark's context menu choose Show In Navigator.
3. Notice that the Navigator view is now visible and the file JanesFile.txt is automatically selected. JanesFile.txt is the file My Bookmark was associated with.

Removing bookmarks

In this section you will learn how to remove the bookmarks you have created.

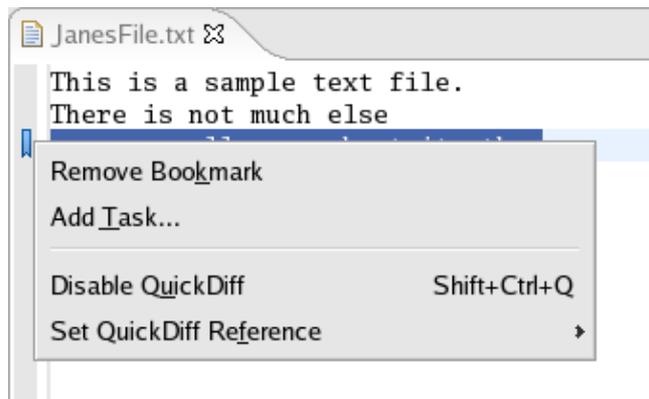
1. In the Bookmarks view, select JanesFile.txt (the second bookmark you created) and do one of the following:
 - ◆ Click the Delete button  on the view's toolbar.
 - ◆ Right-click on the bookmark and choose Delete.

Notice that the bookmark is removed from the Bookmarks view.



2. You should have one remaining bookmark. This bookmark is associated with a line in the file JanesFile.txt. There are two other approaches to removing this bookmark.

- ◆ Right-click on the bookmark icon in the marker bar of the JanesFile.txt editor and select Remove Bookmark. You may recall that you used Add Bookmark in the marker bar when you first created the bookmark.



- ◆ Select the bookmark in the Bookmarks view, then right-click on the bookmark and select Delete from the bookmark's pop-up menu.

Use the second approach:

- Ensure there is an editor open on JanesFile.txt.

Although you do not actually need to open the editor, doing so enables you to watch the editor update as you delete the bookmark.

- In the Bookmarks view, select JanesFile.txt (the remaining bookmark). Click the Delete button  on the view's toolbar. Notice that the bookmark is removed from the Bookmarks view and the JanesFile.txt editor.

Rearranging views and editors

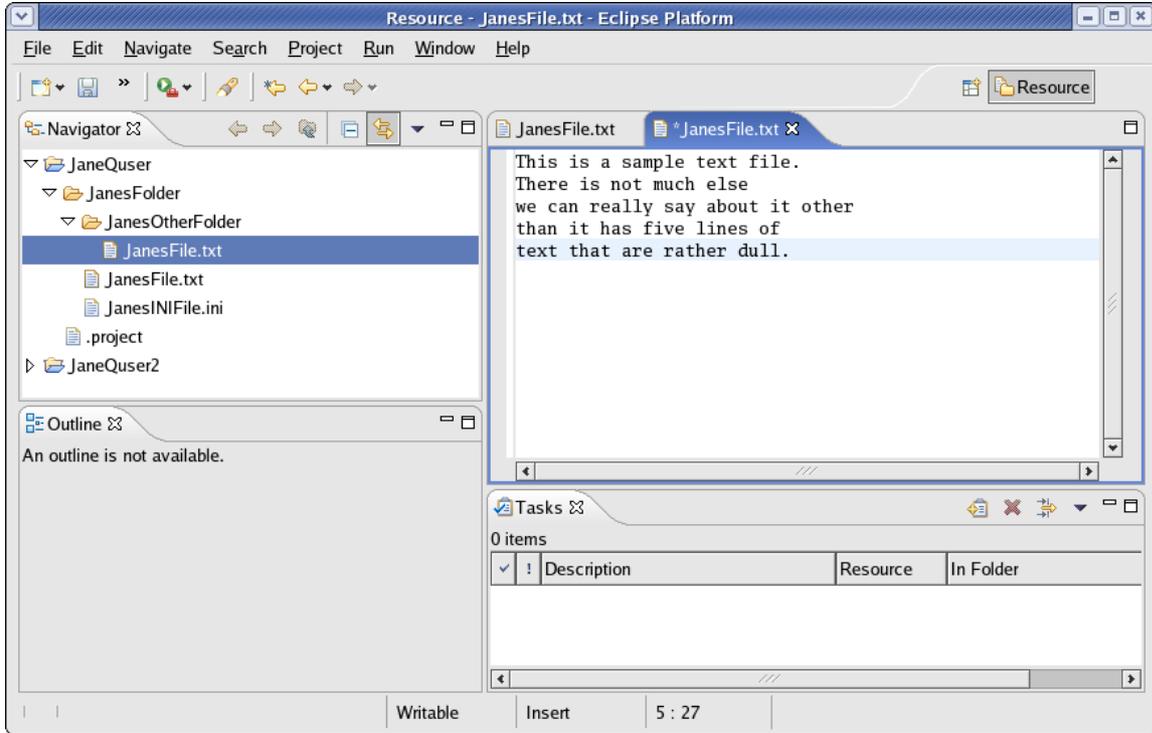
You have seen how to work with editors and views. In this section you will learn how to rearrange these views and editors to customize the layout of the Workbench.

Set up

Before rearranging the Workbench, perform a little setup.

- Start by choosing Window > Reset Perspective, then click OK. This resets the current perspective to its original views and layout.
- Ensure there are editors open for JanesFile.txt and JanesText.txt. Close any other open editors including the Welcome Page if it is still open. (You can always get back to the Welcome Page by choosing Help > Welcome and selecting Eclipse Platform.)

Your Workbench should now look like this:



Drop cursors

Drop cursors indicate where you can dock views in the Workbench window. Several different drop cursors may be displayed when you are rearranging views.

↑	Dock above: if you release the mouse button when a dock above cursor is displayed, the view will appear above the view underneath the cursor.
↓	Dock below: if you release the mouse button when a dock below cursor is displayed, the view will appear below the view underneath the cursor.
→	Dock to the right: If you release the mouse button when a dock to the right cursor is displayed, the view will appear to the right of the view underneath the cursor.
←	Dock to the left: if you release the mouse button when a dock to the left cursor is displayed, the view will appear to the left of the view underneath the cursor.
📁	Stack: if you release the mouse button when a stack cursor is displayed, the view will appear as a Tab in the same pane as the view underneath the cursor.
⊘	Restricted: if you release the mouse button when a restricted cursor is displayed, the view will not dock there. For example, you cannot dock a view in the editor area.

Rearranging Views

You can change the position of your Navigator view in the Workbench window.

1. Click in the title bar of the Navigator view and drag the view across the Workbench window. Do not release the mouse button yet.

2. While still dragging the view around on top of the Workbench window, note that various drop cursors appear. These drop cursors indicate where the view will dock in relation to the view or editor area underneath the cursor when you release your mouse button.
 3. Dock the view in any position in the Workbench window and view the results of this action.
 4. Click and drag the view's title bar to re-dock the view in another position in the Workbench window. Observe the results of this action.
 5. Finally, drag the Navigator view over the Outline view. You will see a stack cursor. If you release the mouse button, the Navigator will be stacked with the Outline view into a tabbed notebook.
-

Tiling Editors

The Workbench allows you to create two or more editors in the *editor area*. You can resize the editor area, but you cannot drag views into it. The editors that you open can be *stacked* (so that one editor can be viewed while the other open editors are shown only as tabs) or *tiled* (so that each open editor is visible in the editor area). Stacking is the default configuration; in this tutorial you will learn how to tile editors.

In the steps that follow, it is important to observe the color of an editor tab:

- **blue** highlighting indicates that the editor is currently active
- **gray** indicates an inactive editor.

To tile editors:

1. Open at least two editors in the editor area by double-clicking editable files in the Navigator view.
2. Click the tab of the editor that is not highlighted and drag it out of the editor area. Do not release the mouse button.

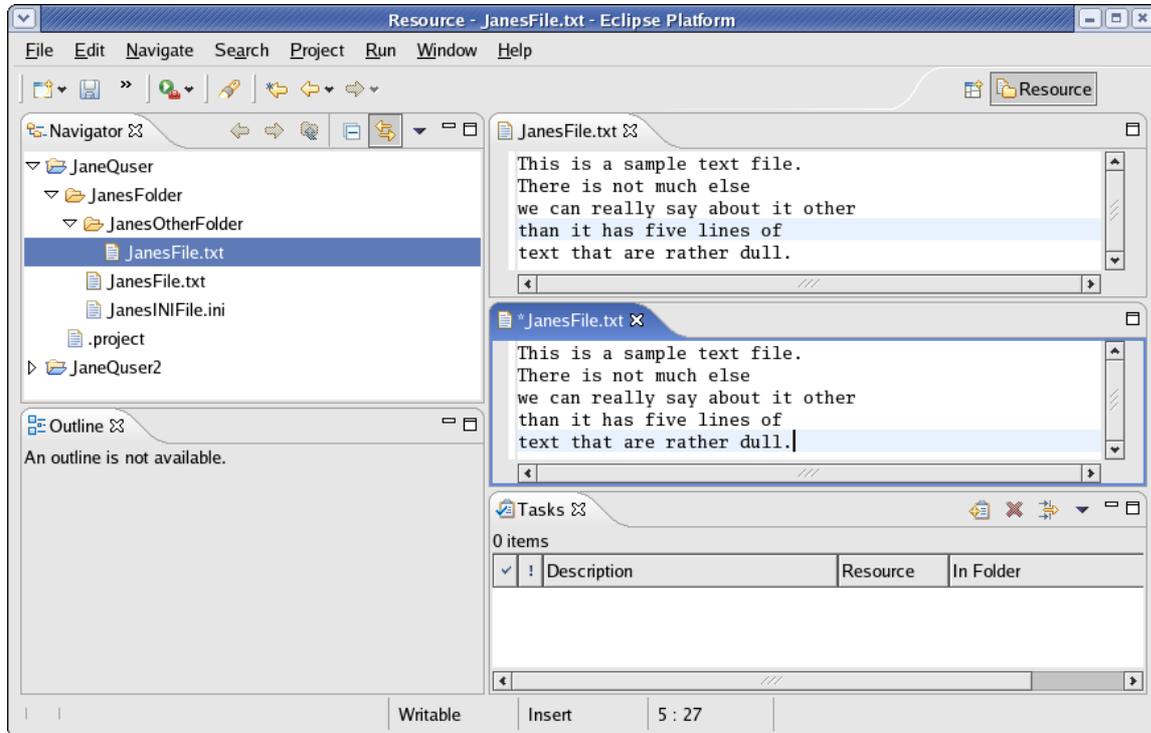
Notice that the restricted cursor displays if you attempt to drop the editor either on top of any view or outside the Workbench window.

3. Still holding down the mouse button, drag the editor over the editor area and move the cursor along all four edges as well as in the middle of the editor area, on top of another open editor. Notice that along the edges of the editor area the directional arrow drop cursors appear, and in the middle of the editor area the stack drop cursor appears.
4. Dock the editor on a directional arrow drop cursor so that two editors appear in the editor area.

Notice that you can also resize each editor as well as the entire editor area to accommodate the editors and views as necessary.

If there is an active view, it will be the view that the active editor is currently working with. This is important when working with views such as Outline and Properties that work closely with the editor.

5. Drag and dock the editor somewhere else in the editor area, noting the behavior that results from docking on each kind of drop cursor. Continue to experiment with docking and resizing editors and views until you have arranged the Workbench to your satisfaction. The figure below illustrates the layout if you drag and drop one editor below another.



Rearranging tabbed views

In addition to dragging and dropping views on the Workbench you can also rearrange the order of views within a tabbed notebook.

1. Click Window > Reset Perspective to reset the Resource perspective back to its original layout.
2. Click on the Outline title bar and drag it on top of the Navigator view. The Outline will now be stacked on top of the Navigator.
3. Click on the Navigator tab and drag it to the right of the Outline tab.



4. Once your cursor is to the right of the Outline tab and the cursor is a stack cursor, release the mouse button.

The Navigator tab is now to the right of the Outline tab.



Maximizing

Sometimes it is useful to be able to maximize a view or editor. Maximizing both views and editors is easy.

To maximize a view or an editor you can:

- Double-click on its tab.

- Right-click on its tab and choose Maximize.
- Click the "Maximize" icon from the top-right corner of the view or editor area.

Restoring a view to its original size is done in a similar manner (double-click on the menu bar or choose Restore from the menu).

Fast views

Fast views are hidden views that can quickly be made visible. When hidden they do not take up screen space on your Workbench window, but otherwise they work identically to normal views.

In this section you will learn how to convert the Navigator view into a fast view.

Creating fast views

Fast views are hidden views that can quickly be made visible. You will start by creating a fast view from the Navigator view and then show how to use the view once it is a fast view.

There are two ways to create a fast view:

- Using drag and drop
- Using a menu operation available from the view icon menu.

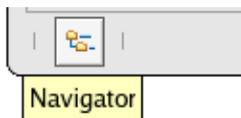
Start by creating a fast view using drag and drop:

1. In the Navigator view, click on the title bar and drag it to the shortcut bar at the bottom-left of the Workbench.

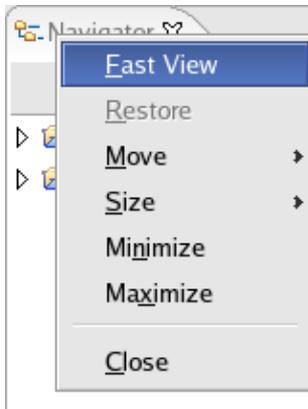
Once you are over the shortcut bar the cursor changes to a fast-view cursor.

2. Release the mouse button to drop the Navigator onto the shortcut bar.

The shortcut bar now includes a button for the Navigator fast view.



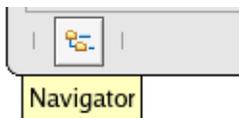
To create a fast view using the second approach, you would have started by right-clicking on the Navigator tab and selecting Fast View.



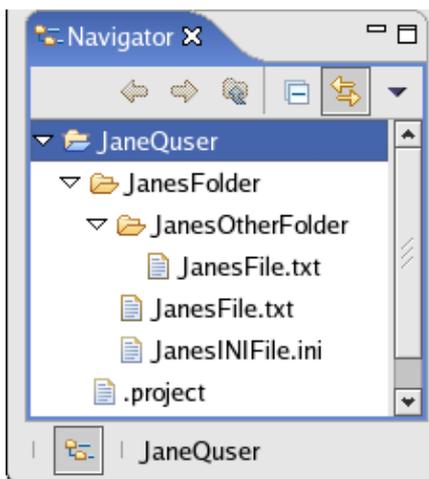
Working with fast views

The **Navigator** has been converted into a fast view. The next question is, "What can you do with it?" The answer to this question and others are about to be revealed.

Confirm that your shortcut bar at the far left of the window still has the **Navigator** view and looks like this:



1. In the shortcut bar, click on the **Navigator** fast-view button.
2. Observe the **Navigator** view slides out from the shortcut bar.



3. You can use the **Navigator** fast view as you would normally. To resize a fast view, move the mouse to the right edge of the fast view where the cursor will change to a double-headed arrow. Then hold the left mouse button down as you move the mouse.
4. To hide the fast view, simply click on another view or editor or click on the **Minimize** button on the fast view's toolbar.



Note: If you open a file from the **Navigator** fast view, the fast view will automatically hide itself to allow you to work with the file.

To convert a fast view back to a regular view, right-click on the tab of the view and choose **Fast View** from the context menu.

Perspectives

A perspective defines the initial set and layout of views in the Workbench window. A perspective provides you with a set of capabilities aimed at accomplishing a specific type of task, or working with a specific types of resources. One or more perspectives can be available in a single Workbench window.

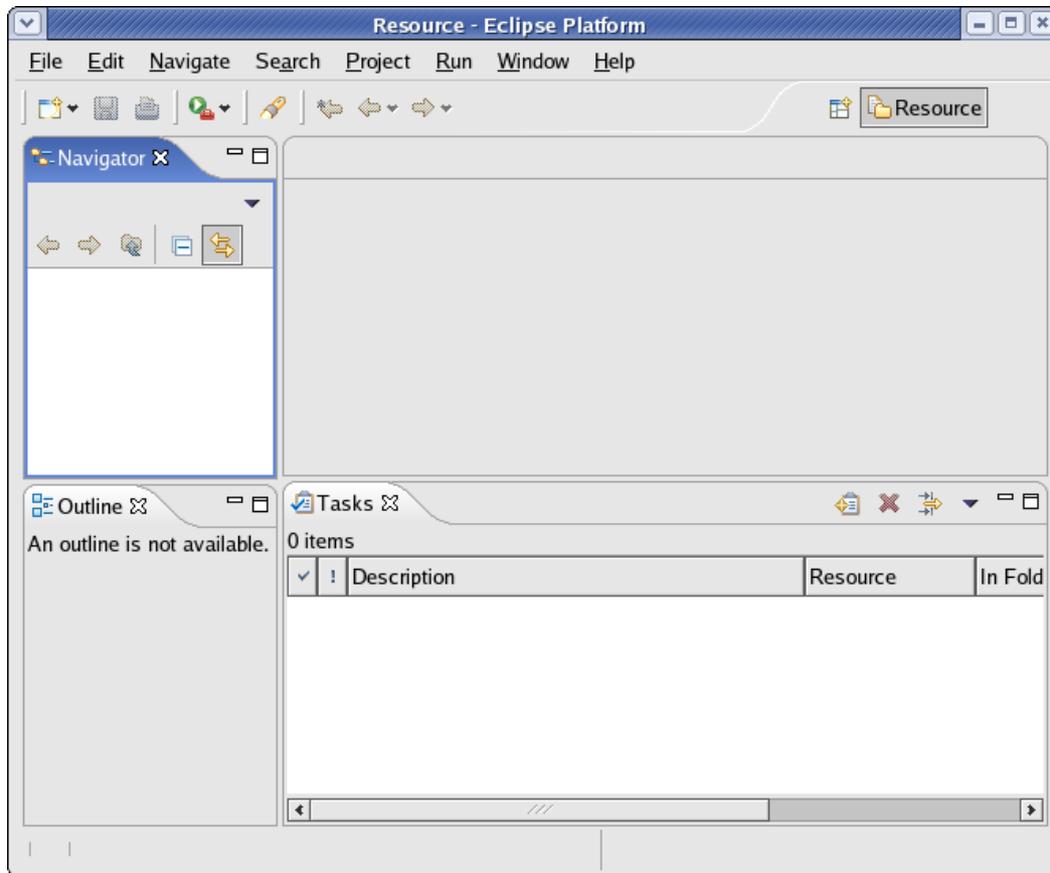
Perspectives can be opened in one of two ways:

- In the same (existing) Workbench window
- In a new Workbench window.

Perspectives are collections of visible action sets, which you can change to customize a perspective. You can also save a customized perspective so that you can open it again later.

The Workbench window displays one or more perspectives. Initially one perspective, the Resource perspective, is displayed. A perspective consists of views like the Navigator view and editors for working with your resources. More than one Workbench window can be open at any given time.

So far you have only used the Resource perspective (shown below). In this section you will explore how to open and work with other perspectives.



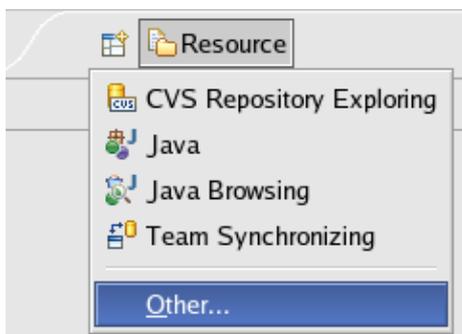
New perspectives

You can open a perspective in the following ways::

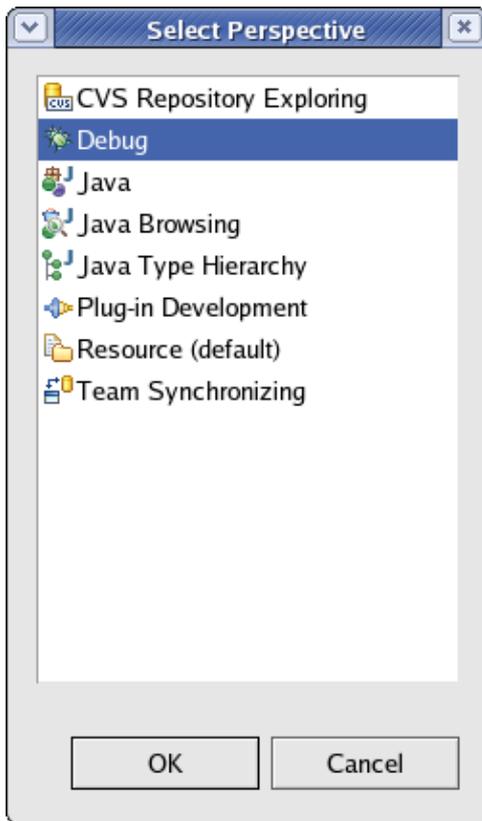
- Using the Open Perspective button  on the shortcut bar.
- From the menu bar, choosing Window > Open Perspective > *perspectivename*.

You will open one by using the shortcut bar button.

1. Click on the Open Perspective button .
2. A menu appears showing the same choices as shown on the Window > Open Perspective menu. Choose Other from the menu.



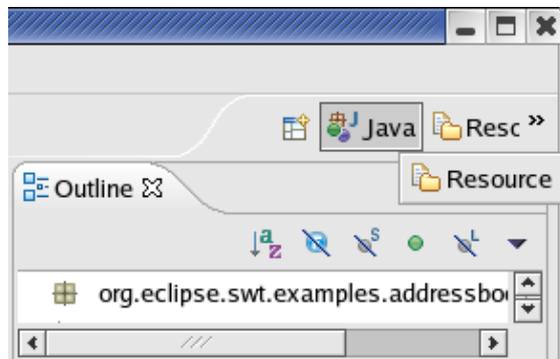
3. In the Select Perspective dialog, choose Debug and click OK.



The Debug perspective is displayed.

4. There are several other interesting things to note:

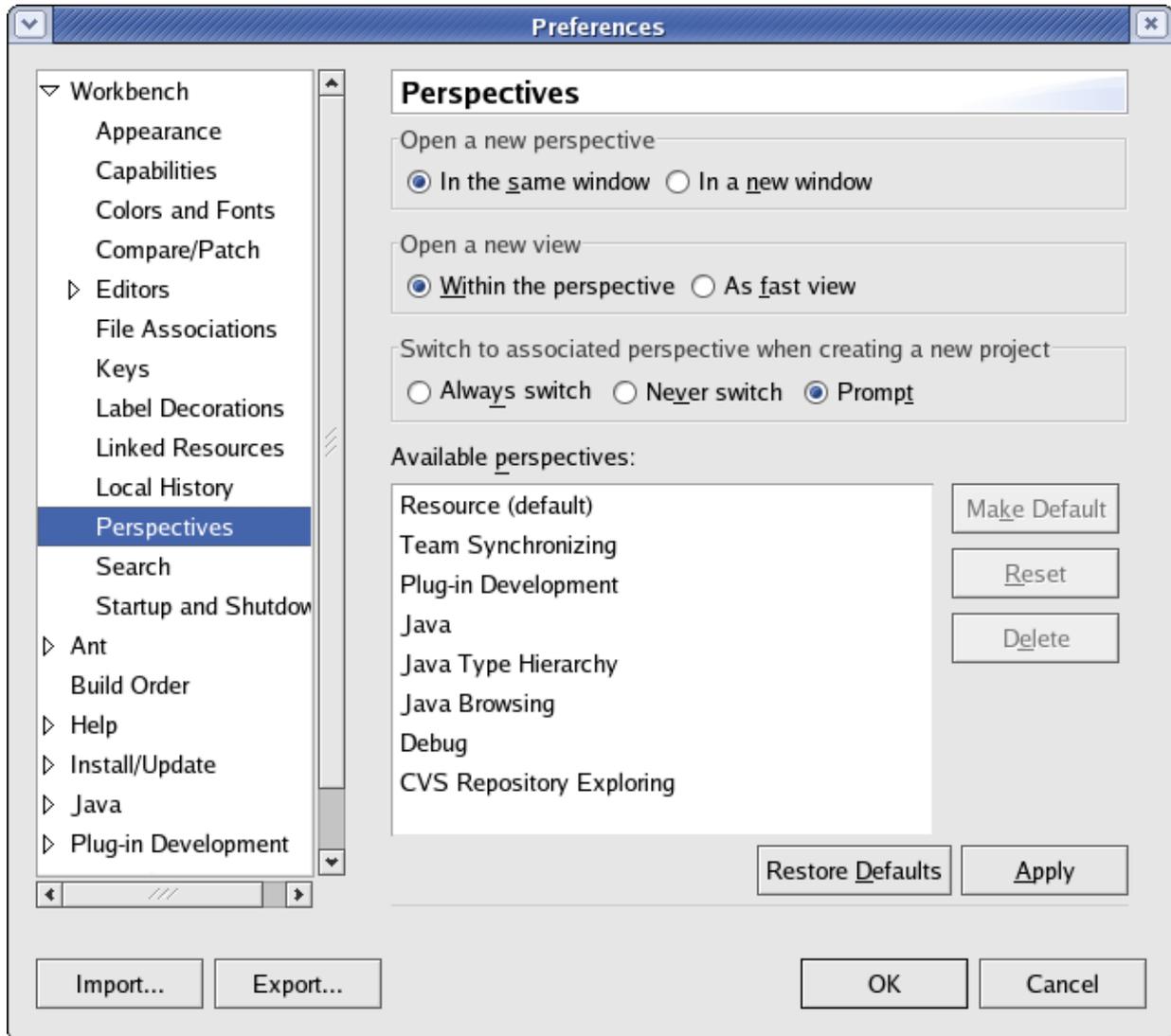
- ◆ The title of the window now indicates that you are using the Debug perspective.
- ◆ The shortcut bar now contains several perspectives, the original Resource perspective and the new Debug perspective, and others. The Debug perspective button is pressed-in, indicating that it is the current perspective.
- ◆ To display the full name of the perspective, right-click the perspective bar and check Show Text.



5. In the shortcut bar, click on the Resource perspective button. The Resource perspective is once again your current perspective. Notice that the set of views is different for each of the perspectives.

New windows

You have just seen how to open a perspective inside the current Workbench window. You can also open a new perspective in its own window; you can configure this default behavior using Window > Preferences > Workbench > Perspectives.



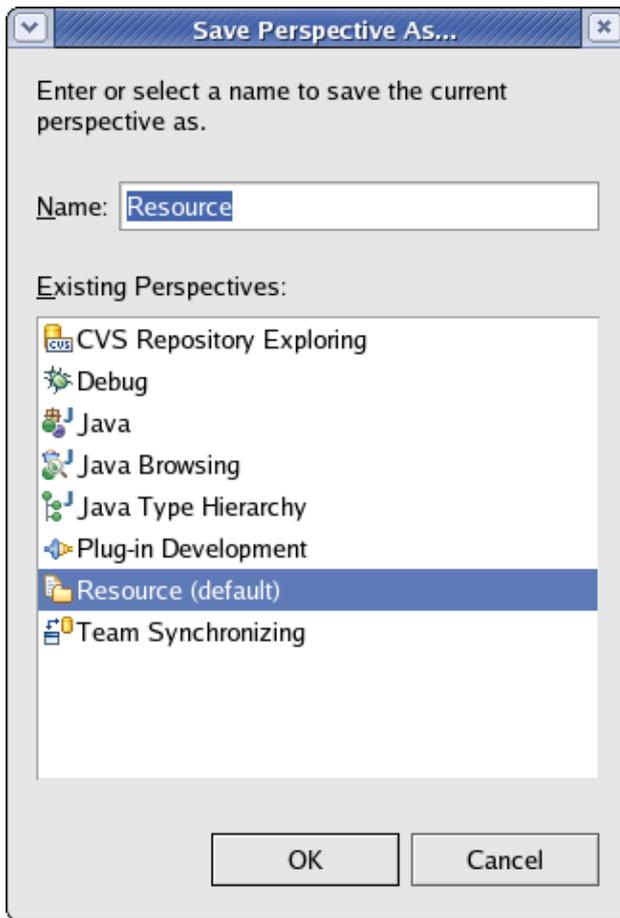
Saving perspectives

In this tutorial you have seen how to add new views to your perspective, rearrange the views and convert views into fast views. The Workbench also enables you to save this layout for future use.

1. In the shortcut bar, click on the Resource perspective. The Resource perspective is now active.
2. Drag the Outline view and stack it with the Navigator view.
3. Click Window > Save Perspective As.

The Save Perspective As dialog enables you to redefine an existing perspective or create a new

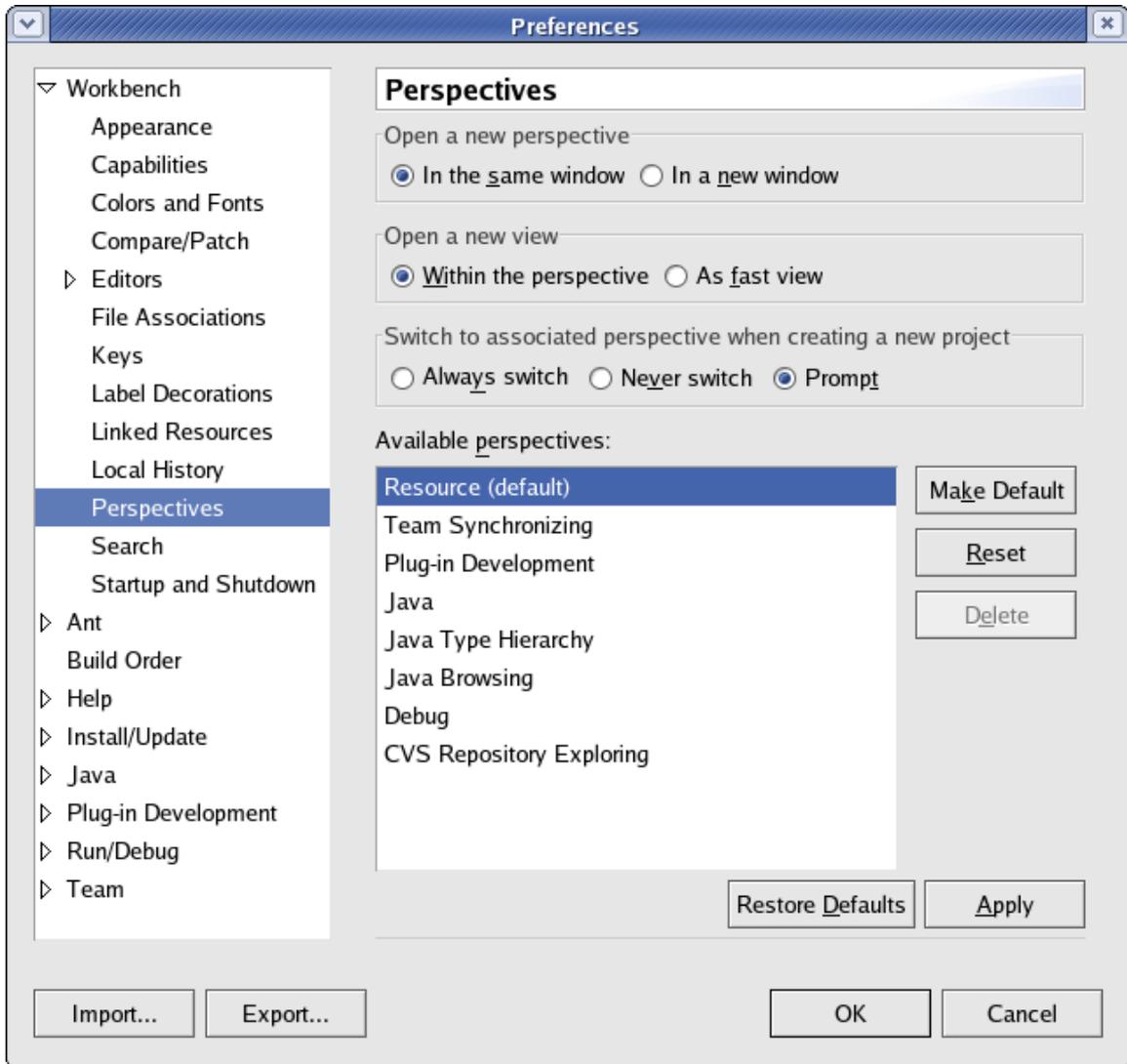
perspective.



4. Click OK to update the Resource perspective and Yes to the subsequent confirmation dialog. The new perspective layout will be used if you reset the perspective or open a new one.
5. In the Resource perspective, move the Outline view so that it is now stacked with the Tasks view.
6. Click Window > Reset Perspective. Notice the Outline view is stacked with the Navigator. Originally when you first started the Workbench it was below the Navigator, but because you saved the perspective with Repositories and Outline stacked, Eclipse now considers this to be the default layout.
7. Click Window > New Window to open a second window showing the Resource perspective. Observe that it uses the newly saved layout.
8. Close the second window.

Now that you have changed the default for the Resource perspective, you are probably hoping there is some way to get back the original layout. To reset the Resource perspective to its original layout:

1. Click Window > Preferences > Workbench > Perspectives.
2. Select the Resource perspective, click the Reset button, and then click OK.



3. You have undone any changes to the saved state of the perspective. To update the current copy of the Resource perspective you are working with, click Window > Reset Perspective.

Configuring perspectives

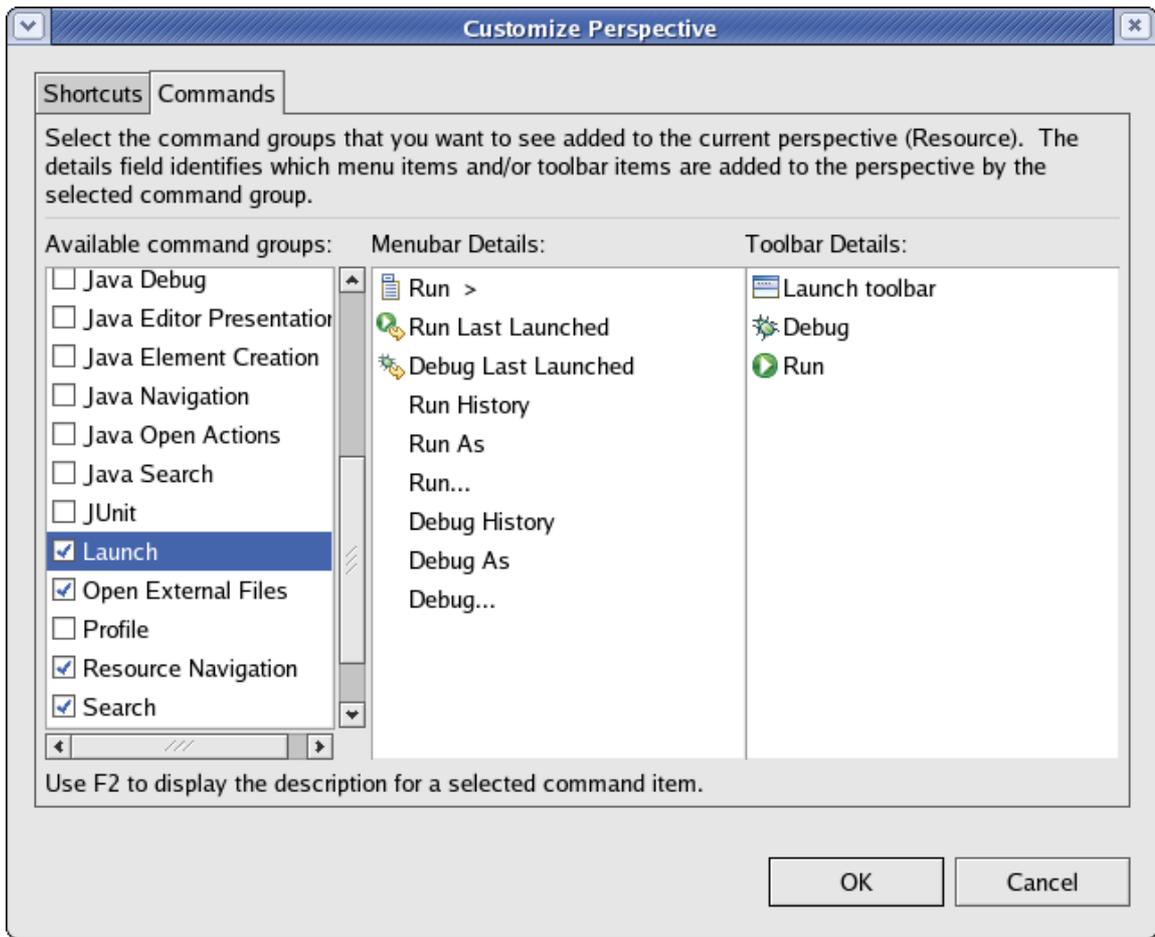
In addition to configuring the layout of your perspective you can also control several other key aspects of a perspective. These include:

- The File > New menu
- The Window > Open Perspective menu
- The Window > Show View menu
- Action sets (buttons and menu options) that show up on the toolbar and menu bar.

Try customizing one of these items:

1. In the shortcut bar, click on the Resource perspective.
2. Select Window > Customize Perspective.
3. Select the Commands tab.

4. Check Launch and click OK.



Observe that the toolbar now includes buttons for debug/run launching.



5. When you have finished experimenting with the other options on the Customize Perspective dialog, to return the perspective to its original state, click Window > Reset Perspective.

Comparing

The Workbench enables you to compare multiple resources and present the results in a special compare editor.

Set up

Before you get started with the compare feature, you need to create a few files. This will also be a good time to recap some of the basic features you have already learned about.

1. Start by selecting each project in the Navigator and deleting it by right-clicking and selecting Delete on the pop-up menu.

2. Create a new Simple project using File > New > Project. Be sure to give the project a distinct name by typing your name as the name of your new project (for example, **JaneQuserCompare**). Do not use spaces or special characters in the project name.
3. In the Navigator, right-click on the project and use the pop-up menu to create a file called: file1.txt
4. In the editor for file1.txt, type the following lines of text and save the file:

```
This is line 1.
This is line 2.
This is line 3.
This is line 4.
This is line 5.
```

5. In the Navigator select file1.txt and press [Ctrl]+[C] to copy the file.
6. Press [Ctrl]+[V] (paste) to create the copy. In the Name Conflict dialog that appears, rename the file to: file2.txt

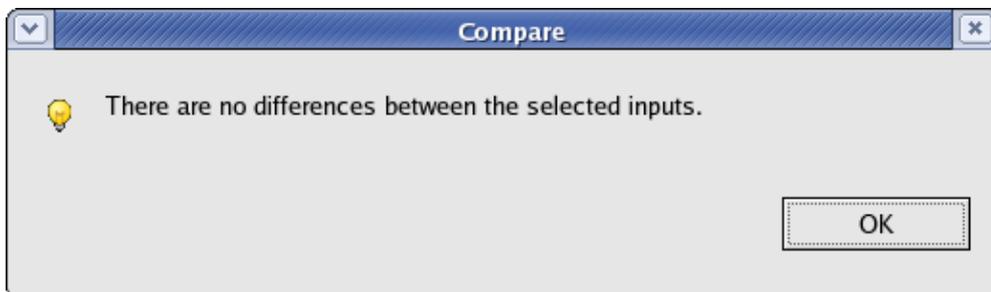
You now have two files, file1.txt and file2.txt, that are identical. Now you can use the compare feature.

Simple compare

Comparing files is easy:

1. In the Navigator, select file1.txt and file2.txt (hold down [Ctrl] when clicking the two file names), right-click on them, then choose Compare With > Each Other from the context menu.

A dialog like the one below appears, saying that there are no differences between the two files.



2. Now edit file1.txt as follows:
 1. Delete line 1 "**This is line 1.**"
 2. Change line 3 to be "**This is a much better line 3.**"
 3. Insert a line 4a before line 5 that reads "**This is line 4a and it is new**"

Your file (file1.txt) should now look like this:

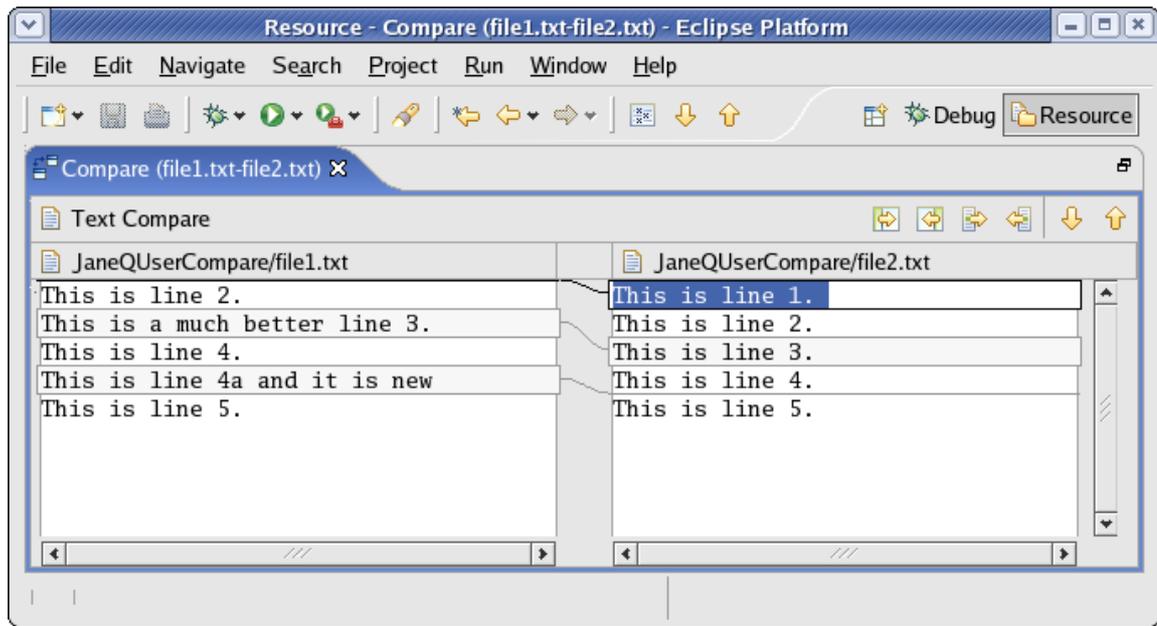
```
This is line 2.
This is a much better line 3.
This is line 4.
This is line 4a and it is new
This is line 5.
```

3. Save the contents of the file from the menu bar by choosing File > Save (or by pressing [Ctrl]+[S]).

Now you are ready to do a proper compare.

4. Once again select file1.txt and file2.txt in the Navigator, then right-click and choose Compare With > Each Other in the context menu.

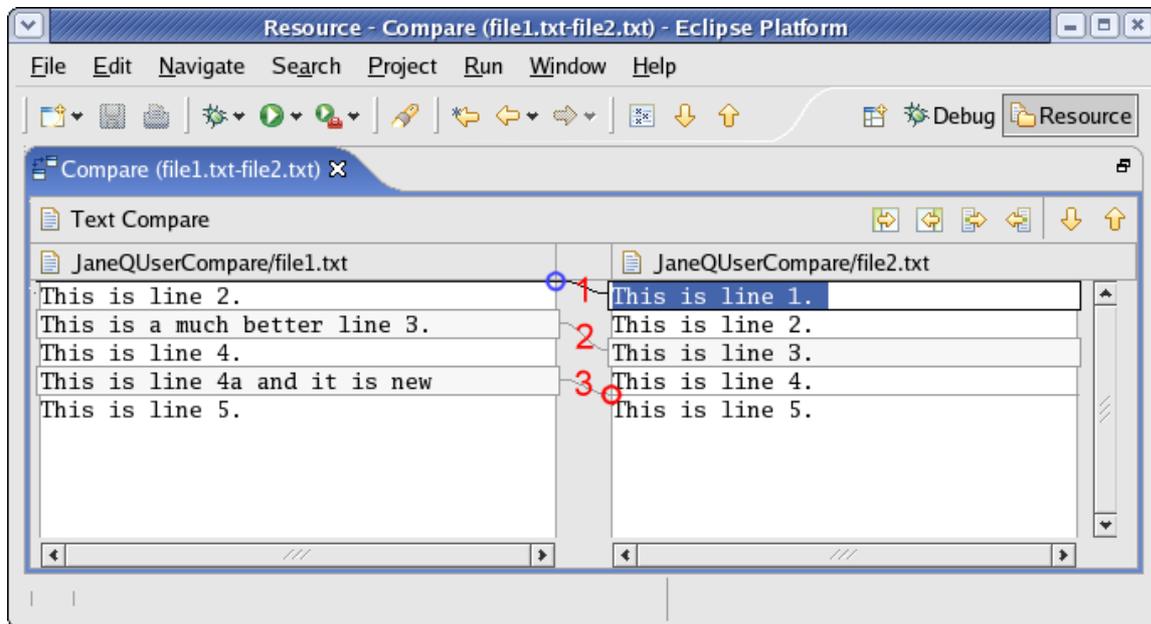
A special compare editor opens. In the next section you will see how to use this compare editor.



Understanding the comparison

Comparing file1.txt and file2.txt resulted in the following compare editor. The left side shows the contents of file1.txt and the right side shows the contents of file2.txt. The lines connecting the left and right panes indicate the differences between the files.

If you need more room to look at the comparison, you can double-click on the editor tab to maximize the editor.



Looking at the numbered changes, work your way down the left side of the difference editor.

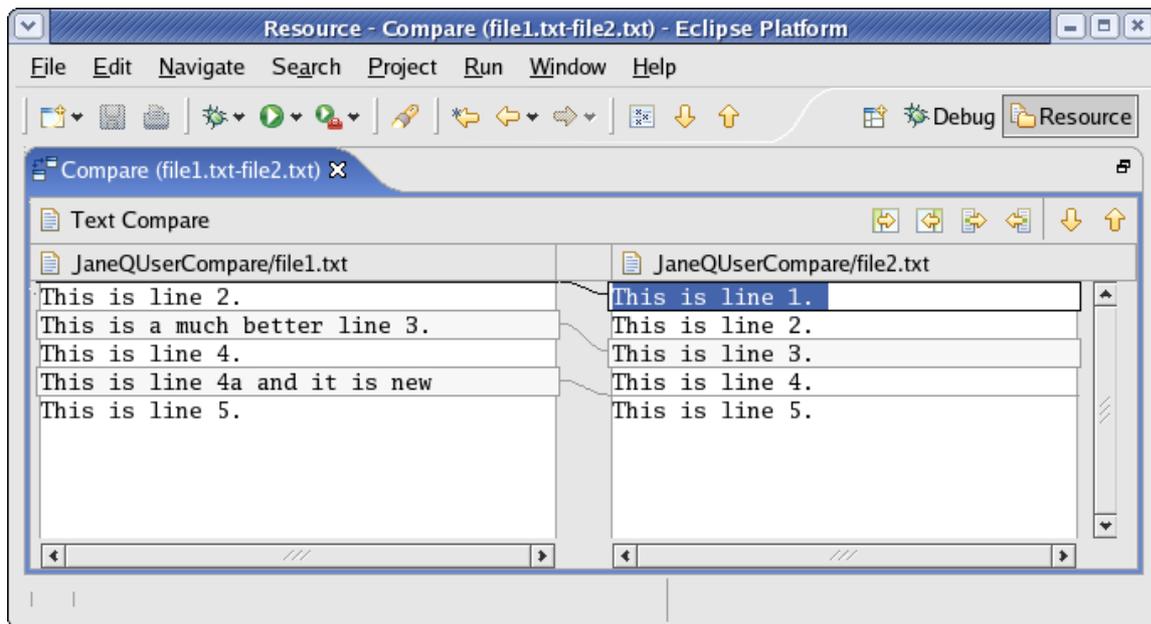
- Starting with the top line (in the left pane) you can see that the difference bar (in the area of the blue circle) indicates something is missing from the very top of the left file. If you follow the difference band (see #1) to the right file you can see that the it contains "**This is line 1.**"
- The next line "**This is line 2.**" is white indicating it matches the right file.
- Moving onto the next line (colored in the background color), you can see that the left file and right file have different contents for this line (see #2).
- The next line (**This is line 4**) is once again in white, so both files are the same.
- The next line exists in the left file but since it is gray you follow its difference bar to the right (see #3) and notice that the right file does not contain the line (see red circle).

Initially the compare editor might seem a bit daunting, but just you work down the left side and focus on the items marked as gray and those items missing from the left side.

Note: Spaces are significant; if two lines contain the same text but one has an extra space character, those lines will be displayed as being different.

Working with the comparison

Comparing file1.txt and file2.txt resulted in the following compare editor. In this section you will learn how to use the compare editor to resolve the differences between the two files.



There are two parts to the compare editor's local toolbar. The right group of local toolbar buttons (#1) enables you to move to the next or previous change.



1. Click the Select Next Change button . Observe how it selects the next difference.
2. Click the Select Next Change button a second time to go to the next change.
3. Click the Select Previous Change  button.

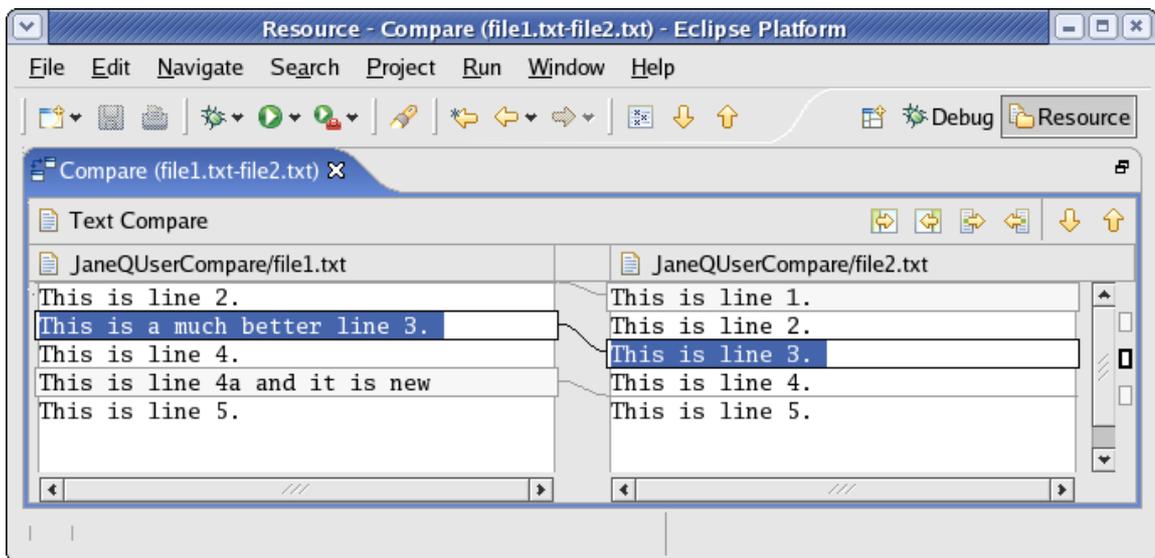
Eclipse Tutorials

The left group of local toolbar buttons (see #2) enables you to merge changes from the left file to the right file and vice versa. There are four types of merges you can perform:

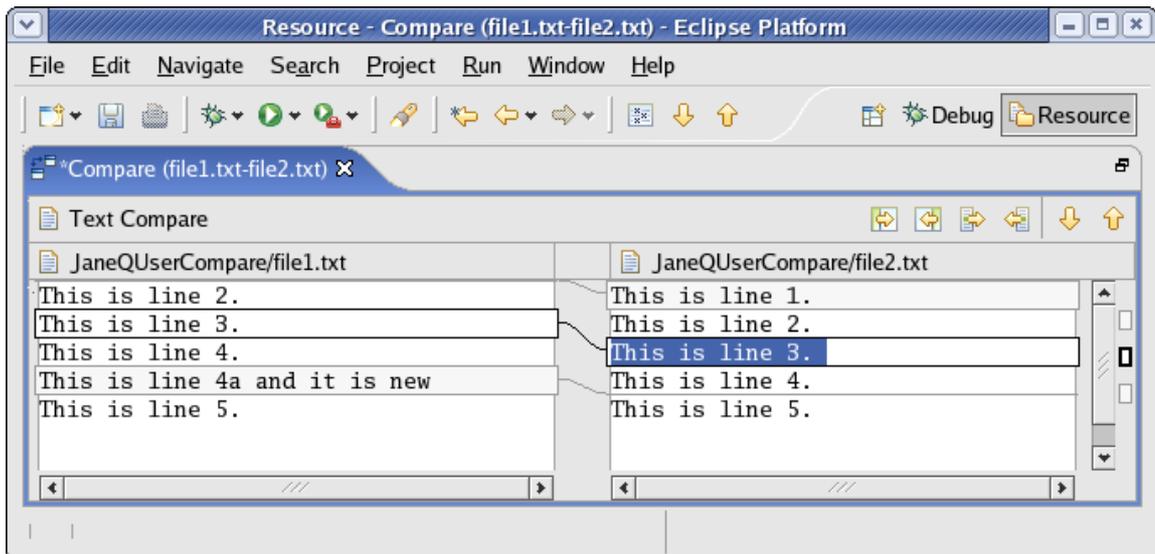
- Copy whole document from left to right.
- Copy whole document from right to left.
- Copy current change from left to right.
- Copy current change from right to left.

Typically the "copy whole document" actions are used when you know the entire file on either the left or right can just be replaced by the contents of the other file. The "copy current change" buttons allow you to merge a single change.

1. Ensure that the second difference is selected (as shown below):



2. Click the Copy current change from right to left button. Observe that the selected text from the right file is copied to the left file.



3. Close the compare editor and choose Yes to save the changes. Alternatively you can save the changes by choosing File > Save or by pressing [Ctrl]+[S].

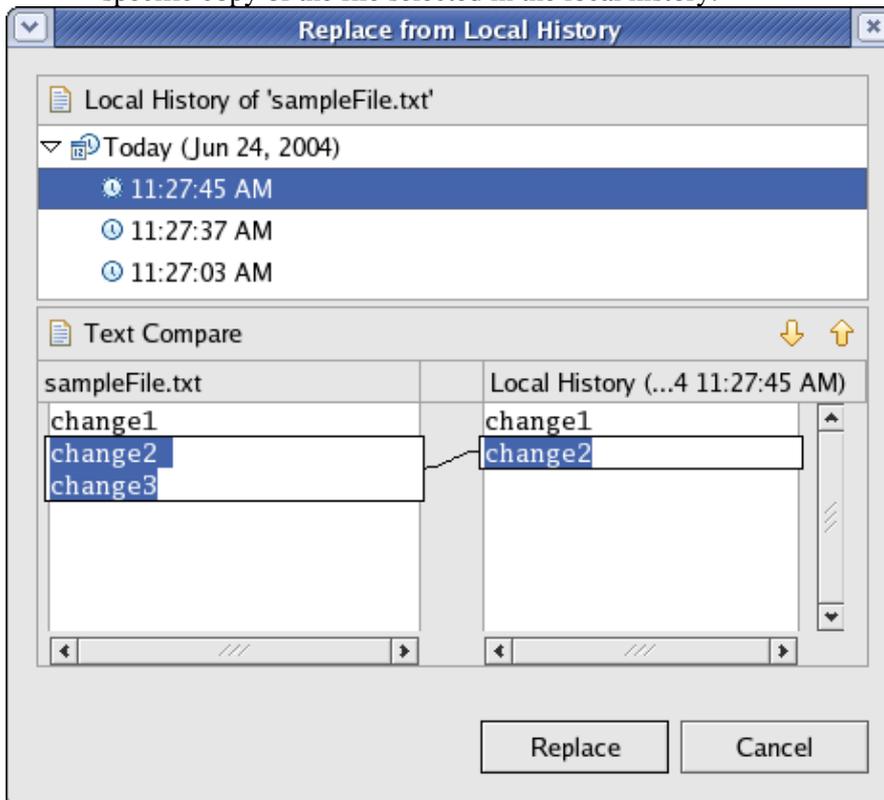
Local history

Every time you save an editable file in the Workbench, the Workbench updates the local history of that file and logs the changes that you have made to it. You can then access the local history of a file and revert to a previously saved copy of the file, as long as the desired state is recent enough in the save history.

1. Create a new file named: sampleFile.txt
2. In the editor for sampleFile.txt, modify the resource by adding the line **change1** and saving the file.
3. Enter a new line **change2** and save it again.
4. Add a third line **change3** and save it again.
5. In the Navigator view, right-click on sampleFile.txt and select Replace With > Local History.

The Replace from Local History dialog opens and shows the previous local history of the file (see the figure below):

- ◆ The local history section is at the top of the dialog.
- ◆ The left pane of the dialog contains the Workbench's current copy of the file.
- ◆ The right pane in the bottom area of the dialog shows the contents of the file selected in the local history tree. This enables you to see the differences between the Workbench file and the specific copy of the file selected in the local history.



6. The first item in the local history contains the last saved copy of the file. Click on it. The right pane in the bottom area of the dialog shows that this is the version that has only two lines of text.

7. The second item in the local history contains the next saved copy of the file. Click on it. The right pane shows that this is the version that has only one line of text.

The bottommost entry in the tree is the creation time of the file. It has no contents.

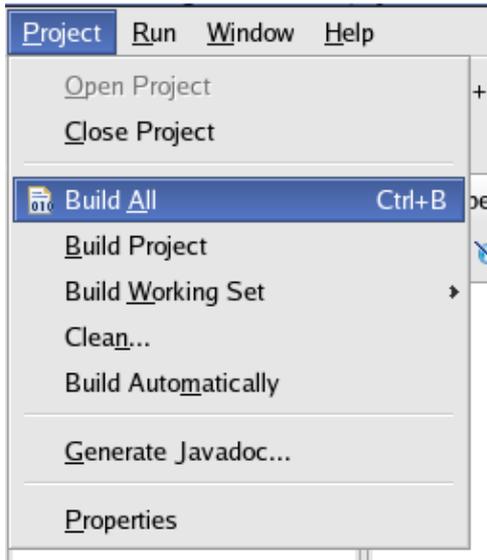
8. Select the first item (shown above) in the local history.
9. Click Replace. This replaces the Workbench's copy of sampleFile.txt with the chosen local history item.

Observe that the sampleFile.txt editor now contains two lines.

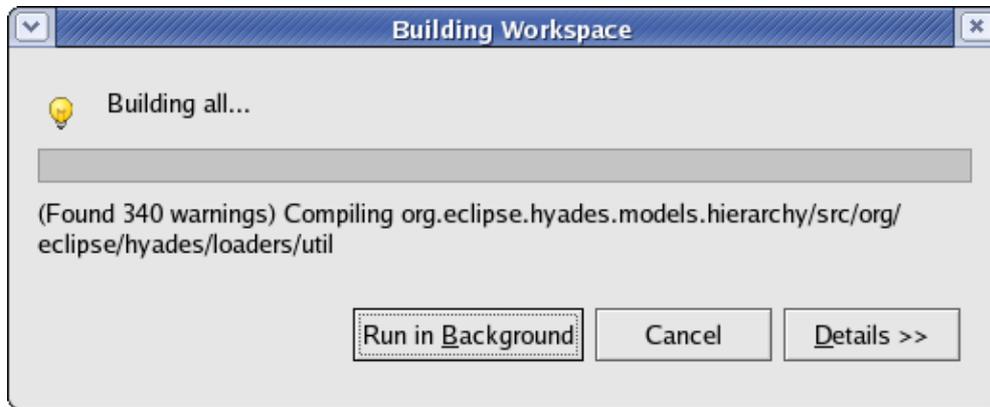
Responsive UI

By default, all Eclipse operations run in the user-interface thread. Employing the Responsive UI, which allows the threading of sequential code, enables you to continue working elsewhere in Eclipse. Without the Responsive UI activated in the user-interface thread, you would be locked out of performing other actions during a slow operation.

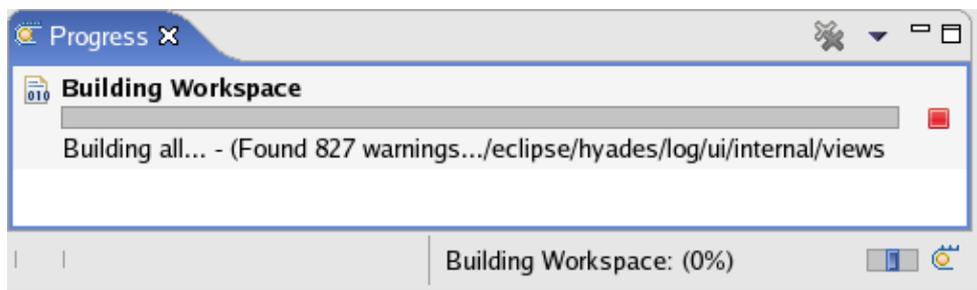
While operations with auto-build enabled are automatically run in the background, in many cases a dialog is displayed to provide you with the option to run an operation in the background. For example, building a project can sometimes take more than a few minutes, during which time you may wish to continue to use other functions in Eclipse.



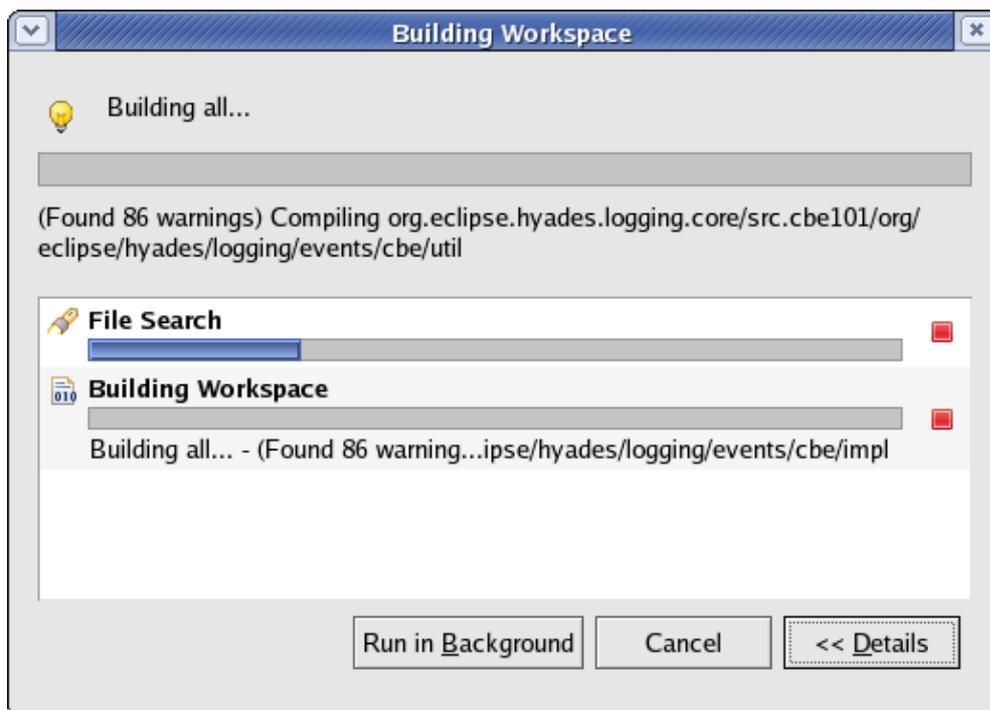
While the project is being built, select Run in Background from the Building Workspace dialog and the Responsive UI will allow you to carry on with other tasks in Eclipse.



For information on the status of the action and additional operations that are currently running, click Details.

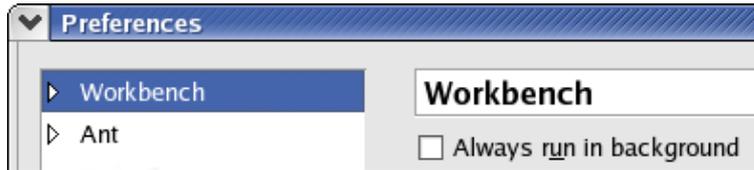


The Details panel displays the status information of the operation at hand as well as any additional operations that may be running simultaneously.



The Progress Information view also indicates when one operation is being blocked by another.

To have operations running in the background set as the default, select **Window > Preferences > Workbench** and check **Always run in background**.



Exiting the Workbench

Each time you exit the Workbench, the Workbench is automatically saved, including all open perspectives and windows. The next time you reopen the Workbench, it will appear exactly as it was when you closed it.

To exit the Workbench, click **File > Exit**.

Note: If you close the Workbench by clicking the window–close button (X), a dialog asks if you really want to quit and presents the option to turn off this prompt in the future. If you accept this option now, you can later restore prompting by clicking **Window > Preferences > Workbench > Startup and Shutdown** and check **Confirm exit when closing last window**.

Team CVS tutorial

In this tutorial, you will learn to use the CVS team capabilities built into the Workbench. The steps in this chapter are team oriented; you cannot (easily) complete these tutorials if others are not simultaneously working through this chapter.

You will learn how to work on your project, then commit changes to the repository for others on the team to use. As you continue to work on the project alongside other users, you will see how to commit resources that other users may be simultaneously working on, and how to update your workspace with changes others make in the project.

Remember, before you get started, you need to have at least one co-worker working through these steps with you and you need to have access to a CVS repository.

Setting up a CVS repository

A repository is a persistent store that coordinates multi-user access to the resources being developed by a team. The Workbench comes with CVS support built-in. The CVS server is available at <http://www.cvshome.org>.

Please refer to this site for information on how to install and configure your CVS repository including user access and passwords.

In order to continue with this tutorial you must have access to a CVS repository.

Starting offline

You will start your team CVS tutorial by working offline and creating a simple project. Once the project is made you will see how to commit it to the repository.

1. Create a new "Simple" project: from the menu bar select File > New > Project. Use your name as the project name (for example, JanesTeamProject).

The new project appears in the Navigator.

2. In the Navigator, right-click on JanesTeamProject and choose New > Folder. Create a folder named folder1.
3. In the Navigator, right-click on Folder and choose New > File. Create two text files in folder1 called file1.txt and file2.txt. Their contents should be as follows:

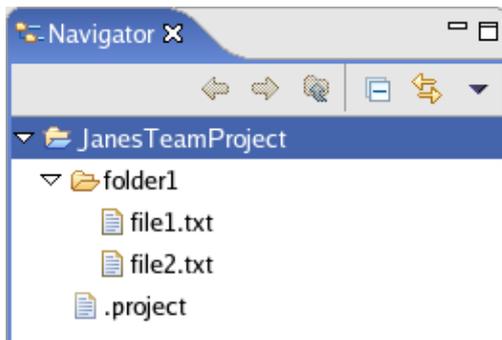
file1.txt

```
These are the contents  
of file 1.
```

file2.txt

```
File2 is a small file  
with simple text.
```

The Navigator should now appear as follows:



You can continue to work with your project in this mode but unless you commit the project into the repository others on your team will be unable to work on the project with you. In the next few sections you will see how to commit your project to the repository.

Sharing your project

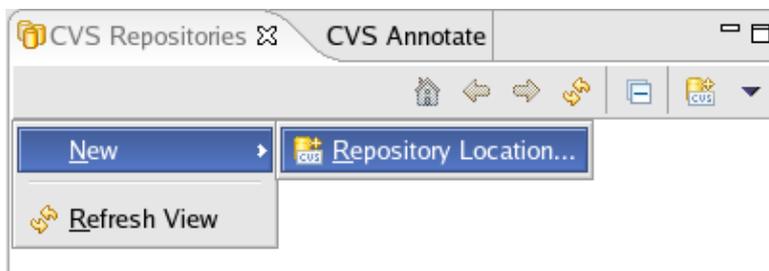
You have created our project in offline mode. To make this project available to other team members you need to do two things:

1. Specify the location of your team's CVS repository.
2. Commit the work into that repository.

Specifying a repository location

Before you can share your project with other users, you must first specify an available repository.

1. Open the CVS Repository Exploring perspective. (From the menu bar select Window > Open Perspective > CVS Repository Exploring.) The top left view shows all of the CVS repositories that you are currently working with. As you can see, the view is empty, meaning you still need to specify a repository.
2. Right-click on the CVS Repositories view choose New > Repository Location.



3. In the CVS Repository Location wizard you need to fill in the location of your repository and your login information. You may require assistance from your repository administrator in order to fill in the necessary information.

Add a new CVS Repository

Add a new CVS Repository to the CVS Repositories view

Location

Host:

Repository path:

Authentication

User:

Password:

Connection

Connection type: pserver

Use Default Port

Use Port:

Validate Connection on Finish

Save Password

⚠ Saved passwords are stored on your computer in a file that's difficult, but not impossible, for an intruder to read.

Finish Cancel

4. In the Host field, type the address of the host (for example, **teamsamples.com**).
5. In the Repository path field, type the path for the repository at the host address (for example, **/home/cvsroot/repositoryName**).
6. In the User field, type the user name under which you want to connect.
7. In the Password field, type your password.
8. In the Connection type field, select the type of CVS connection for the repository. (The default is pserver).
9. Leave Use Default Port enabled and Validate location on finish checked.
10. Click Finish when you are done.

Because Validate location on finish was checked, the wizard attempts to validate the information by connecting to the repository. In doing so it may prompt you for your password. Note that the repository connection is used only to validate the information.

Observe that the Repositories view now shows the new repository location.



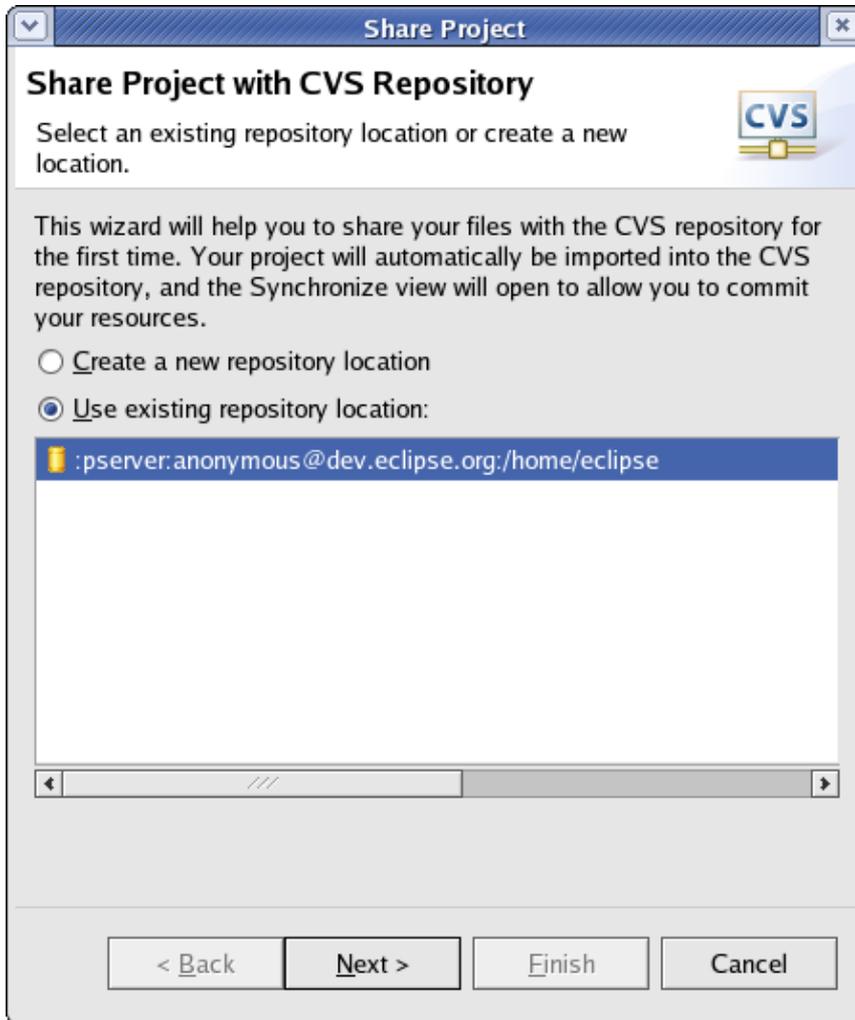
What is a repository location?

You have just added a new repository location to the CVS Repositories view. This is probably a good time to describe what a "repository location" is and what it is not: a repository location is an address, not an active connection; a connection is made from the address as required to perform CVS operations.

Synchronizing your project

You have created your project and specified your repository location. Now you need to make your project available to other team members.

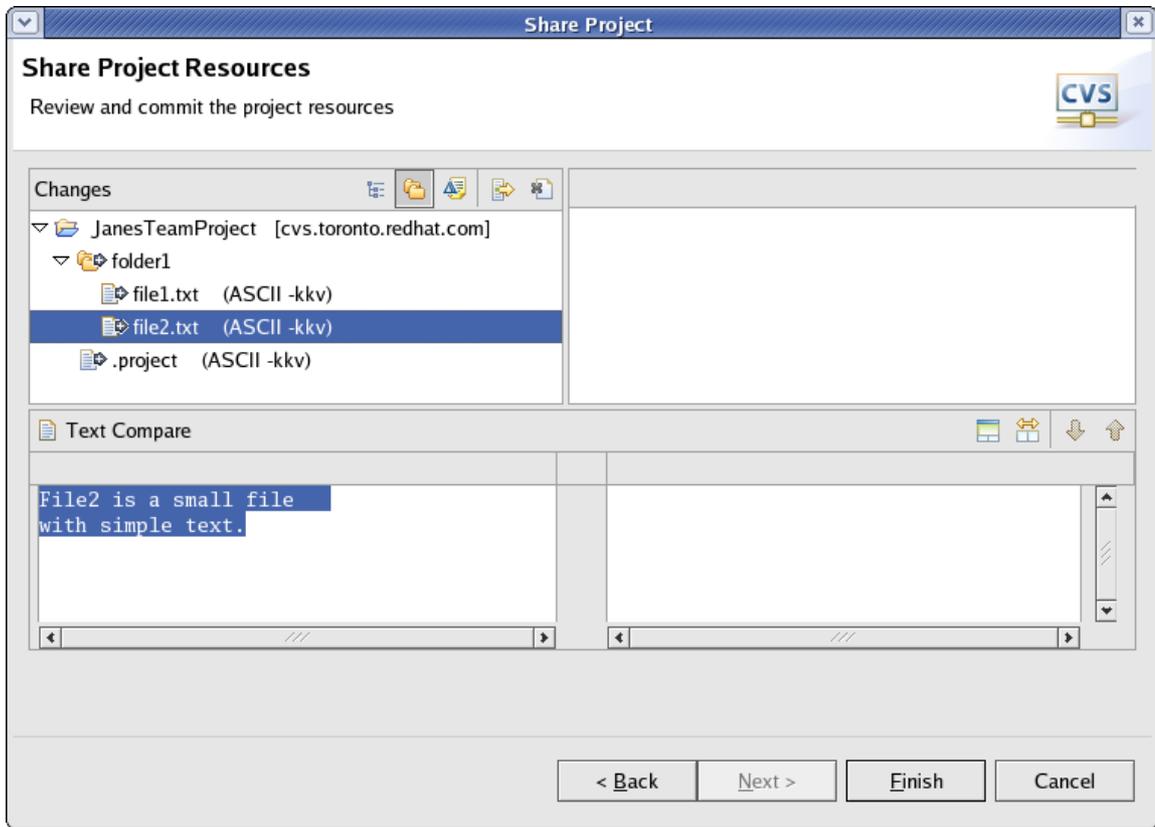
1. In the Navigator view select the project JanesTeamProject.
2. Right-click on the project and choose Team > Share Project. If you have more than one repository provider installed, select CVS and click Next.
3. In the sharing wizard page, select the location you created previously.



4. The next page asks you the module name to create on the server. Use the default value and use the name of the project you are sharing. Click Next.



The next page displays the files to be shared with your team. The arrows with the plus sign show that the files are new outgoing additions. They do not yet exist on the server.



5. Click Finish. Answer Yes and then Yes again when asked to add the new files. Finally, you have to enter a commit comment to describe the commit you are making. You can enter anything you like.

Now you have shared the project and all the files have been committed to the CVS repository. Others can now see your files.

Working with another user

You have seen how to create a project and commit it to the repository. You first specified your repository location. Next you shared your project with that location and the HEAD branch. Then the resources were added and committed.

The repository currently contains your project and the three files that you committed (file1.txt, file2.txt, and file3.txt).

It is time to find that co-worker (call him Fred) to work through some steps with you. In this section you will learn how two people can work on the same project and simultaneously commit changes to the repository. Specifically, you will do the following:

- Fred will add the project to his Workbench.
- Fred will make changes to file1.txt and file2.txt.
- You will update these new changes and also add a new file called file3.txt.
- Fred will make more changes to some files while you change the same files. Fred will release his changes first.

- You will have to resolve the conflicts between files that you both changed.

CVS terminology

This is some CVS terminology commonly used to describe changes:

incoming

Changes that are in the repository, which you have not yet updated to.

outgoing

Changes that you are committing.

conflict

Both you and the repository have modifications to the same resource.

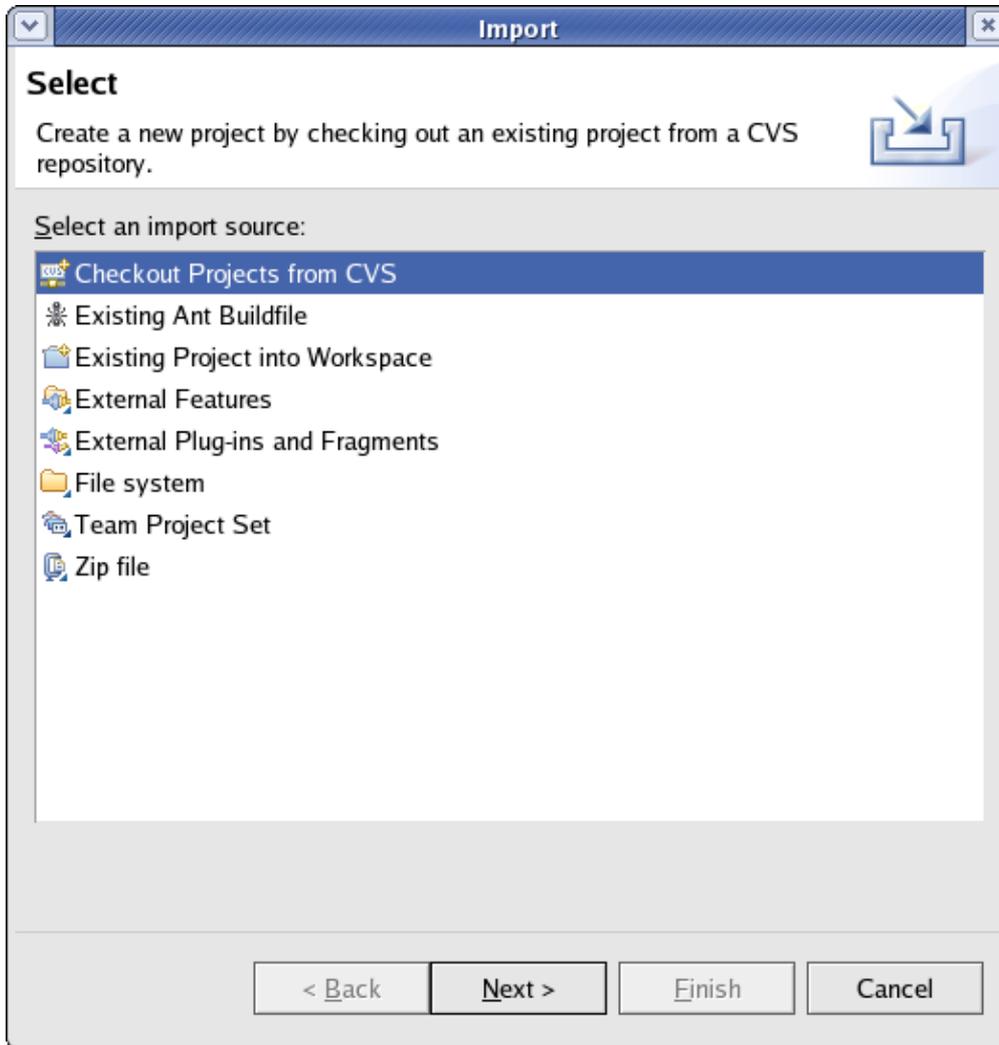
Checking out a project

Co-worker Fred has several tasks in front of him:

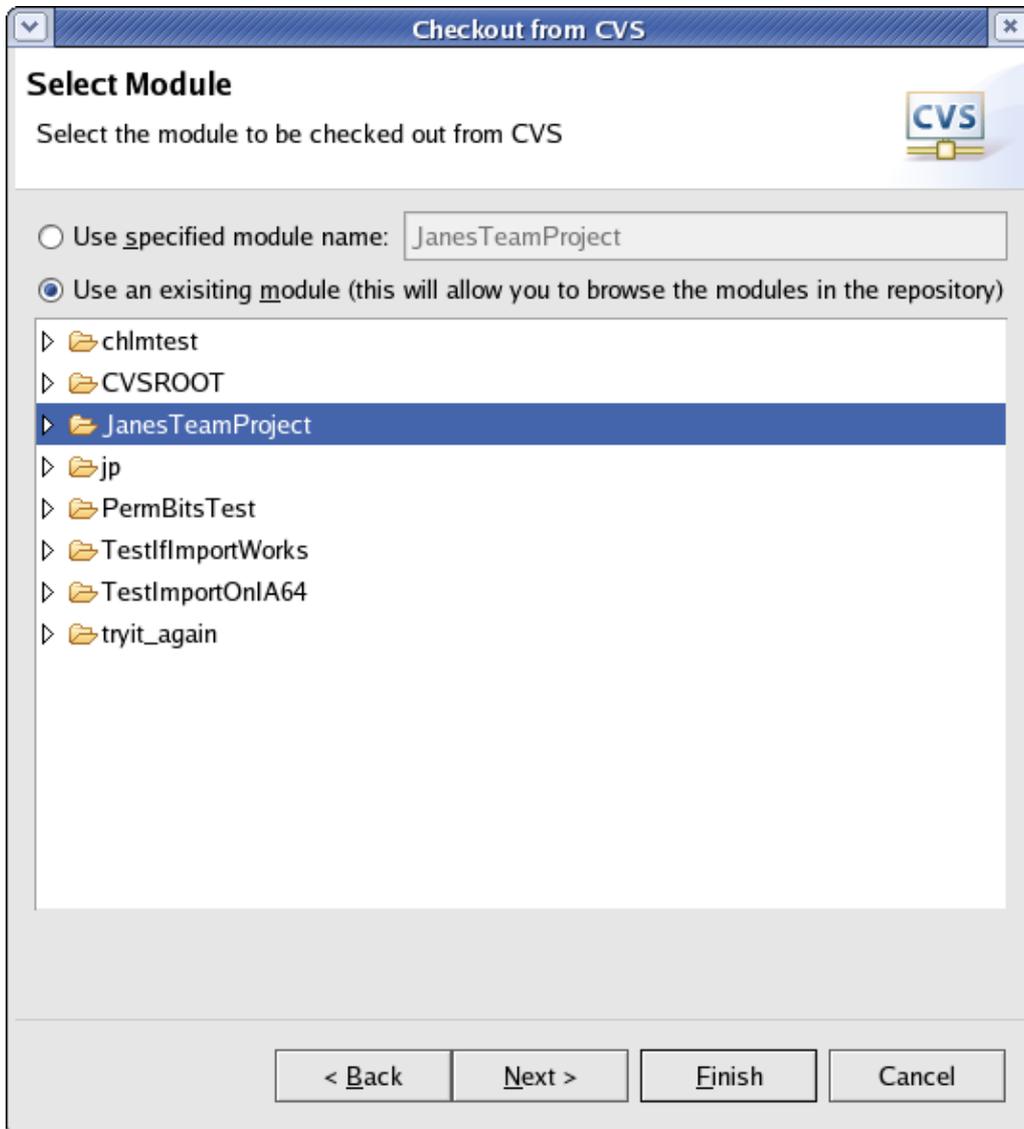
- Fred will import into his Workbench the project that Jane committed to the CVS repository.
- Fred will make changes to file1.txt and file2.txt.
- Fred will synchronize and commit his outgoing changes to the two files.

Fred's first step is to import the project into his workspace as follows:

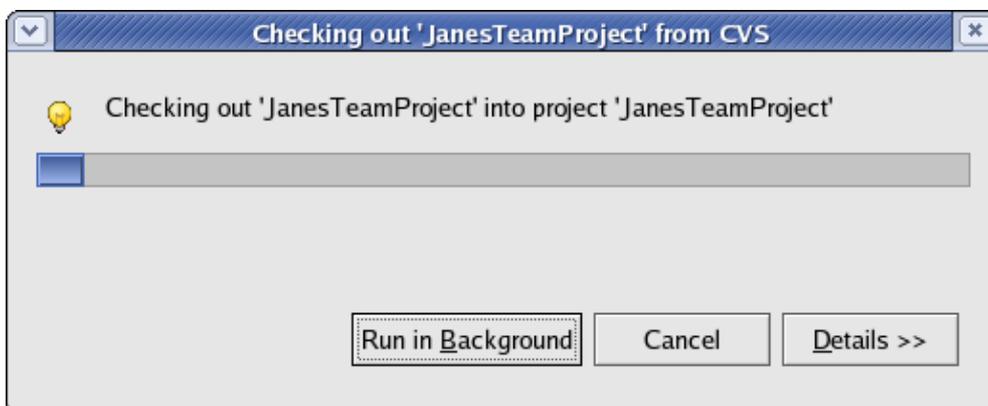
1. Click File > Import to run the Import wizard.



2. Select the Checkout Projects from CVS item and click Next.
3. Create a repository location as described when Jane created a repository.
4. On the next page, select the Use an existing module radio button and wait while the repository is contacted.



5. From the list of available projects, select `JanesTeamProject` and click `Finish`. A progress dialog shows the status of the import operation.



6. Open the Navigator view and observe that it now includes the project `JanesTeamProject`. Notice that there are CVS decorators indicating the file revisions, the repository name, and the file types.

Another user making changes

Now that Fred has the project in his Workbench, you and he need to modify several of the files and synchronize with the repository in order to commit them.

- Fred will make changes to file1.txt and file2.txt.
- Fred will synchronize and commit his (outgoing) changes to the two files.

Fred should proceed as follows:

1. Modify file1.txt as follows

Original Contents:

```
These are the contents  
of file 1.
```

New Contents (changes shown in bold)

```
These are the contents  
Fred-update  
of file 1.
```

2. Modify file2.txt as follows

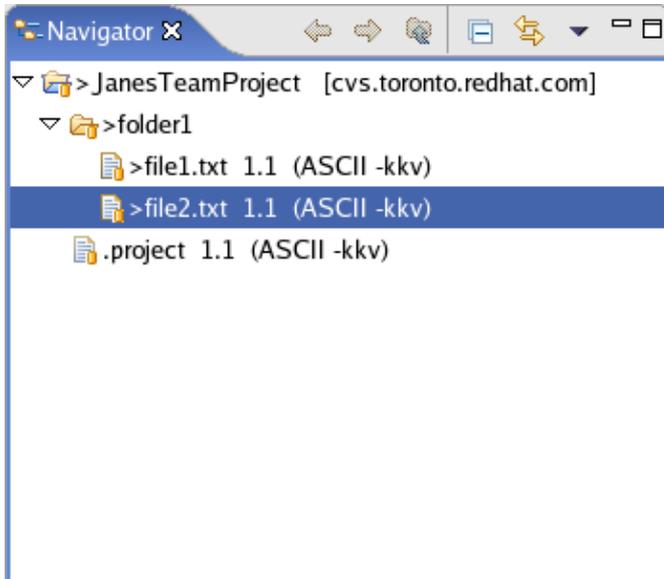
Original Contents:

```
File2 is a small file  
with simple text.
```

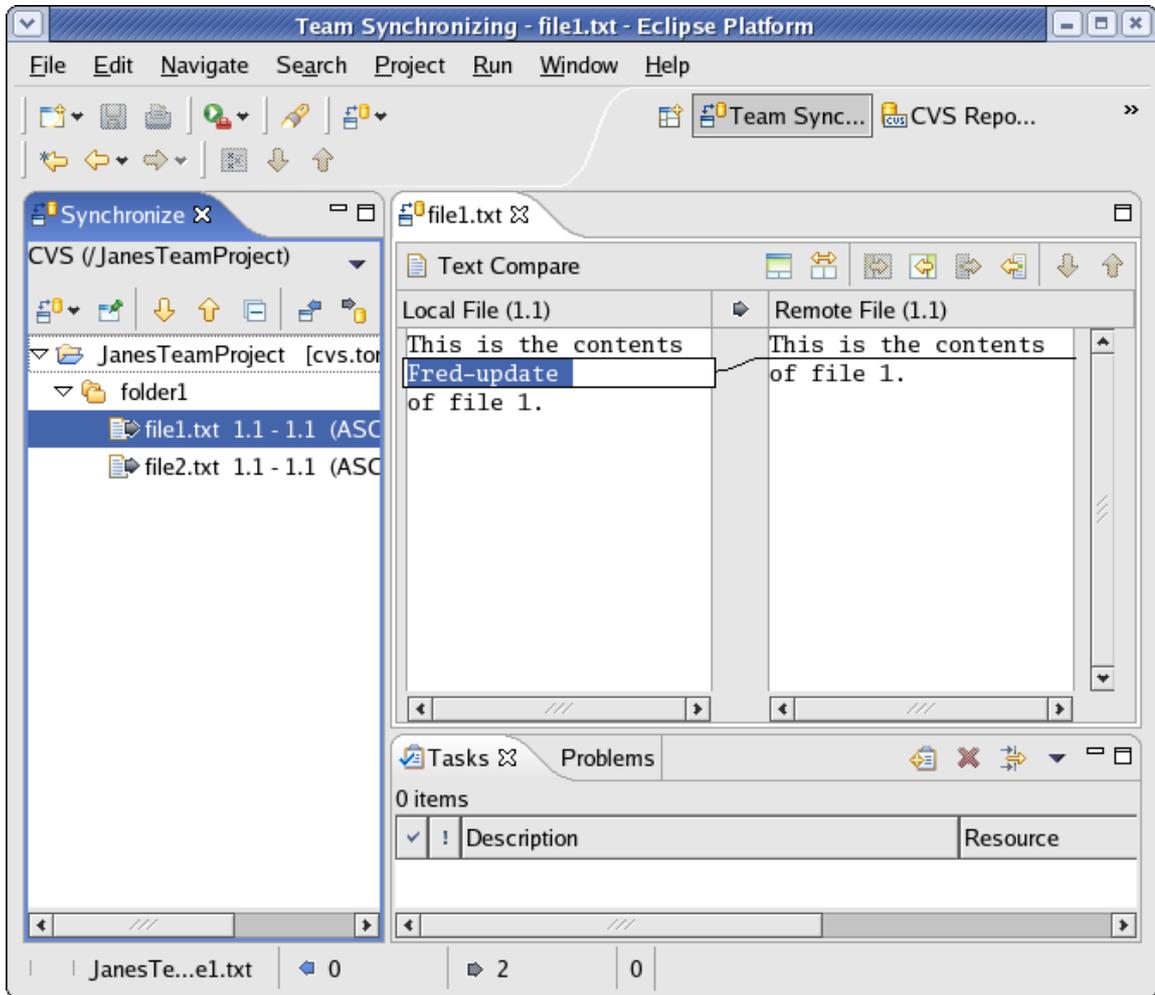
New Contents (changes shown in bold)

```
File2 is a (Fred was here) small file  
with simple text.
```

Observe that the Navigator updates to show the CVS state of a resource. Notice that the two files that Fred has changed are preceded by ">".



3. To commit his changes to the repository, Fred can either:
 - ◆ Select the two files and choose Team > Synchronize with Repository
 - ◆ Select the project and choose Team > Synchronize with RepositoryOften choosing the project is the easiest thing to do. To do that, right-click on the project and choose Team > Synchronize with Repository.
4. When the Synchronize view opens, Fred should commit his changes to file1.txt and file2.txt. To do that, right-click on the project and choose Team > Synchronize with Repository from its context menu. When you are asked to switch to the Team Synchronizing perspective, select Yes.



- When the Synchronize view opens, Fred can browse the changes he made. At the end, he should commit his changes to file1.txt and file2.txt by selecting JanesTeamProject in the Synchronize view, right-clicking, and selecting Commit from the context menu. A commit comment must be entered before committing the changes.

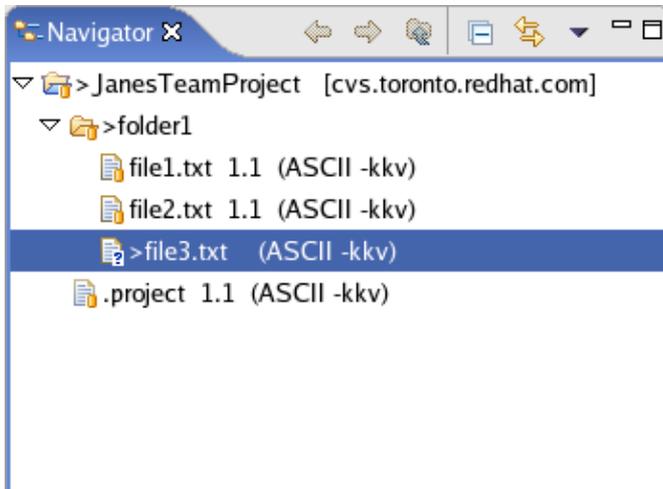
Making your own changes

Fred has made several changes to file1.txt and file2.txt and committed them to the repository. You now need to make some changes and then synchronize with the repository. When you synchronize, expect to see the changes you have made along with changes that have been made by Fred.

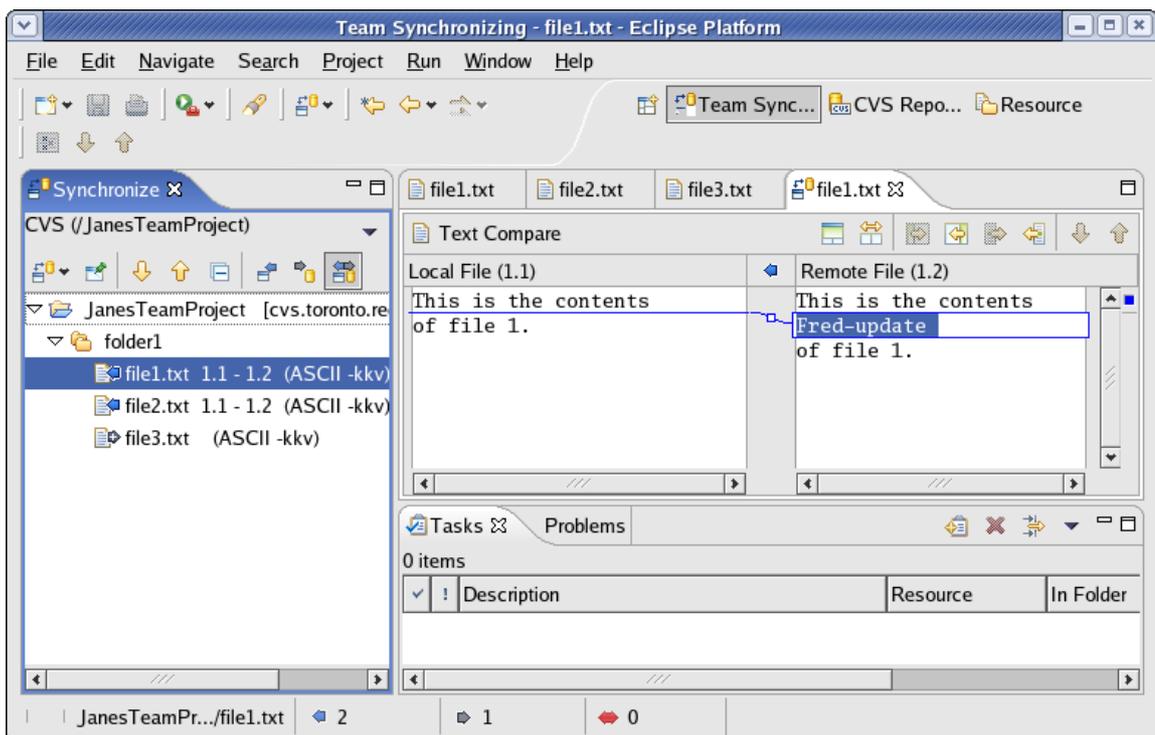
- Add file3.txt as follows:

```
This is the brief contents
of file 3
```

Observe that the Navigator updates the CVS state of a resource. Notice that the two files Fred changed are preceded by ">".



2. Select the project JanesTeamProject, right-click on it, and select Team > Synchronize with Repository. The Team Synchronizing perspective opens and the files you have changed appear in the Synchronize view. Double-click on file1.txt and the compare editor opens:

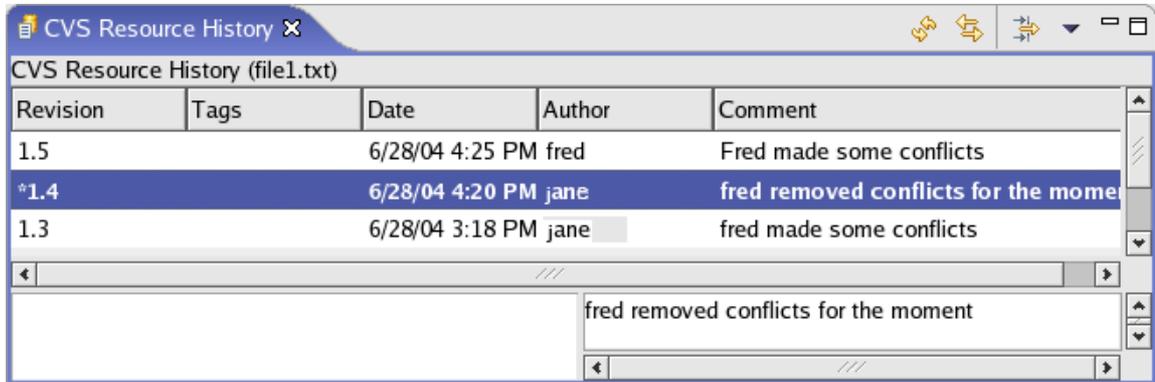


There are couple of other things to note. First, the icon next to file1.txt (in the Structured compare area) indicates that file1.txt has an incoming change. This means a change was released to the server which you need to take. Looking at file1.txt you can see the text that Fred has added to the file. Also, notice in the status line at the bottom of the window there are arrows with numbers beside them. These show the number of files you have incoming, outgoing, and in conflict. The first number beside a file is the revision you have in the workspace, and the other is the revision on the server when you last synchronized.

Normally you should update the changes made by others, test your workspace with the changes, then

commit your changes once you are sure that the new changes by others did not break anything in your workspace.

Before deciding to accept Fred's changes, you may want to find out why he made the changes. Select file1.txt, right-click and select Show in Resource History from the context menu.



The row starting with a * indicates that this is the current revision loaded. In this case, you can see the comment made by Fred when he released revision 1.3.

Tip: You can select the 'Link with editor' toolbar button in the History view to have the history automatically update when a new editor is opened. This allows for quick browsing of comments.

- To update, select JanesTeamProject in the Synchronize view, right click, and select Update from the context menu.

The Synchronize view updates to reflect the fact that file1.txt and file2.txt are no longer out-of-sync with the repository. You should have only file3.txt visible now.

- You can commit file3.txt.

Working with conflicting changes

There are cases where two users are editing the same files and, when the second to commit to the repository tries to commit changes, the repository does not allow the commit to succeed because of a conflict. Simulate this by making Fred and Jane change the same files.

- In Fred's workspace, open the Navigator and edit file1.txt. Make the text the following:

```
Fred line 1
This is the contents
Fred-update
of file 1.
```

- Have Fred also edit file2.txt with the following change:

```
File2 is a (Fred was here again) small file
with simple text.
```

- Fred commits his changes to the repository.
- At the same time, Jane makes changes to file1.txt. She adds the following line to the end of the file:

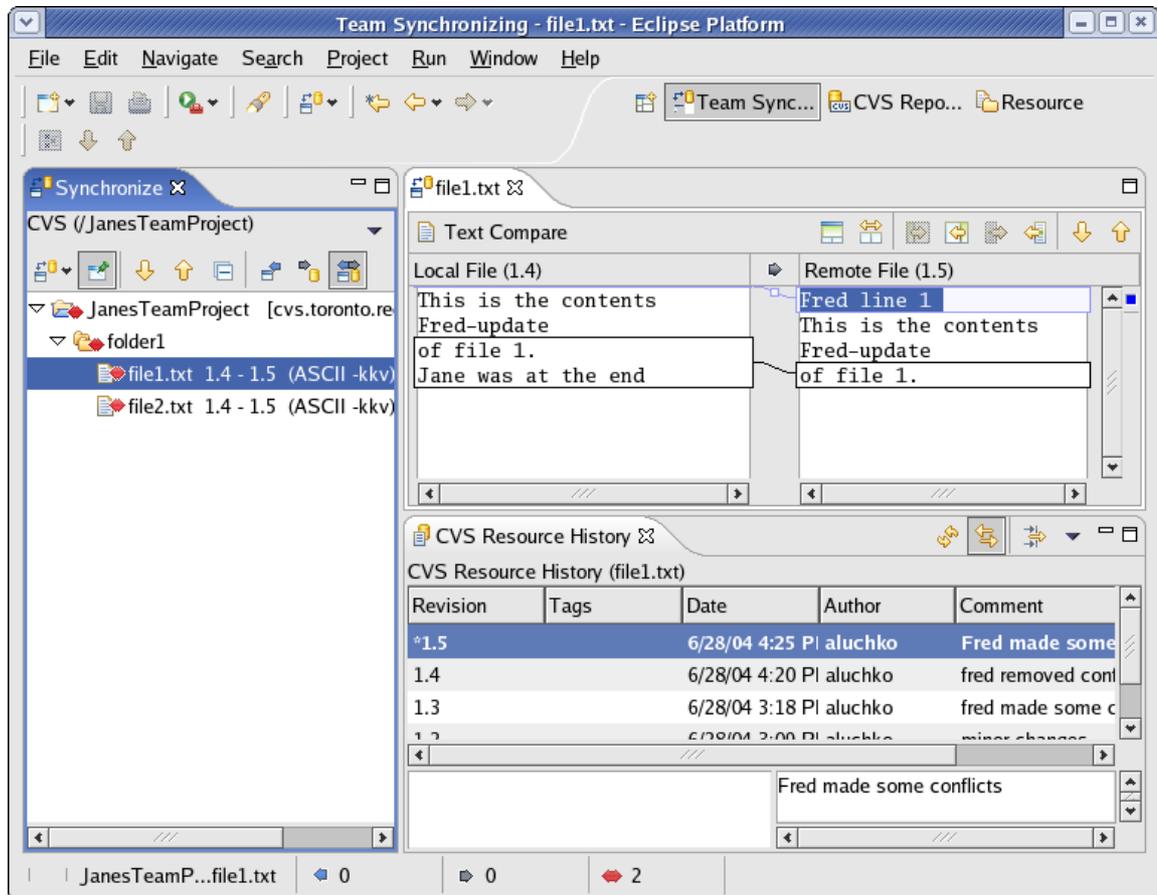
Eclipse Tutorials

```
This is the contents  
Fred-update  
of file 1.  
Jane was at the end
```

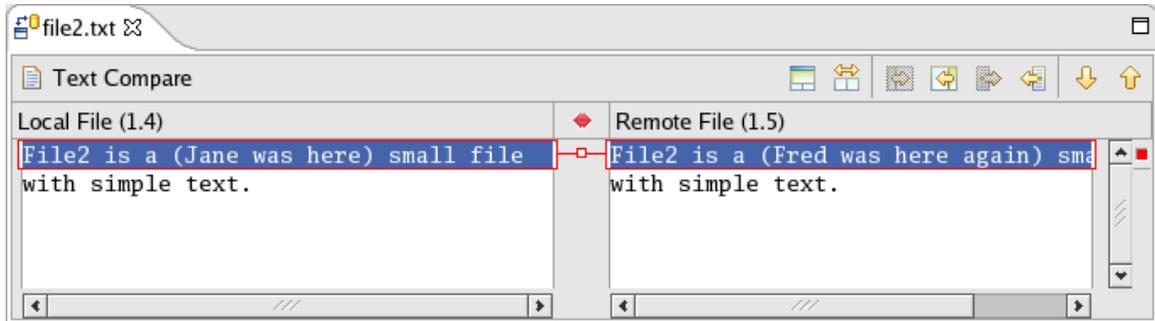
5. And finally, Jane changes file2.txt to the following:

```
File2 is a (Jane was here) small file  
with simple text.
```

6. When Jane finishes making changes, she synchronizes the project and finds the following appear in the Synchronize view:



7. Both file1.txt and file2.txt show with a red icon, indicating that they have conflicting changes. You cannot commit the files until the conflicts are resolved. Click on file1.txt and notice that Fred and Jane made changes to two different parts of the file. In this case, Jane can simply update the file and the lines added by Fred will be merged into Jane's local file. Select file1.txt, right-click, and select **Update**.
8. Next, double-click on file2.txt to see the conflict. In this case you can see that both Jane and Fred changed the same line. For this type of conflicting change, a regular update cannot resolve the conflict. Here you have three options (with the command to use in brackets): accept the changes from Fred (Override and Update), ignore Fred's changes (Mark as Merged), or manually merge the files within the compare editor.



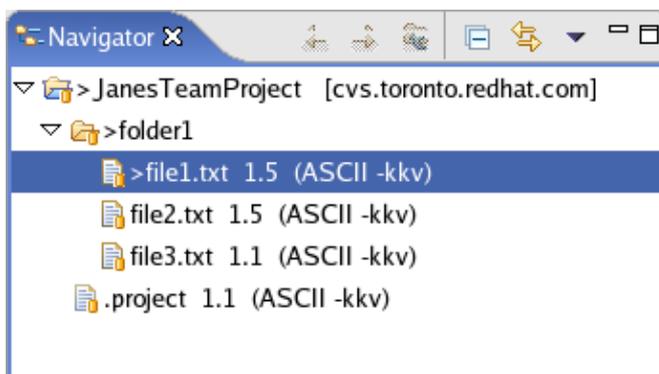
9. For this example, Jane updates file1.txt and selects override and update for file2.txt. After the operations are run, the conflicts turn into outgoing changes. Jane can review the changes and commit them.

Replacing

Suppose after further reflection you realize that the revision of file1.txt that you just received is incorrect and in fact you want an earlier revision. You can replace a Workbench resource with an earlier revision of the resource from the repository. To rollback to an earlier revision:

1. In the Navigator view, right-click on file1.txt and select Replace With > Revision.
2. In the Replace with revision Compare editor that opens, right-click on the original (bottom-most) revision and select Get Contents.

Observe that the Navigator updates to show the CVS state of a resource. Notice that the modified file is preceded by ">" to indicate that file1.txt has changed (by replacing it with the earlier version).



3. Now decide that this older revision is not as good as it initially seemed in fact, the revision in the repository is better after all.

Instead of choosing Team > Synchronize with Repository, choose Replace With > Latest from HEAD.

Observe that file1.txt is updated to be the contents from the repository, and that the leading indicator ">" has been removed since you now have the same revision as the repository.

You have seen how to synchronize with the repository, replace with revision, and replace with the latest from the repository. You can also compare with these revisions/versions in a similar manner by right-clicking on a file in the Navigator and selecting Compare With.

Versioning your project

Now that your project is complete, it is time to version it. If you were paying close attention while you were committing your file resources, you probably noticed that the repository automatically assigned them version numbers, or to be more accurate, "revision numbers". As you committed your files, the only information you needed to supply was a commit comment.

To version your project, proceed as follows:

1. Select the project JanesTeamProject.
2. In the Navigator view, select Team > Tag as Version.
3. When the Tag Resources dialog opens, type the version name **A1** and click OK.
4. Select the CVS Repositories view.
5. Expand Versions and JanesTeamProject. Observe that under JanesTeamProject there is now a version of this project whose version number is A1. Looking closely at the contents of folder1 you can also see that the version numbers of the individual files match the version numbers in your Workbench.

A quick review

Here are some of the more important but subtle issues associated with working in a repository:

- When you versioned the project, you did so by versioning the project as it appeared in your Workbench. For this reason it is important to synchronize the project with the repository (that is, the HEAD or the branch you are working in) prior to versioning it. Otherwise another user may have committed interesting changes to the project which you have yet to update to. If you proceed to version the project without updating, you will version it without these changes.
- The repository contains all projects in the repository. Individual users pick which projects they are interested in and check them out into the workspace. From that point on they are synchronizing those projects (only) with respect to the repository.
- The repository represents a large in-progress collection of all known projects. From the repository's perspective, everything in HEAD or on a branch is always open for change.
- The act of versioning a project effectively snapshots it and places it into the Versions section of the repository, however the repository branches are still open for change.
- It is important to first update to changes made to the repository, retest with those changes and your soon-to-be-committed changes and then commit your changes. By first taking the latest changes in the branch, and retesting, it helps to ensure that the changes you are about to commit will actually work with the current state of the branch.
- Each project is associated with a specific repository. Different projects can be associated with different repositories that may in fact be on completely different servers.

Tutorial for Ant and external tools

This section covers Eclipse's external tools framework, and its integration with [Ant](#), the build tool from the [Apache Software Foundation](#).

This tutorial assumes you have a basic knowledge of Ant. If you are not familiar with Ant, refer to the [Apache Ant manual](#).

The tutorial has three main parts:

- [The basics of working with Ant buildfiles in Eclipse](#)
 - [Eclipse examples that employ Ant buildfiles](#)
 - [The external tools framework and how to use non-Ant tools](#).
-

The basics of working with Ant buildfiles in Eclipse

This section covers the basics of working with Ant buildfiles in Eclipse:

- [Creating Ant buildfiles](#)
 - [Editing Ant buildfiles](#)
 - [Running Ant buildfiles](#)
 - [Saving and reusing Ant options](#)
 - [Using the Ant view](#)
-

Creating Ant buildfiles

Ant buildfiles are just text files, so the most straightforward way to create an Ant buildfile in Eclipse is:

1. Click **File > New > File**.
2. Give this new file the name: build.xml
3. Click Finish.

Note: By default, Eclipse recognizes build.xml as an Ant buildfile and enables Ant-related actions when it is selected. You can configure eclipse to recognize other files as Ant files from the Ant preferences dialog (Window > Preferences > Ant).

As you will see in a later section, you can create an Ant buildfile for an Eclipse plug-in that contains predefined targets. This is useful for deploying a plug-in.

Editing Ant buildfiles

Because Ant buildfiles are simple text files, you can use any text editor to edit them. But there are several advantages to using the Eclipse Ant editor, including syntax coloring, content assist, and an Outline view. To get familiar with the Eclipse Ant editor, you will create a simple Ant buildfile using this editor.

1. Create an Ant buildfile called build.xml (see the previous topic, if necessary).

- In the Navigator view, select build.xml, right-click, and choose **Open With > Ant Editor** from the file's context menu.
- Enter the following content in the editor:

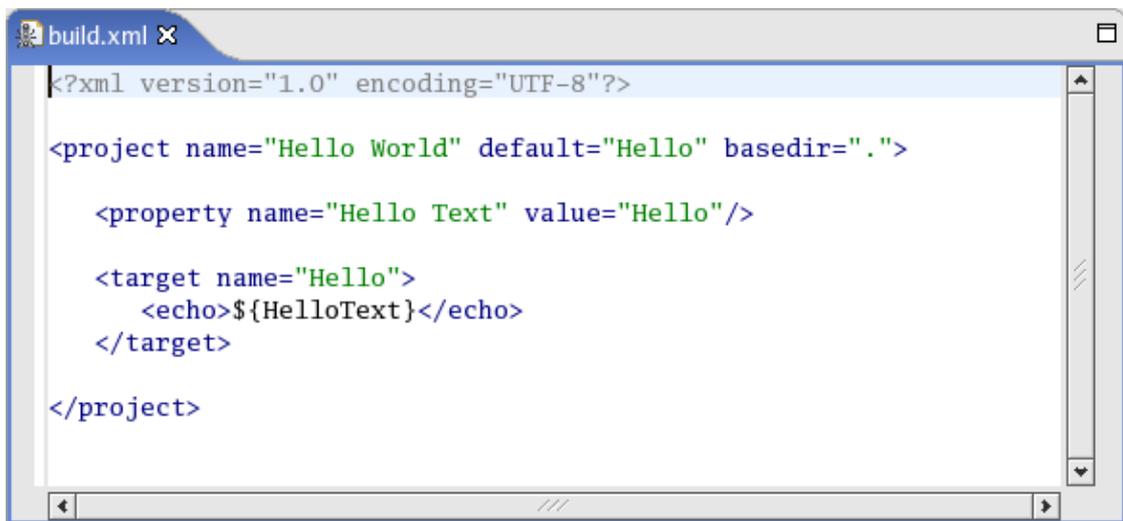
```
<?xml version="1.0" encoding="UTF-8"?>

<project name="Hello World" default="Hello" basedir=". ">

    <property name="Hello Text" value="Hello"/>

    <target name="Hello">
        <echo>${HelloText}</echo>
    </target>

</project>
```

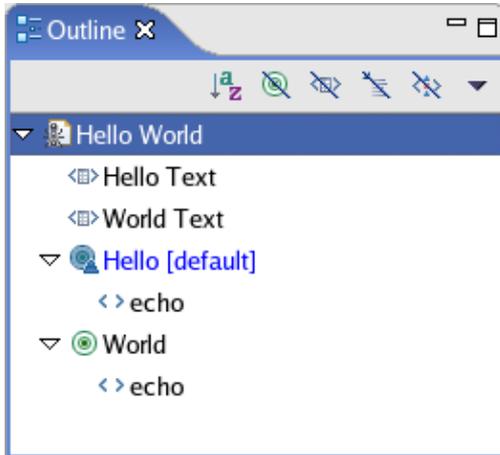


Notice the syntax coloring for property values. (You may have to save the file, close it, and re-open it to see the highlighting.)

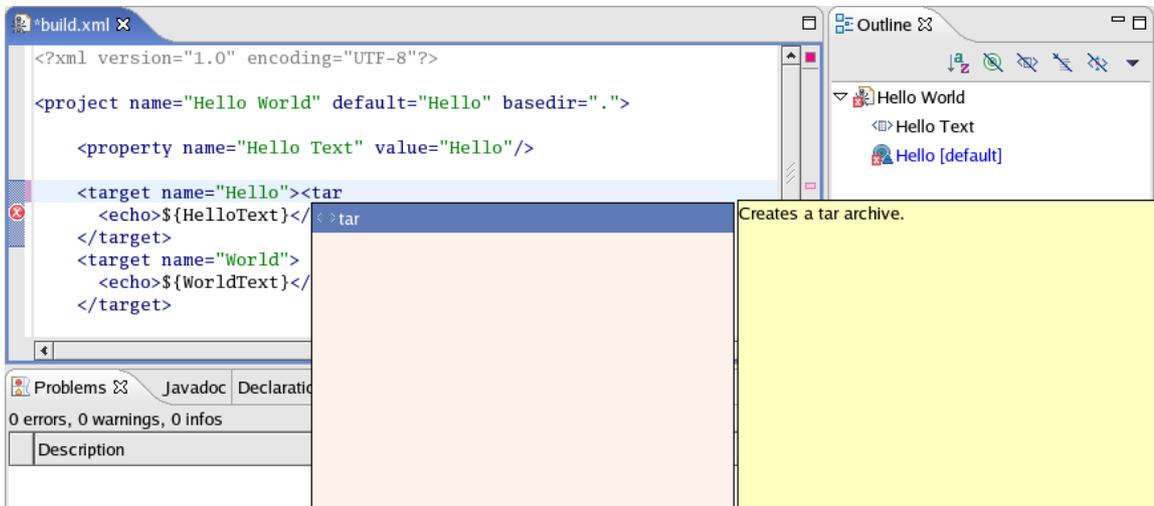
- Begin to enter a second target by typing: **<targ**

Content assist presents you with a list of valid completions. In this case there is only one, the **<target>** tag.

- Select the **<target>** completion. Notice that the editor inserts both the opening and closing tags and leaves the cursor positioned for you to enter properties for this tag.
- Name this target **World**.
- Enter an 'echo' task for this target that is similar to that for the Hello target, but change the text to **World**.
- Save the changes to build.xml.
- Make the Outline view visible and notice that there are entries for each property and each target. In addition, each task under a target has an entry.



10. Clicking on an entry in the Outline view scrolls the editor to that entry. In addition, when the Outline view has focus, typing a character will move the selection in the Outline view to the next visible entry beginning with that character.
11. Position the cursor just past the end of one of the '<target>' tags, then type '<tar>'. If necessary, press [Ctrl]–[Space] to activate content assist. Notice that now the only valid completion is the 'tar' tag. This is because the Ant editor knows that nested targets are not allowed. Previously, when you used content assist to create a target tag, the editor knew that a tar task was not allowed outside of a target.



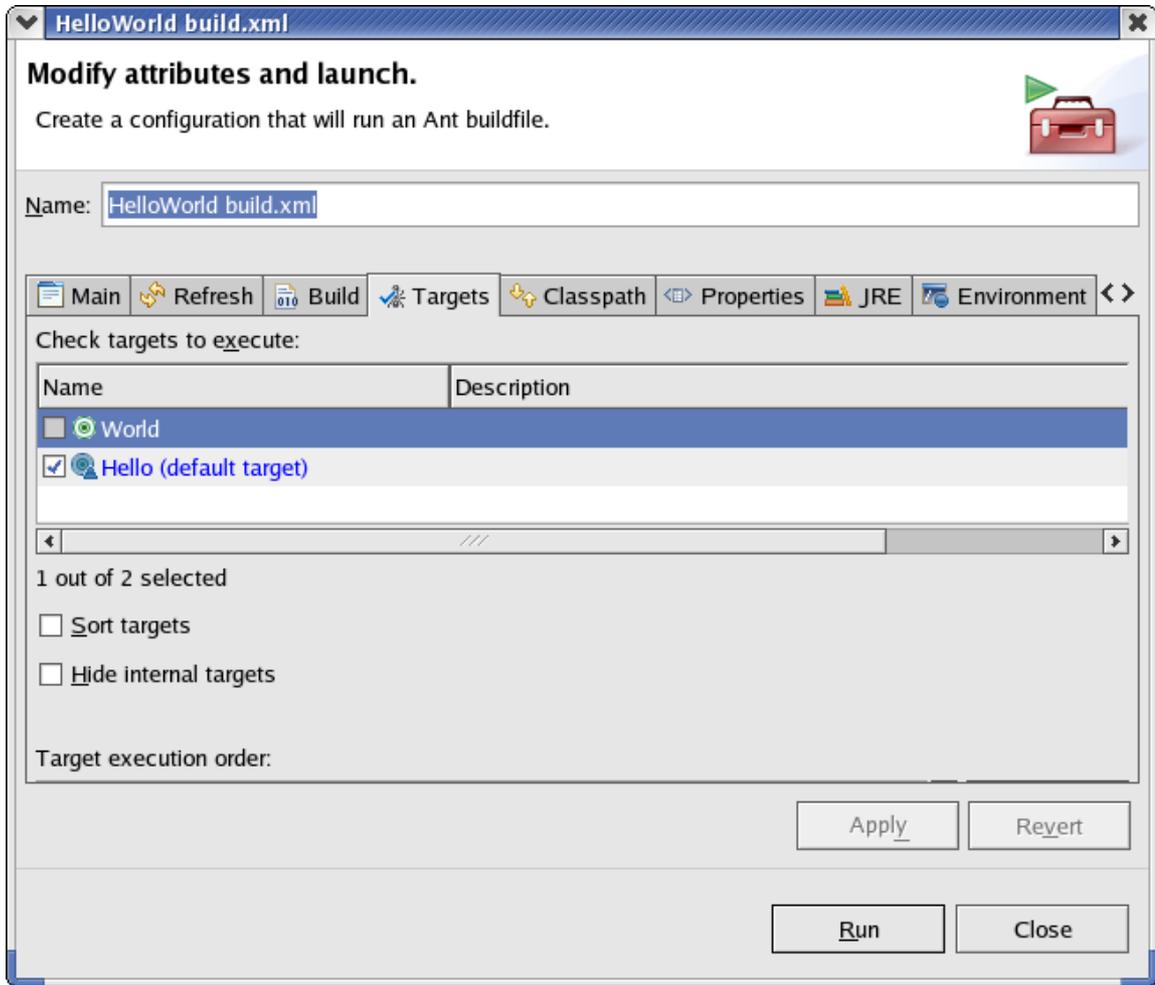
12. Close the editor and do not save changes.

Running Ant buildfiles

Any file with a .xml extension can be run as an Ant buildfile. Of course, not all such files really are Ant buildfiles, but no harm is done if you mistakenly attempt to run a non–Ant .xml file as an Ant buildfile.

This section covers the basic mechanisms for running Ant buildfiles in Eclipse, using the build.xml file created in the last section.

1. Select build.xml in the Navigator, right–click, and choose **Run > Ant Build** from the context menu. The Modify Attributes and Launch dialog appears.



This dialog enables you to configure many aspects of the way your Ant buildfile is run, but for now concentrate on the Targets tab, which allows you to select the Ant targets to run and their order.

2. Select both targets and leave the order as the default.
3. Click Run.

The Ant buildfile is run, and the output is sent to the Console view.

Note: If the buildfile is run so that the Hello target is executed first and other times so that the World target is first, the dialog does not need to be brought up each time nor does the ordering need to be changed.

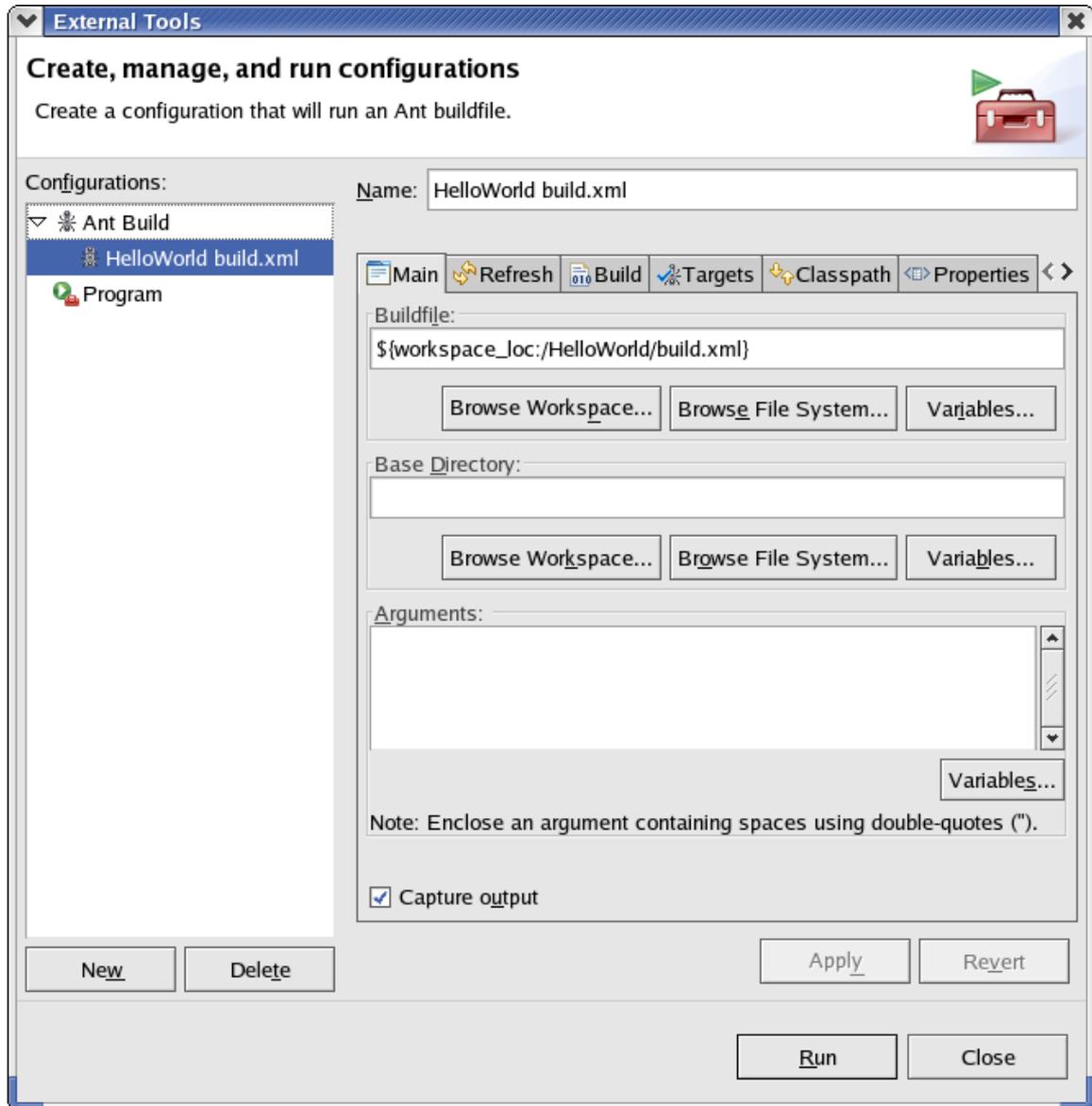
Saving and reusing Ant options

When you ran the build.xml Ant buildfile in the last section, the choice of targets, along with all other options in the Run Ant dialog, were saved in an entity called a 'launch configuration'. Launch configurations contain all details necessary to run a single Ant buildfile in a particular way. It is perfectly valid to have multiple launch configurations associated with a single Ant buildfile. So, in addition to the launch configuration that was created in the last step, specifying that your build.xml buildfile should execute the targets Hello and World in that order, you could create a second launch configuration for this same buildfile specifying the same targets but in the reverse order. So far, so good. But the really nice thing about launch configurations is that now you can quickly run your Ant buildfile in either configuration by simply specifying the

corresponding launch configuration.

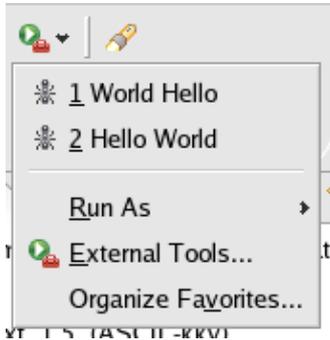
1. Click Run > External Tools > External Tools.

The External Tools dialog opens. This should look a lot like the Run Ant dialog you saw previously. In fact, it is identical except that in the External Tools dialog, you have a choice of which launch configuration you wish to view and edit. The launch configuration you created when you ran the build.xml buildfile is selected in the tree at the left, and the tabs on the right show the options for this launch configuration.



2. At the top of the dialog, change the Name to 'Hello World' and Apply the change.
3. In the tree at left, bring up on the context menu on the selected launch configuration and choose **Duplicate**. A copy of the launch configuration for the Hello World buildfile is created, '(1)' is appended to the name, and the new launch configuration is selected in the tree.
4. Rename the new configuration to **World Hello**.
5. In the Targets tab:

- a. Make sure the Hide internal targets box is unchecked.
 - b. In Check targets to execute make sure that both targets are checked.
 - c. Click the Order button
 - d. Change the order of the targets so that the World target executes first, then Apply the change.
6. Click Run.
 7. As before, the HelloWorld.xml buildfile runs and sends its output to the Console view. This time, however, because the targets were reversed, the output is reversed as well.
 8. Go to the External Tools drop down in the toolbar.

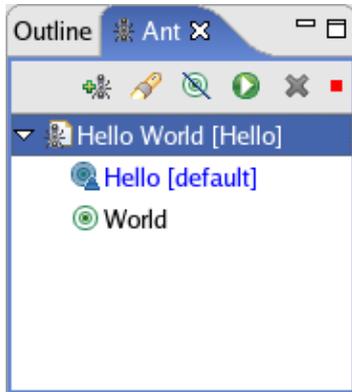


Notice now that there are two entries in the history, one for Hello World and one for World Hello. In order to rerun the Ant buildfile so that it outputs Hello World, just select this launch configuration in the history. To rerun the launch configuration that outputs World Hello, select this configuration in the history. Note that the history is ordered so that the most frequently run configurations are at the top of the dropdown.

Using the Ant view

Eclipse provides a standard view, the Ant view, that lets you work with your Ant buildfiles. This view is tree-structured, showing Ant buildfiles as top-level entries, then targets and subtargets as children. The main advantage of this view is that you can work with all of your Ant buildfiles in one place, as opposed to hunting them down in the Navigator.

1. Open the Ant view by clicking **Window > Show View > Other > Ant > Ant**.
2. By default, the Ant view is empty. There are two ways to add Ant buildfiles to this view:
 - ◆ Click the Add Buildfile button . This brings up a dialog in which you explicitly select those Ant buildfiles you want to add.
 - ◆ Click the Add Buildfiles with Search button . This brings up a search dialog in which you can specify a filename pattern for your Ant buildfiles and search within the entire workspace or a specified working set.
3. Once added to the Ant view, Ant buildfile entries remain in the view across workbench invocations until explicitly removed.
4. Click the Add Buildfiles with Search button. Suppose you only remember that the buildfile you want to work with starts with 'H'. Type **H* .xml** for the buildfile name. Make sure Workspace is selected for the scope, then click Search. The HelloWorld.xml file is found and placed in the Ant view.
5. Expand the top-level entry to see the default target Hello, and the subtarget World.



6. Select the World subtarget and click the Run the Default Target of the Selected Buildfile button . Notice that just the World subtarget gets executed.
7. Select the top-level HelloWorld buildfile and click Run the Default Target of the Selected Buildfile. Notice that just the default target, Hello, gets executed.
8. To edit your buildfile, bring up the context menu on the HelloWorld file and select **Open With > Ant Editor**.
9. To edit the default launch configuration, select Properties from the context menu.

The Run Ant dialog appears. Here you can modify the way in which the buildfile is run from the Ant view. Note that the choice and order of targets is ignored when running from the Ant view. If the top-level file is selected, just the default target is run and if one of the targets or subtargets is selected, just that target is run. It is not possible to run multiple targets from the Ant view.

Note: The choice and order of targets is ignored when running from the Ant view.

10. Select the HelloWorld file, then click the Remove button. The buildfile is removed from the view. Note that this does not delete the file from the workspace.

This concludes a quick look at the basics of Ant integration in Eclipse. The next sections consider several real-world use cases for running Ant buildfiles inside Eclipse.

Use cases for Ant in Eclipse

Now that you have seen how to work with Ant buildfiles in Eclipse, let's look at two ways that Ant buildfiles can be useful in Eclipse.

- [Deploying Eclipse plug-ins](#)
- [Using Ant buildfiles as project builders](#).

You will start by using an Ant buildfile to handle various aspects of deploying Eclipse plug-ins.

Deploying Eclipse plug-ins

The first use of Ant within Eclipse discussed here will be as a plug-in deployment tool.

When developing Eclipse plug-ins, it is frequently handy to build the JAR files for your plug-in and test them as part of an Eclipse install. The Ant buildfile in the next section describes how to do this and much more.

Creating a HelloWorld plug-in

To work through this example, you need to create a simple 'Hello World' Eclipse plug-in. Creating Eclipse plug-ins is covered in more detail elsewhere in this documentation, so just create a simple HelloWorld plug-in as follows:

1. Click **File > New > Other**.
2. Select **Plug-in Development** in the left pane, **Plug-in Project** in the right, and click Next.
3. Enter a name for your project, then click Next.
4. On the next page, accept all defaults and click Next.
5. Make sure **Create a plug-in project using a code generation wizard** is selected, then select **Hello, World** from the list below and click Next.
6. Accept all the defaults on this page and click Finish.

Now that you have a plug-in project, you can manage plug-in deployment by creating a build.xml file for it.

Generating the build.xml file

Open the Navigator view and locate your new plug-in project.

1. Select the file plugin.xml underneath your new project and right-click **Create Ant Buildfile**.

A new file, build.xml is created underneath your project.

2. Open the Ant editor on the build.xml file by double-clicking it in the Navigator. Notice that for this file, the Ant editor is the default editor. (This is because of the default file association in the workbench preferences.)

```

<?xml version="1.0" encoding="UTF-8"?>
<project name="HelloWorld" default="build.jars" basedir=".">

  <property name="bootclasspath" value=""/>
  <property name="basews" value="{ws}"/>
  <property name="baseos" value="{os}"/>
  <property name="basearch" value="{arch}"/>
  <property name="basenl" value="{nl}"/>
  <property name="javacFailOnError" value="false"/>
  <property name="javacDebugInfo" value="on"/>
  <property name="javacVerbose" value="true"/>
  <property name="javacSource" value="1.3"/>
  <property name="javacTarget" value="1.2"/>
  <property name="compilerArg" value=""/>

  <target name="init" depends="properties">
    <condition property="pluginTemp" value="{buildTempFolder}/plugins">
      <isset property="buildTempFolder" />
    </condition>
    <property name="pluginTemp" value="{basedir}" />
    <condition property="build.result.folder" value="{pluginTemp}/HelloWorld">
      <isset property="buildTempFolder" />
    </condition>
    <property name="build.result.folder" value="{basedir}" />
    <property name="temp.folder" value="{basedir}/temp.folder" />
    <property name="plugin.destination" value="{basedir}" />
  </target>

```

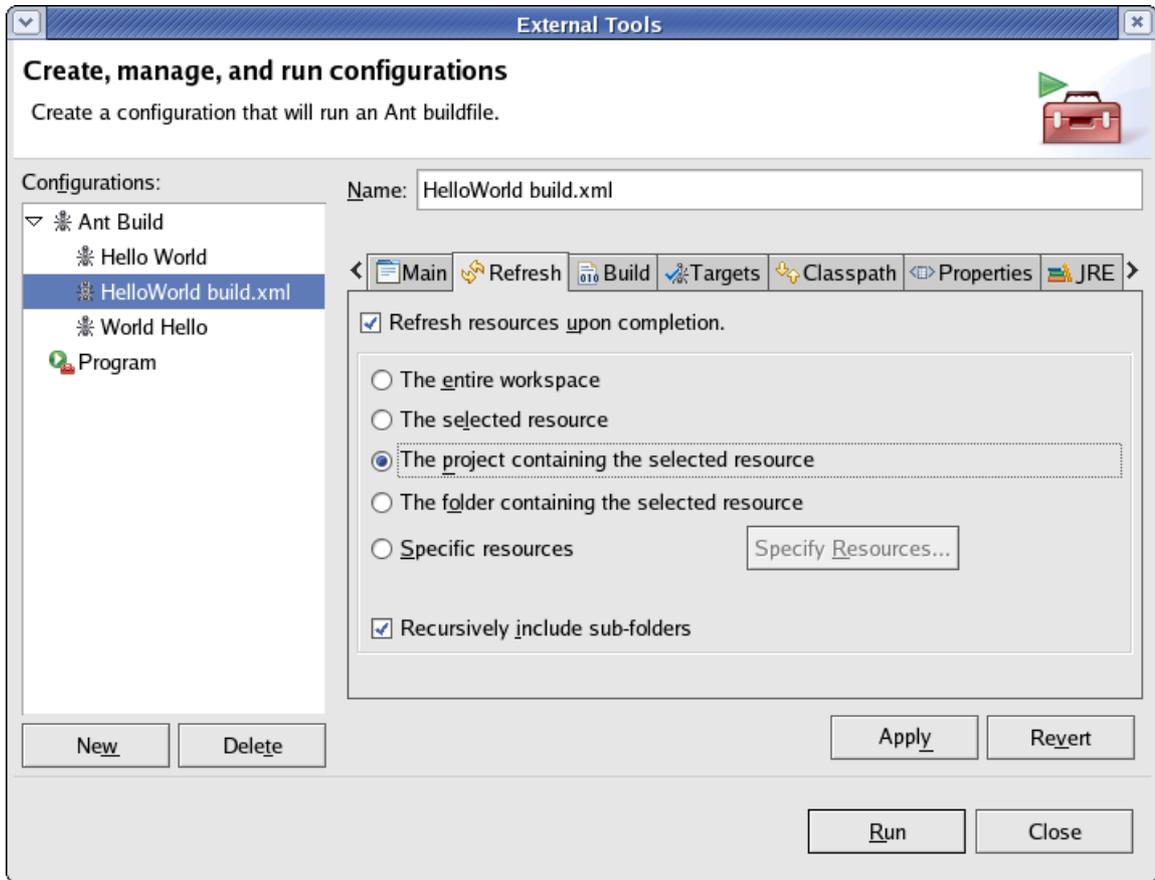
The build.xml file is a default Ant buildfile with many targets useful for deploying your plug-in. For example, there are individual targets to build all necessary jar files for your plug-in, to create a .zip file containing everything in your plug-in, to clean up any .zip files that have been created, and so on.

Now that you have the build.xml file, you can use it to build a .jar file for your plug-in.

Building a .jar file for the plug-in

Now you are ready to use the build.xml Ant buildfile that was created in the last section:

1. Select the build.xml file in the Navigator and choose **Run > External Tools** from the context menu.
2. In the External Tools dialog, click on the Targets tab. Make sure that build.jars is the only item selected in the list.
3. Click on the Refresh tab. The options on this tab control if and how the workspace looks for new, changed and deleted resources after the buildfile finishes execution. Because refreshing can be an expensive operation, you should refresh the smallest subset of the workspace necessary for your buildfile.
4. Check **Refresh resources upon completion**, then select **The project containing the selected resource**.



5. Click Run.

The Ant buildfile runs, executing the build.jar target. Output from the buildfile is visible in the Console view.

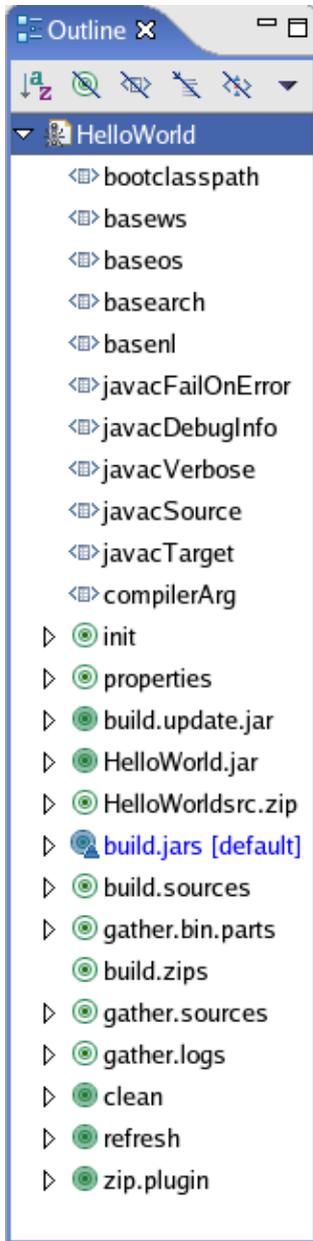
6. When the buildfile finishes, notice that there is a new .jar file under your project containing the two class files that comprise your plug-in. If you had chosen not to refresh the project, the .jar file would still have been created, but you would not see it in the Navigator.

You have run the default target for the build.xml Ant buildfile, but there are many other targets and options to be explored.

More plug-in deployment options

Now that you have run the default target for your build.xml buildfile, it is time to explore some more options.

As can be seen in the Outline view, the default build.xml Ant buildfile has a number of targets.



Solid green arrows denote targets, while subtargets are represented by hollow green arrows. The following are the useful top-level targets:

build.update.jar

This target builds jars for the plug-in, then zips them into a form suitable for use with the Eclipse Update Manager.

HelloWorldPlugin.jar

This target does the work of the building jars for the plug-in.

build.jars

This is the default target we ran in the last section. It defers to the HelloWorldPlugin.jar target to do its work.

clean

This target deletes all zips, jars, and temporary directories that may have been created by other targets in this buildfile.

zip.plugin

This target builds executable and source jars, then zips everything into a single archive.

Remember that this build.xml file is simply the *default* deployment buildfile for an Eclipse plug-in. There will be times when you will want to modify this buildfile to handle your particular projects. For example, if you have a directory that contains resources necessary for your plug-in, you will want to update the build.xml file to include this directory in the jars and zips it creates.

This completes the look at using Ant buildfiles to deploy Eclipse plug-ins. The key points to remember are that you can easily create a default deployment buildfile for a plug-in that contains many useful targets, and that while this default buildfile is useful, you may have to modify it to suit your needs.

Ant buildfiles as project builders

Your second practical example of using Ant buildfiles in Eclipse is a 'project builder'. This is an Ant buildfile that has been designated to run whenever a project is built. There are many uses for such a buildfile:

- Generate a .jar file containing class files from your project
- Perform some type of pre- or post-build processing on source or binary files in your project. For example:
 - ◆ Pre-processing source files to instrument them for performance analysis
 - ◆ Obfuscating binary files to prevent reverse engineering.
- Copy class files to some location (for instance, on a network)

For this example, you will create an Ant buildfile that creates a .jar archive of the class files in a project. Note that this is similar to the example in which you used an Ant buildfile to generate .jar files for an Eclipse plug-in. This example differs in that it works for any Eclipse project, whether or not it is also an Eclipse plug-in.

Creating a project builder Ant buildfile

To see how project builders work, we will create a simple project with a single source file and an Ant buildfile that jars up the single class file. Though this example uses Java, it should be noted that project builders are available for all projects, Java or otherwise.

1. Create a Java project named 'HW'.
2. Create a Java source file named 'HelloWorld' with a main method.
3. Put a single 'System.out.println()' statement in the main method, and make it print a greeting of your choice.
4. Save changes.
5. Create a file named projectBuilder.xml, open the Ant editor on it, enter the following content, and save changes.

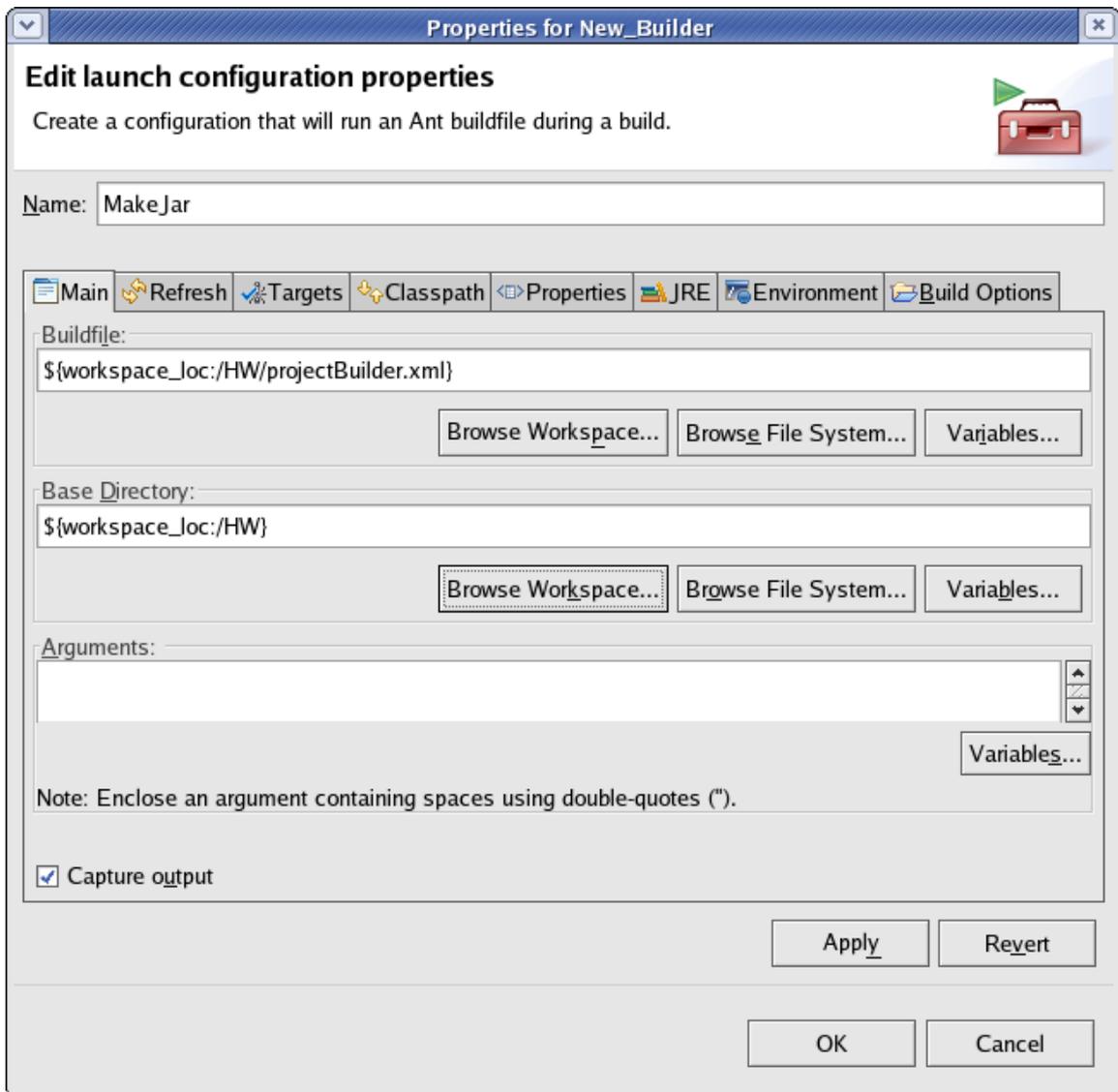
```
<?xml version="1.0" encoding="UTF-8"?>
<project name="HW.makejar" default="makejar" basedir=".">

  <target name="makejar" description="Create a jar for the HW project">
    <jar jarfile="HelloWorld.jar" includes="*.class" basedir="."/>
  </target>
</project>
```

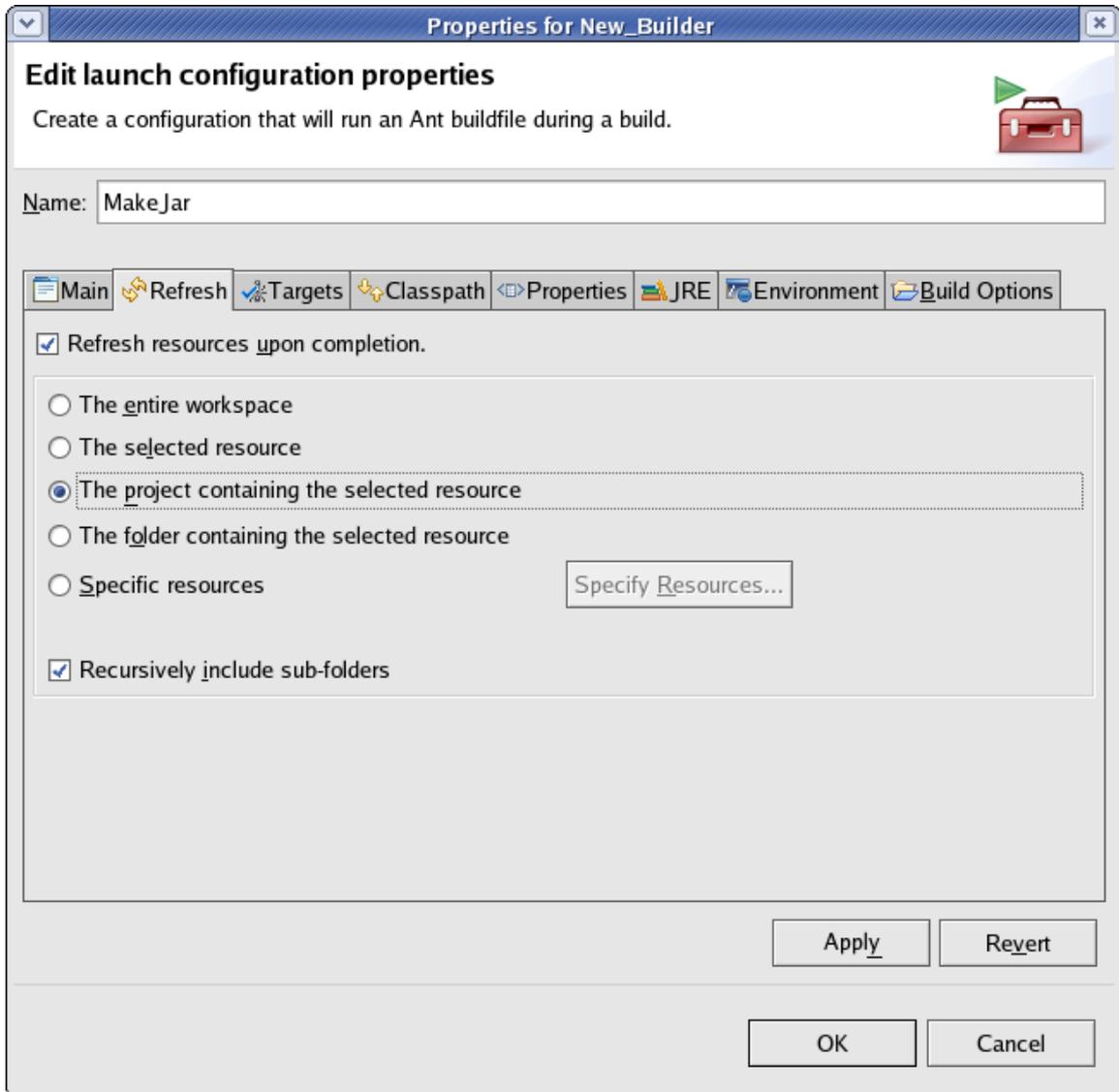
```
</target>
```

```
</project>
```

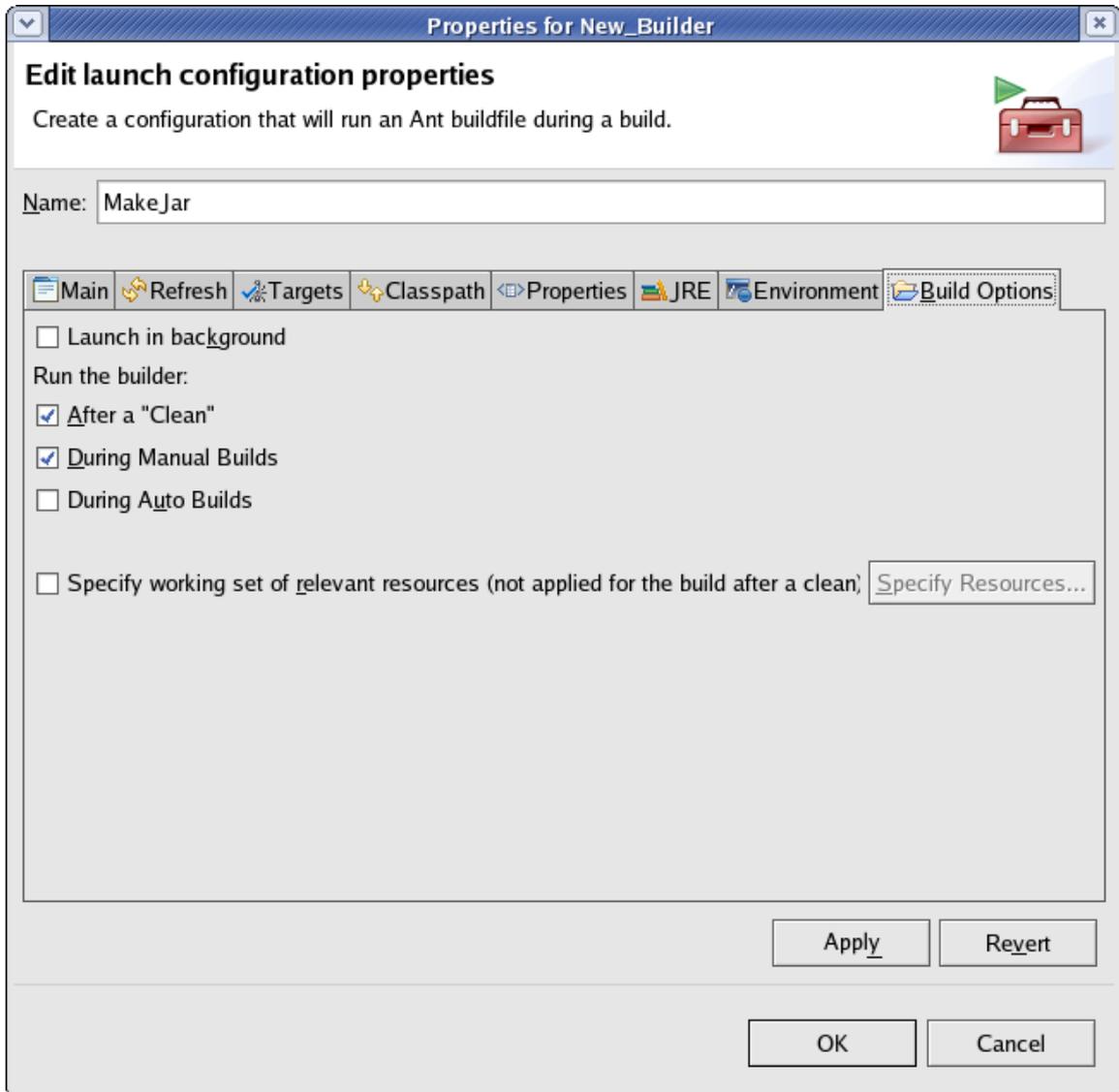
6. In the Navigator, select the HW project and choose **Properties** from its context menu.
7. In the project properties dialog, select **External Tools Builders**, then click New.
8. In the Choose configuration type dialog, make sure 'Ant build' is selected, and click OK.
9. The External Tools dialog appears. Set the name to 'Makejar'. In the Main tab, use the Buildfile Browse Workspace button to set the **Location** to be the projectBuilder.xml buildfile created above. Then use the Base Directory Browse Workspace button to set the base directory to be the HW project.



10. In the Refresh tab, you want to be sure that when your HelloWorld.jar is created, you see it in Eclipse. By default, no refreshing is done when a project builder finishes running, so check **Refresh resource after running tool**, then select **The project containing the selected resource** in the list of scope variables. Because refreshing can be expensive, you should in general refresh the smallest entity that contains all resources that will be affected by your buildfile.

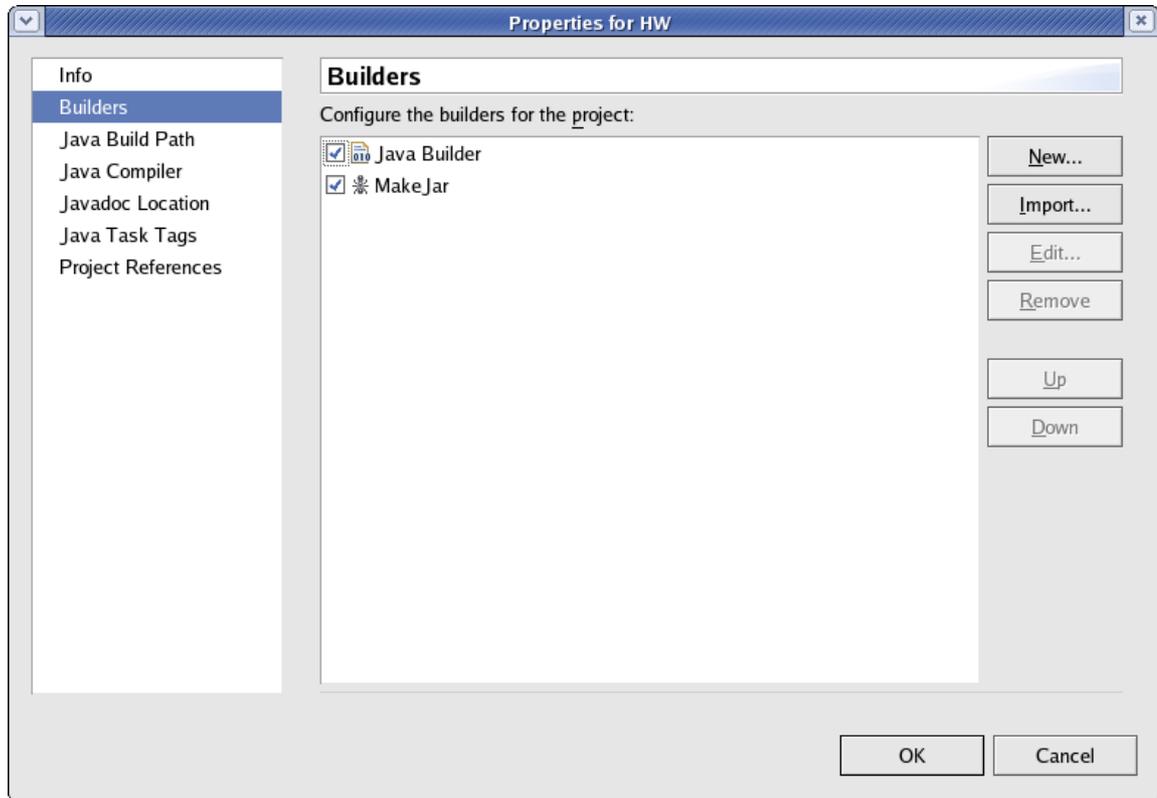


11. In the Targets tab, the default target should be selected.
12. In the Build Options tab, you can specify when this project builder is executed. By default, this is set to After a "Clean" and During Manual Builds. Running your project builder during auto builds is possible, but this is not recommended because of performance concerns.



13. Apply the changes and click OK.
14. Back in the project properties dialog, you will now see a project builder named 'Makejar' that is set to run after the default Java Builder. Note that you can change the order so that your Ant buildfile runs before the Java builder, although that would not make sense in this example. Click OK to save the project builder and close the dialog.

Note: You can change the order so that your Ant buildfile runs before the Java builder, although that would not make sense in this example.



For a Java project, the default Java Builder will always be run and cannot be removed. The Java Builder runs the internal Eclipse Java compiler which in turn is responsible for indexing your source so that searching, refactoring, and many other features are available. Thus it is not possible to replace the internal Eclipse Java compiler by using a project builder. Your only option with the Java Builder is when it runs with respect to the project builders that you define.

Executing project builders

The whole point of project builders is that they are not explicitly run by you, but instead are run any time a qualifying build takes place for the project that owns the buildfile. Remember that the types of builds that trigger project builders are defined in the Build Options tab in the External Tools dialog and can be any combination of full, incremental, or auto builds. Here is how this works:

1. Select the HW project in the Navigator. Click **Project > Clean > Clean selected projects** and click OK.

The project is rebuilt and the projectBuilder.xml buildfile is run. Notice the output from this buildfile in the Console view.

2. Make sure the Autobuild preference is turned on, then make some trivial change to HelloWorld.java and save the change. The save triggers an autobuild, but the autobuild does not trigger the project builder.
3. Suppose you do not want to see the buildfile output every time it runs. Go back to the External Tools Builders page of the project properties dialog on HW. Select the Makejar entry and click Edit. On the Main tab, uncheck the **Capture Output** option, apply the change and exit back to the workbench.

This concludes the look at Ant buildfiles as project builders in Eclipse. It is worth repeating that though this example used a Java project, project builders are not tied to Java, and may be used for any type of project.

External tools

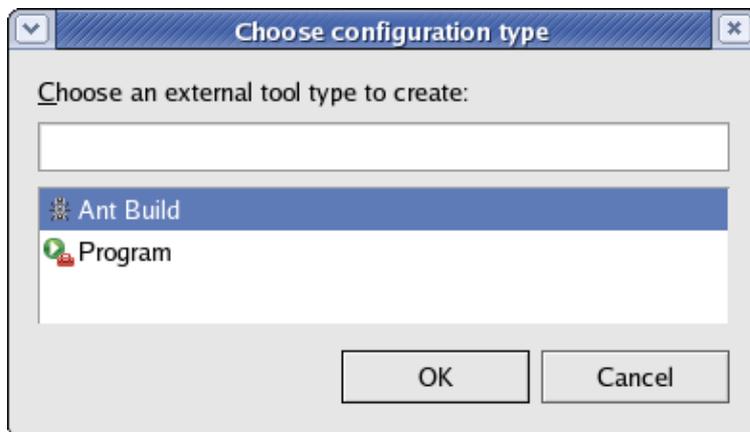
Up to this point in the tutorial, the discussion has been as if Ant were the only type of external tool available for use in Eclipse. The fact is that Eclipse's external tools framework handles two broad classes of external tools:

- Ant buildfiles
- Everything else.

This section looks at ways to set up and use non–Ant external tools.

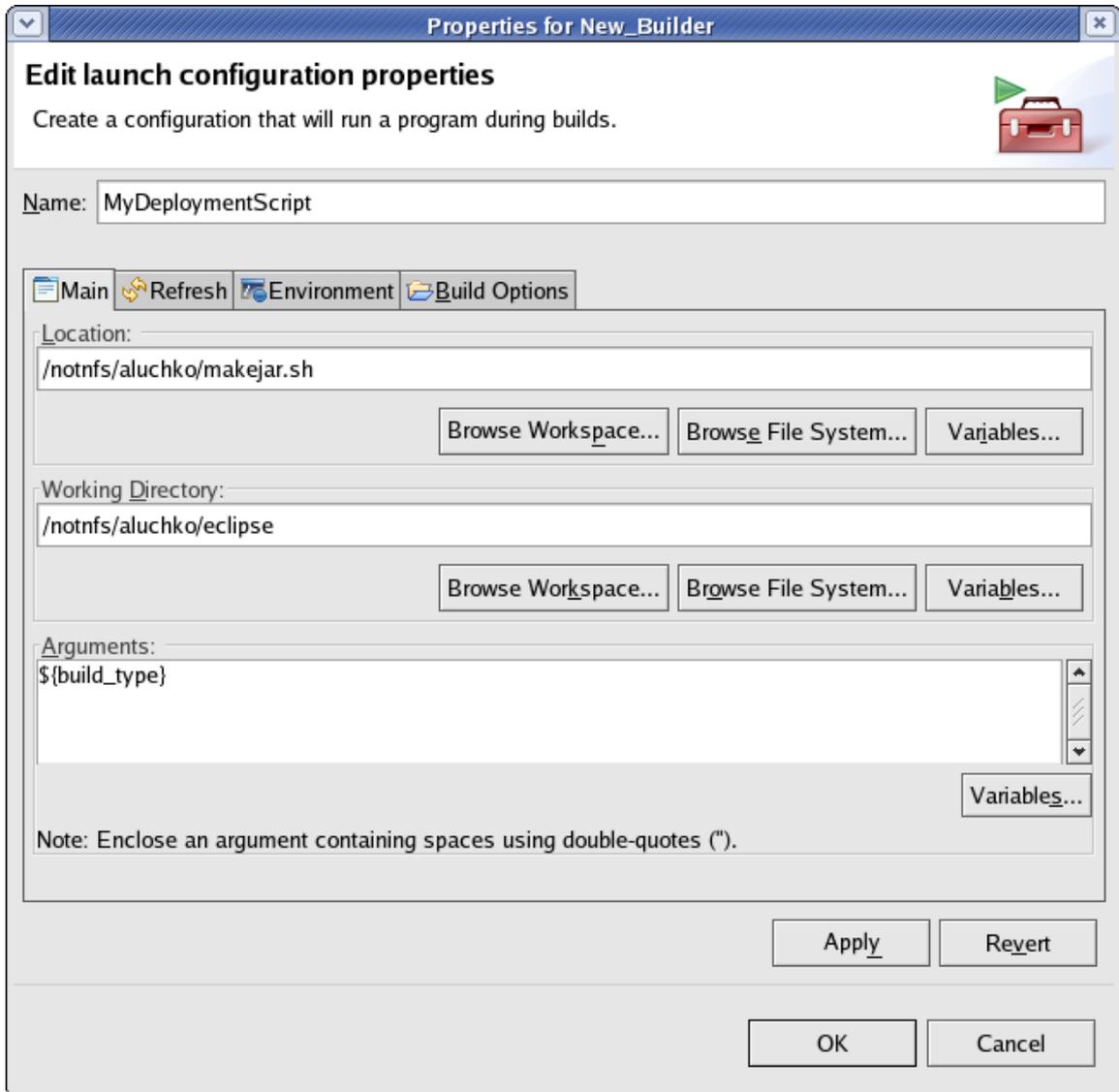
Non–Ant project builders

When you worked through your project builder example, you may have noticed that when you created your project builder Ant buildfile, you had a choice of external tool type.



The **Program** option is essentially a catch–all, allowing you to define an external tool for any executable file that is accessible on your local or network file system. Suppose that instead of Ant, you prefer to use your own shell scripts to create jar files and deploy your Eclipse projects. You would then create a Program external tool that specified where and how to execute your script.

1. Create a script that performs your preferred deployment steps.
2. Select the project that you wish to build in the Navigator, and choose **Properties** from the context menu.
3. Select **Builders**, click New, select **Program** and click OK.
4. The External Tools dialog appears, configured for Program–type tools.
5. Enter the location of your script, its working directory, and any required arguments.



6. The script could be a Linux shell script, a Perl script, or just about anything else that can be executed on your system.
7. The Refresh and Build Options tabs are identical to the tabs you saw for Ant project builders. In particular, the Build Options tab allows you to control what types of builds trigger your project builder buildfile.
8. Apply the changes, and click OK.

As with Ant project builders, you can control the ordering of this project builder with respect to other project builders (such as the default Java Builder for Java projects).

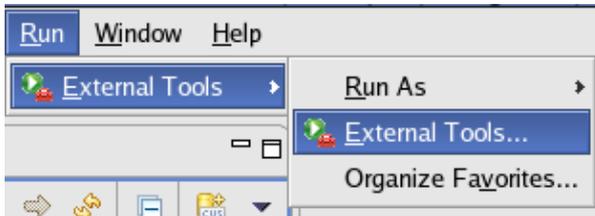
9. Rebuild your project. This triggers your script to execute. Any output it generates is sent to the Console view.

Ant is a popular tool for configuring and deploying projects. However, if you prefer some other tool, or prefer to do it yourself, you can set up a Program external tool project builder. This allows you customize the deployment of your project as you see fit, while keeping the convenience of automatically running your script every time your project is built.

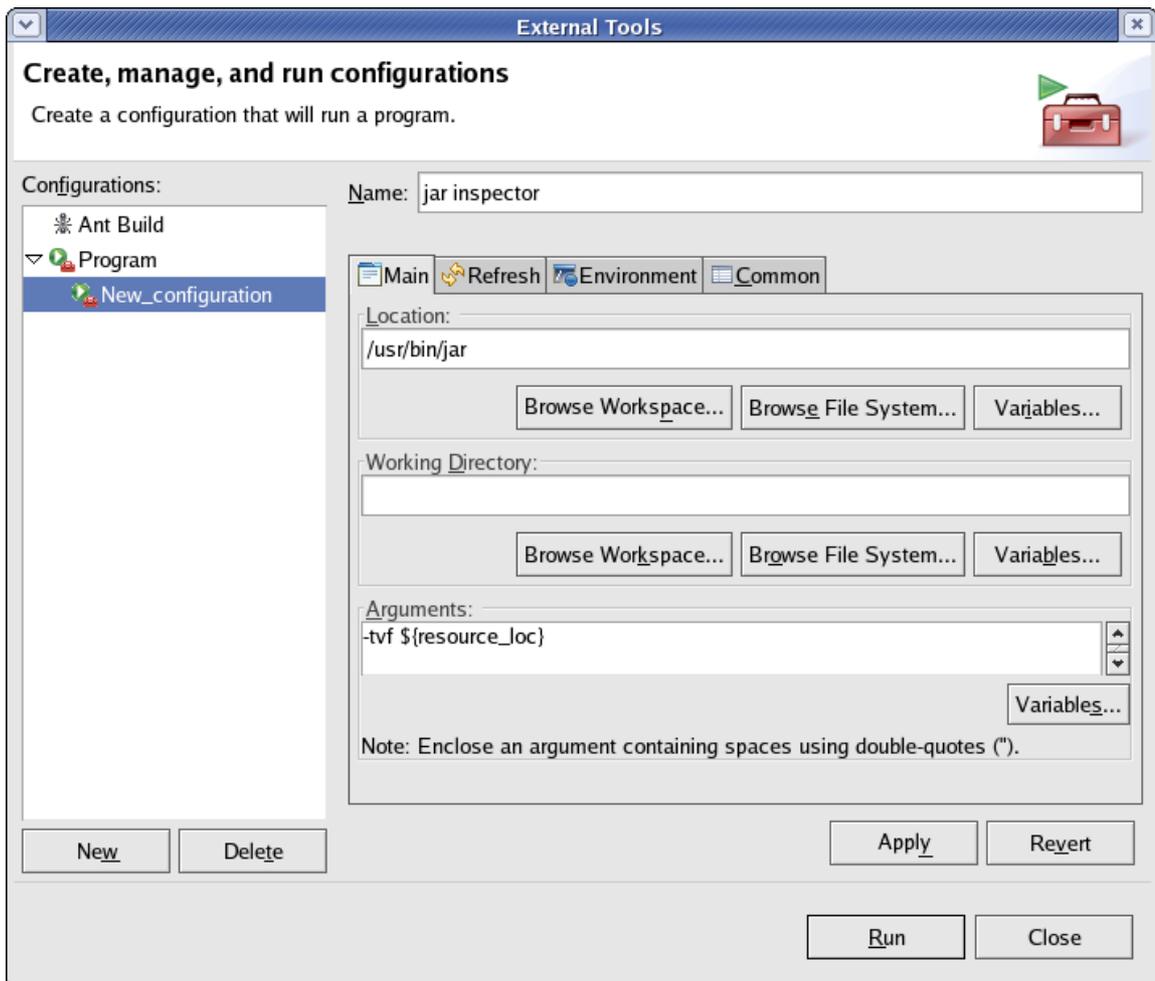
Stand-alone external tools

For the ultimate in external tool flexibility, create a 'stand-alone' external tool launch configuration. This is similar to the project builder launch configurations discussed in the last section, except that it need have nothing to do with project building, and you can explicitly run it whenever you choose. Suppose you wanted to have a way to quickly see the contents of a .jar file in your workspace using the jar utility.

1. Select some .jar file in your workspace.
2. Click **Run > External Tools > External Tools** from the Workbench toolbar.



3. Select **Program** in the tree, then click New.



4. Name the launch configuration **jar inspector**.
5. Use the first Browse File System button to locate the jar executable.
6. In the **Arguments** field, type **-tvf** and a space, then click Variables.

Eclipse Tutorials

7. In the Select Variable dialog, there are a number of variables you can pass as arguments to the program specified in Location. Select **resource_loc** and click OK.

When this buildfile is run, the absolute path of the resource selected in the workbench will be passed to the jar utility in the position specified.

8. Click Run.

Notice that the buildfile sends the jar utility output to the Console view.

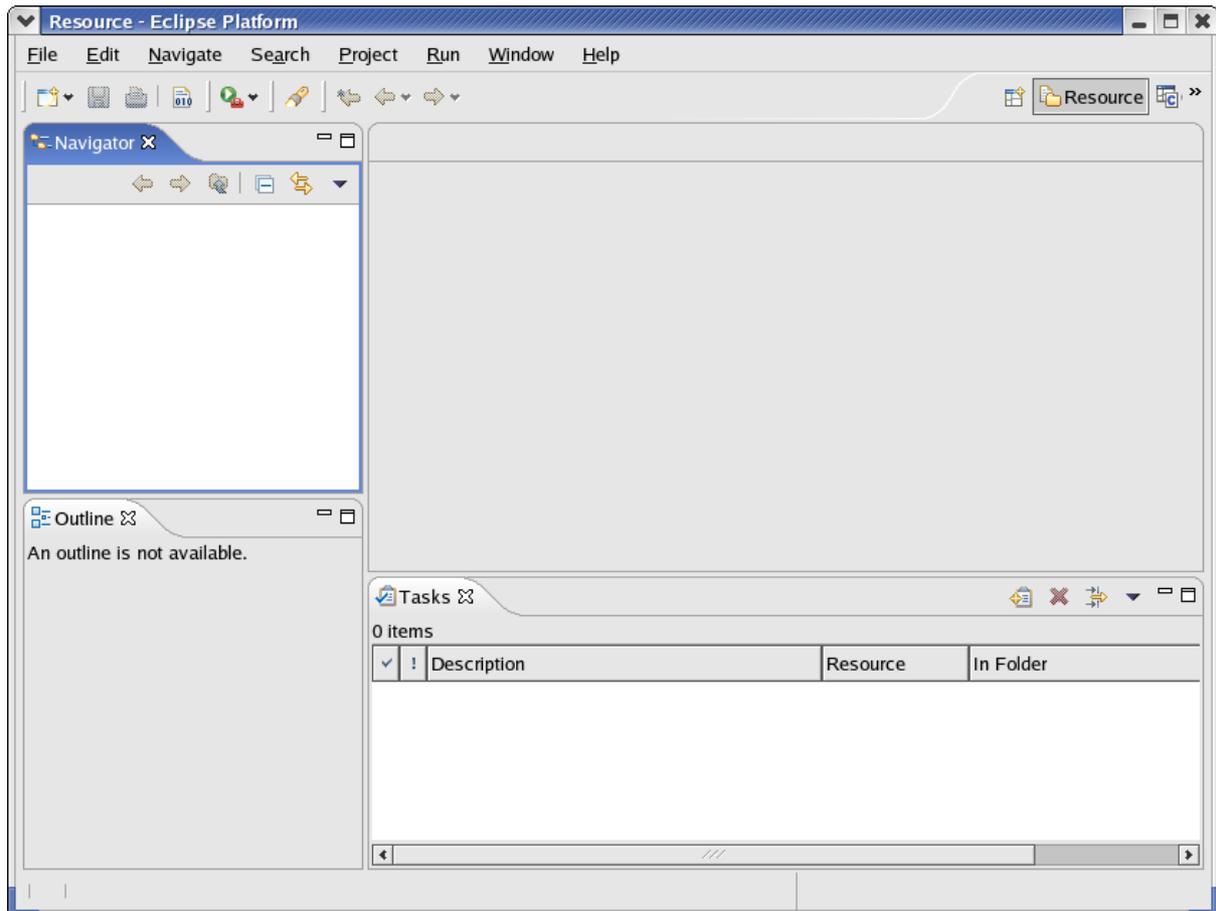
9. Select a different .jar file in your workspace.
10. Click the **External Tools** button in the toolbar. Notice the contents of this jar are sent to the Console view as well. Now you have a quick and easy way to see the output of the jar utility for any .jar file in your workspace.

This example has only scratched the surface of what you can do with external tools. The important things to remember are that you can create an external tool for anything you can run on your system, and that you can pass arguments to the external tool related to the current workbench selection. In many cases, this allows you to loosely integrate tools that do not have corresponding Eclipse plug-ins.

CDT "Managed Make" Tutorial

We will now walk through the process of creating a simple 'Hello World' application using a Managed Make project in the CDT.

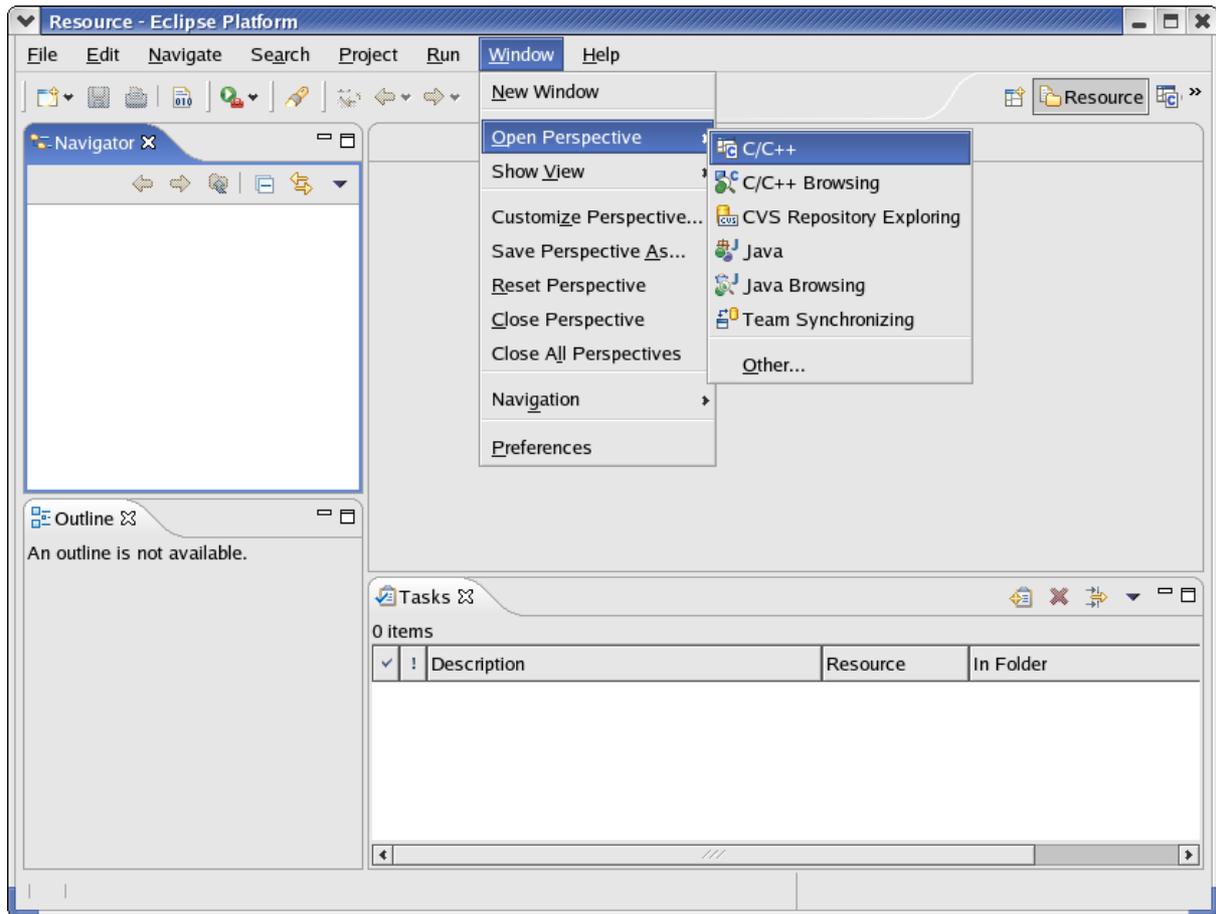
The image below shows the standard Workbench.



Open the C/C++ Perspective

Click **Window > Open Perspective > C/C++**.

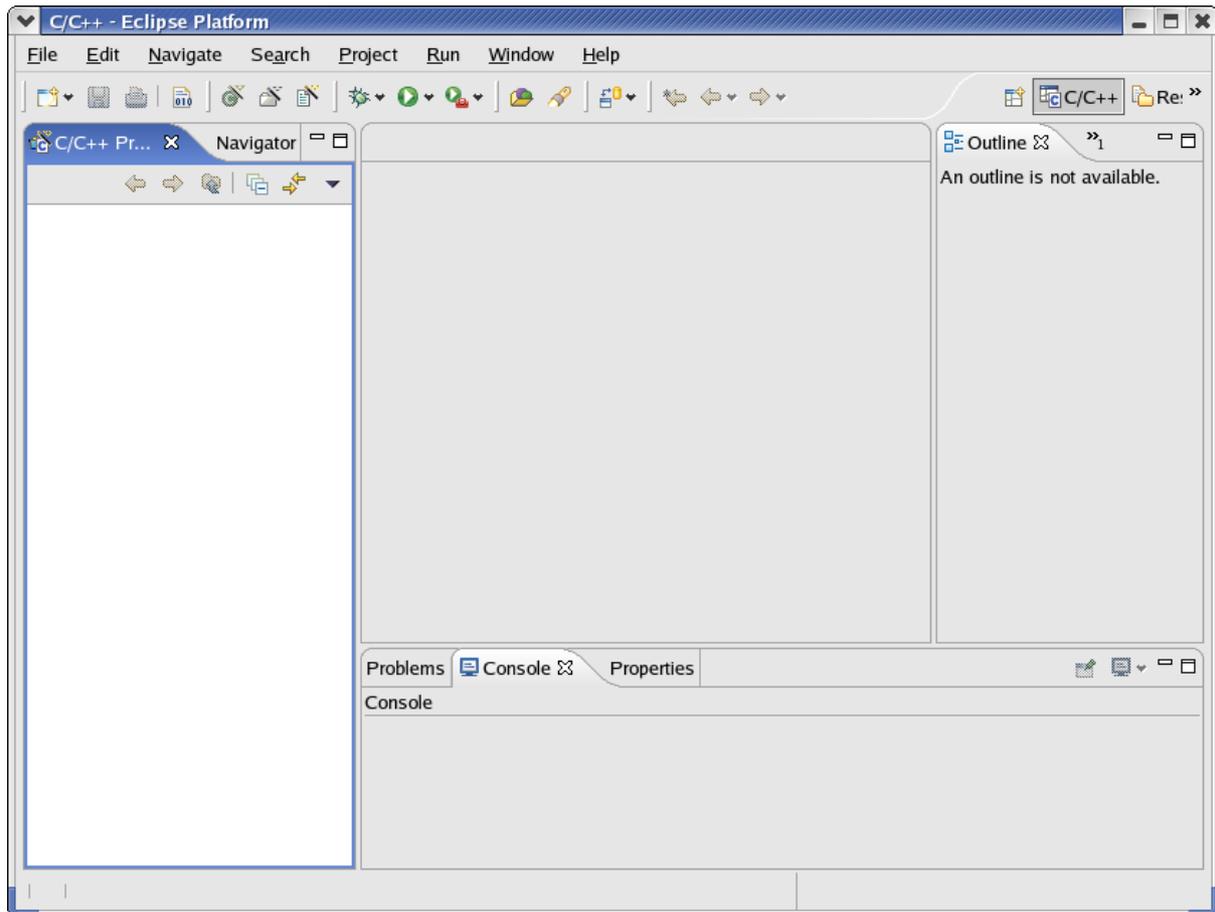
Note: If the C/C++ perspective is not listed, click **Other...** and select **C/C++** from the **Select Perspective** dialog box.



C/C++ Perspective

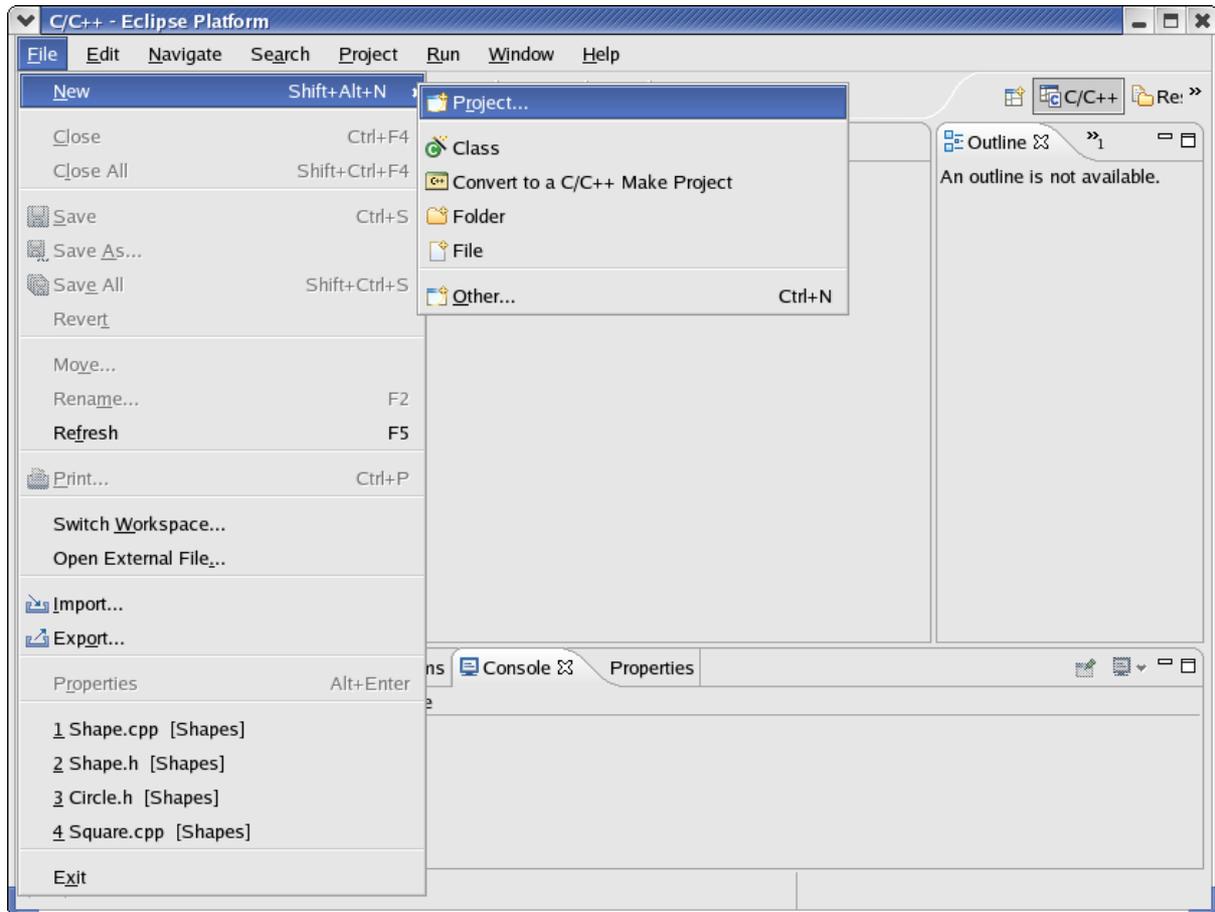
This is the **C/C++ Perspective**. Notice the **C/C++ Projects** view is on the left and the **Outline** view has moved to the right. The center area is reserved for your editor.

There may be other views available in your workbench, such as the **Navigator** or **Make Targets** views.



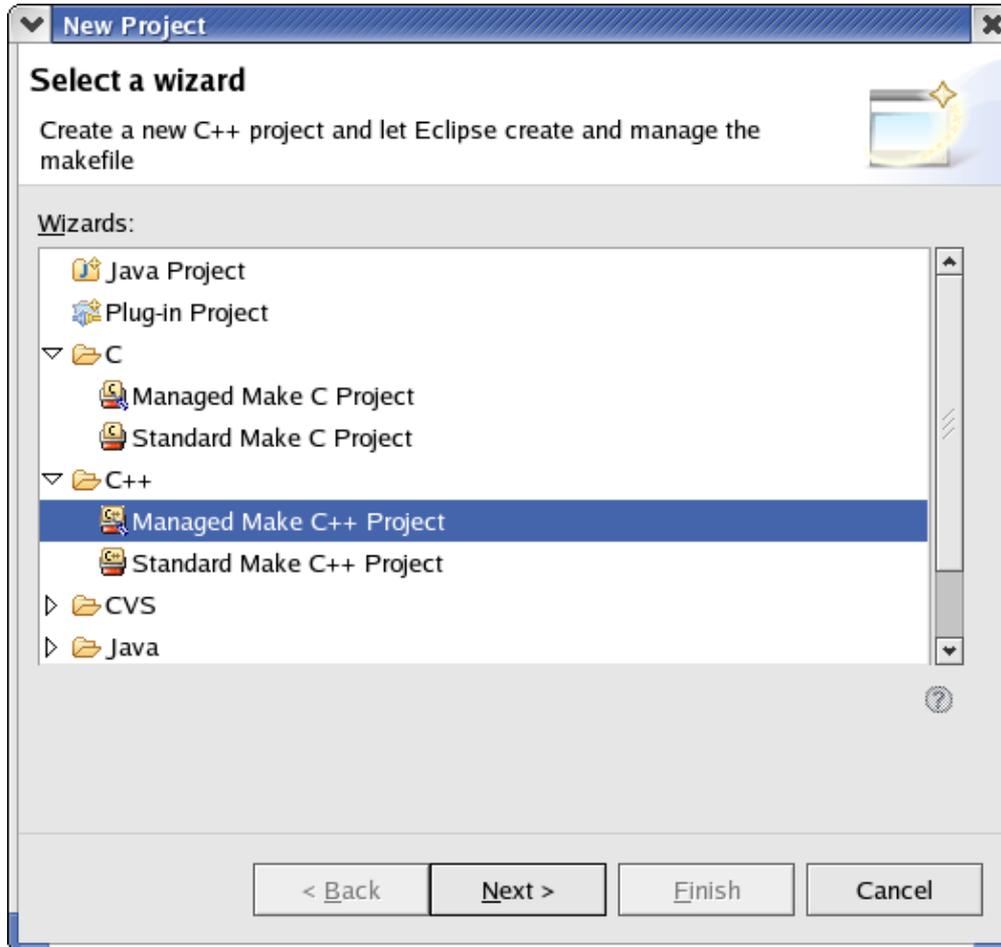
Create a project

You can now create a C/ C++ **project** by clicking **File > New > Project**.



New Project Wizard

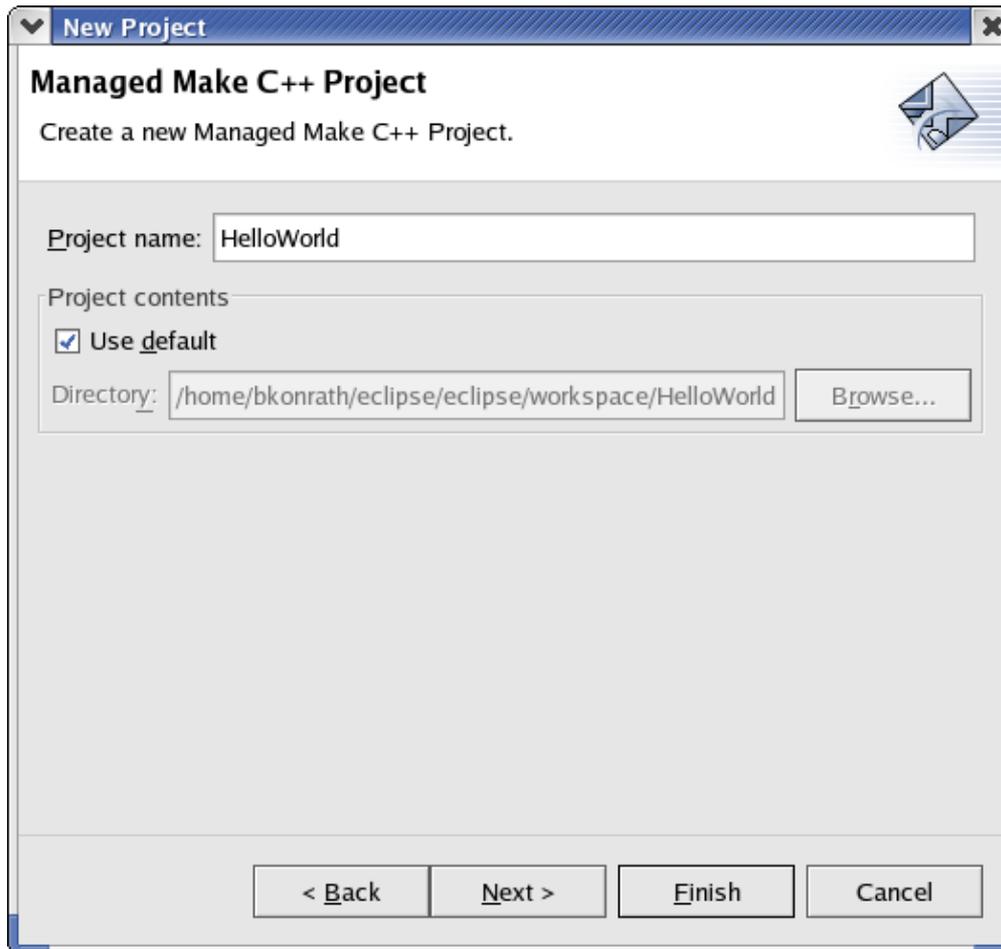
You will now see the **New Project** wizard. Open a C or C++ project and select a **Managed Make** project. A Standard Make C/C++ project requires you to provide a makefile; a Managed Make project will create a makefile for you.



New Project

Enter a name for the project. You can also enter a new path for your project by deselecting the **Use Default Location** checkbox and entering the new path in the **Location** text box.

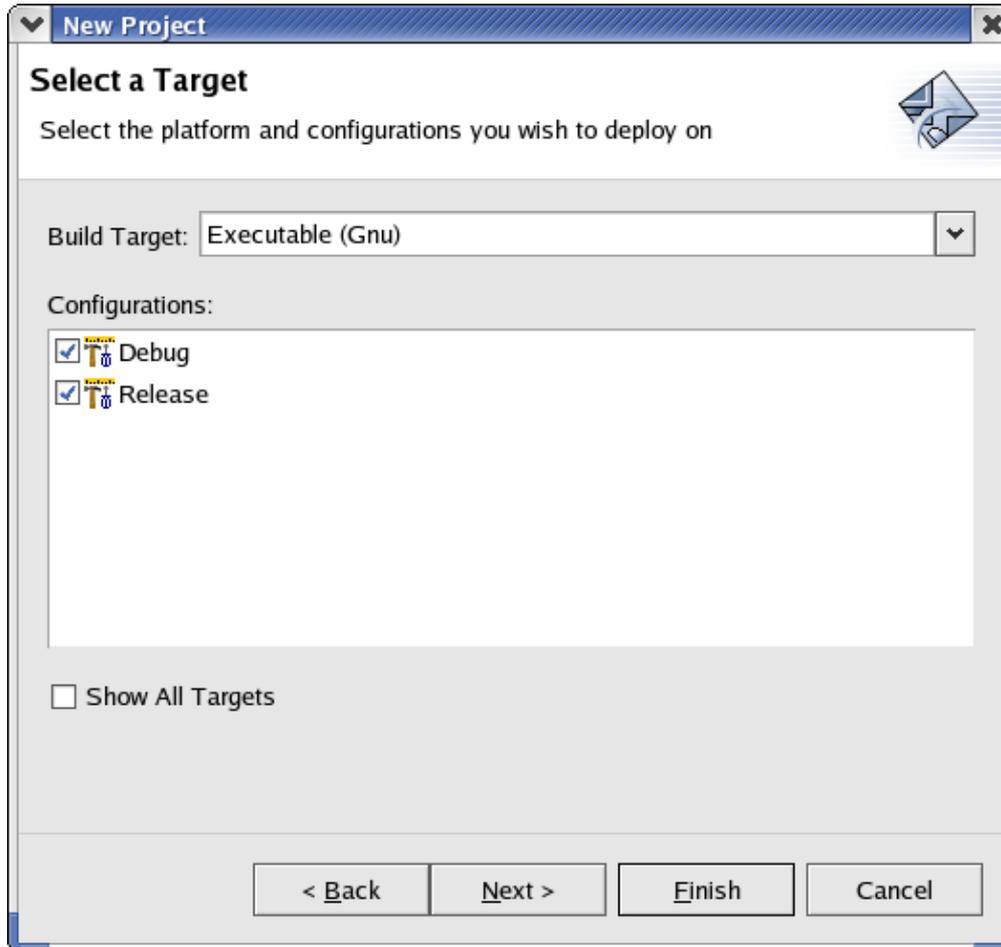
Click **Next**.



Select a Target

Select the appropriate Executable Build Target.

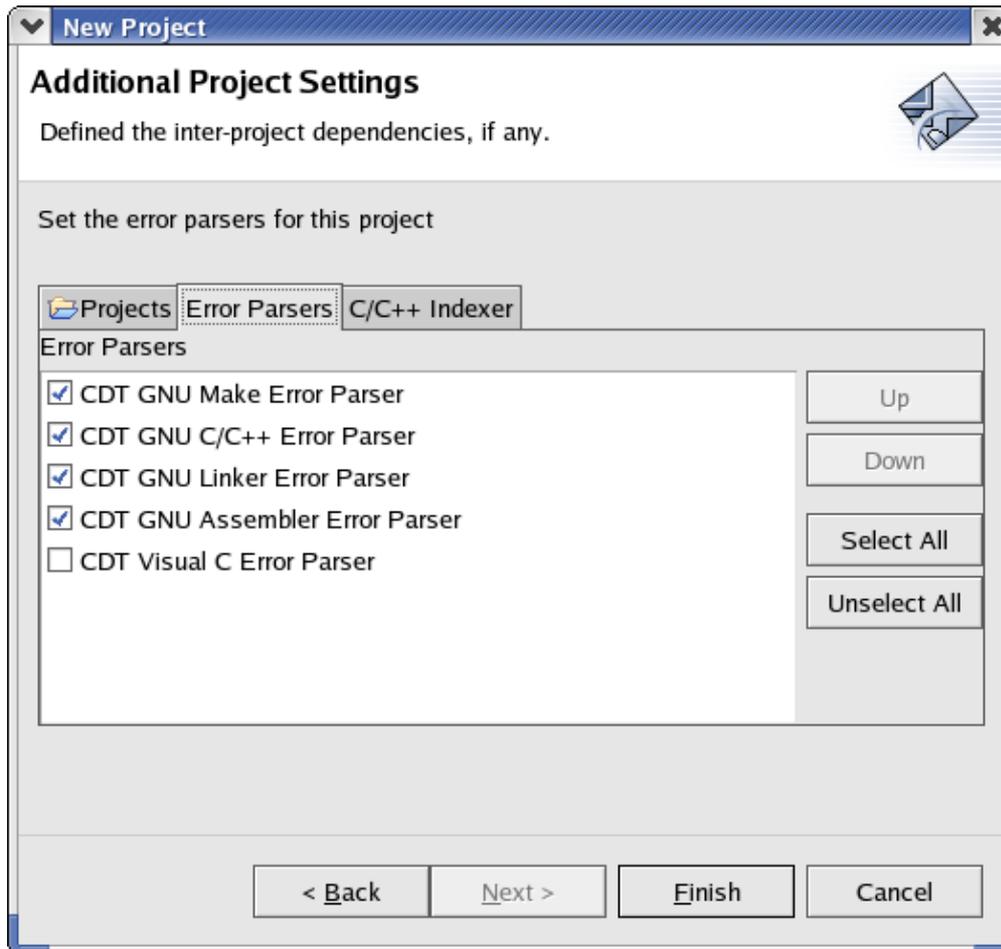
Click **Next**.



Error Parser Settings

Click the **Error Parsers** tab and select the error parsers you require for the project. (You can leave all parsers selected.)

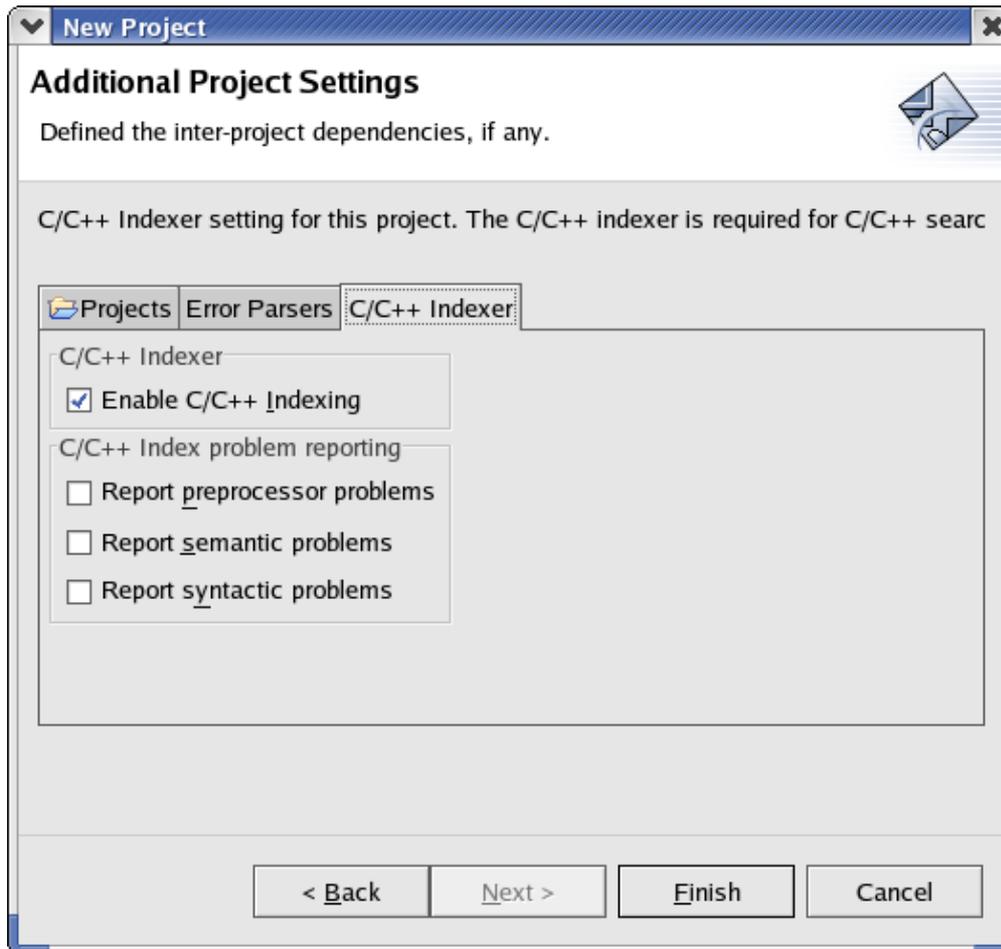
You can also change the order in which Error Parsers are called.



Additional Project Settings

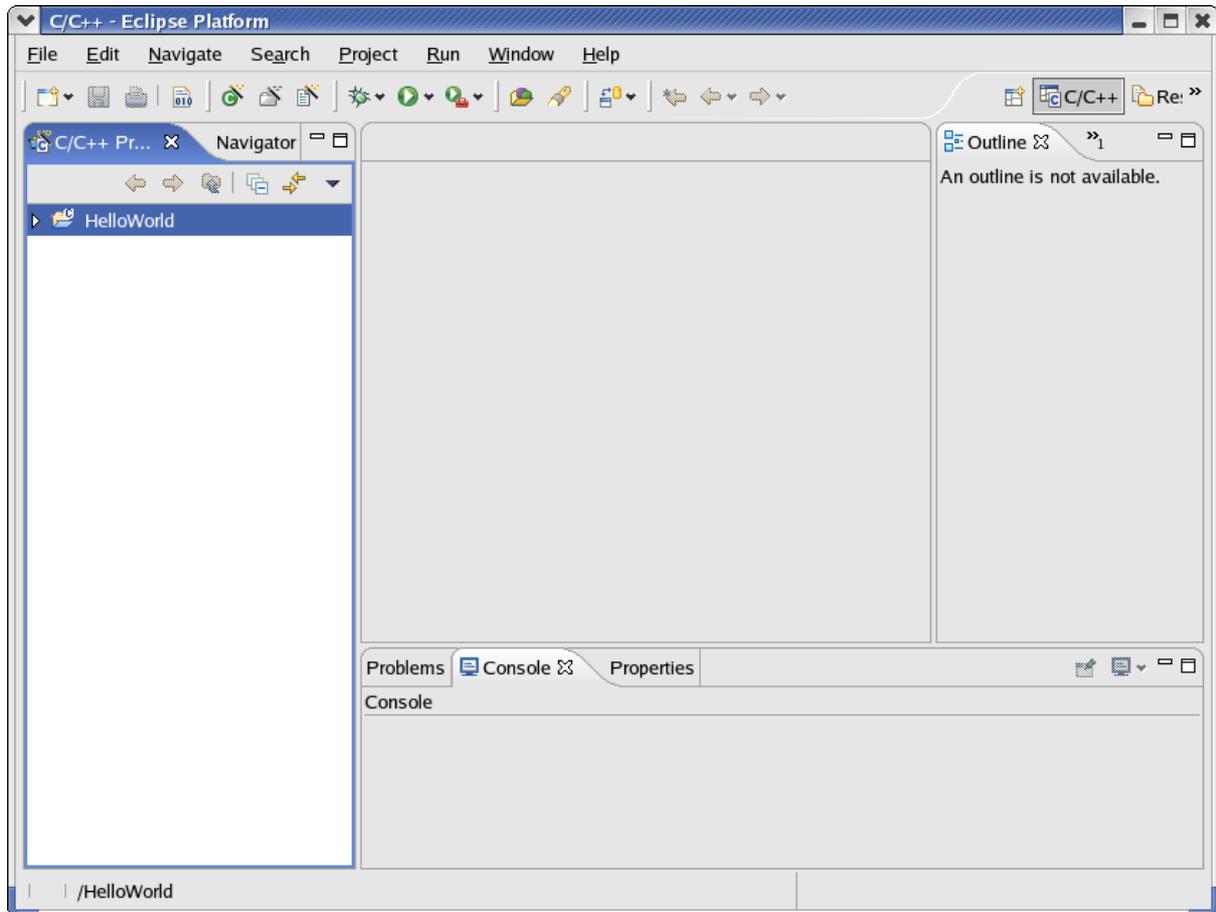
Click the **C/C++ Indexer** tab, ensure **Enable C/C++ Indexing** is selected otherwise search capabilities and other tools that require search will be disabled for your project.

Click **Finish**.



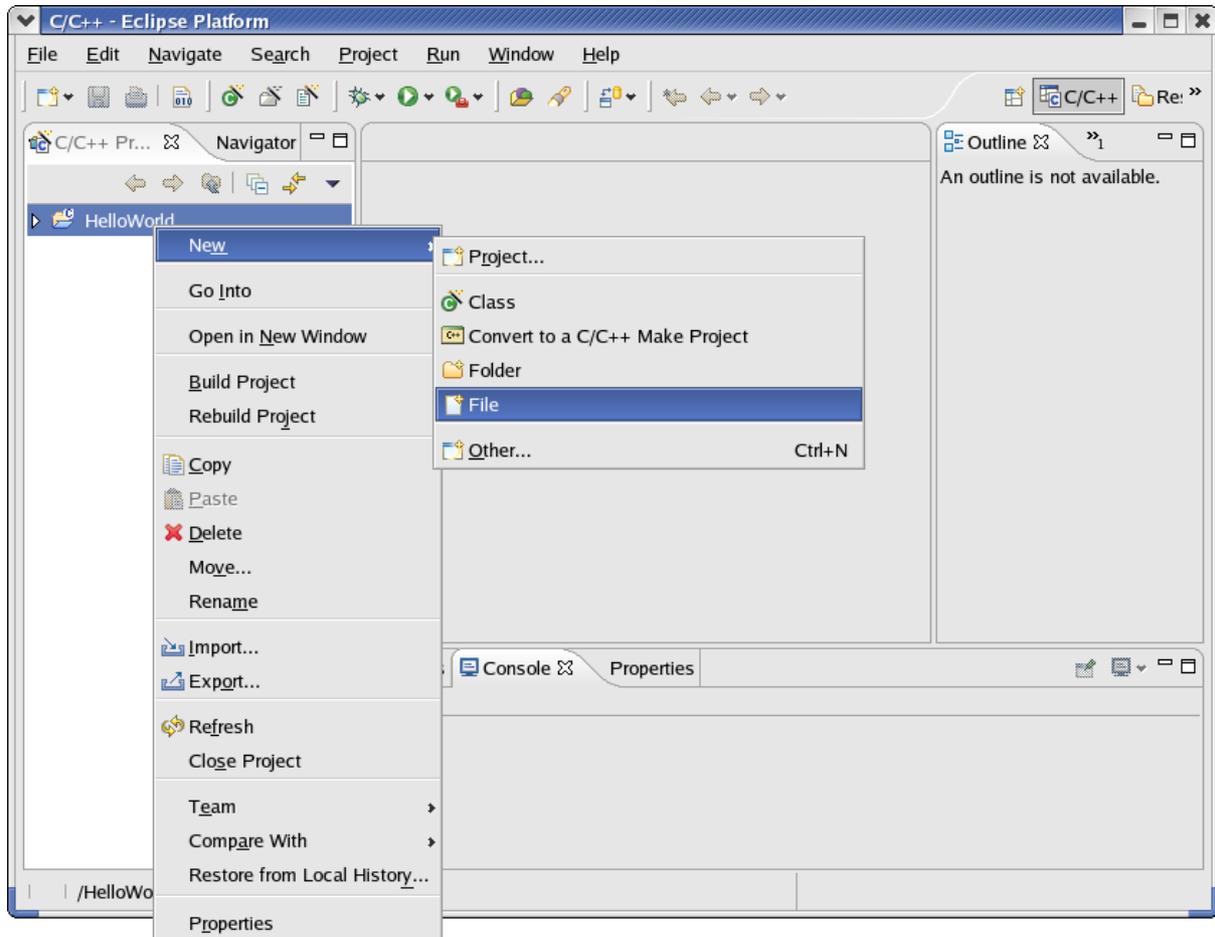
New Project

You should now see the new project in the **C/C++ Projects** view.



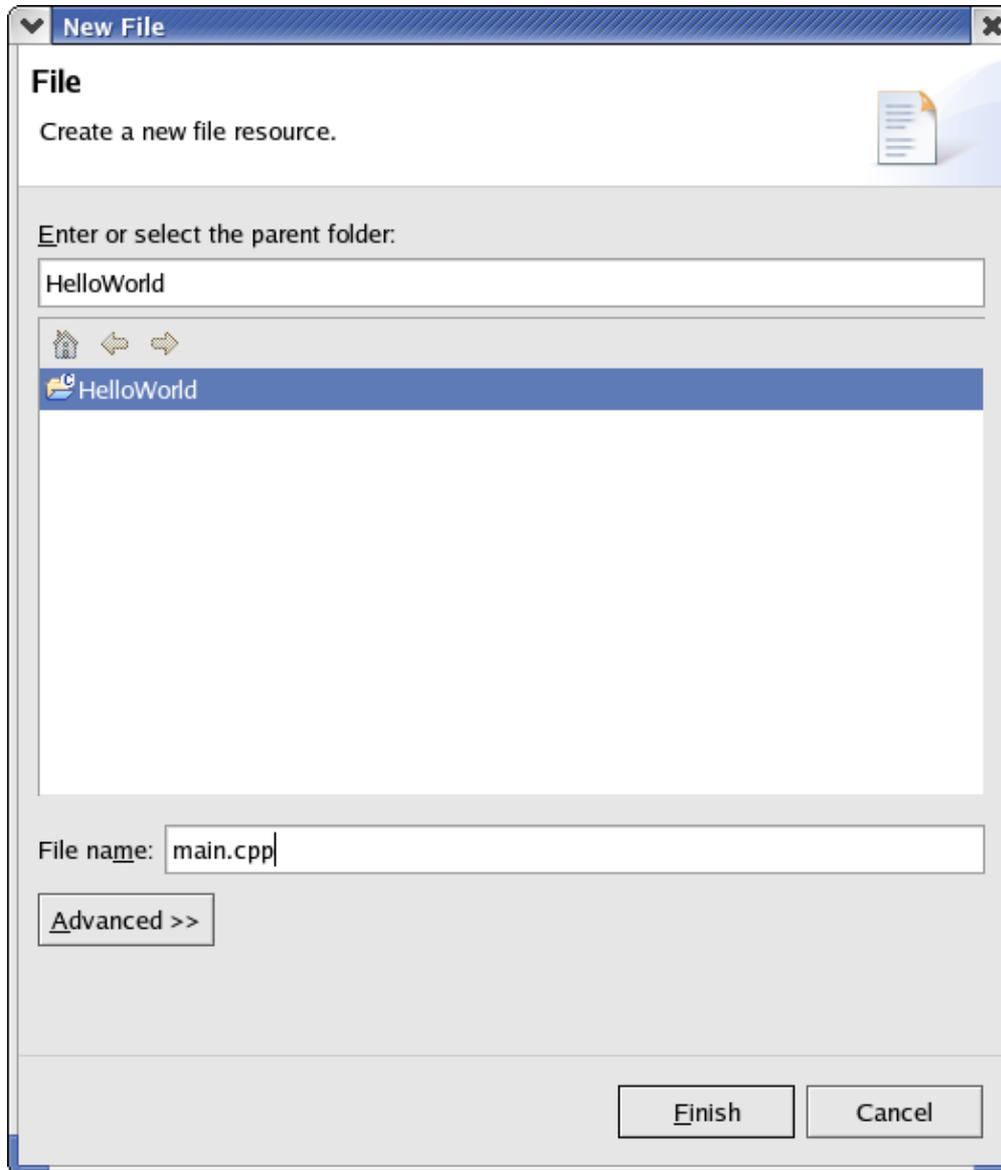
Create a file

Create a new cpp file (such as main.cpp) by right-clicking your project and selecting **New > File**.



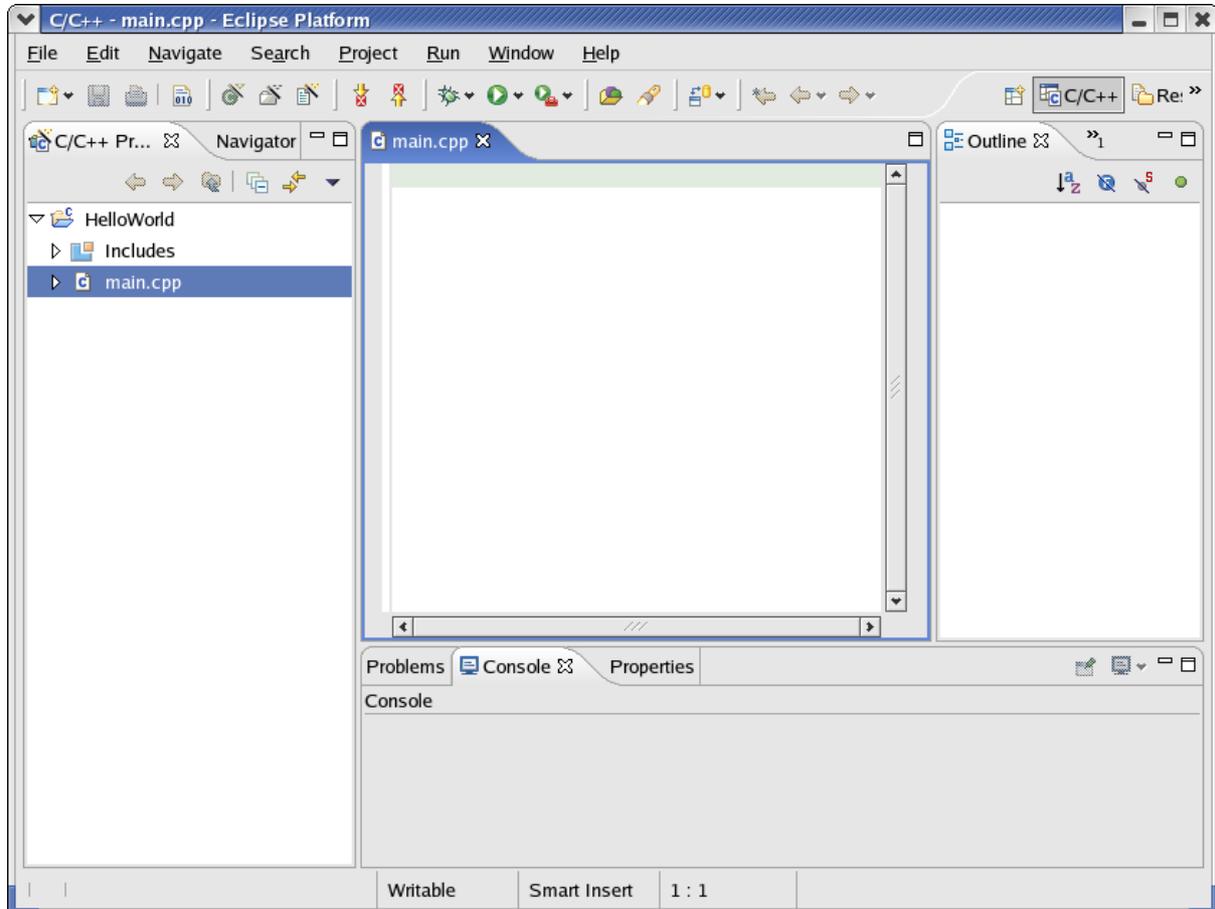
Name the new file

In the **New File** wizard, enter the name of your file in the **File name** text area, then click **Finish**.



New project file

You should now see the new file located in the **Projects** view under the project, and the new file should be open in the editor.

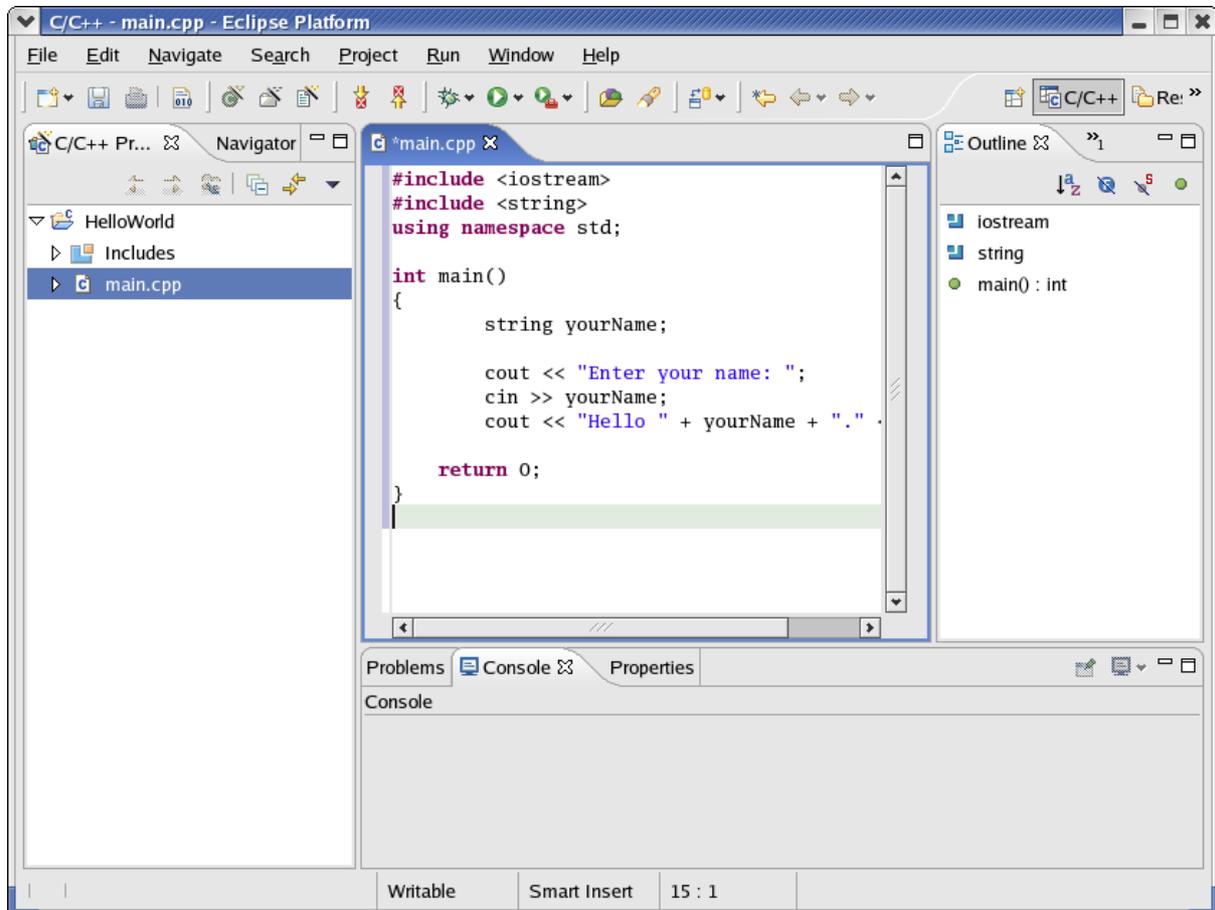


Enter code

Enter the code in the `main.cpp` file that was just created. You can double click the **main.cpp** tab in the **Editor** view to expand the view.

Note: Leave a blank line at the end of the code as some compilers require this.

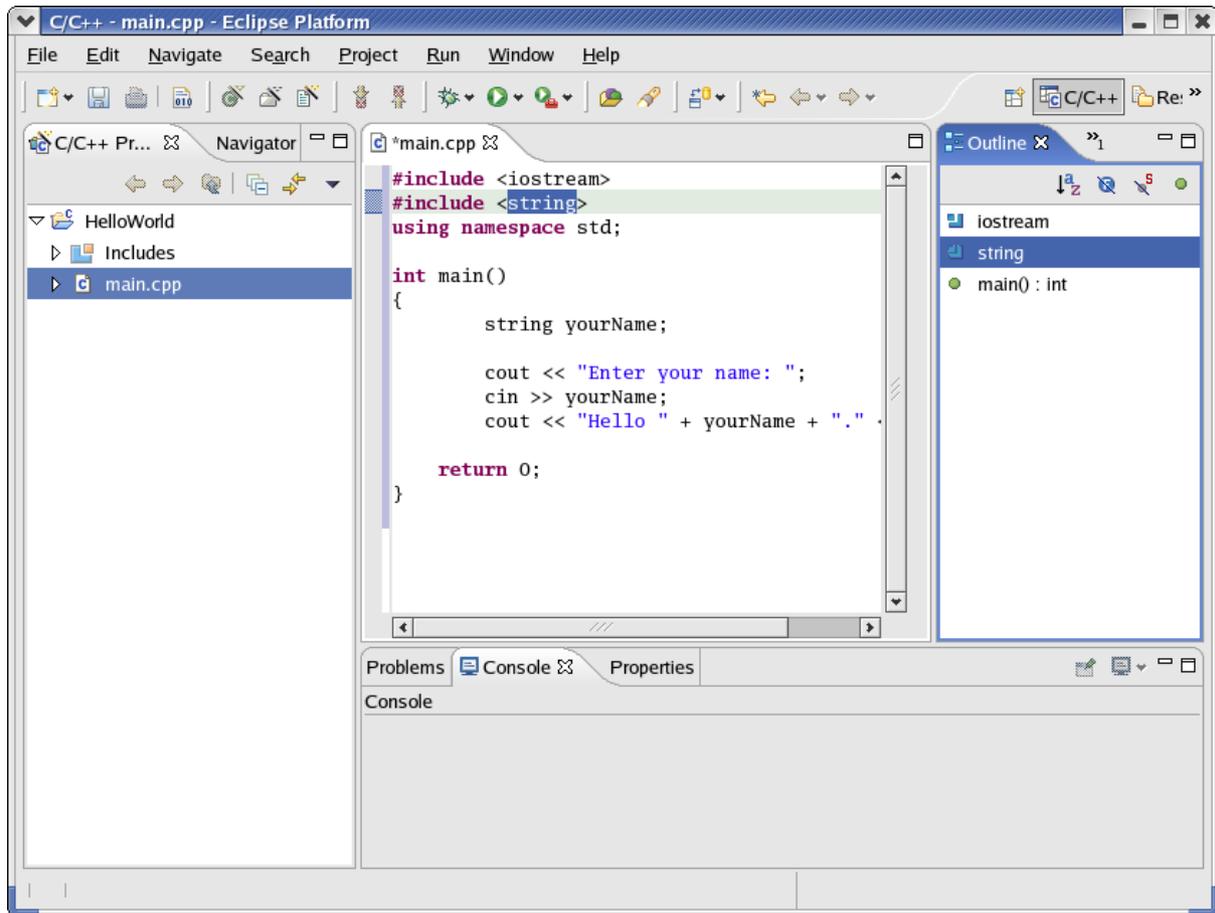
You will notice an asterisk in front of the file name on the tab in the **Editor** view; this tells you the file changed but has not been saved. An editor in this state is known as a "dirty" editor.



Step through the code

You will also see the **Outline** view has also been populated with objects created from your code.

If you select an item from the **Outline** view, the corresponding text in the editor is highlighted.

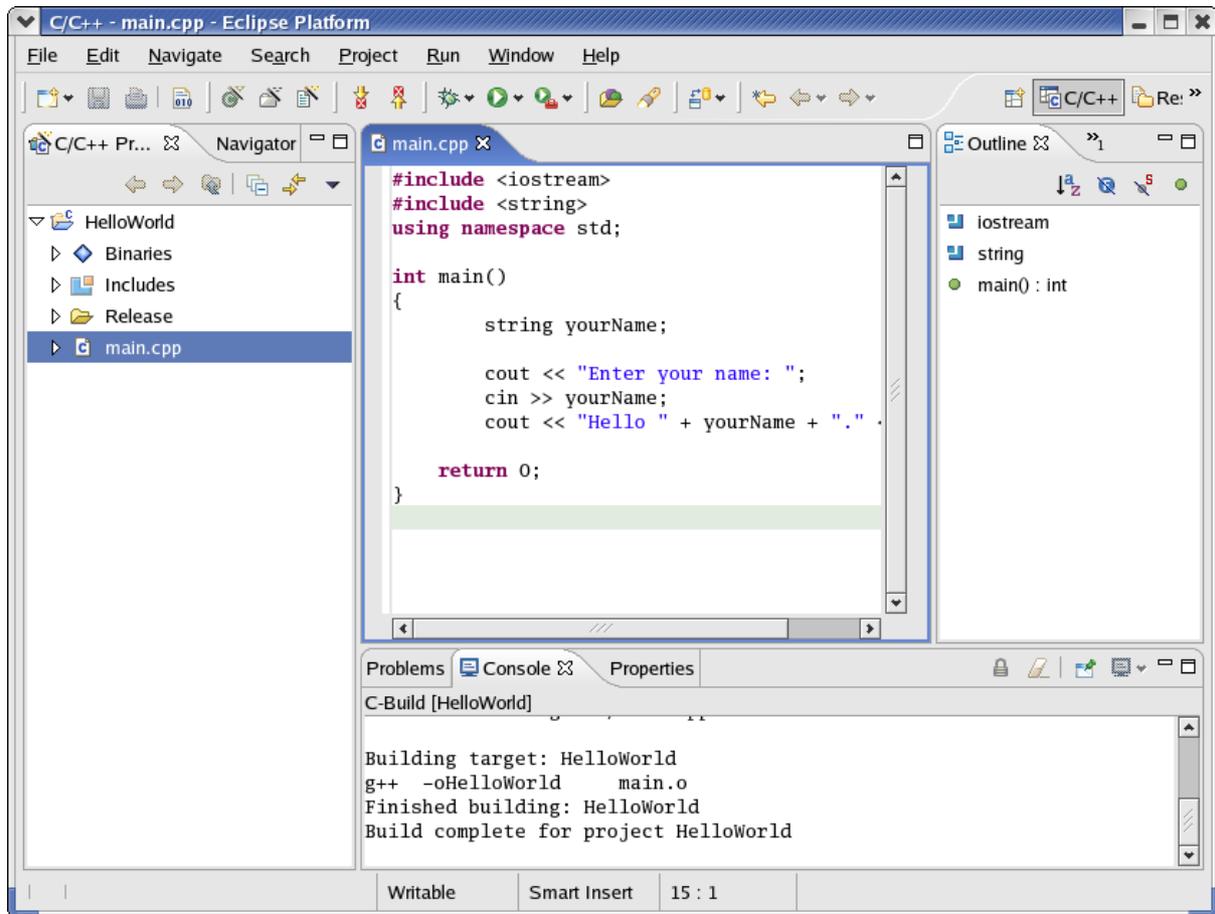


Save the file

Now save the `main.cpp` file, and build your project by typing `[CTRL]+[B]`.

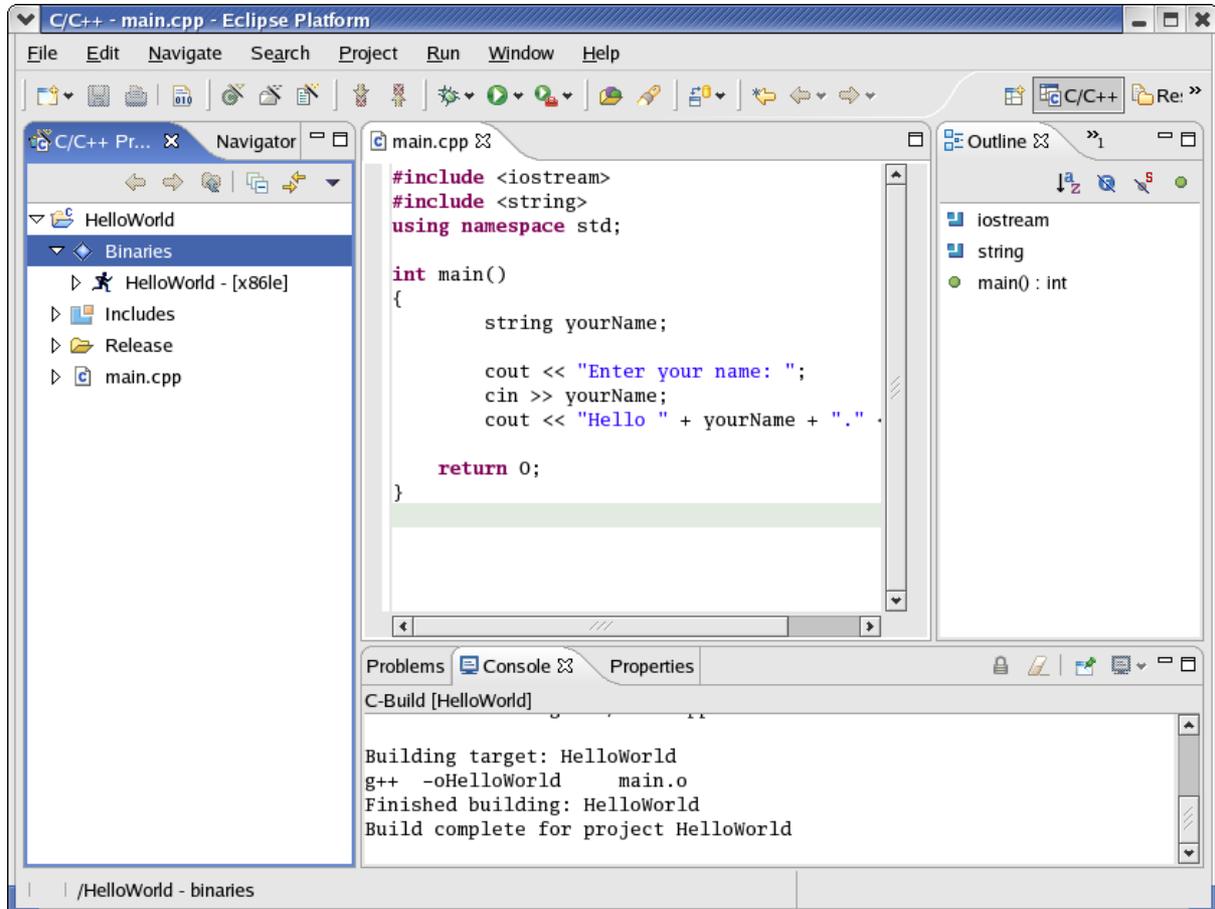
You can read through the build messages in the **Console** view. The project should build successfully showing the following message:

```
Build complete for project HelloWorld
```



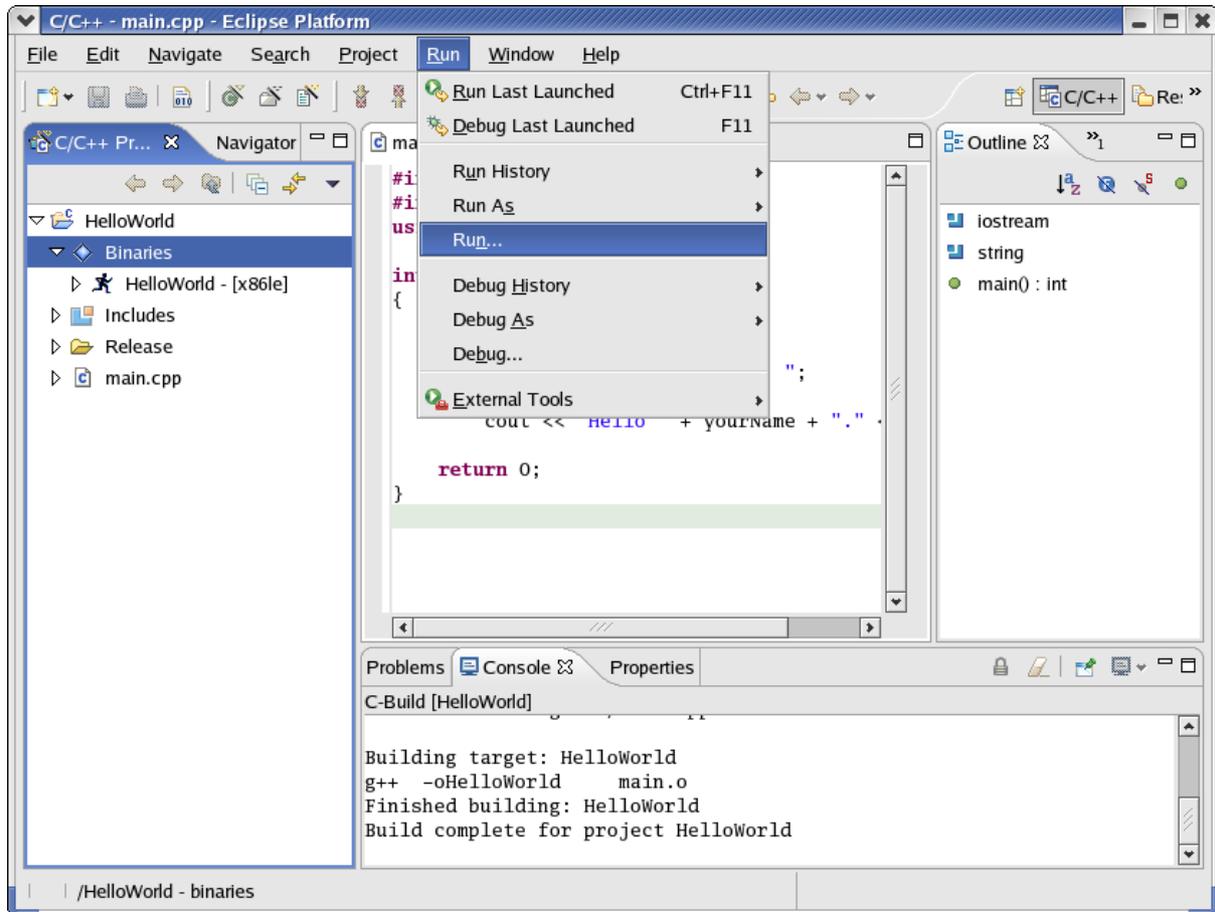
View Executable

Navigate to the C/C++ **Projects** view and expand the **Binaries** object. You will now see the executables listed.



Run the application

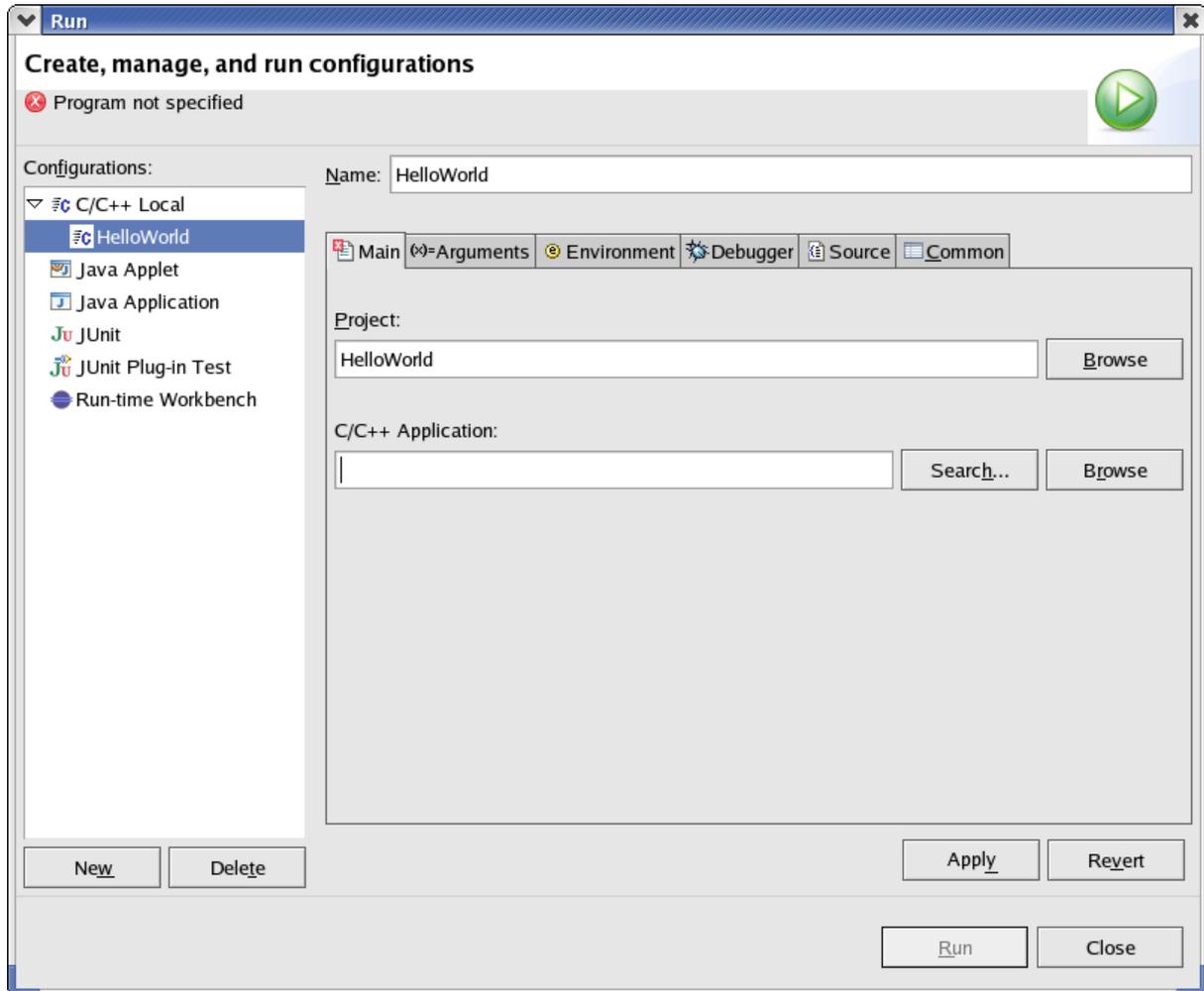
You can run your application within the *C/C++* perspective. To do so, click **Run > Run...**



Run Configuration

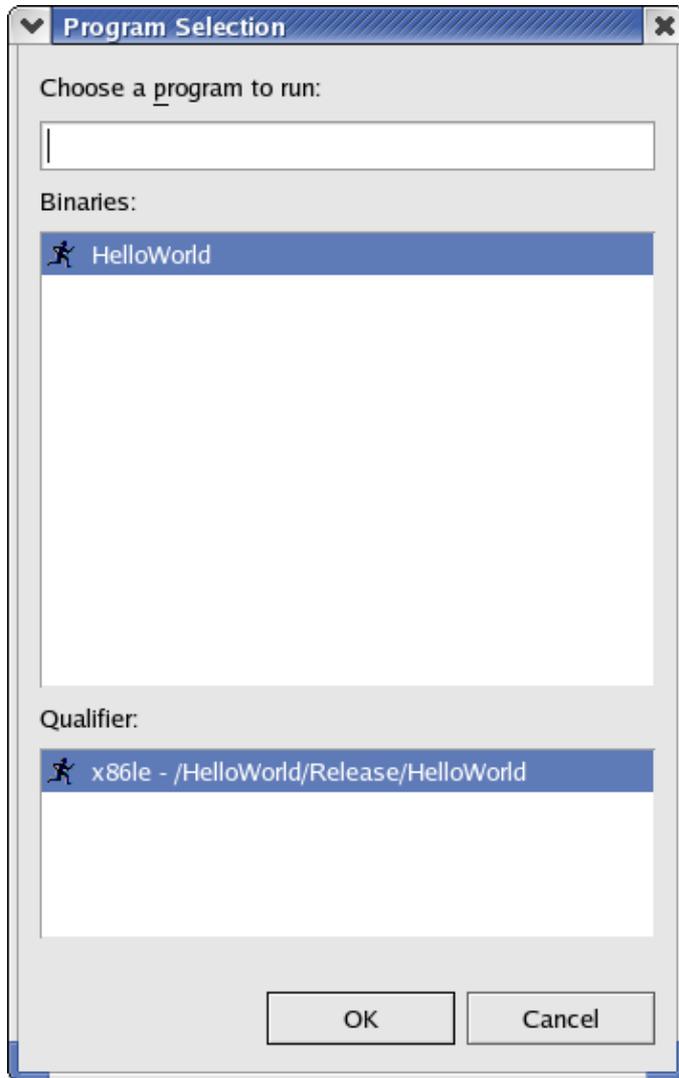
You should now see the **Run Configurations** dialog. Navigate to the **Configurations:** view, and then double-click **C/C++ Local**. This creates your Run Configuration. Select the new Run Configuration in the **Configurations** view.

Now click the **Main** tab and then the **Search** button.



Program Selection

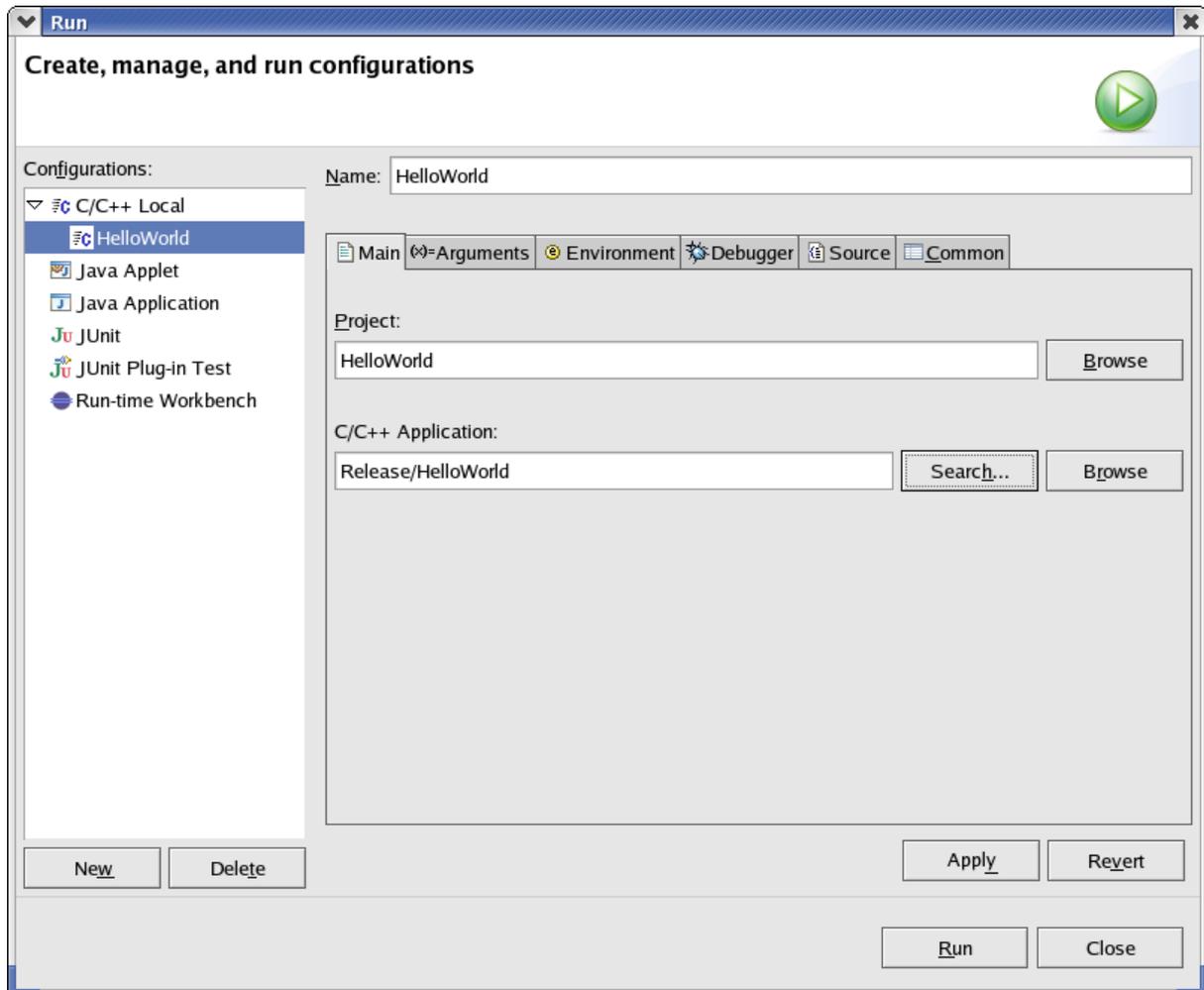
In the **Program Selection** dialog, select the executable you would like to run, then click **OK**.



Run Configuration

The executable file should appear in the **C/C++ Application** text area.

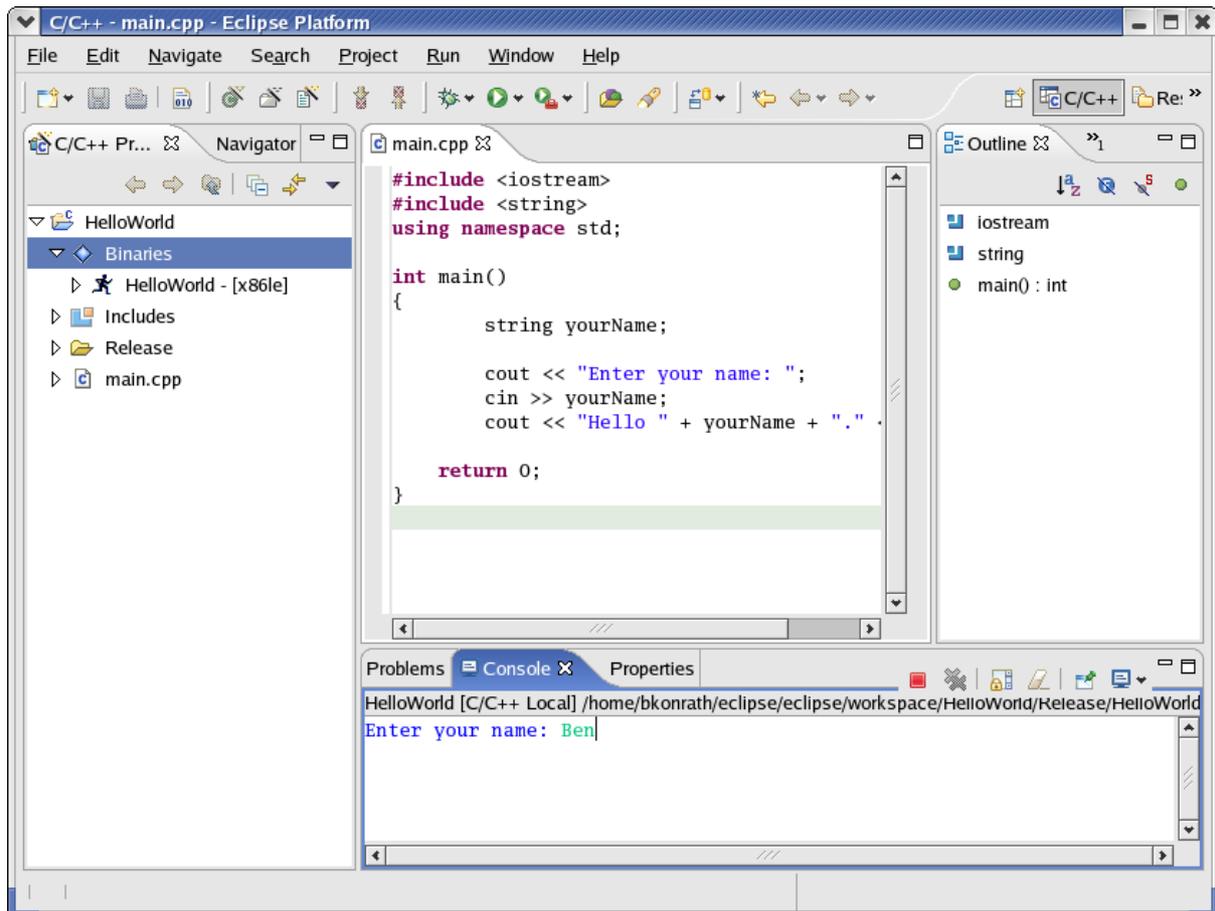
Click **Apply**, and then click **Run** to run the application.



Console View

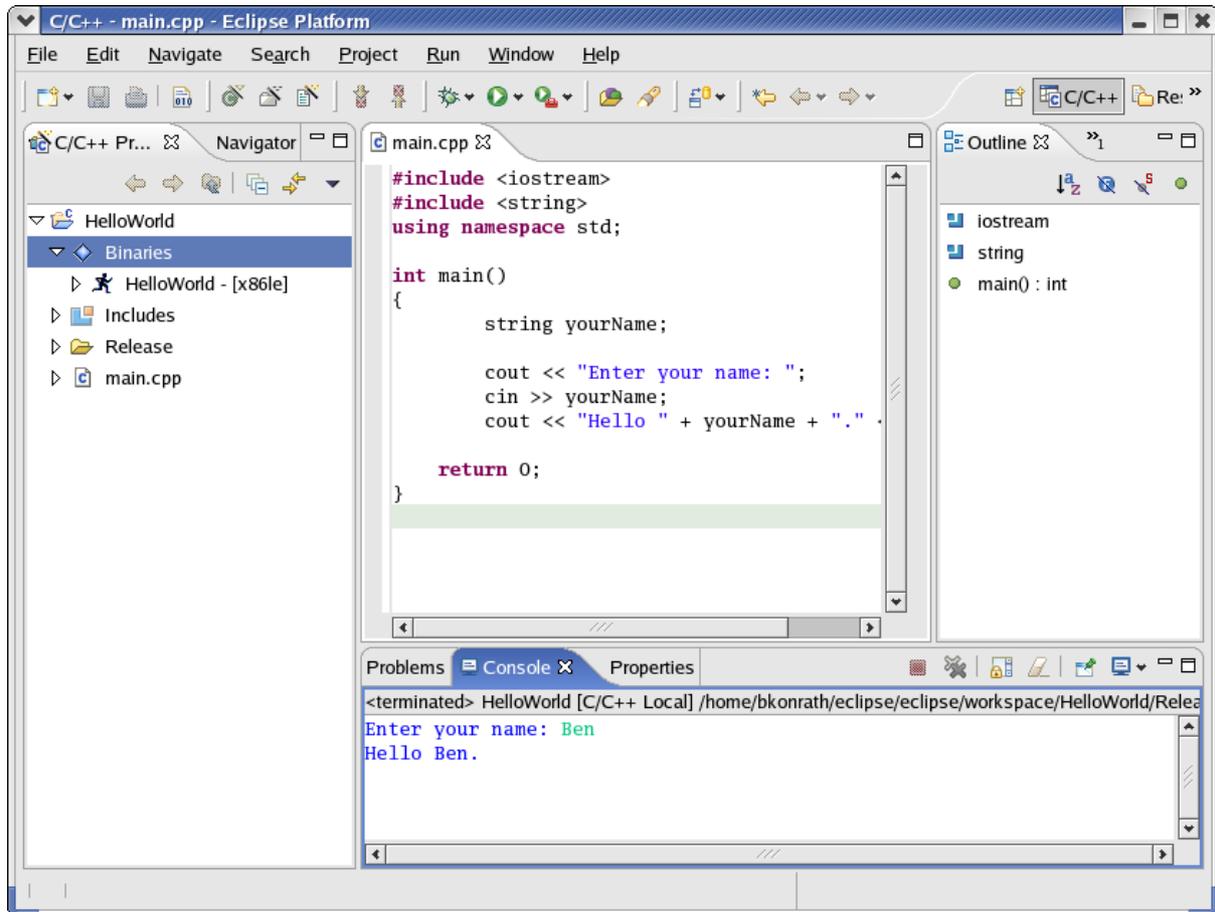
You should now see the application running in the **Console** view. The **Console** will also show which application is running in a title bar. Note that the view can be configured to display different elements (such as user input elements) in different colors.

Type in your name and press [Enter].



Run complete

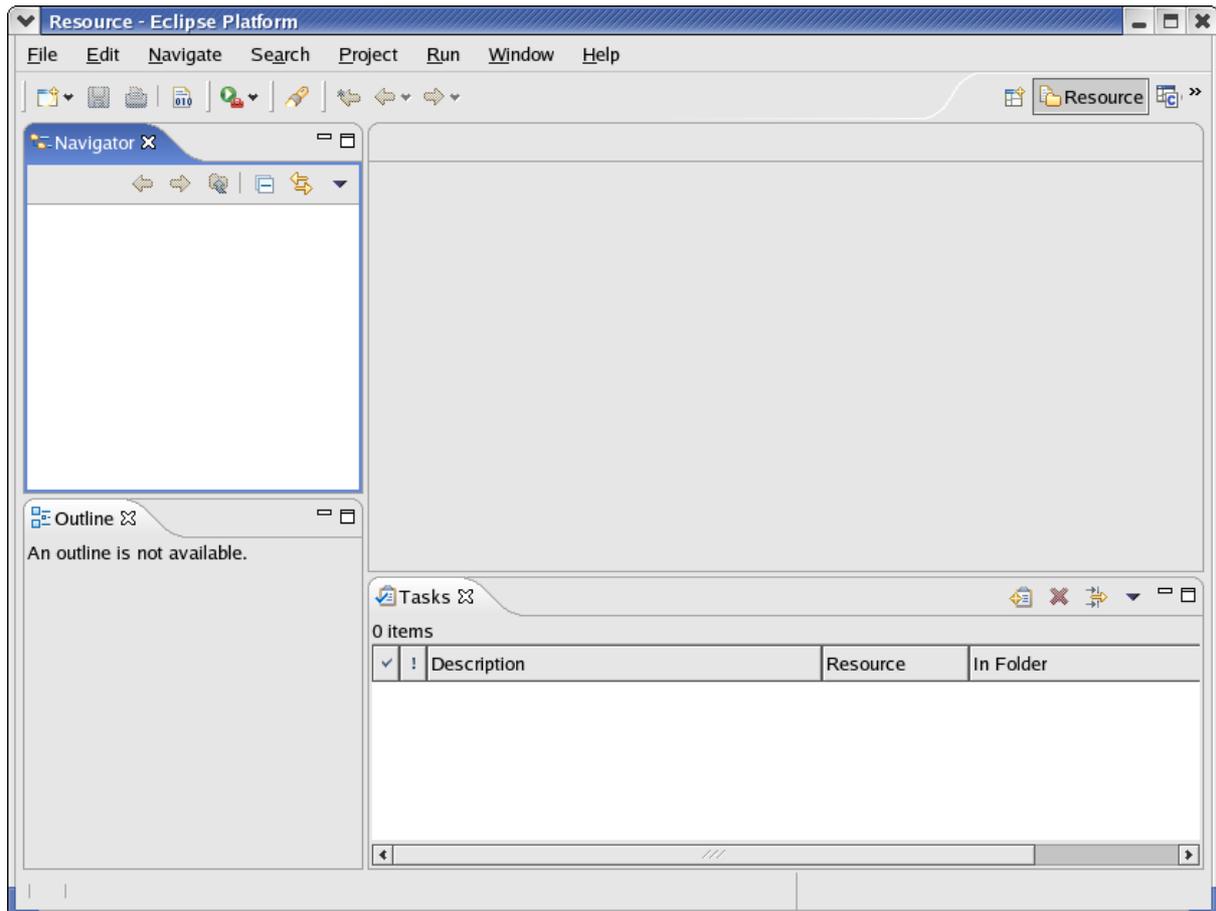
The title bar in the **Console** view shows you when the program has terminated.



CDT "Standard Make" Tutorial

We will now walk through the process of creating a simple 'Hello World' application using a Standard Make project in the CDT.

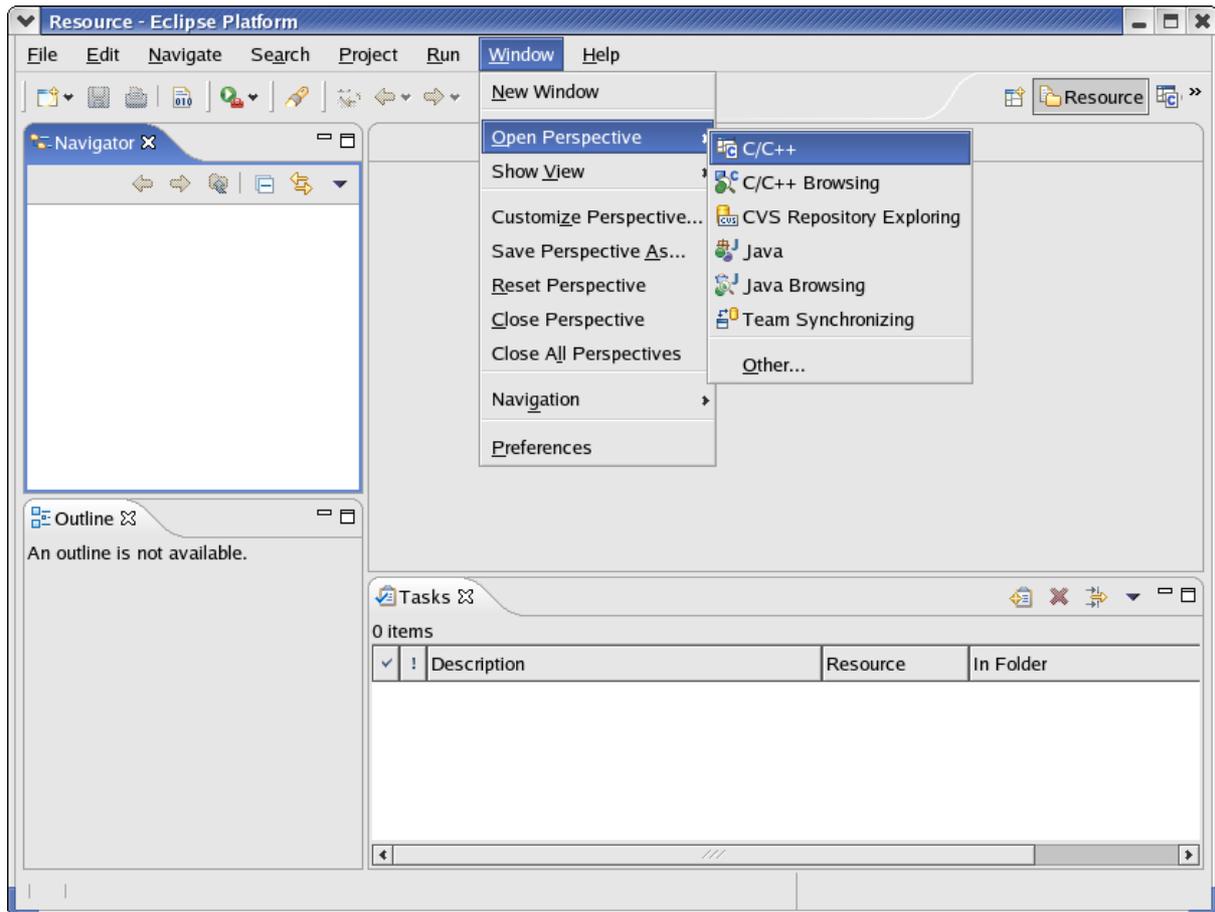
The image below shows the standard Workbench.



Open the C/C++ Perspective

Click **Window > Open Perspective > C/C++**.

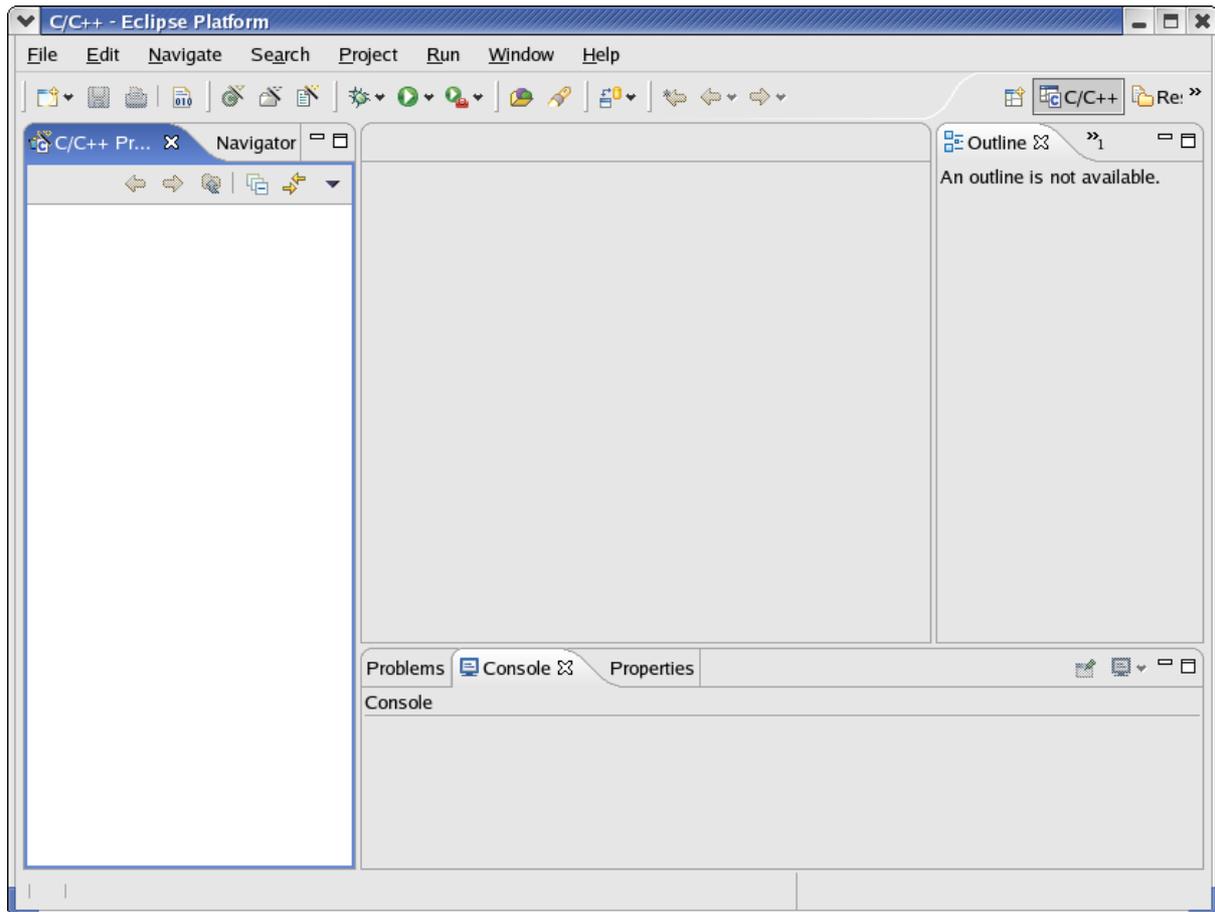
Note: If the C/C++ perspective is not listed, click **Other...** and select **C/C++** from the **Select Perspective** dialog box.



C/C++ Perspective

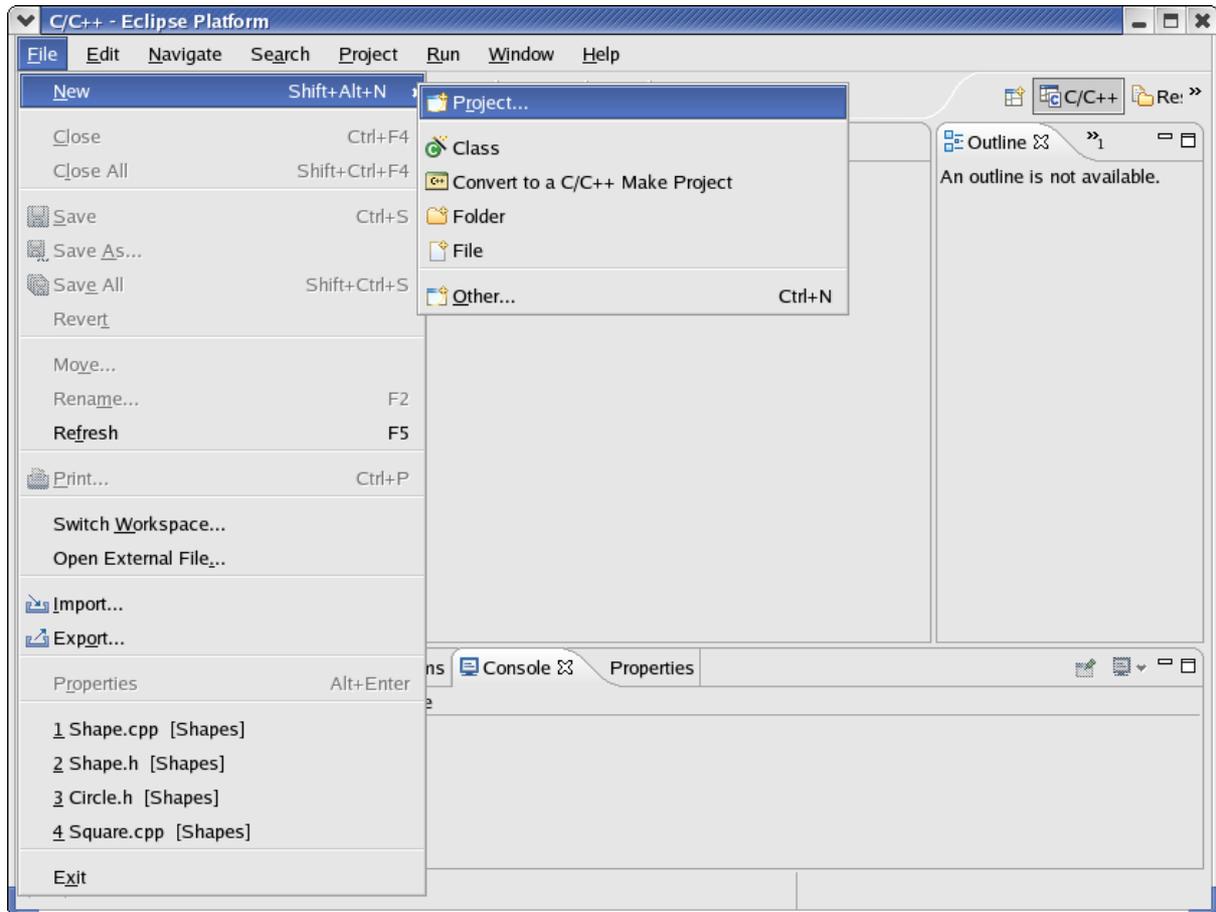
This is the **C/C++ Perspective**. Notice the **C/C++ Projects** view is on the left and the **Outline** view has moved to the right. The center area is reserved for your editor.

There may be other views available in your workbench, such as the **Navigator** or **Make Targets** views.



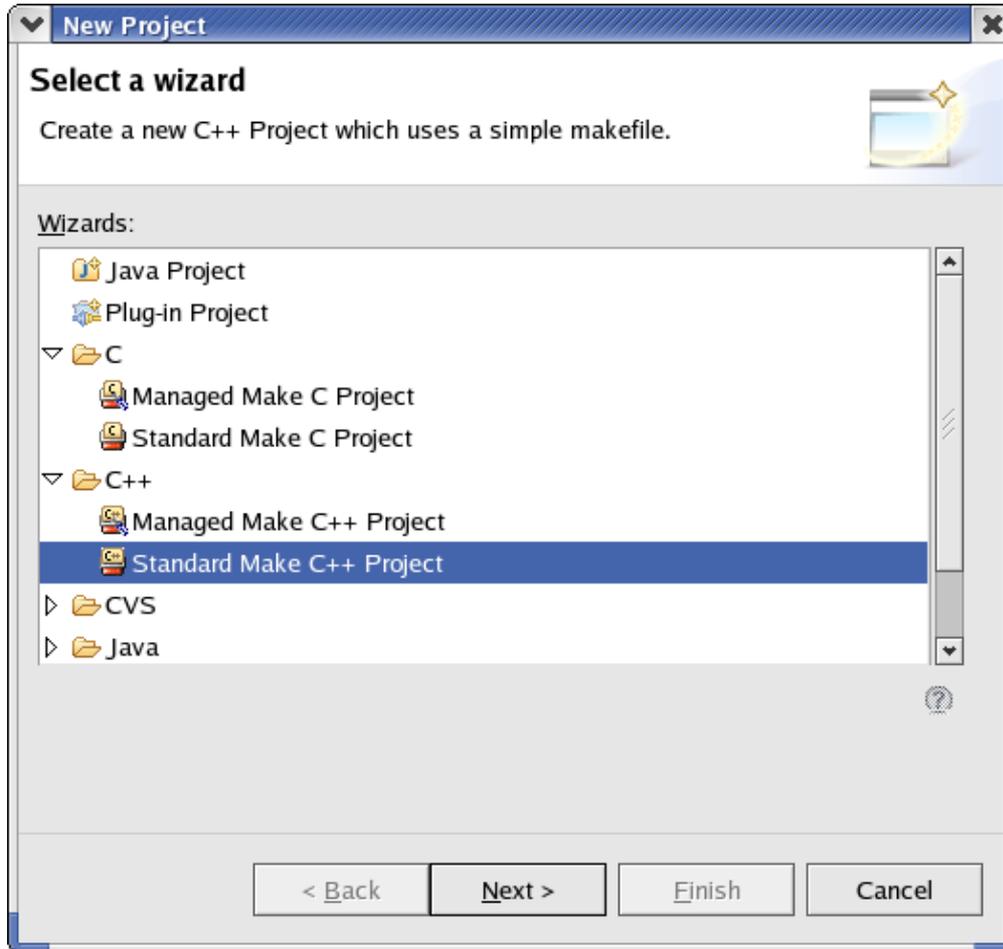
Create a project

You can now create a **C/ C++ project** by clicking **File > New > Project**.



New Project Wizard

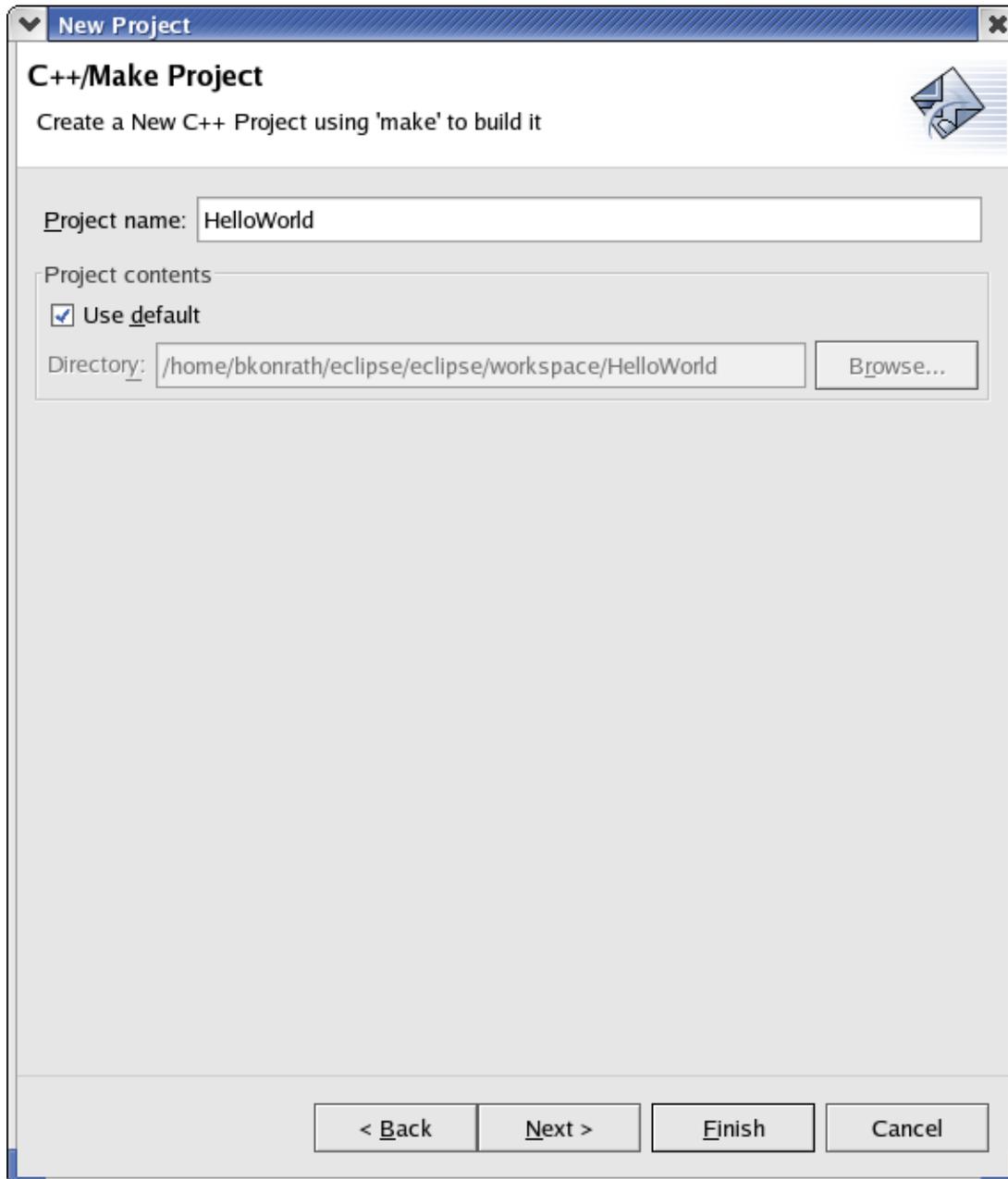
You will now see the **New Project** wizard. Open a C or C++ project and select a **Standard Make** project. A Standard Make C/C++ project requires you to provide a makefile; a Managed Make project creates a makefile for you.



New Project

Enter a name for the project. You can also enter a new path for your project by deselecting the **Use Default Location** checkbox and entering the new path in the **Location** text box.

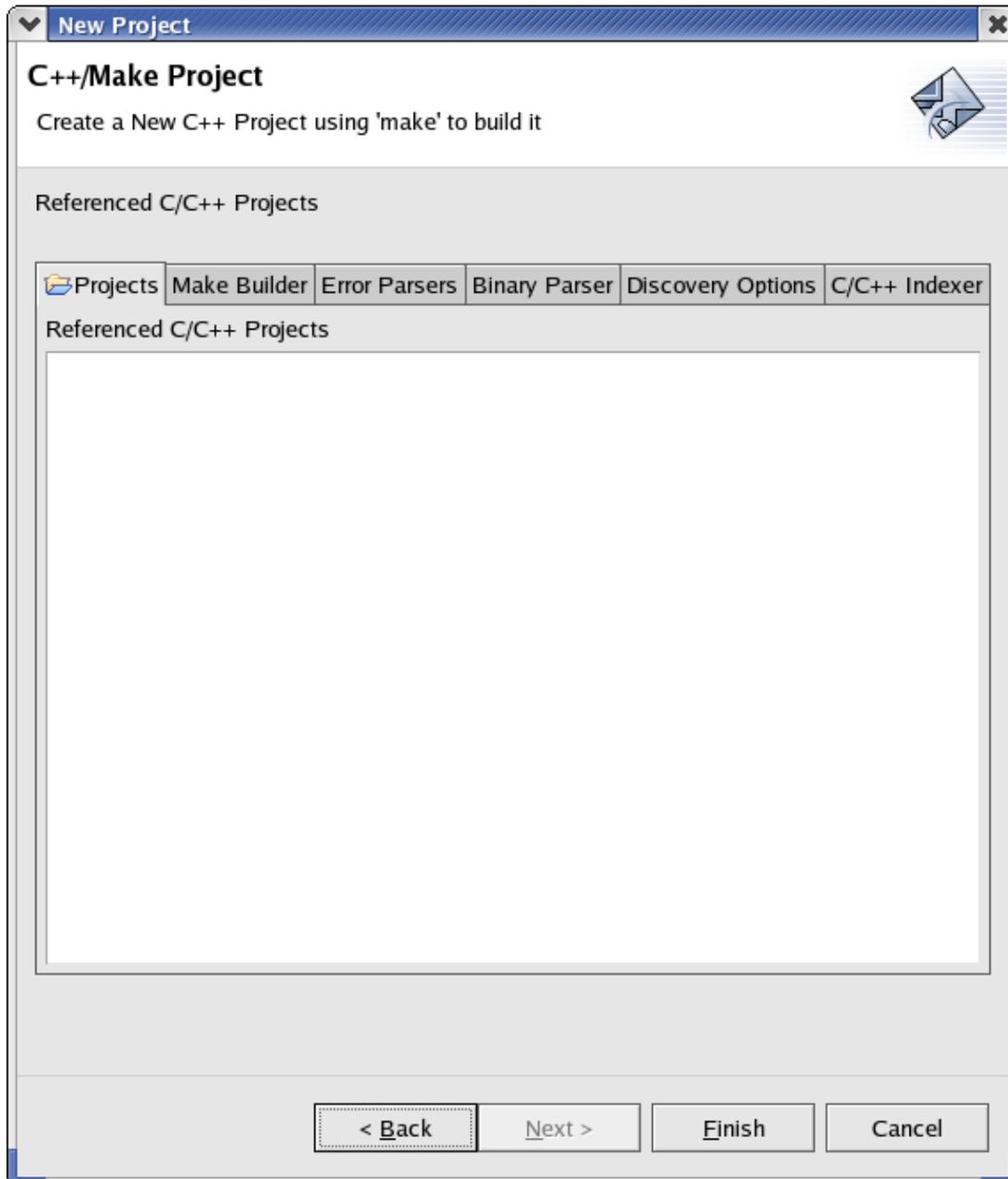
Click **Next**.



Project References

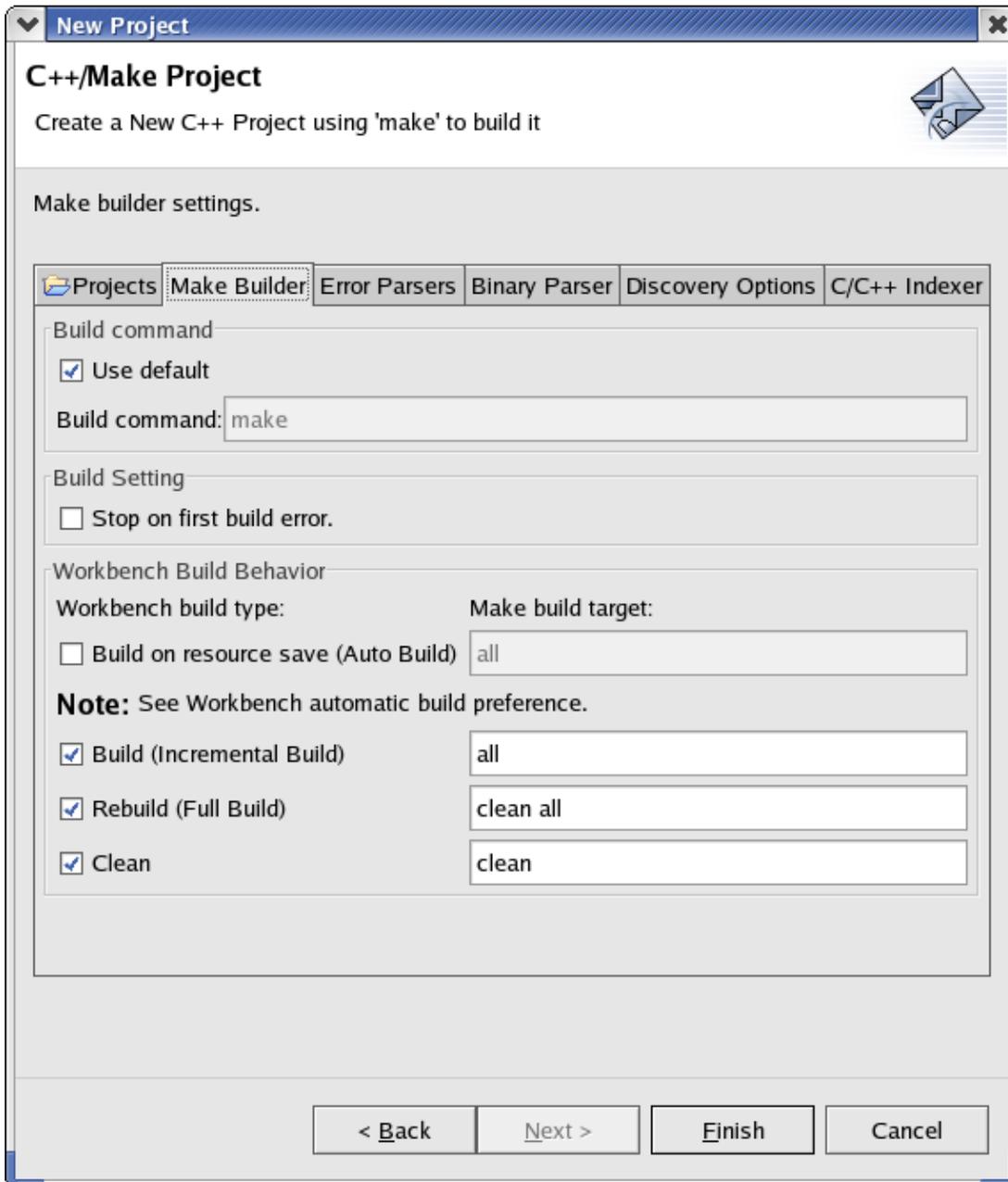
Now you will select your preferences for the project.

This page is where you would select other referenced C/C++ projects. As this is a simple Hello World project, you will not need to make any selections here.



Make Builder Settings

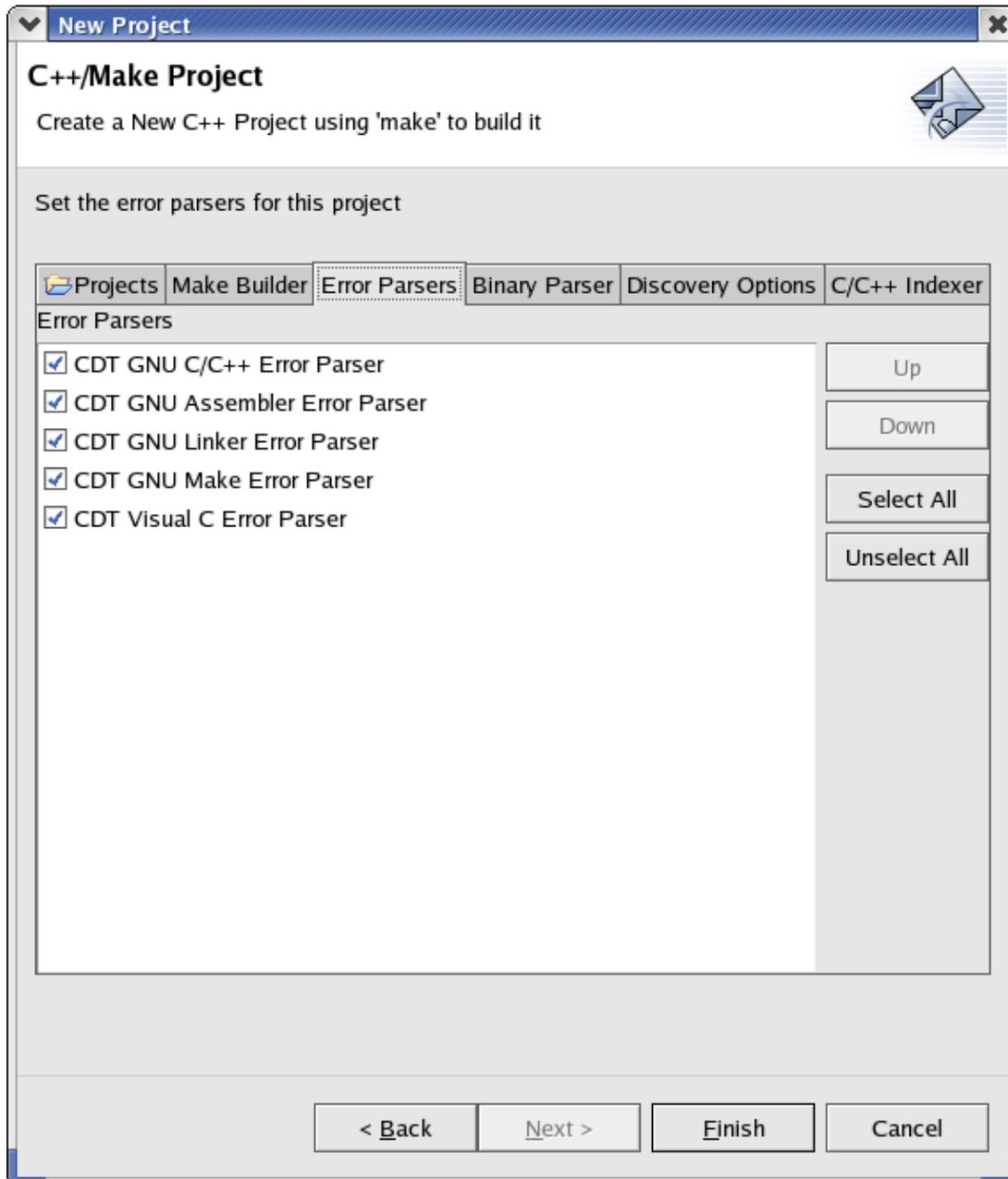
Click the **Make Builder** tab. If your build environment has a build command that is not make, you need to change your **Build Command** from the default. To do this, deselect the **Use default** checkbox and enter the build command in the text box.



Error Parsers

Click the **Error Parsers** tab and select the error parsers you require for the project. (You can leave all parsers selected.)

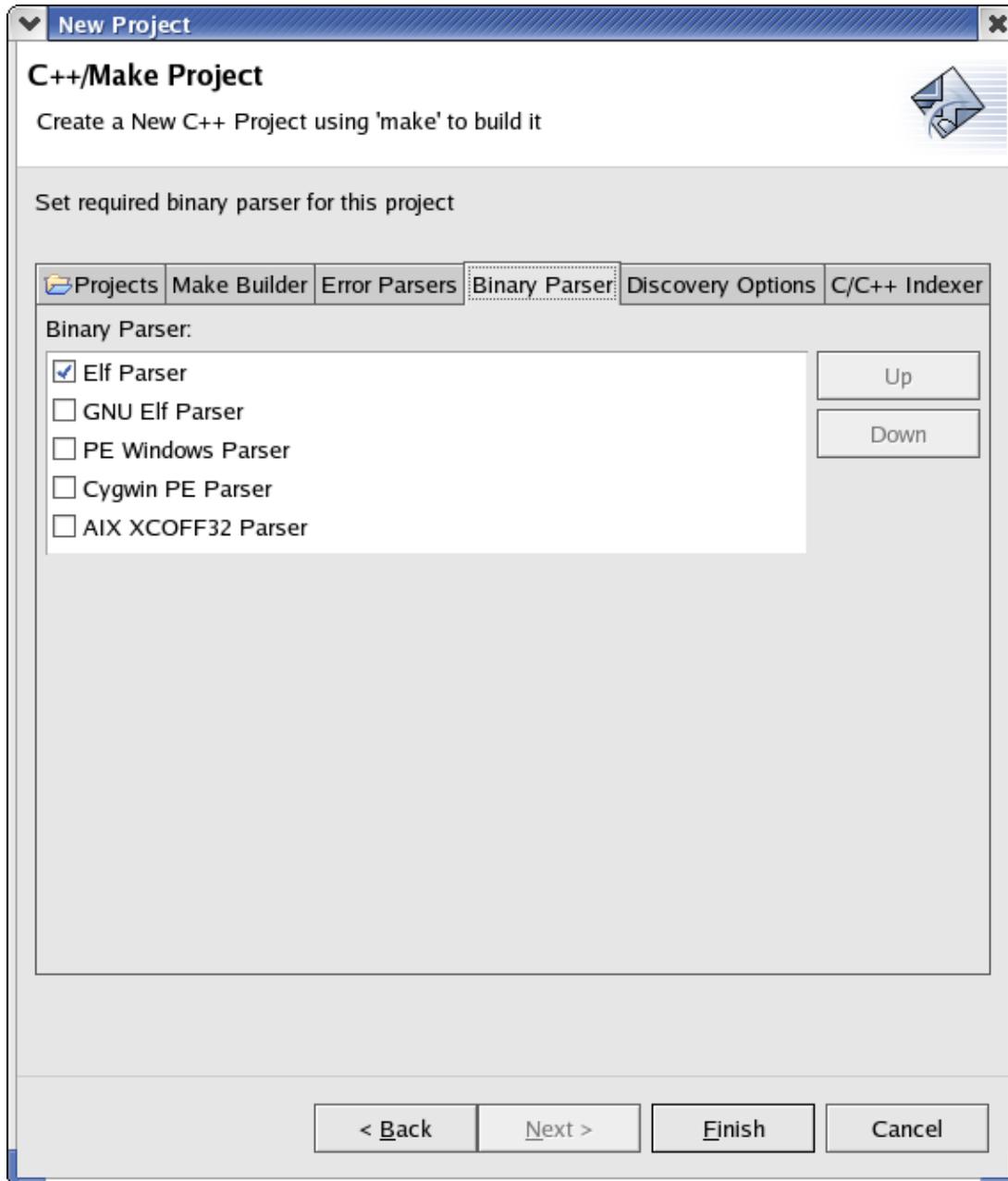
You can also change the order in which Error Parsers are called.



Binary Parser Settings

Click the **Binary Parser** tab, select the Binary Parsers you require for the project, and set the order in which they are to be used.

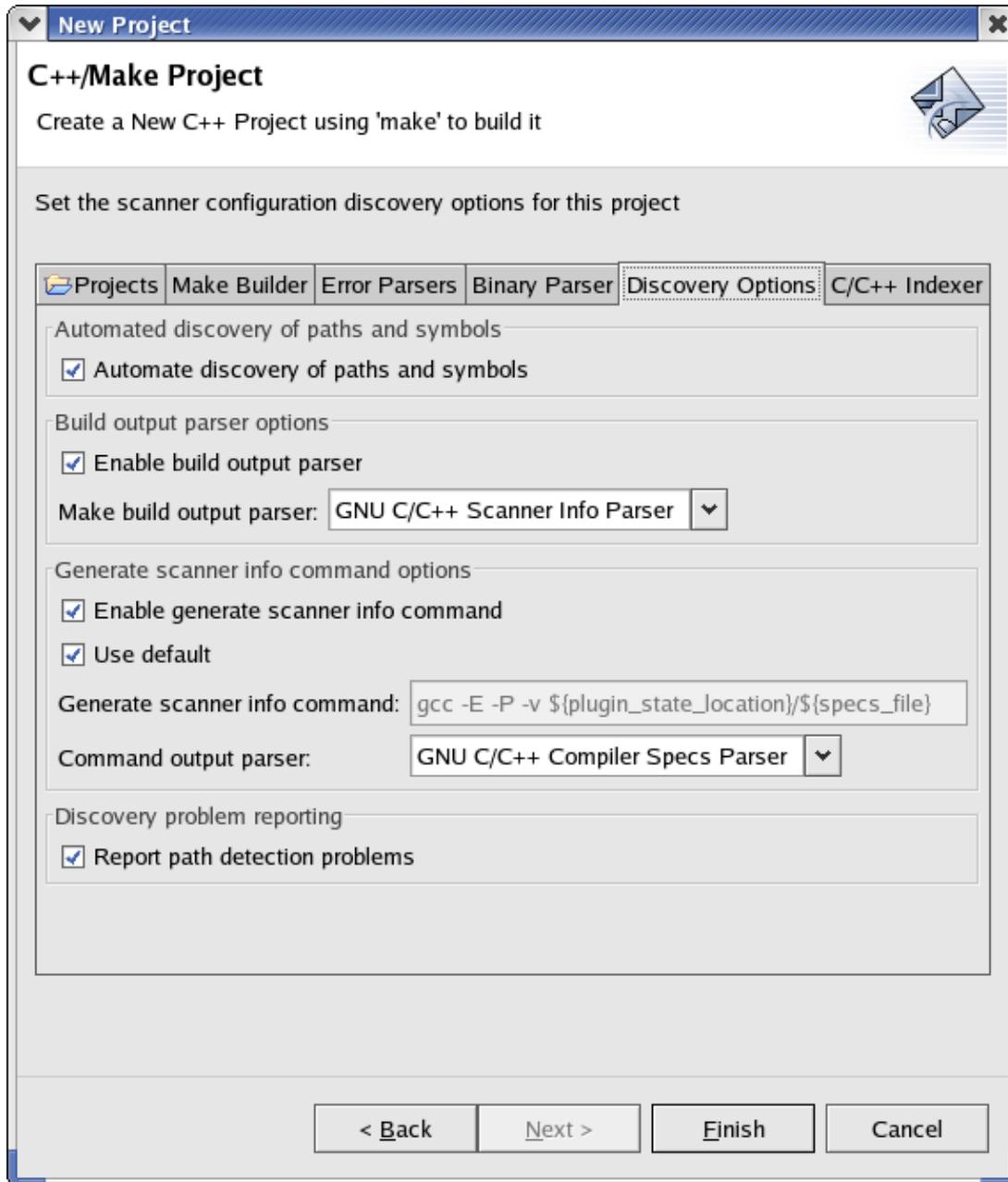
It is important you chose the proper parser settings to ensure the accuracy of the **C/C++ Projects** view and the ability of Eclipse to successfully run and debug your programs. After you select the correct parser for your development environment and build your project, you can view the symbols of the `.o` file in the **C/C++ Projects** view.



Discovery Options

Click the **Discovery Options** tab. This page defines the configuration for autodiscovery of include paths and preprocessor symbols for the parser. This enables the parser to understand the contents of the C/C++ source code so that you may more effectively use the search and code-completion features.

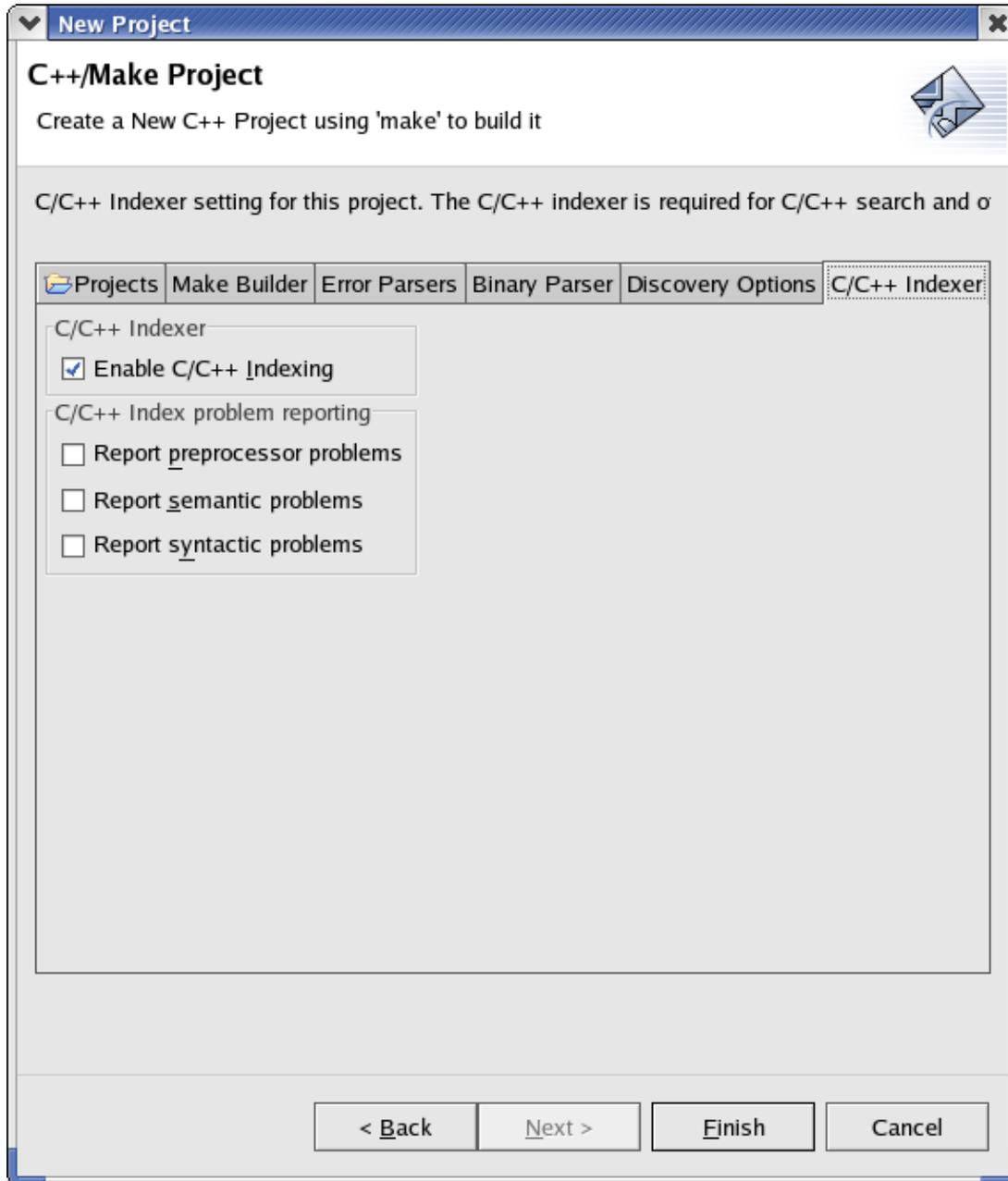
Select the **Automate scanner configuration discovery** checkbox to configure the scanner discovery to run automatically.



C/C++ Indexer

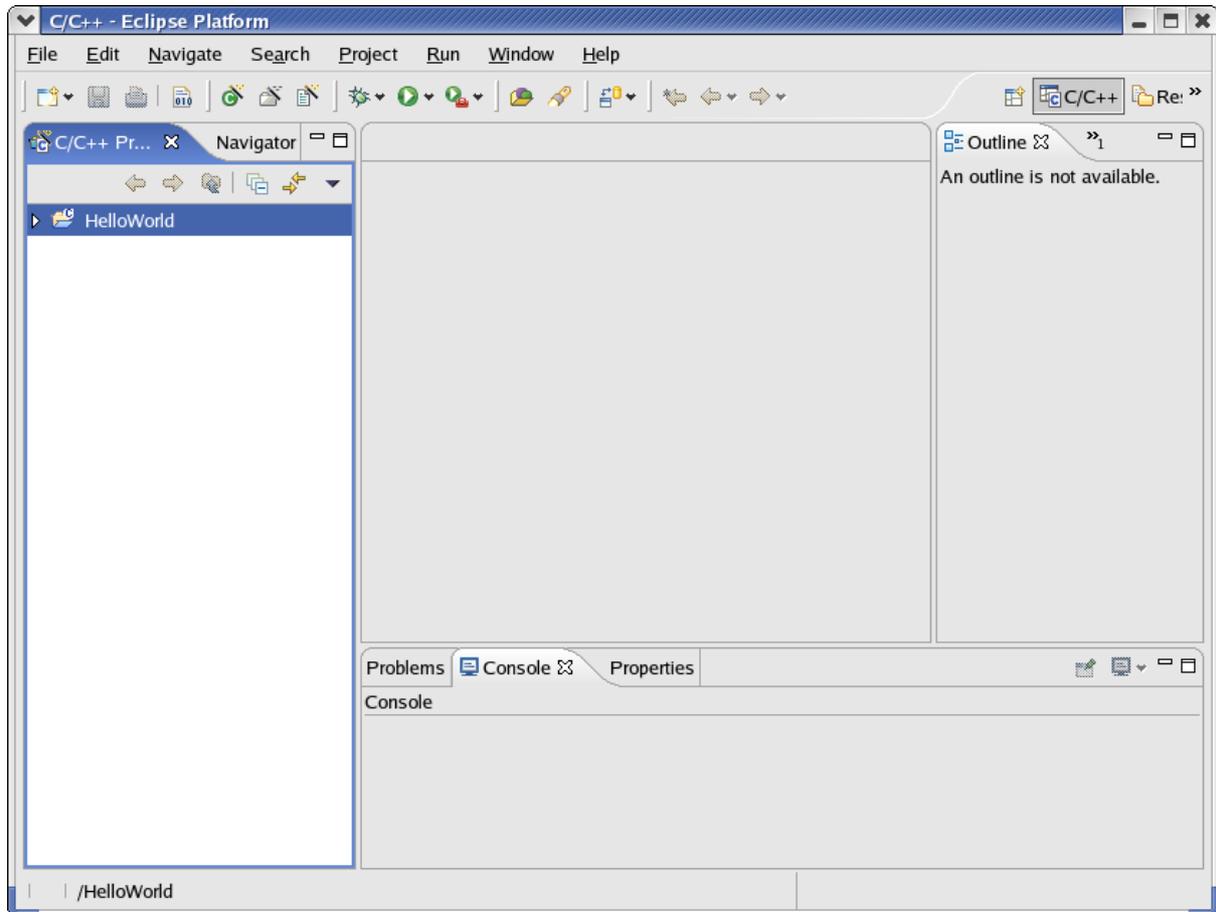
You can enable or disable the **C/C++ Indexer** here. The indexer is necessary for search and related features, such as content assist and refactoring. While there may be situations where the indexer is not required, for this project you can leave it enabled.

When you have completed setting your C/C++ project preferences, click **Finish**.



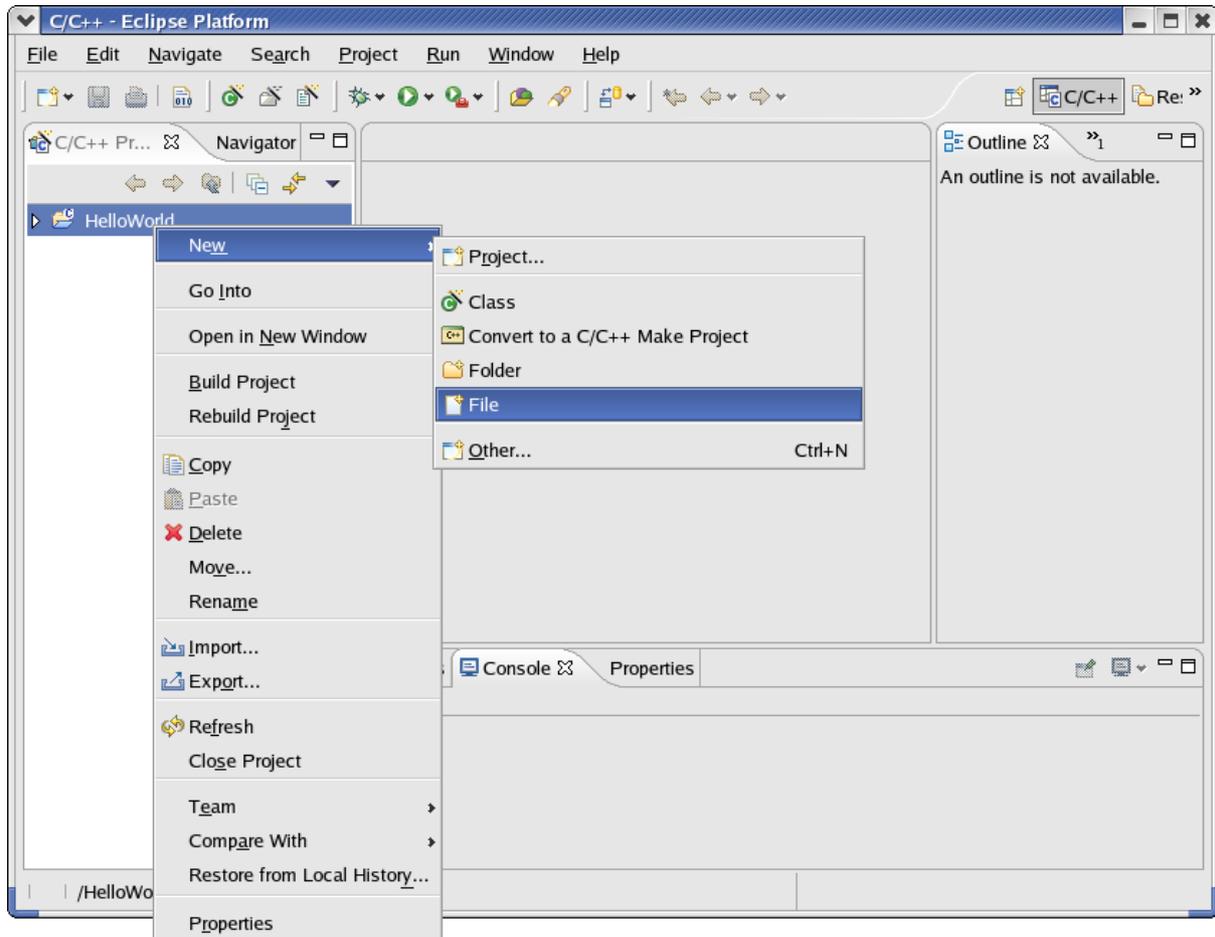
New Project

You should now see the new project in the **C/C++ Projects** view.



Create a makefile

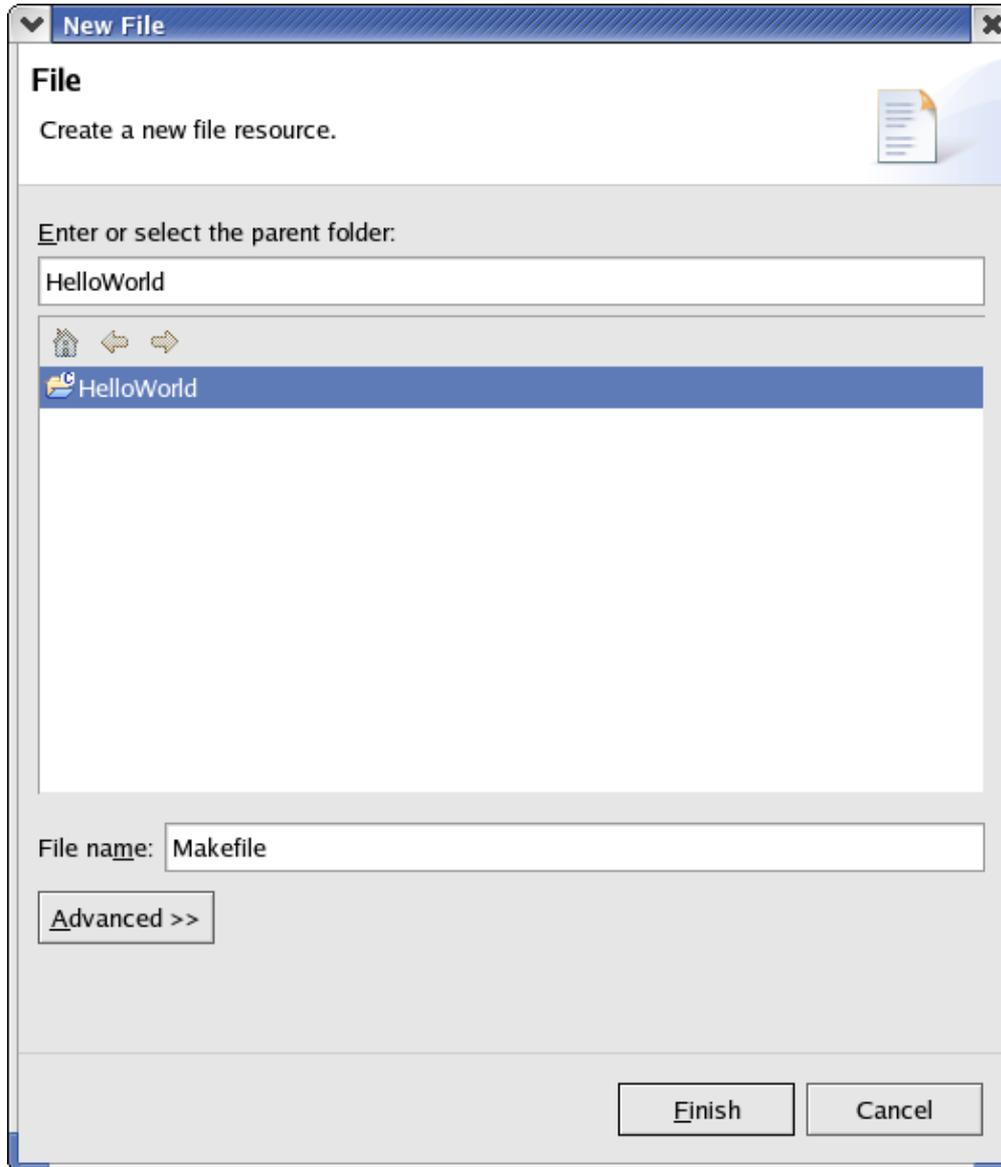
Create the makefile by right-clicking your project and selecting **New > File**.



Create a makefile

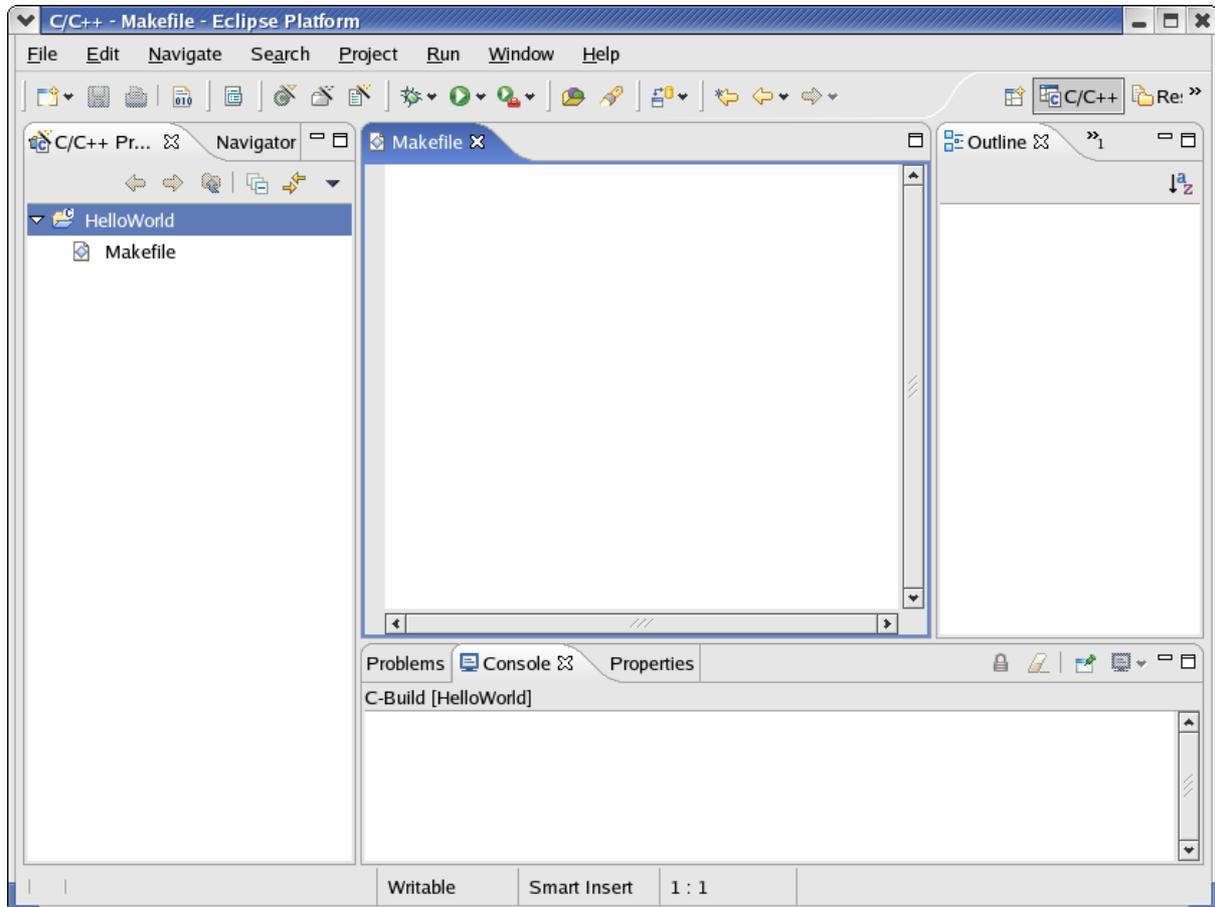
You should now see the **New File** wizard.

Enter **makefile** for the name of the file in the **File name** text area, then click **Finish**.



New Makefile

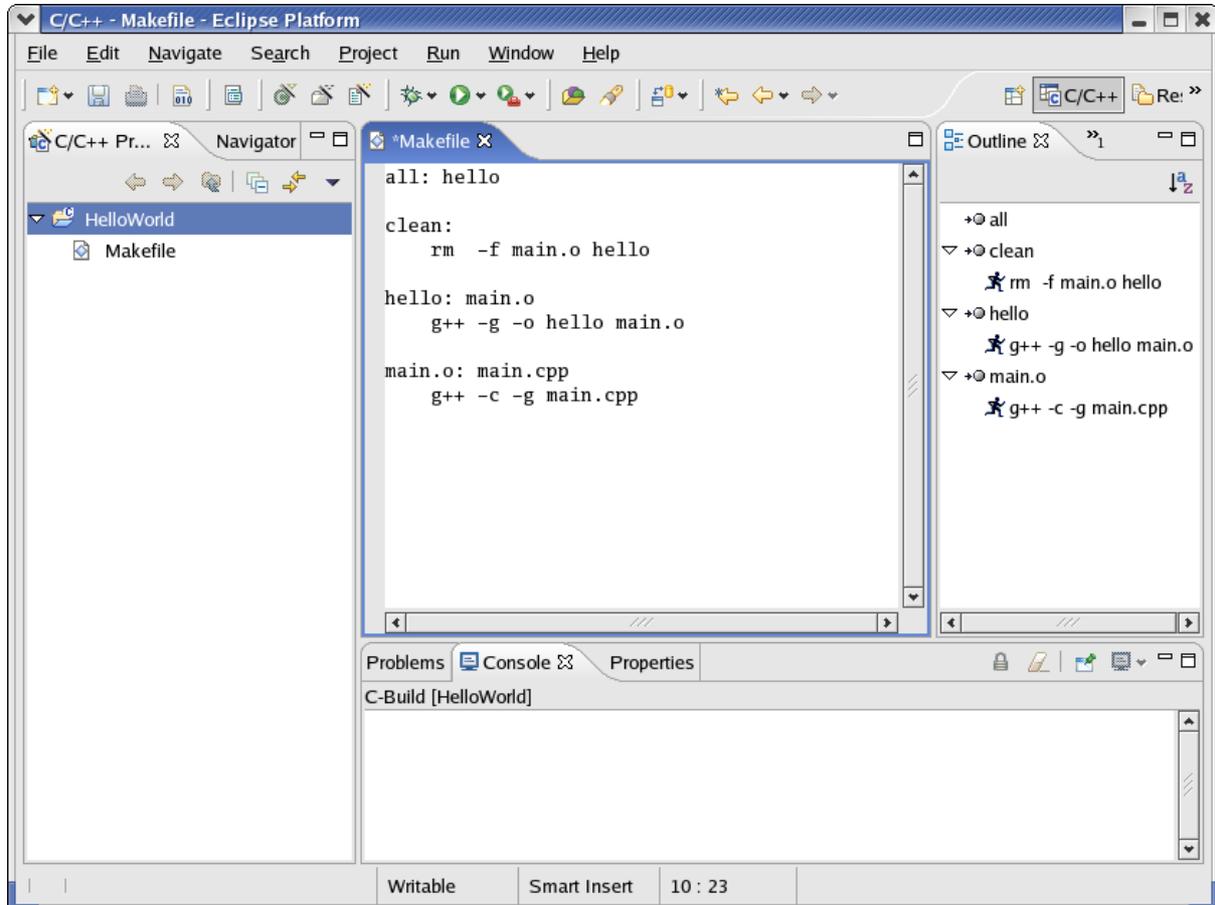
You should now see the new makefile located in the **Projects** view under the project, and the makefile should be open in the editor.



Enter the make script

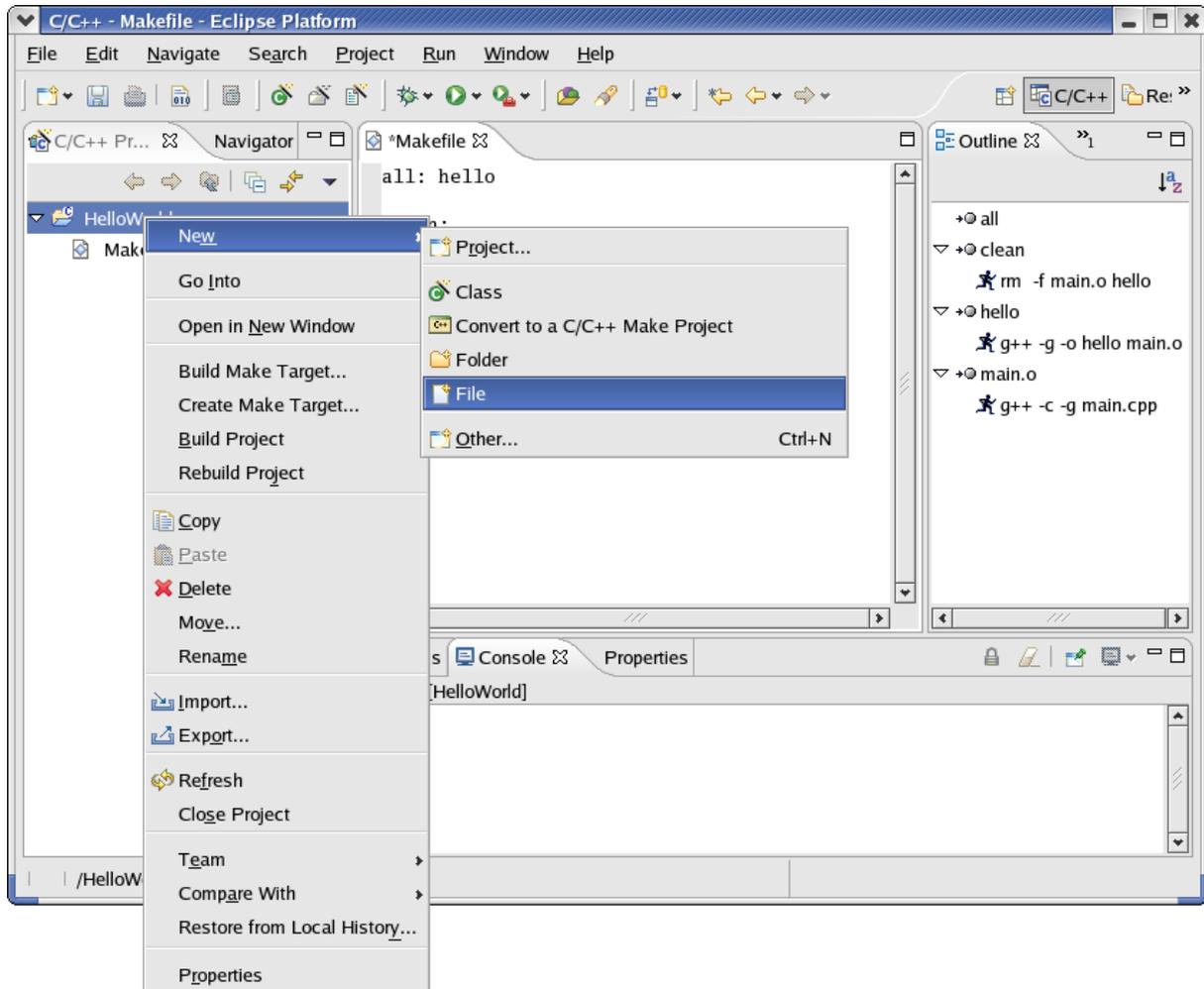
Enter the build instruction for your `makefile` that was just created. The **Outline** view displays the structure of the makefile as you add components. When you enter the code you will notice an asterisk in front of the file name on the tab in the **Editor** view; this tells you the file changed but has not been saved. A file in this state is known as a "dirty" file.

Note: Ensure you have tabs, not spaces, before your indented code, as `make` *will not* accept spaces before commands.



Create a CPP file

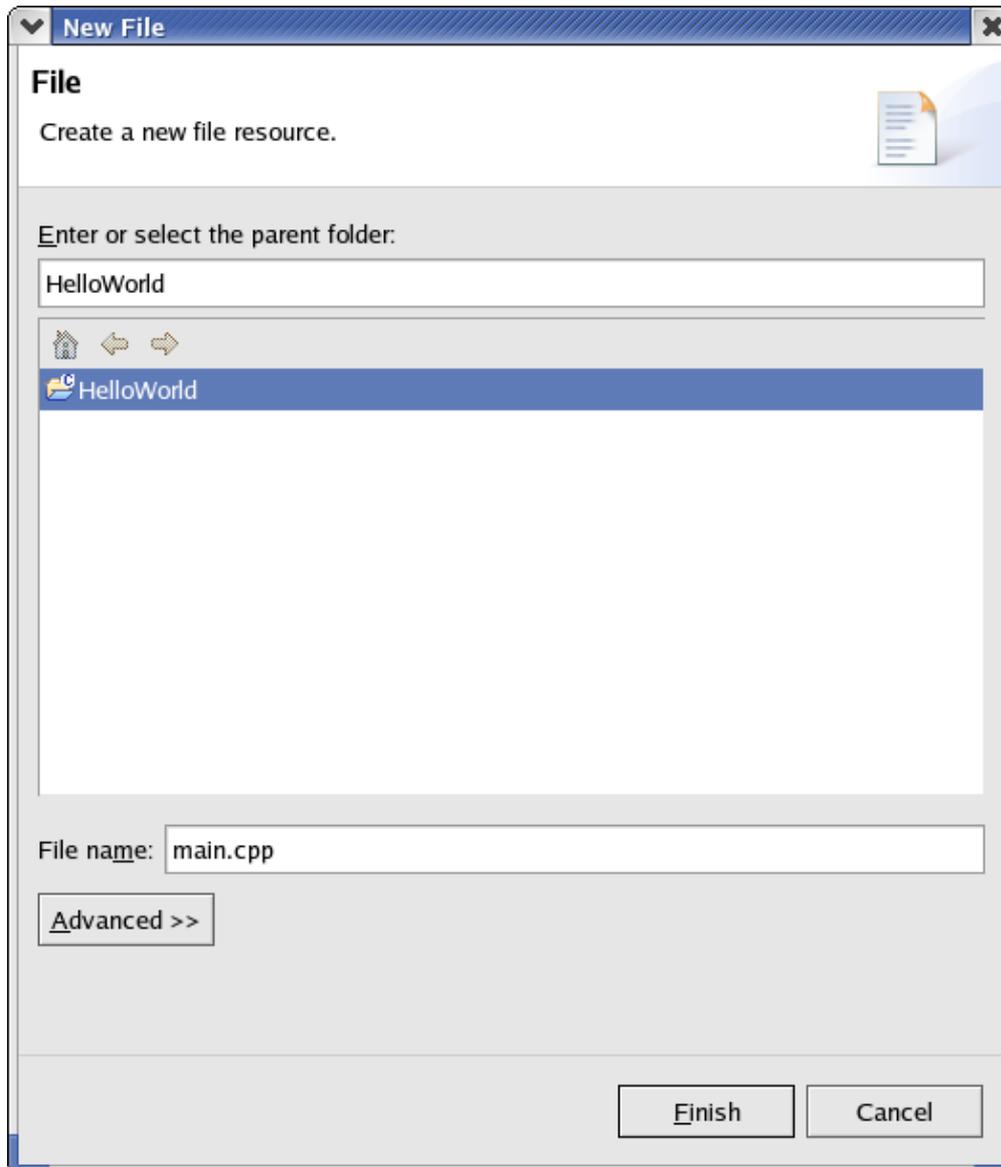
Create the main.cpp file by right clicking your project and selecting **New > File**.



Create a CPP file

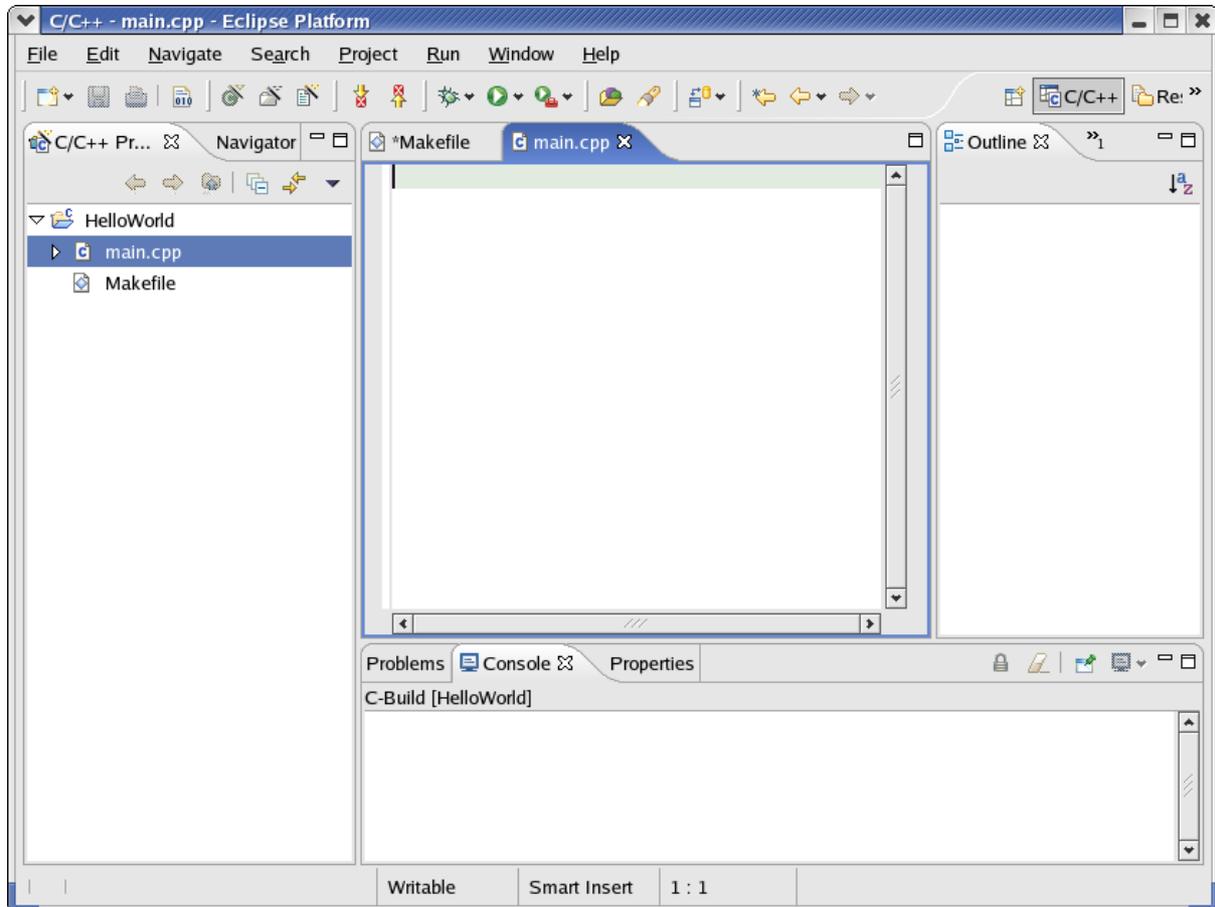
You should now see the **New File** wizard.

Enter **main.cpp** in the **File name** text area then click **Finish**.



New Project Files

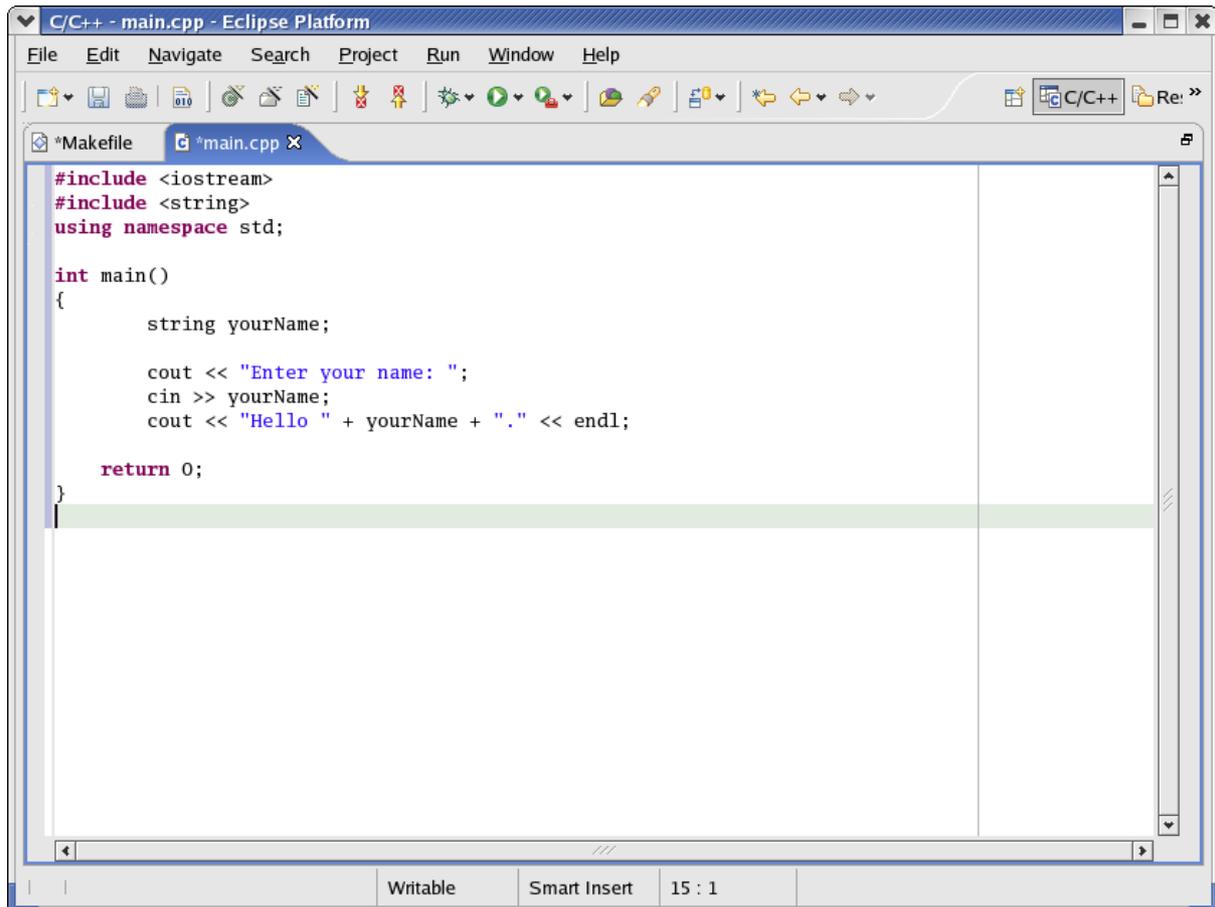
You should now see the **main.cpp** file located in the **Projects** view under the project, and the **main.cpp** file should be open in the editor.



Enter code

Enter the code in the `main.cpp` file that was just created. You can double click the **main.cpp** tab in the **Editor** view to expand the view.

Note: Leave a blank line at the end of the code; some compilers require one.



```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string yourName;

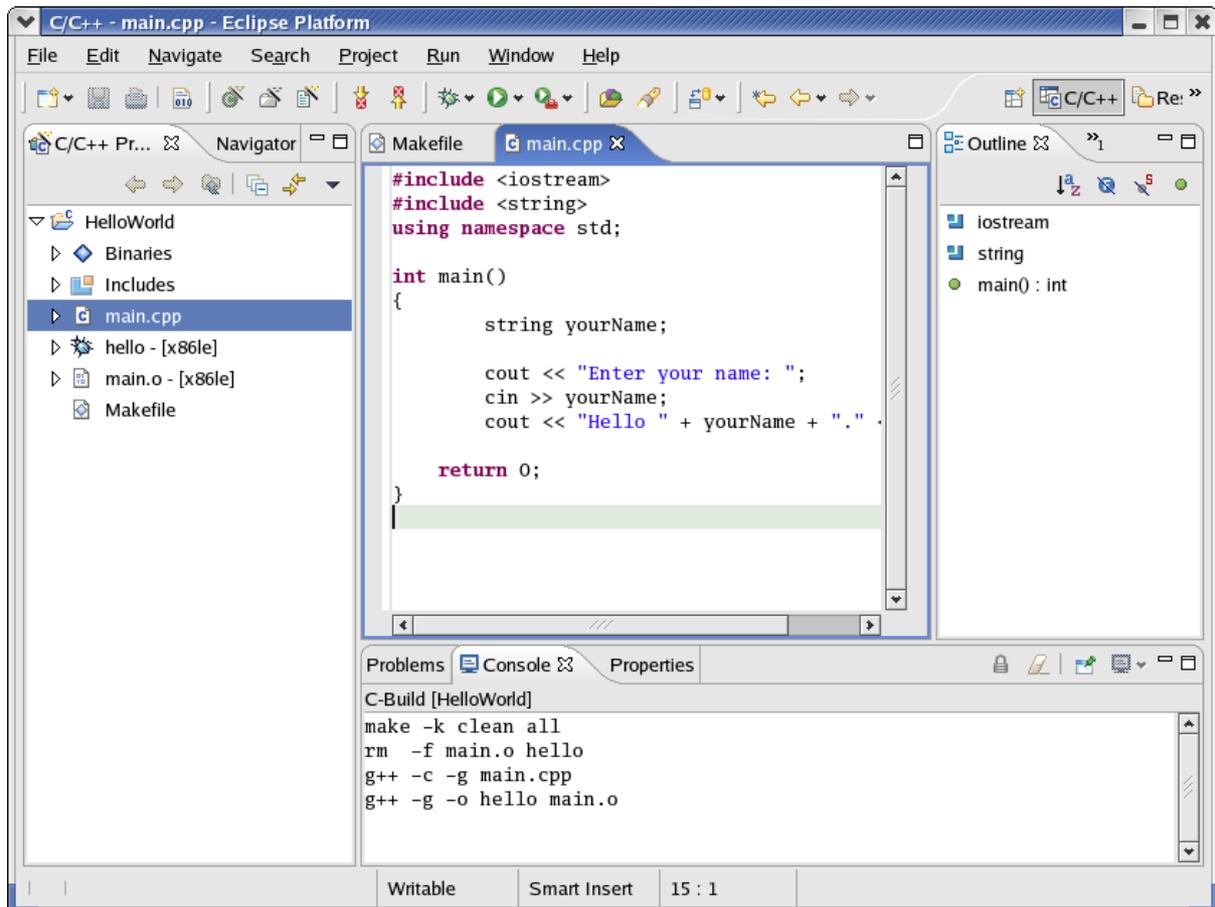
    cout << "Enter your name: ";
    cin >> yourName;
    cout << "Hello " + yourName + "." << endl;

    return 0;
}
```

Build the project

Save the `makefile` and `main.cpp` files by selecting them and typing [CTRL]+[S], and then build your project by typing [CTRL]+[B].

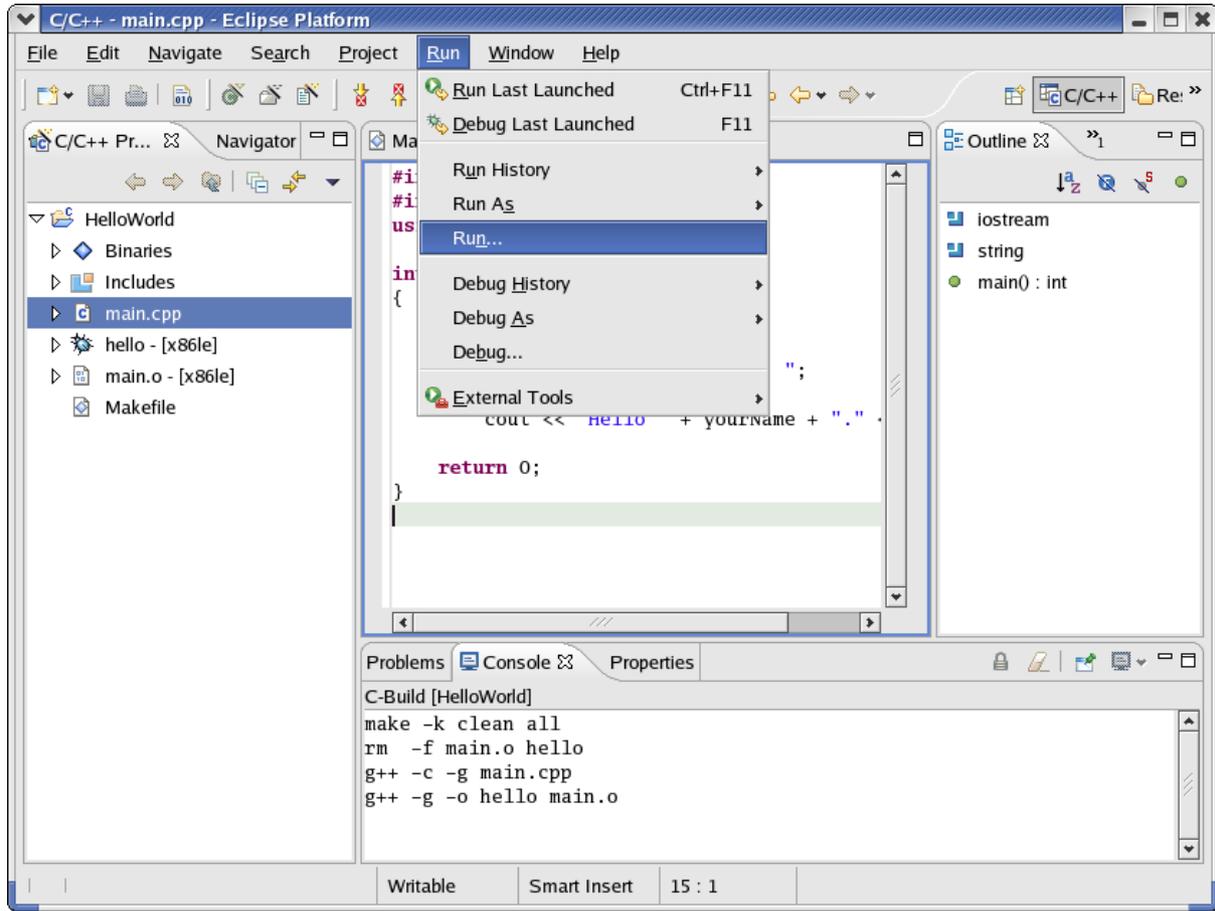
You can read through the build messages in the **Console** view, if there are any errors in the code you can review them in the **Problems** view.



Run the application

Navigate to the **C/C++ Projects** view, you will now see the executables listed.

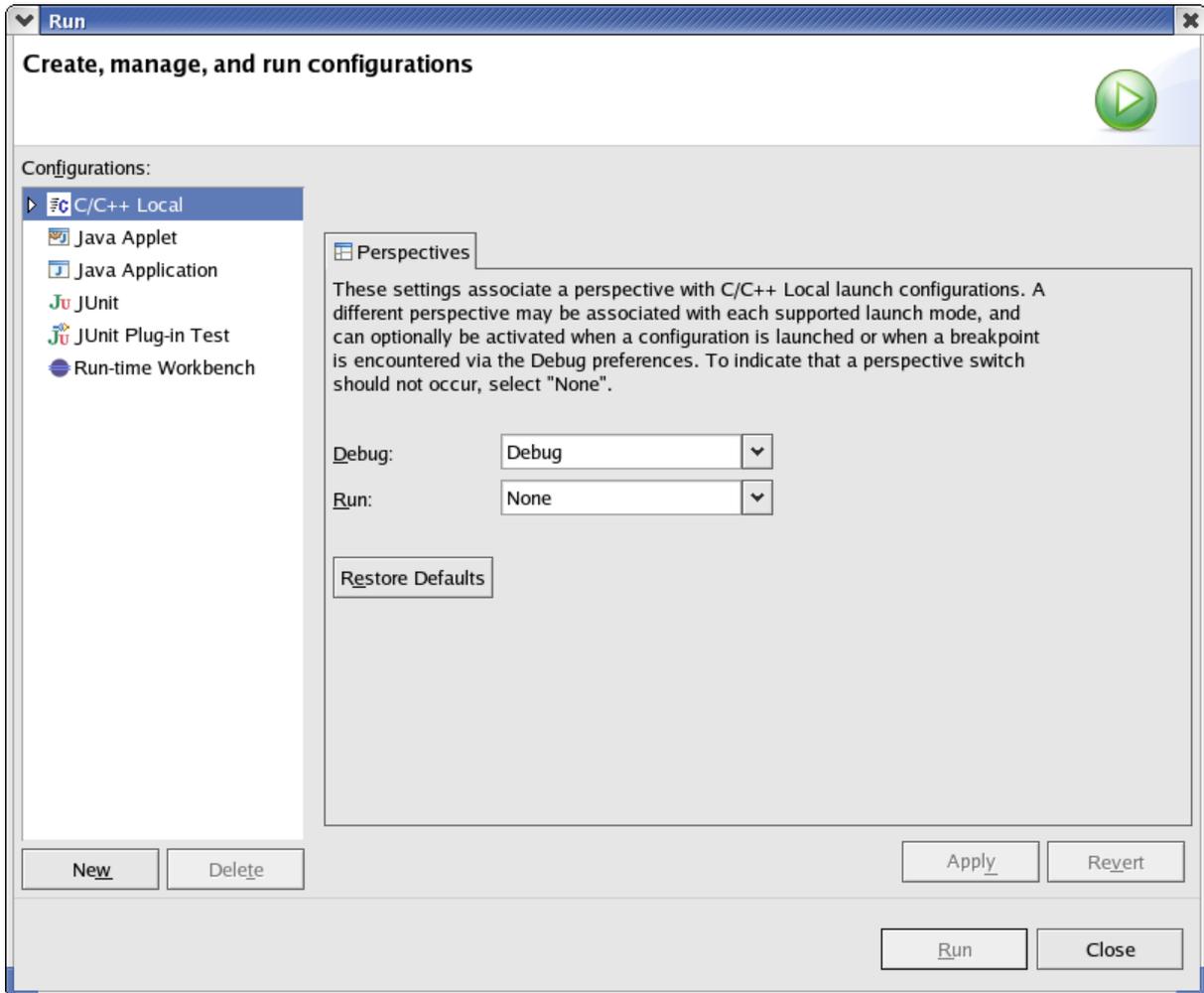
You can run your application within the **C/C++ Perspective**; to do so click **Run > Run**.



Run Configuration

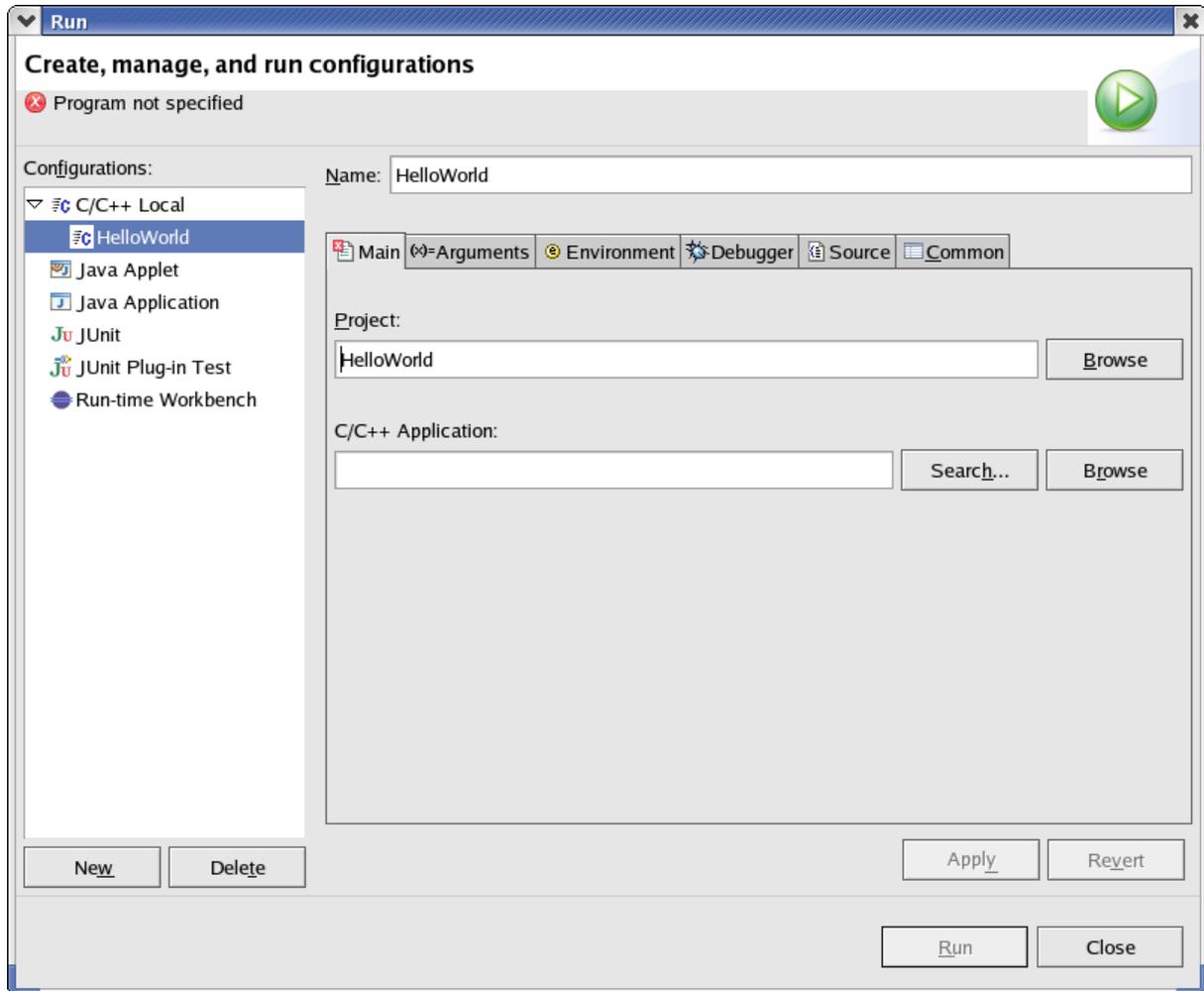
You should now see the **Run Configurations** dialog box. Navigate to the **Configurations** view, and then double-click **C/C++ Local**. This creates your Run Configuration. Select the new Run Configuration in the **Configurations** view.

Now click the **Main** tab and then the **Search** button.



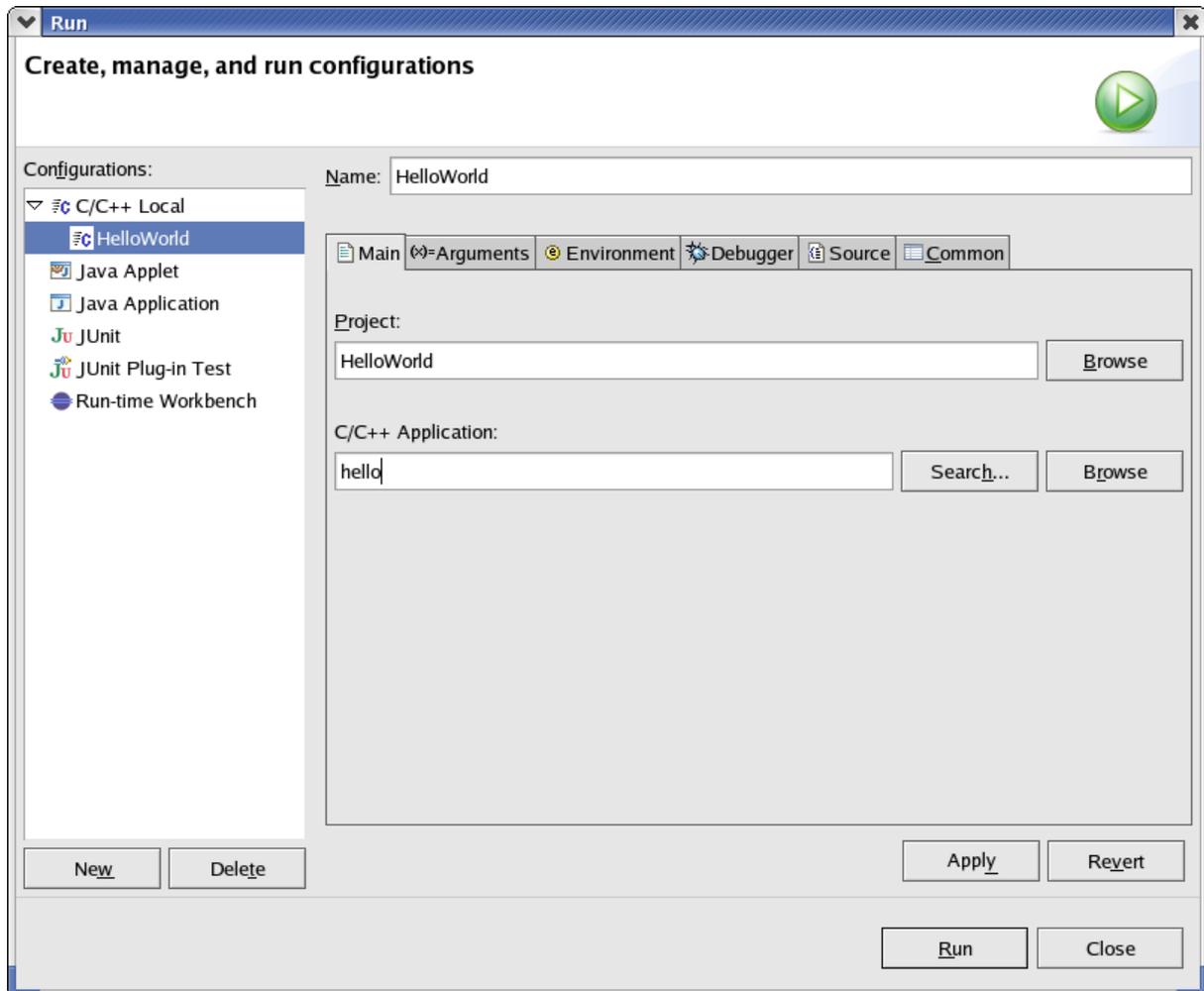
Run Configuration

Select the new Run Configuration in the **Configurations:** view and click the **Main** tab. This is where you select your C/C++ application and other settings that will impact how your application is run.



Program Selection

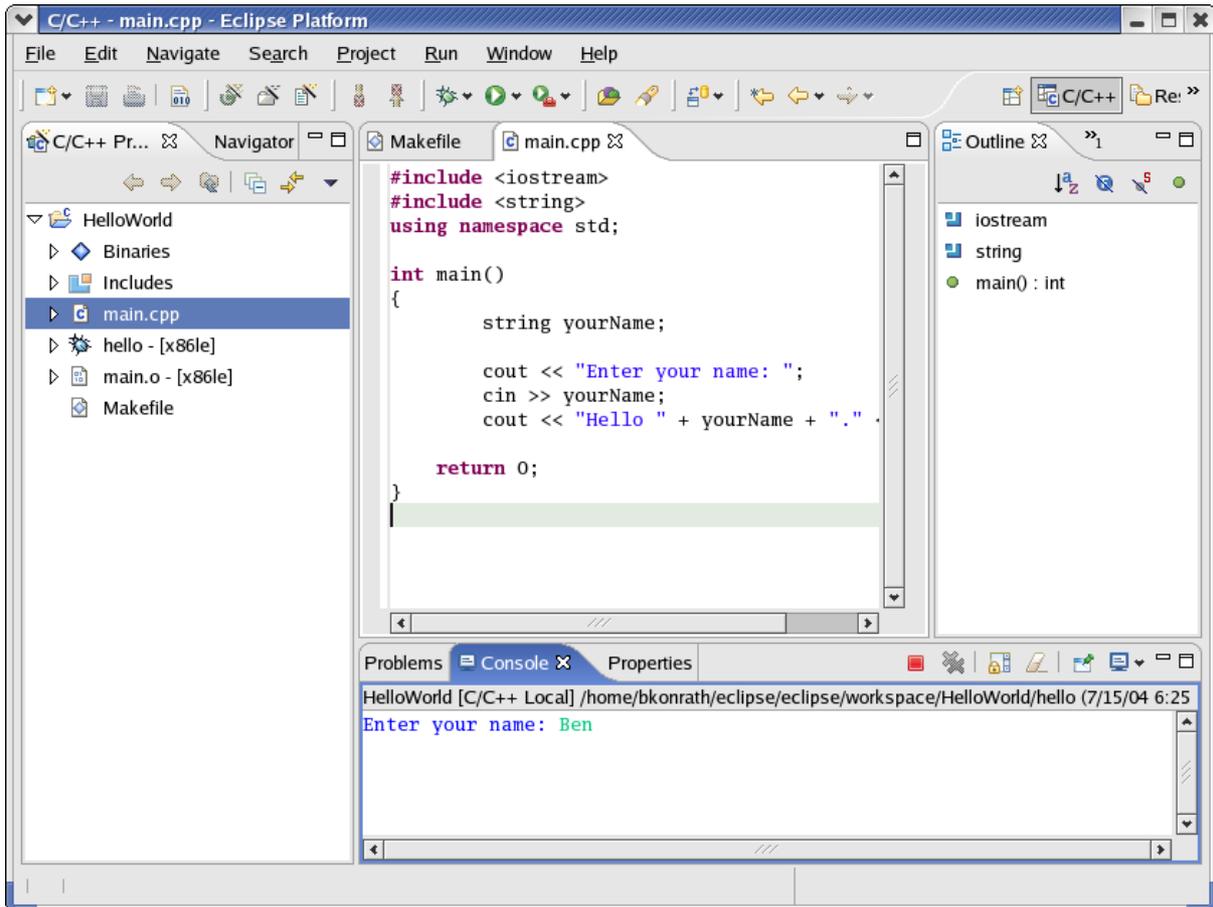
Type `hello` in the C/C++ **Application** field, click **Apply**, and then **Run**.



Console View

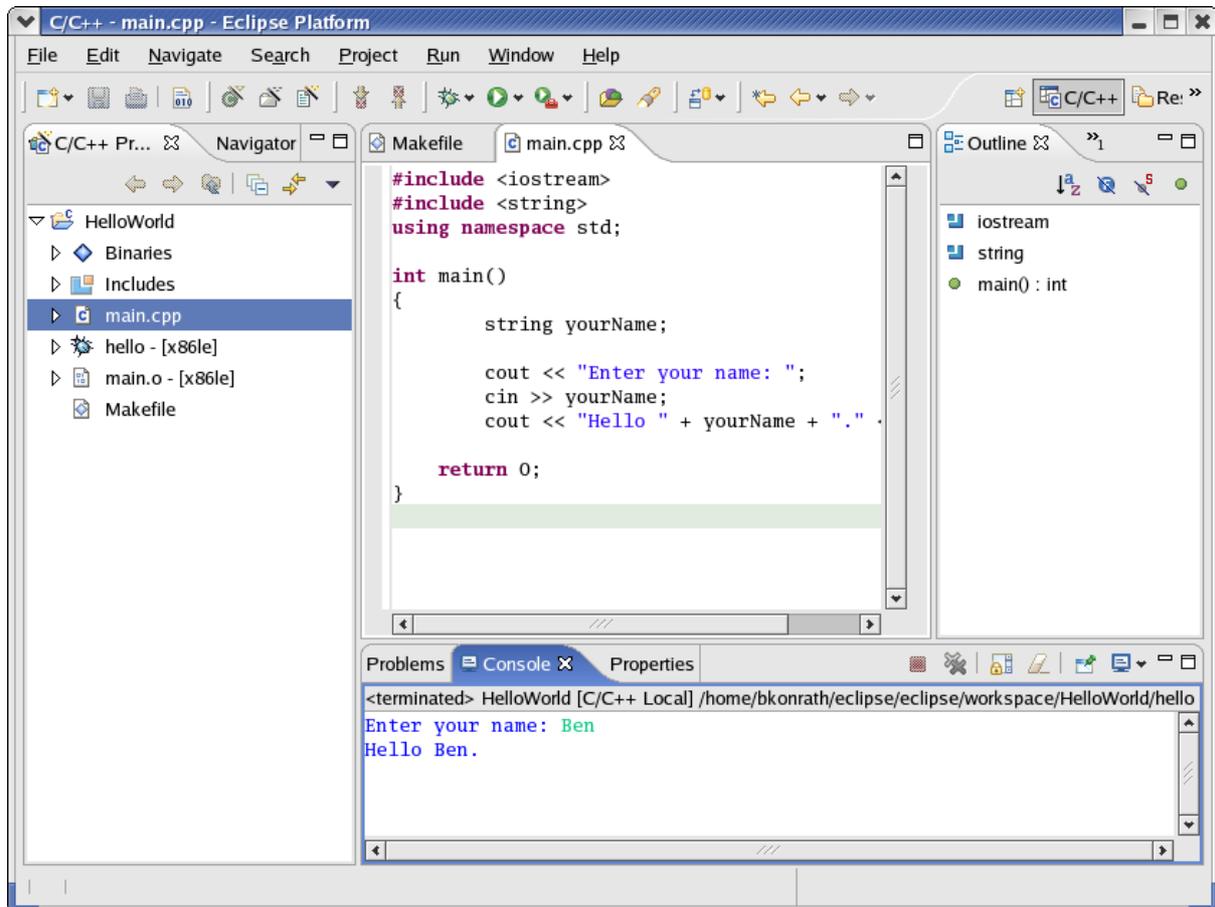
You should now see the application running in the **Console** view. The **Console** will also show which application is running in a title bar. You will notice that the view can be configured to display different elements (such as user input elements) in different colors.

Type in your name and press [Enter].



Run complete

The title bar in the **Console** view shows you when the program has terminated.



A Tour of the Java Development Toolkit (JDT)

This tutorial provides a step by step walk-through of the Java development tools.

Preparing the workbench

In this section, you will verify that the workbench is properly set up for Java development.

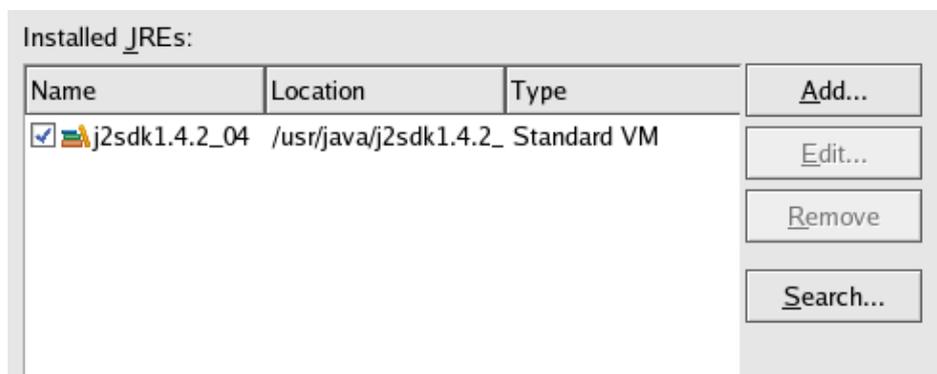
The following is assumed:

- You are starting with a new workbench installation with default settings.
- You are familiar with the basic workbench mechanisms, such as views and perspectives.

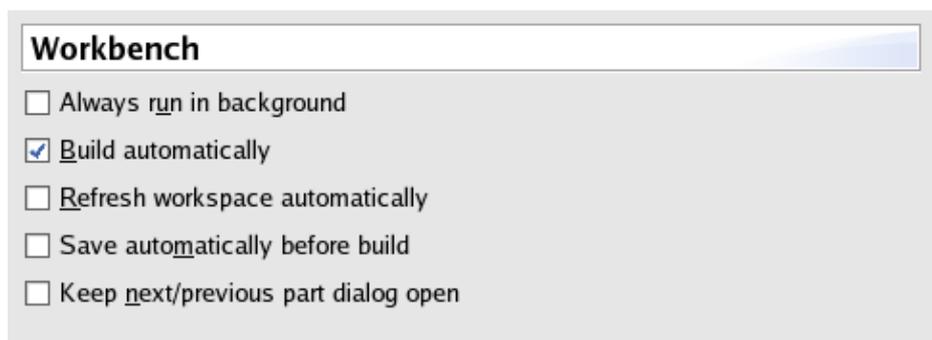
If you are not familiar with the basic workbench mechanisms, see the *Getting Started* section of the *Workbench User Guide*.

Verifying JRE installation and classpath variables

1. Click **Window > Preferences > Java > Installed JREs** to display the **Installed Java Runtime Environments** preferences page. Confirm that a JRE has been detected. By default, the JRE used to run the workbench is used to build and run Java programs. It should appear with a check mark in the list of installed JREs.

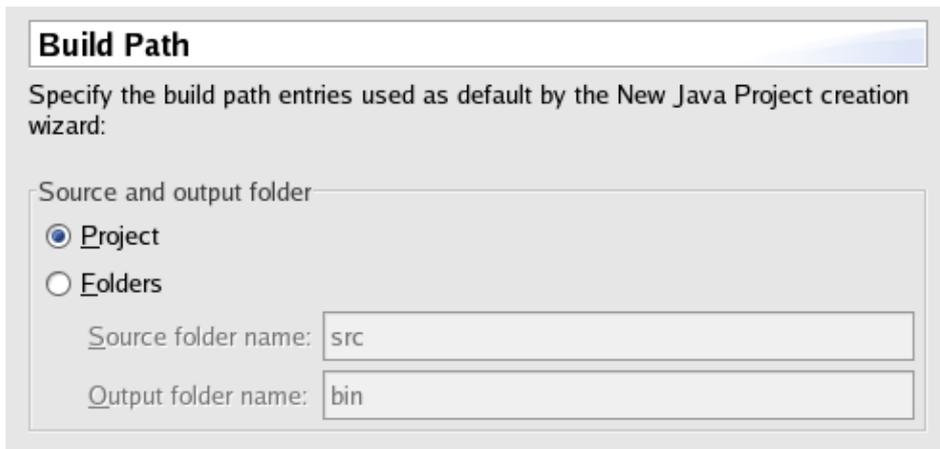


2. Select **Workbench** in the left pane to display the **Workbench** preference page. Confirm that the **Perform build automatically on resource modification** option is checked.

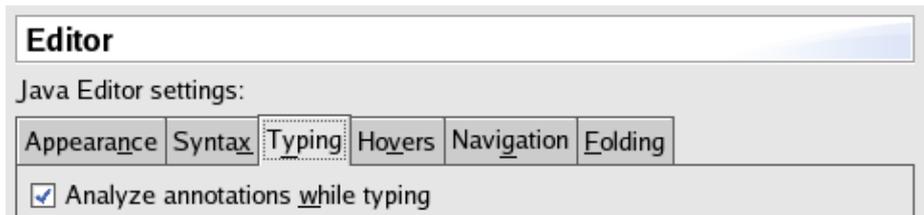


3. Select **Java > Build Path** in the left pane to go to the **Build Path** preference page. Confirm that As

source and output location use is set to *Project*.



4. Select **Java > Editor** in the left pane to go to the **Java Editor** preference page. On the preference page, click the Typing tab. Confirm that option Analyze annotations while typing is checked.



5. Click OK to save the preferences.

Note: The screenshots in this section exercise the advance highlighting feature in version 3.0. In order for your syntax highlighting to look exactly like the screenshots, make sure that Window > Preferences > Java > Editor > Syntax has Enable advance highlighting checked on.

Creating your first Java project

In this section, you will create a new Java project. You will be using JUnit as your example project. JUnit is an open-source unit-testing framework for Java.

Getting the Sample Code (JUnit)

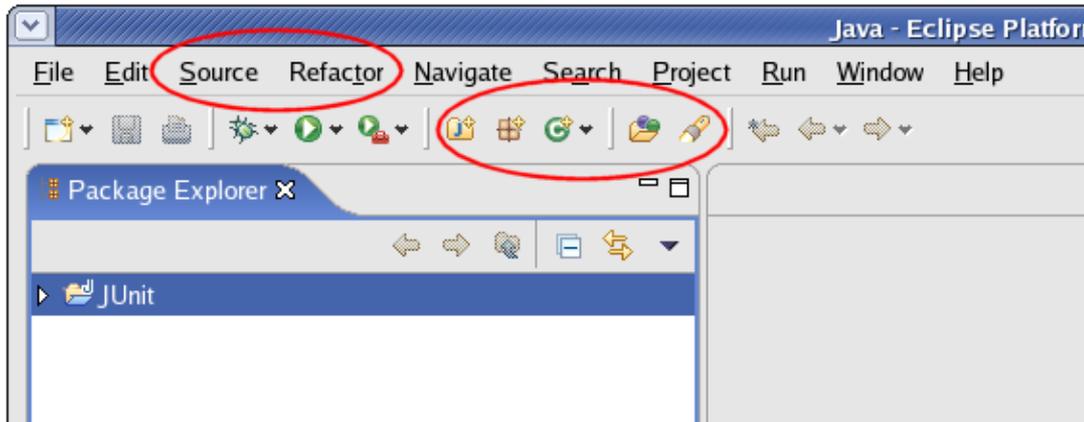
First you need to download the JUnit source code.

1. Go to the <http://www.eclipse.org/downloads/> page and locate the release that you are working with.
2. Scroll down to the *Example Plug-ins* section and download the examples archive.
3. Extract the contents of the Zip file to a directory from now on referenced as <ExamplesDownloadPath>.

Creating the project

1. Inside Eclipse, select the menu item **File > New > Project** to open the **New Project** wizard.

2. Select **Java Project**, then click **Next**. On the next page, type "JUnit" in the **Project name** field and click **Finish**. A Java perspective opens inside the workbench with the new Java project in the **Package Explorer**. When the Java perspective is active, new menu options and Java-specific buttons are loaded in the workbench toolbar. Depending on which view or editor is active, other buttons and menu options will be available.

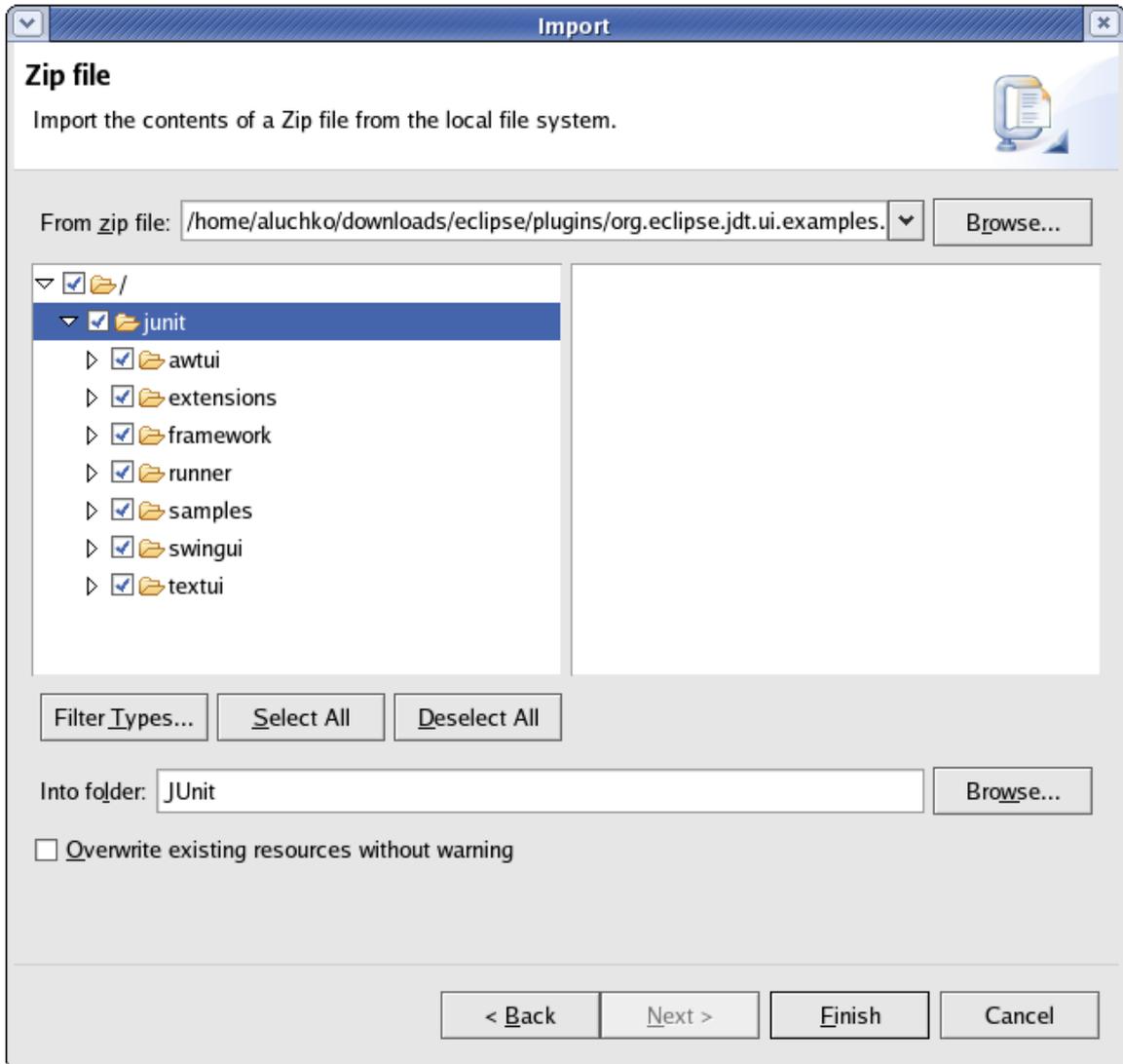


3. In the **Package Explorer**, make sure that the **JUnit** project is selected. Select the menu item **File > Import**
4. Select **Zip file**, then click **Next**.
5. Click the **Browse** button next to the **Zip file** field, browse to the `ExamplesDownloadPath` directory, and select `eclipse/plugins/org.eclipse.jdt.ui.examples.projects_3.0.0/archive/junit/junit381src.jar`.

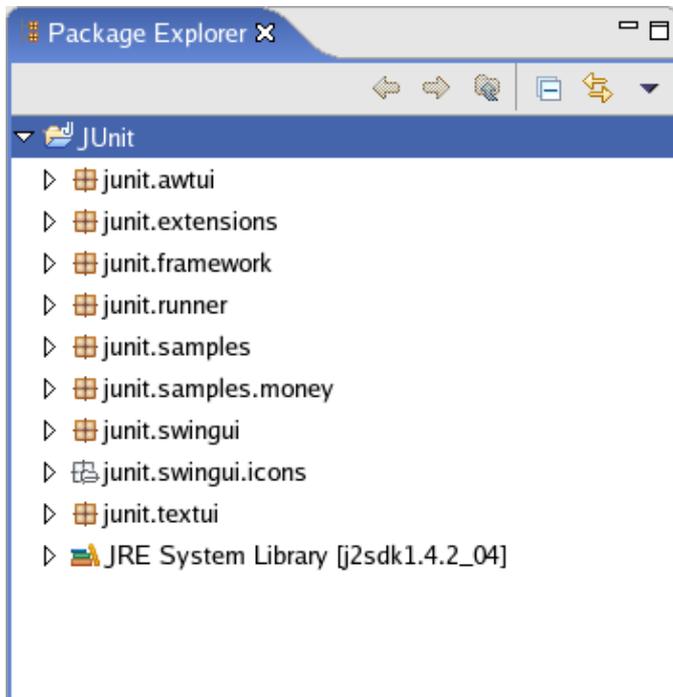
Note: This step assumes that you followed steps 1–3 in the **Getting the Sample Code** section above.

6. In the **Import** wizard, below the hierarchy list, click **Select All**. You can expand and select elements within the `junit` directory on the left pane to view the individual resources that you are importing on the right pane.

Note: Do not deselect any of the resources in the `junit` directory at this time; you will need all of these resources in the tutorial.



7. Make sure that the JUnit project appears in the destination **Folder** field, then click **Finish**. In the import progress indicator, notice that the imported resources are compiled as they are imported into the workbench. This is because the **Build automatically** option is checked on the Workbench preferences page. You are prompted to overwrite the `.classpath` and `.project` files in the JUnit project. This is because the `.classpath` resource was created for you when you created the JUnit project. It is safe to overwrite these files.
8. In the **Package Explorer** view, expand the **JUnit** project to view the JUnit packages.



Browsing Java elements using the Package Explorer

In this section, you will browse Java elements within the JUnit project.

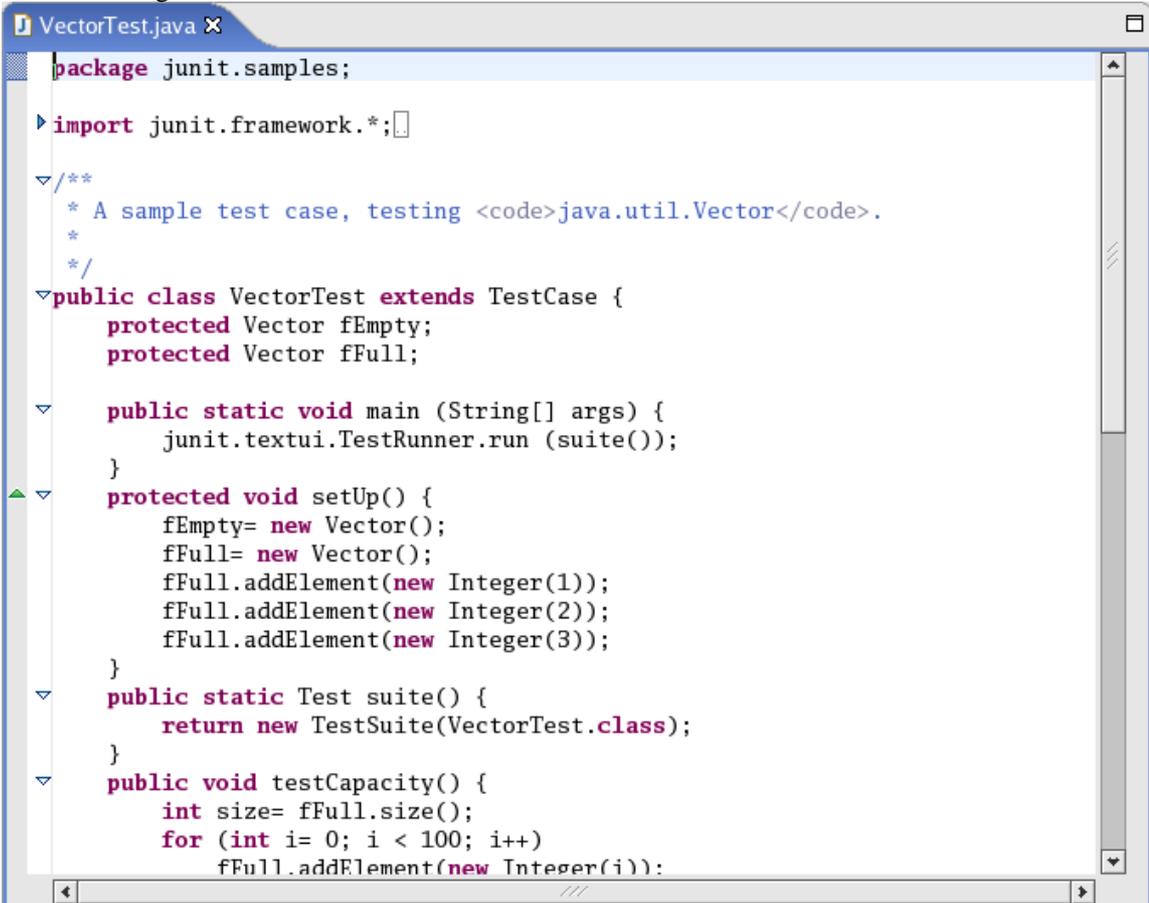
1. In the **Package Explorer** view, expand the JUnit project to see its packages.
2. Expand the package `junit.framework` to see the Java files contained in the package.
3. Expand the Java file `TestCase.java`. Note that the **Package Explorer** shows Java-specific sub-elements of the source code file. The import declaration, the public type, and its members (fields and methods) appear in the tree.



Opening a Java editor

In this section, you will learn how to open an editor for Java files. You will also learn about some of the basic Java editor features.

1. Expand the package `junit.samples` and select the file `VectorTest.java`. Open `VectorTest.java` by double-clicking on it. In general you can open a Java editor for Java files, types, methods, and fields by simply double-clicking on them. For example, to open the editor directly on the method `testClone` defined in `VectorTest.java`, double-click on the method in the **Package Explorer**.
2. Notice the syntax highlighting. Different kinds of elements in the java source are rendered in unique colors. Examples of java source elements that are rendered differently are:
 - ◆ Regular comments
 - ◆ Javadoc comments
 - ◆ Keywords
 - ◆ Strings.



```

package junit.samples;

import junit.framework.*;

/**
 * A sample test case, testing <code>java.util.Vector</code>.
 *
 */
public class VectorTest extends TestCase {
    protected Vector fEmpty;
    protected Vector fFull;

    public static void main (String[] args) {
        junit.textui.TestRunner.run (suite());
    }

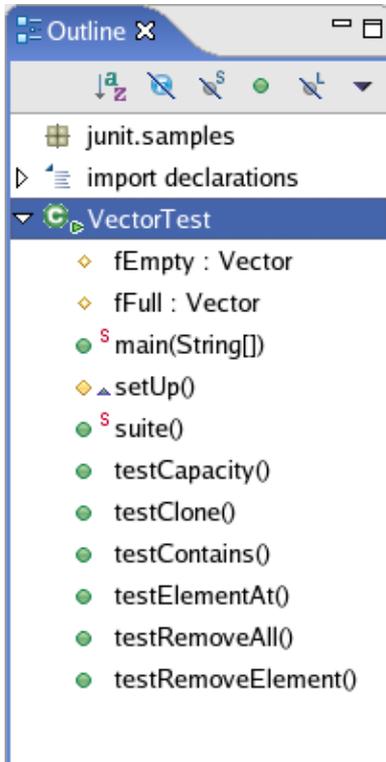
    protected void setUp() {
        fEmpty= new Vector();
        fFull= new Vector();
        fFull.addElement(new Integer(1));
        fFull.addElement(new Integer(2));
        fFull.addElement(new Integer(3));
    }

    public static Test suite() {
        return new TestSuite(VectorTest.class);
    }

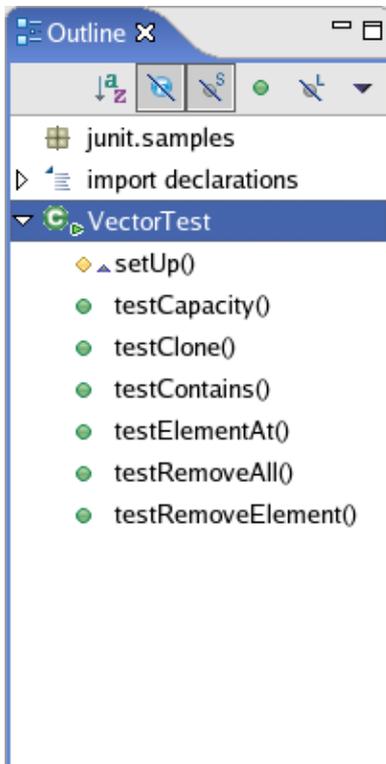
    public void testCapacity() {
        int size= fFull.size();
        for (int i= 0; i < 100; i++)
            fFull.addElement(new Integer(i));
    }
}

```

3. Look at the Outline view. It displays an outline of the Java file including the package declaration, import declarations, fields, types, and methods. The **Outline** view uses icons to annotate Java elements. For example, icons indicate whether a Java element is static, abstract, or final. Different icons show you whether a method is overridden from a base class (⊖) or when it implements a method from an interface (⊕).

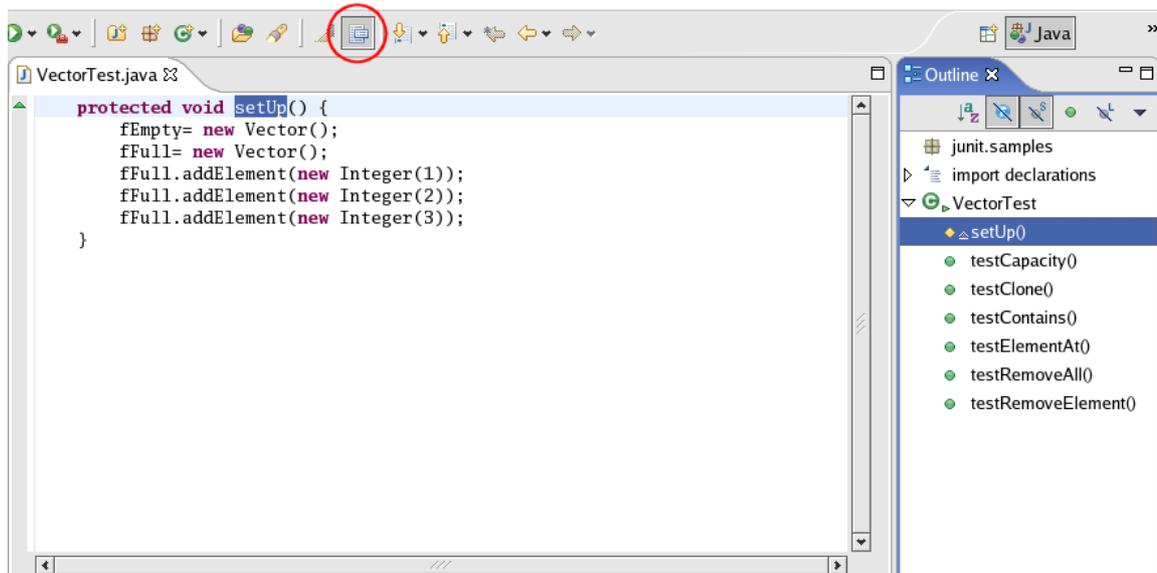


4. Toggle the **Hide Fields**, **Hide Static Members**, and **Hide Non-Public Members** buttons in the **Outline** view toolbar to filter the view's display. Before going to the next step, make sure that the **Hide Non-Public Members** button is not pressed.



5. You can edit source code by viewing the whole Java file, or you can narrow the view to a single Java element. The toolbar includes a button, **Show Source of Selected Element Only**, that causes only the

source code of the selected outline element to be displayed in the Java editor. In the example below, only the `setUp()` method is displayed.



6. Click **Show Source of Selected Element Only** again to see the whole Java file again. In the **Outline** view, select different elements and note that they are again displayed in a whole file view in the editor. The **Outline** view selection now contains a range indicator on the vertical ruler on the left border of the Java editor that indicates the range of the selected element.



Adding new methods

1. Start adding a method by typing the following at the end of the `VectorTest.java` file (but before the closing brackets of the type) in the Java editor:

```
public void testSizeIsThree()
```

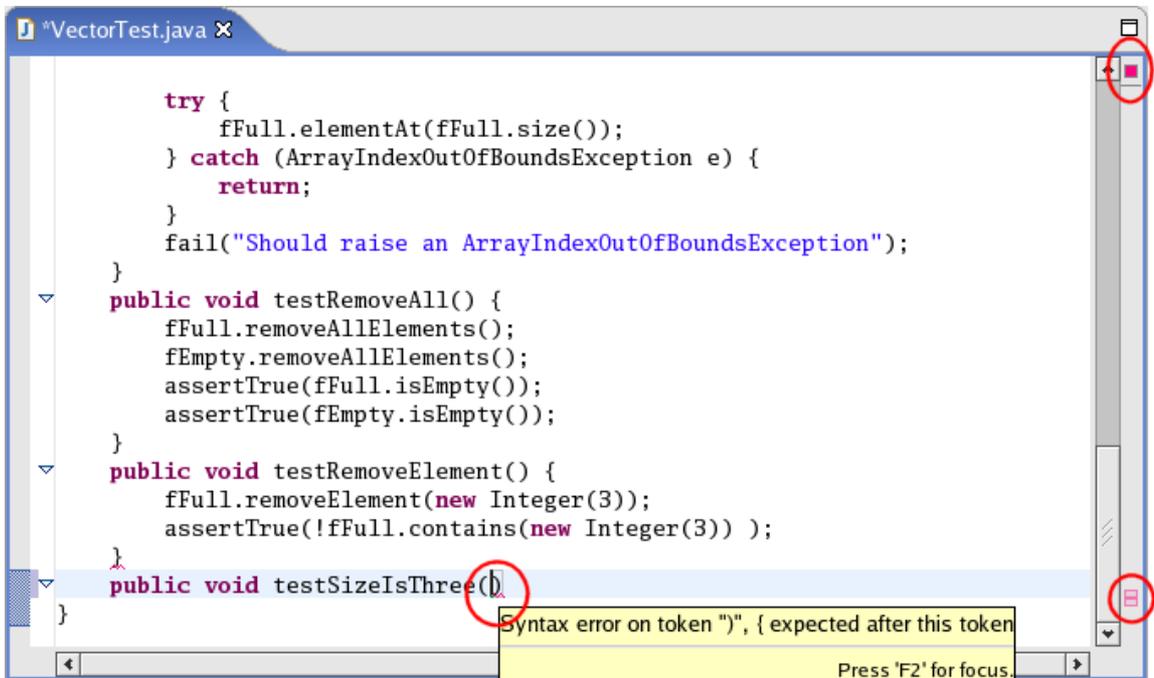
As soon as you type the method name in the editor area, the new method appears at the bottom of the **Outline** view.



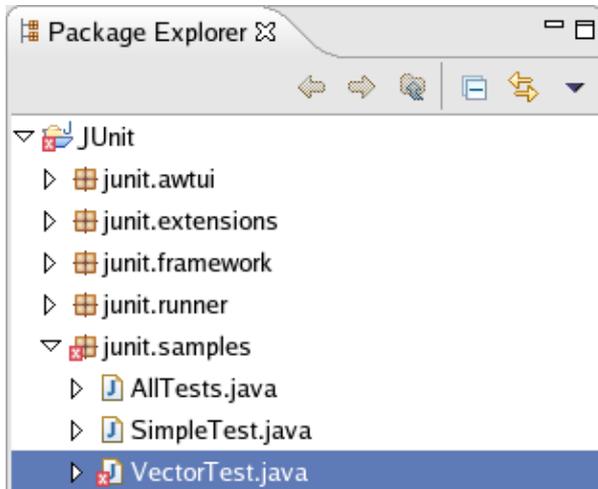
In addition, an error annotation (red square) appears in the overview ruler on the right hand side of the editor. This error annotation indicates that the compilation unit is currently not correct. If you hover over the red square, a tool tip appears:

Unmatched bracket; Syntax error on token ")", "{" expected

This is correct because the method does not have a body yet. Note that error annotations in the editor's rulers are updated as you type. You can control this behavior from the **Window > Preferences > Java > Editor** preference page, under the *Typing* tab, check *Analyze annotations while typing*.



2. Click **Save**. The compilation unit is compiled automatically and errors appear in the **Package Explorer** view, in the **Problems** view, and on the vertical ruler on the left hand side of the editor. In the **Package Explorer** view, the errors are propagated up to the project of the compilation unit containing the error.



3. Complete the new method by typing the following:

```
{
    assertTrue(fFull.size() == 3);
}
```

Note that the closing curly bracket has been auto-inserted.

4. Save the file. Notice that the error indicators disappear because the missing bracket has been added.

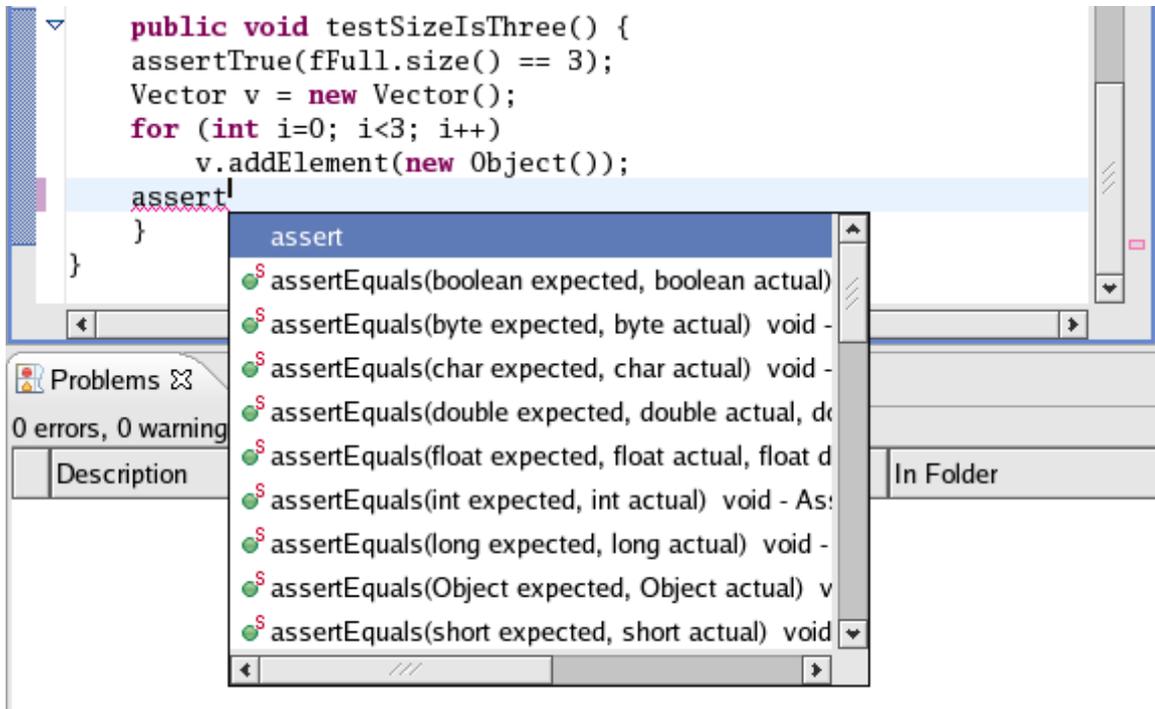
Using content assist

In this section you will use *content assist* to finish writing a new method. Open the `junit.samples.VectorTest.java` file in the Java editor (if you do not already have it open) and select the `testSizeIsThree()` method in the **Outline** view. If the file does not contain such a method, see [Adding new methods](#) for instructions on how to add this method.

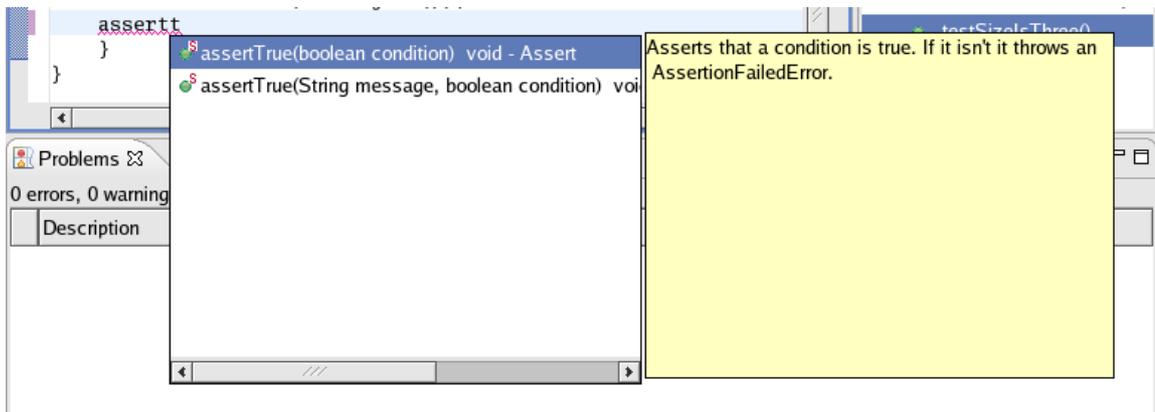
1. Add the following lines to the end of the method:

```
Vector v = new Vector();
for (int i=0; i<3; i++)
    v.addElement(new Object());
assert
```

2. With your cursor at the end of the word `assert`, press [Ctrl]+[Space] to activate content assist. The content assist window appears with a list of proposals. Scroll the list to see the available choices.



- With the content assist window still active, type the letter 't' in the source code after `assert` (with no space in between). The list is narrowed and shows only entries starting with 'assert'. Select and then hover over various items in the list to view any available Javadoc help for each item.



- Select `assertTrue(boolean)` from the list and press [Enter]. The code for the `assertTrue(boolean)` method is inserted.
- Complete the line so that it reads as follows:

```
assertTrue(v.size() == fFull.size());
```

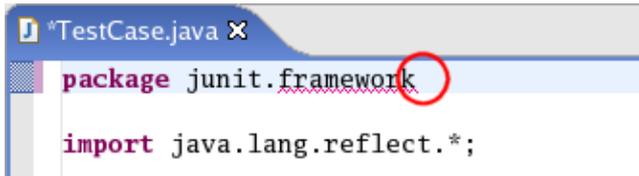
- Save the file.

Identifying problems in your code

In this section, you will review the different indicators for identifying problems in your code.

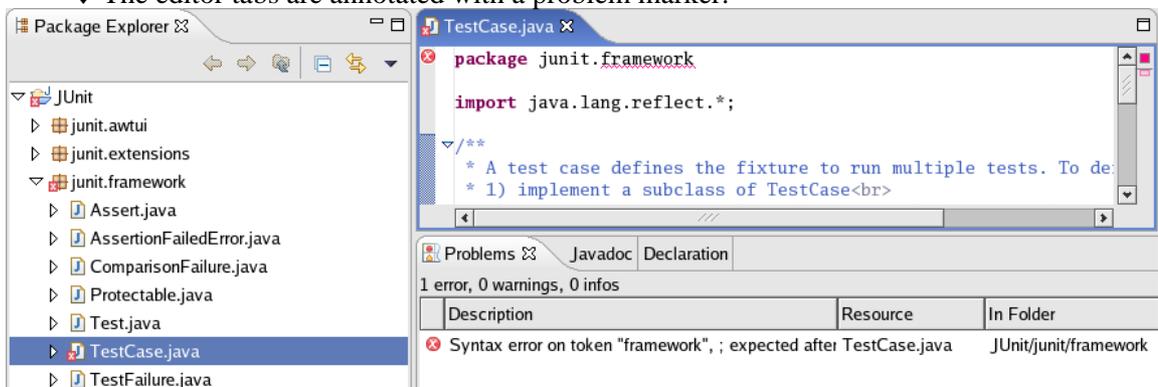
Build problems are displayed in the **Problems** view and marked in the vertical ruler of your source code.

1. Open `junit.framework.TestCase.java` from the **Package Explorer** view.
2. Add a syntax error by deleting the semicolon at the end of the package declaration in the source code.

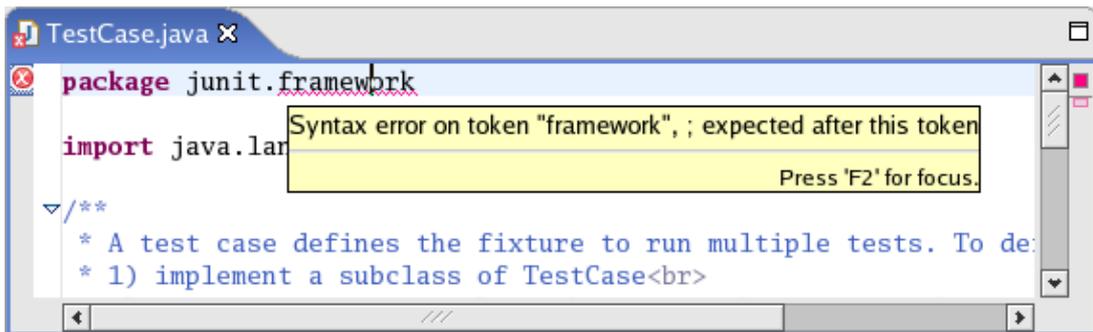


```
package junit.framework
import java.lang.reflect.*;
```

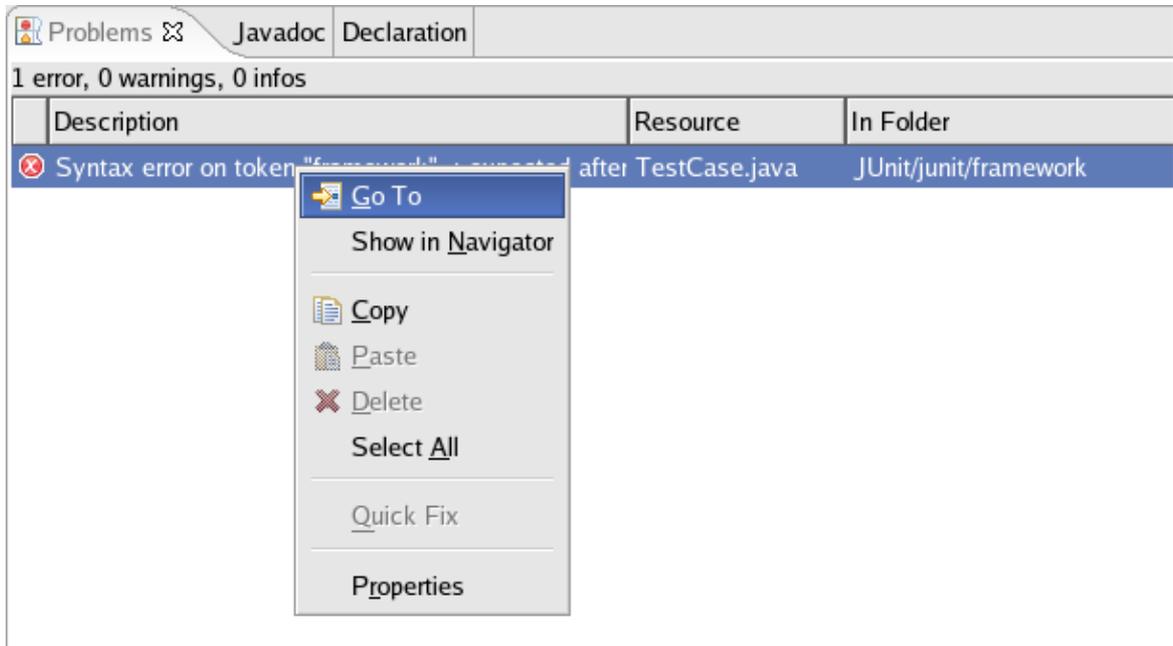
3. Click the **Save** button. The project is rebuilt and the problem is indicated in several ways:
 - ◆ In the **Problems** view, the problems are listed
 - ◆ In the **Package Explorer** view, the **Type Hierarchy**, and the **Outline** view, problem ticks appear on the affected Java elements and their parent elements
 - ◆ In the editor's vertical ruler, a problem marker is displayed near the affected line
 - ◆ Squiggly lines appear under the word that might cause the error
 - ◆ The editor tabs are annotated with a problem marker.



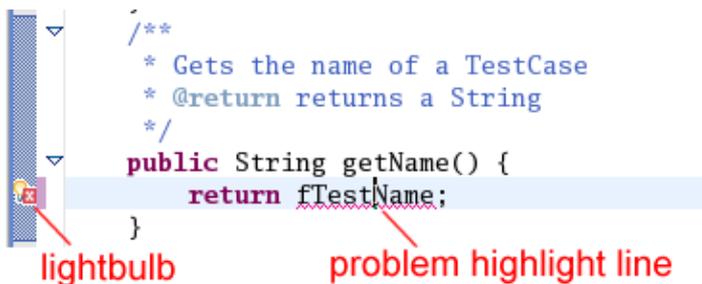
4. You can hover over the problem marker in the vertical ruler to view a description of the problem.



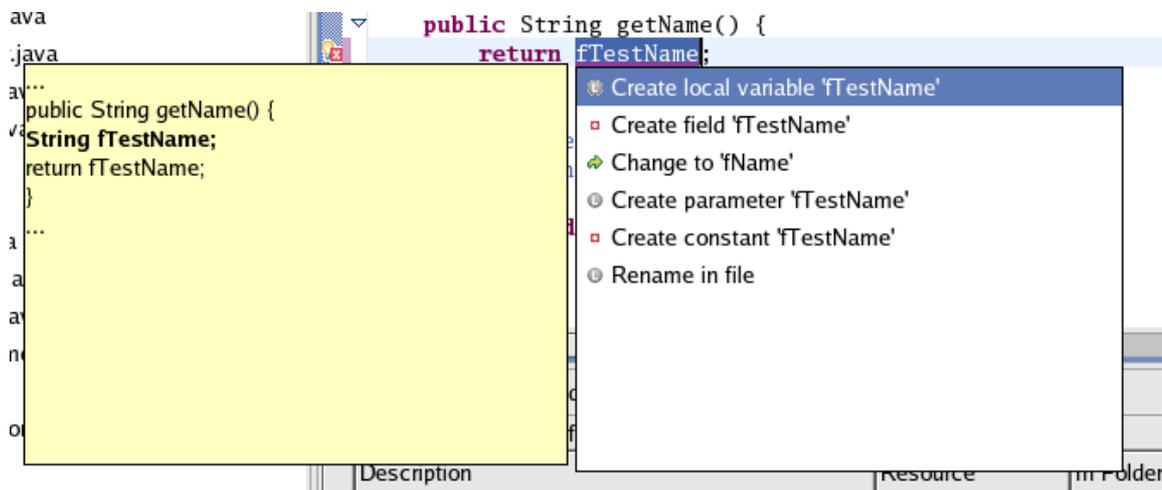
5. Click **Close** on the editor's tab to close the editor.
6. In the **Problems** view, select a problem in the list, right-click, and select **Go To** from its context menu. The file is opened in the editor at the location of the problem.



7. Correct the problem in the editor by adding the semicolon, then click *Save*. The project is rebuilt and the problem indicators disappear.
8. In the **Outline** view, select the method `getName()`. The editor scrolls to this method.
9. On the first line of the method, change the returned variable `fName` to `fTestName`. While you type, a problem highlight underline appears on `fTestName`, to indicate a problem. Hovering over the highlighted problem displays a description of the problem.
10. On the marker bar, a light bulb marker appears. The light bulb signals that correction proposals are available for this problem.



11. Set the cursor inside the marked range and choose **Quick Fix** from the Edit menu bar. (Alternatively, you can press [Ctrl]+[1] or left-click the light bulb. A selection dialog appears with possible corrections.



12. Select **Change to fName** to fix the problem. The problem highlight line disappears as the correction is applied.
13. Close the file without saving.

You can configure how problems are indicated on the **Window > Preferences > Workbench > Editors > Annotations** page.

Using source code templates

In this section you will use content assist to fill in a template for a common loop structure. Open the `junit.samples.VectorTest.java` file in the Java editor if you do not already have it open.

1. Start adding a new method by typing the following:

```
public void testValues() {
    Integer[] expected= new Integer[3];
    for
```

2. With the cursor at the end of `for`, press `[Ctrl]+[Space]>` to enable content assist. You will see a list of common templates for "for" loops. When you hover over a template, you see the code for the template in its help message. Note that the local array name is guessed automatically.
3. Choose the `for - iterate over array entry` and press `[Enter]` to confirm the template. The template is inserted in your source code.
4. Next, change the name of the index variable from `i` to `e`. To do so, simply press `e`, as the index variable is automatically selected. Observe that the name of the index variable changes at all places. When inserting a template, all template variables with the same name are connected to each other. Thus, changing one changes all the other values as well.
5. Pressing the `[Tab]` key moves the cursor to the next variable of the code template. This is the array `expected`.

```

public void testValue(){
    Integer[] expected = new Integer[3];
    for (int e = 0; e < expected.length; e++) {
        expected
    }
}

```

As you do not want to change the name (it was guessed right by the template), press [Tab] again, which leaves the template as there are not any variables left to edit.

6. Complete the for loop as follows:

```

for (int e = 0; e < expected.length; e++) {
    expected[e] = new Integer(e + 1);
}
Integer[] actual = to

```

7. With the cursor at the end of `to`, press [Ctrl]+[Space] to enable content assist. Pick `toArray` – convert collection to array and press [Enter] to confirm the selection (or double-click the selection).

The template is inserted in the editor and `type` is highlighted and selected.

```

public void testValue(){
    Integer[] expected = new Integer[3];
    for (int e = 0; e < expected.length; e++) {
        expected[e] = new Integer(e + 1);
    }
    Integer[] actual = (type[]) collection.toArray(new type[collection.size()])
}

```

8. Overwrite the selection by typing `Integer`. The type of array constructor changes when you change the selection.
9. Press [Tab] to move the selection to `collection` and overwrite it by typing `fFull`.

```

public void testValue(){
    Integer[] expected = new Integer[3];
    for (int e = 0; e < expected.length; e++) {
        expected[e] = new Integer(e + 1);
    }
    Integer[] actual = (Integer[]) fFull.toArray(new Integer[fFull.size()])
}

```

10. Add the following lines of code to complete the method:

```

assertEquals(expected.length, actual.length);
for (int i = 0; i < actual.length; i++)
    assertEquals(expected[i], actual[i]);
}

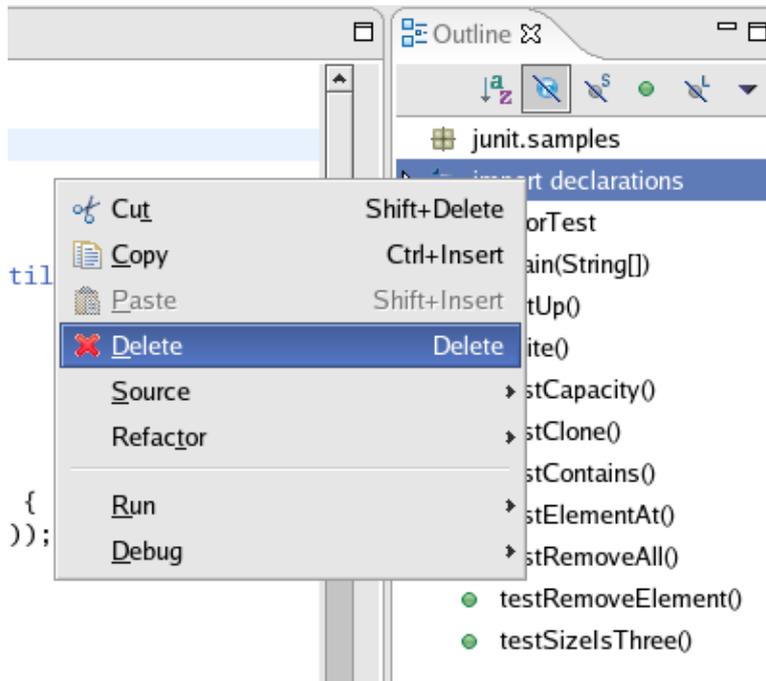
```

11. Save the file.

Organizing import statements

In this section you will organize the import declarations in your source code. If you do not already have `junit.samples.VectorTest.java` open in the Java editor, open it now.

1. Delete the import declarations by selecting them in the **Outline** view, right-clicking, and selecting **Delete** from the context menu. Confirm the resulting dialog with **Yes**. You will see numerous compiler warnings in the vertical ruler since the types used in the method are no longer imported.



2. Right-click in the editor and select **Source** > **Organize Imports** from the context menu. The required import statements are added to the beginning of your code below the package declaration.

```

package junit.samples;

import java.util.Vector;

import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

/**
 * A sample test case, testing <code>java.util.Vector</code>.
 *
 */
public class VectorTest extends TestCase {
    protected Vector fEmpty;
    protected Vector fFull;

    public static void main (String[] args) {
        junit.textui.TestRunner.run (suite());
    }
}

```

You can also choose **Organize Imports** from the context menu of the import declarations in the **Outline** view.

Note: You can specify the order of the import declarations in the **Window > Preferences > Java > Code Style > Organize Imports** preferences dialog.

3. Save the file.

Using the local history

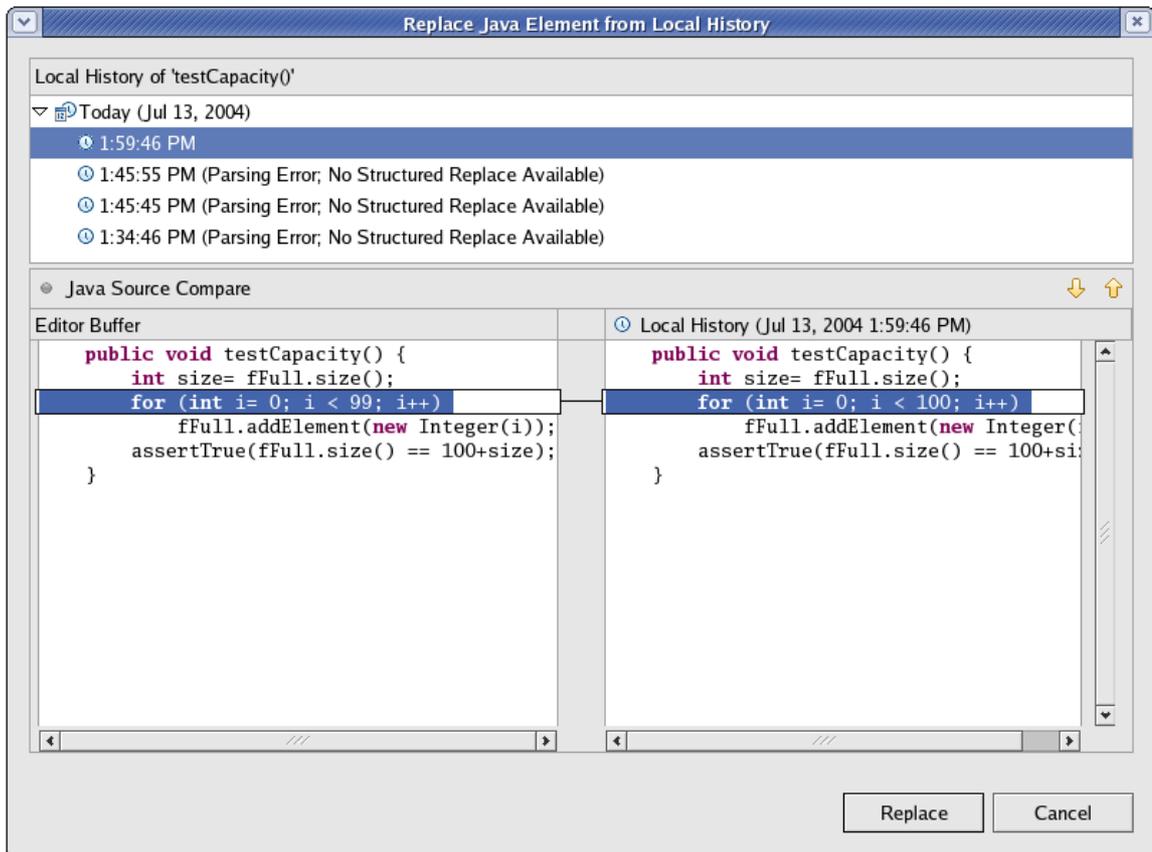
In this section, you will use the local history feature to switch to a previously saved version of an individual Java element.

1. Open the `junit.samples.VectorTest.java` file in the Java editor and select the method `testCapacity()` in the **Outline** view.
2. Change the content of the method so that the 'for' statements reads as:

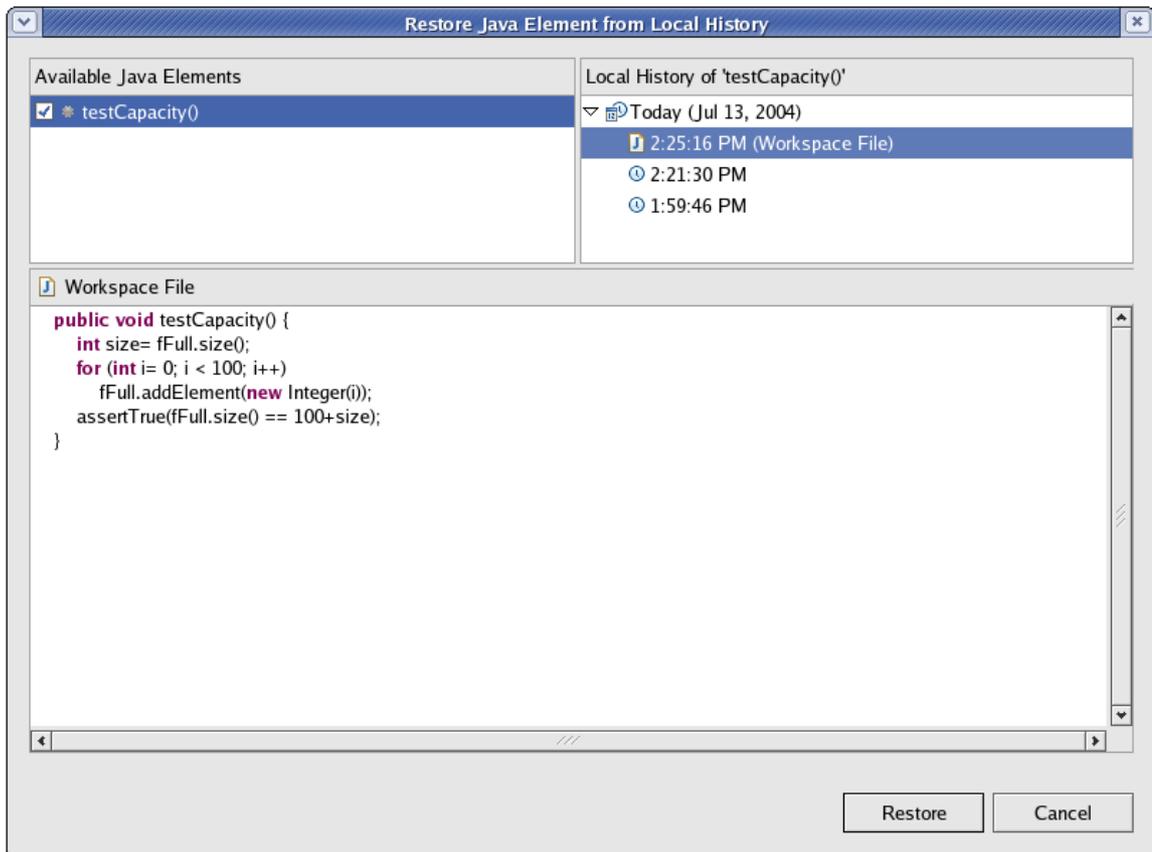
```
for (int i= 0; i < 99; i++)
```

Save the file by pressing [Ctrl]+[S].

3. In the **Outline** view, select the method `testCapacity()`, right-click, and select **Replace With > Element from Local History** from its context menu.
4. In the **Replace Java Element from Local History** dialog, the Local History list shows the various saved states of the method. The Java Source Compare pane shows details of the differences between the selected history resource and the existing workbench resource.



5. In the **Local History** pane, select the method that you changed, then click **Replace**. In the Java editor, the method is replaced with the selected history version.
6. Save the file.
7. Besides replacing a method's version with a previous one, you can also restore Java elements that were deleted. Again, select the method `testCapacity()` in the **Outline** view, right-click, and select **Delete** from its context menu. Confirm the resulting dialog with **Yes** and save the file.
8. Now select the type `VectorTest` in the **Outline** view, right-click, and select **Restore from Local History...** from its context menu. Select and check the method `testCapacity()` in the **Available Java Elements** pane. As before, the **Local History** pane shows the versions saved in the local history. The **Java Source Compare** pane shows the details of the differences between the selected history version and the existing workbench resource.



9. In the **Local History** pane, select the earlier version and then click **Restore**.
10. Press [Ctrl]+[S] to save the file.

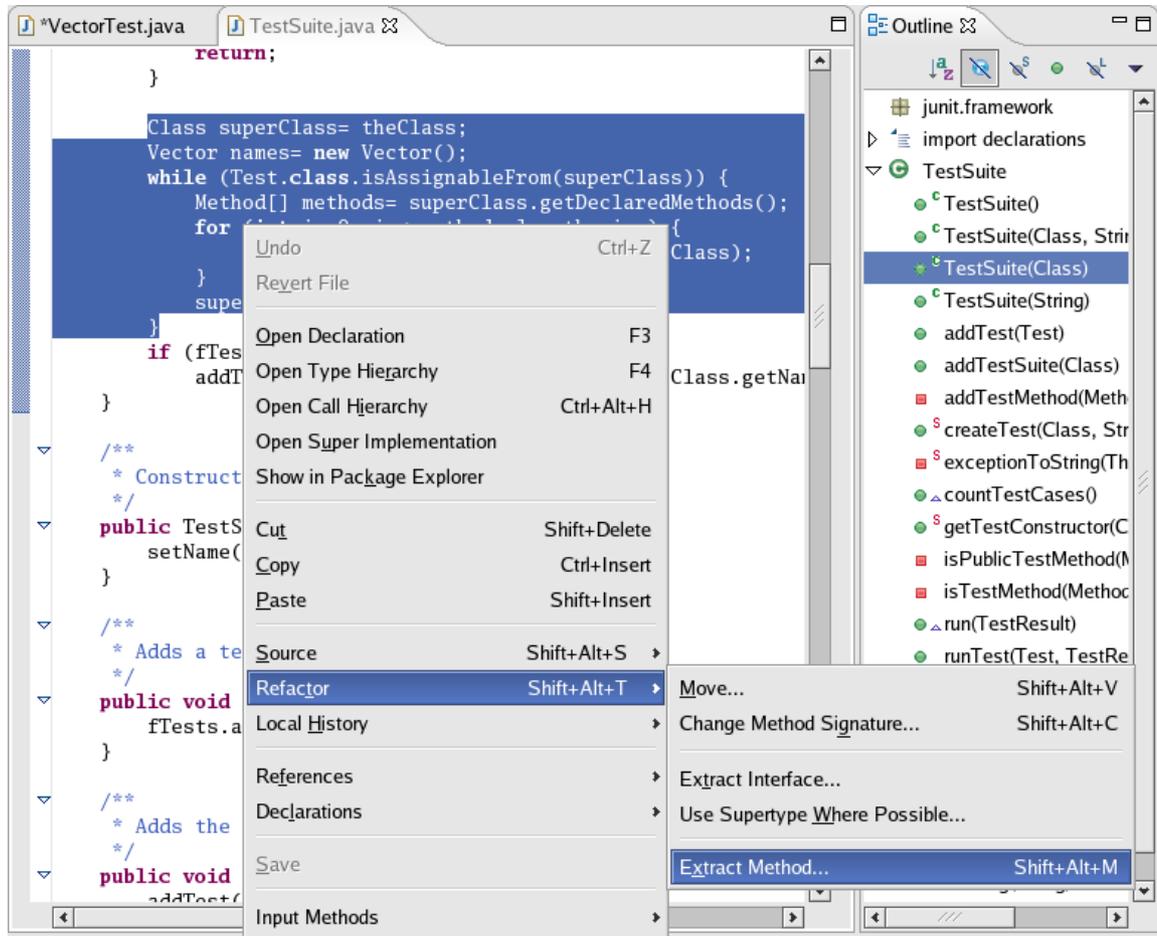
Extract a new method

In this section, you will improve the code of the constructor of `junit.framework.TestSuite`. To make the intent of the code clearer, you will extract the code that collects test cases from base classes into a new method called `collectTestMethods`.

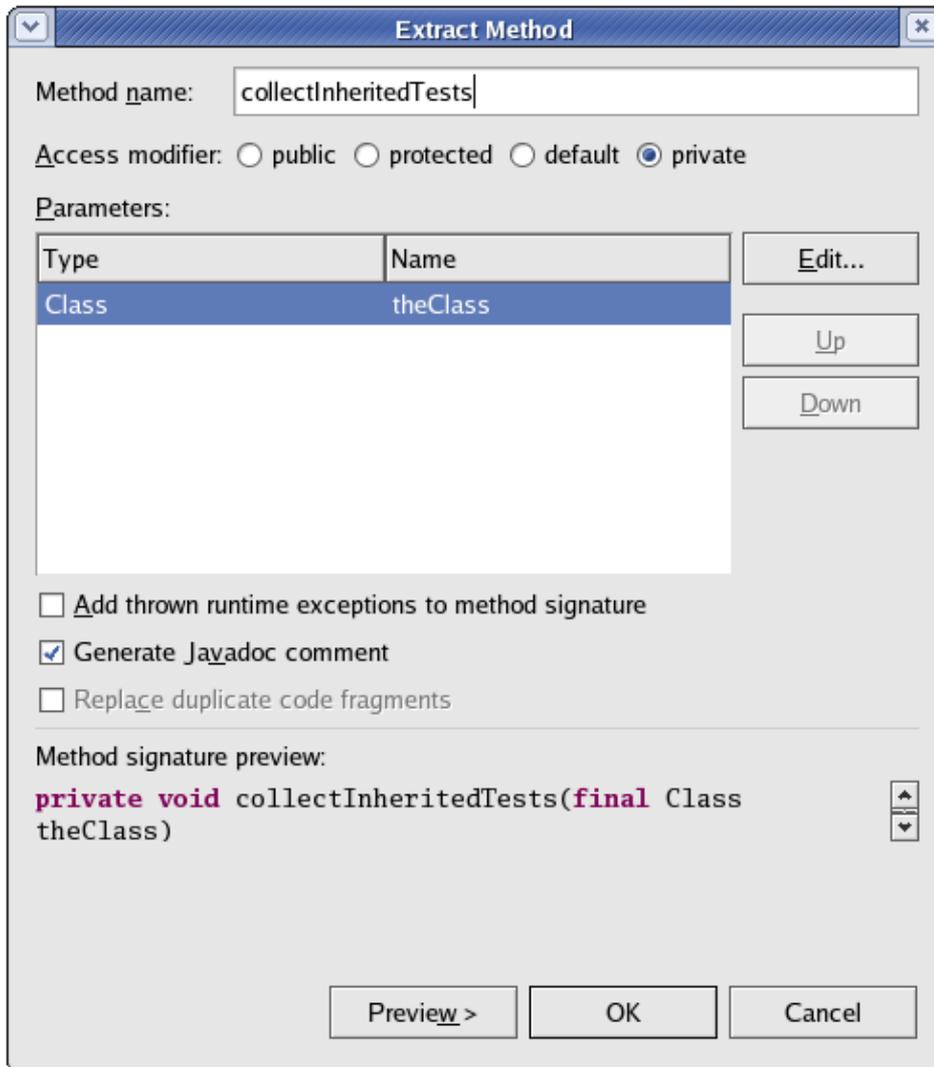
1. In the `junit.framework.TestSuite.java` file, select the following range of code inside the `TestSuite(Class)` constructor:

```
Class superClass= theClass;
Vector names= new Vector();
while(Test.class.isAssignableFrom(superClass)) {
    Method[] methods= superClass.getDeclaredMethods();
    for (int i= 0; i < methods.length; i++) {
        addTestMethod(methods[i],names, constructor);
    }
    superClass= superClass.getSuperclass();
}
}
```

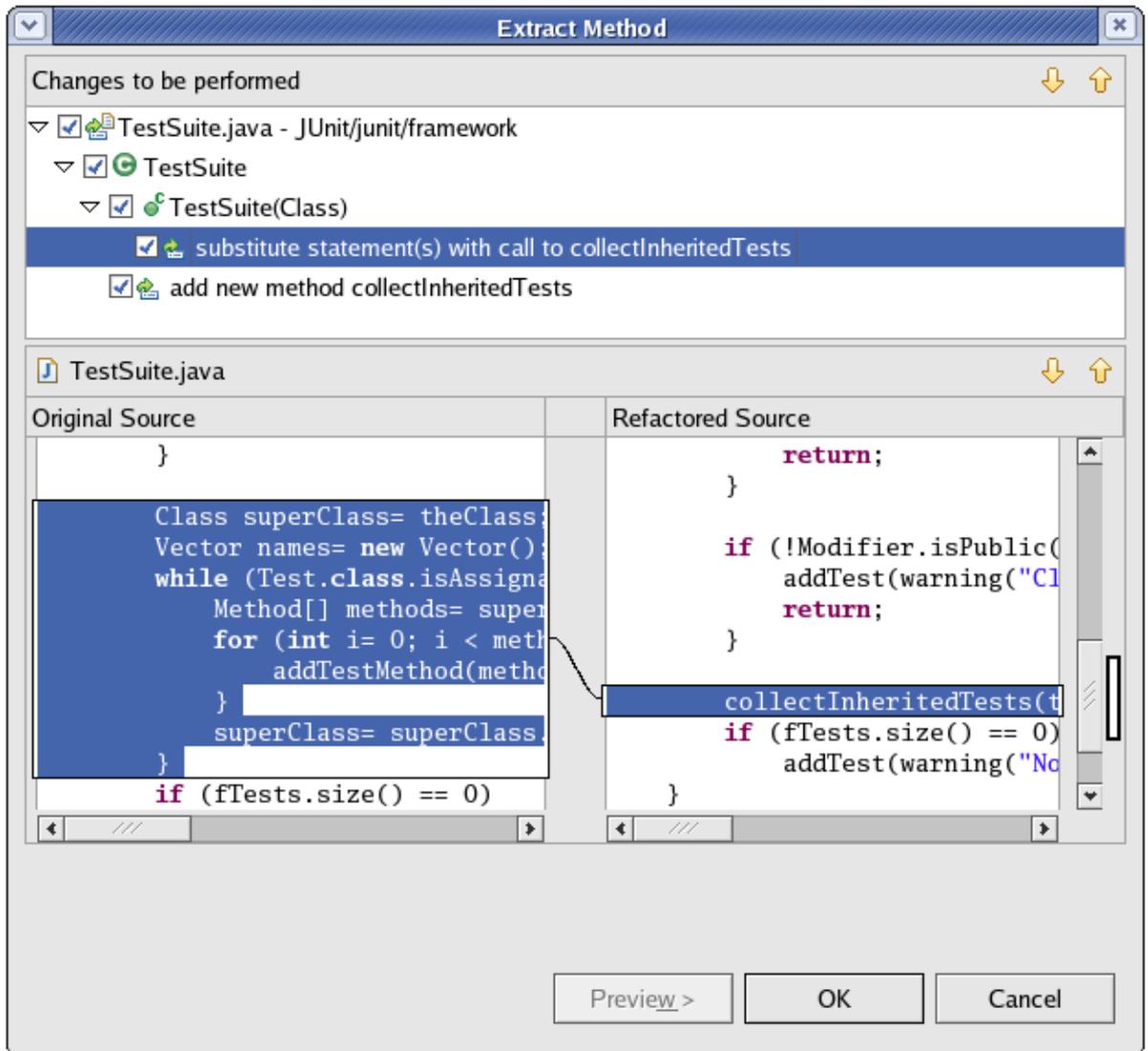
2. Right-click on the selection in the editor and select **Refactor > Extract Method...** from its context menu.



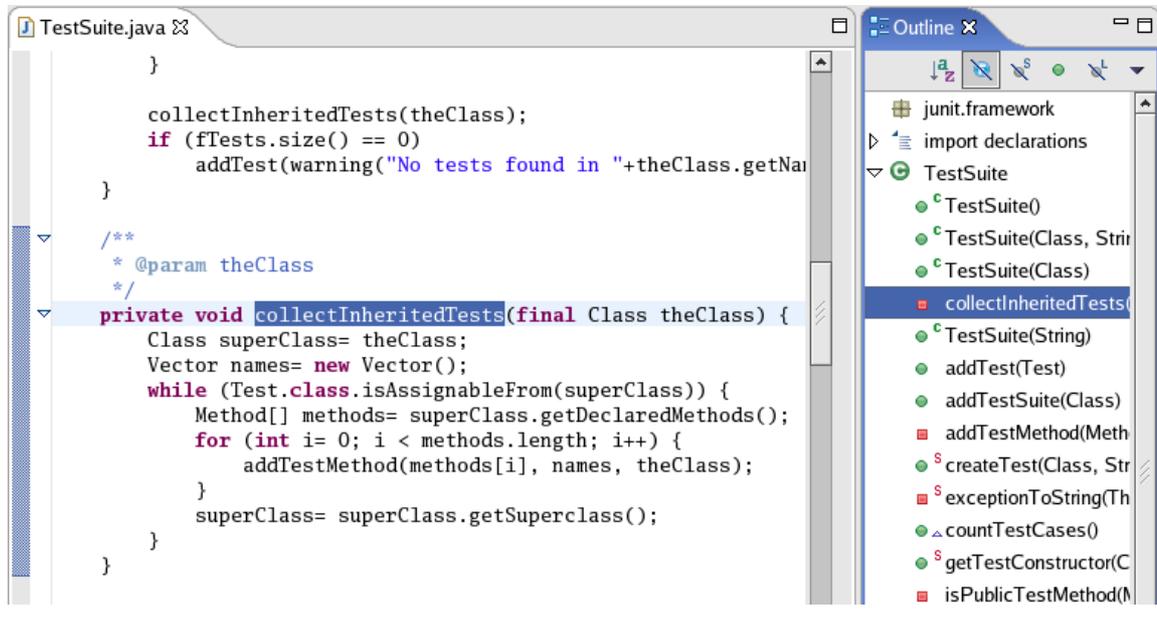
3. In the *Method Name* field, type `collectInheritedTests`.



4. To preview the changes, click **Preview >**. The preview page displays the changes that will be made. Click **OK** to extract the method.



5. Go to the extracted method by selecting it in the **Outline** view.



Creating a Java class

In this section, you will create a new Java class and add methods using code generation actions.

1. In the **Package Explorer** view, select the JUnit project, right-click, and from the project's context menu, select **New > Package**. (Alternatively, you can click the **New Java Package** button in the toolbar.)
2. In the **Name** field, type **test** as the name for the new package, then click **Finish**.
3. In the **Package Explorer** view, select the new **test** package and click the **New Java Class** button in the toolbar.
4. Make sure that **JUnit** appears in the **Source Folder** field and that **test** appears in the **Package** field. In the **Name** field, type **MyTestCase**.

Java Class
Create a new Java class.

Source Folder:

Package:

Enclosing type:

Name:

Modifiers: public default private protected
 abstract final static

Superclass:

Interfaces:

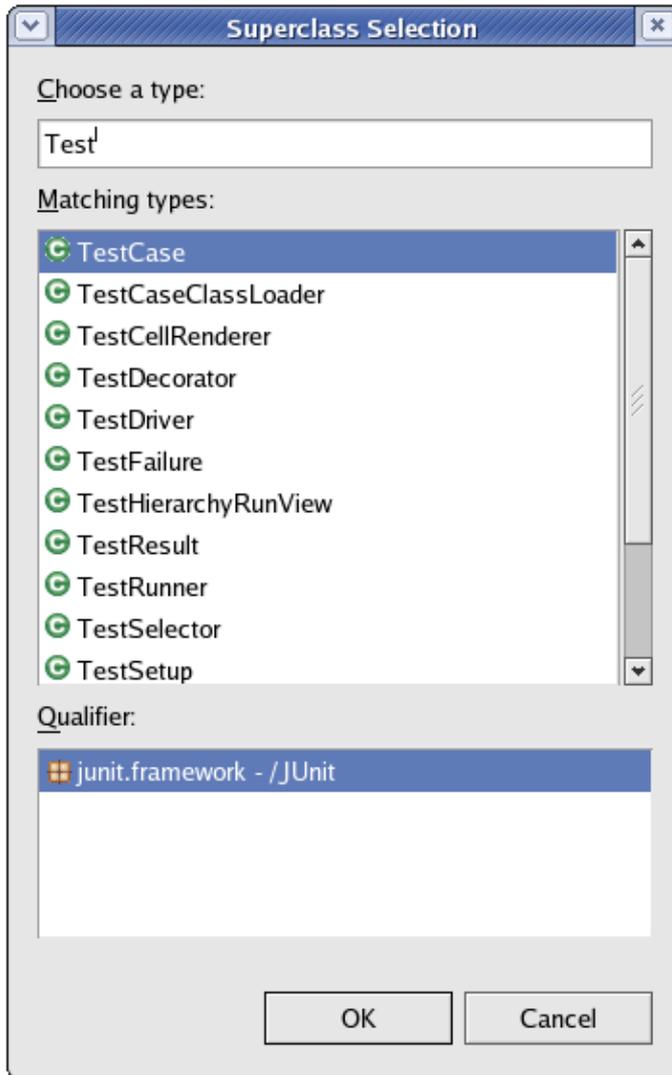
Which method stubs would you like to create?

public static void main(String[] args)

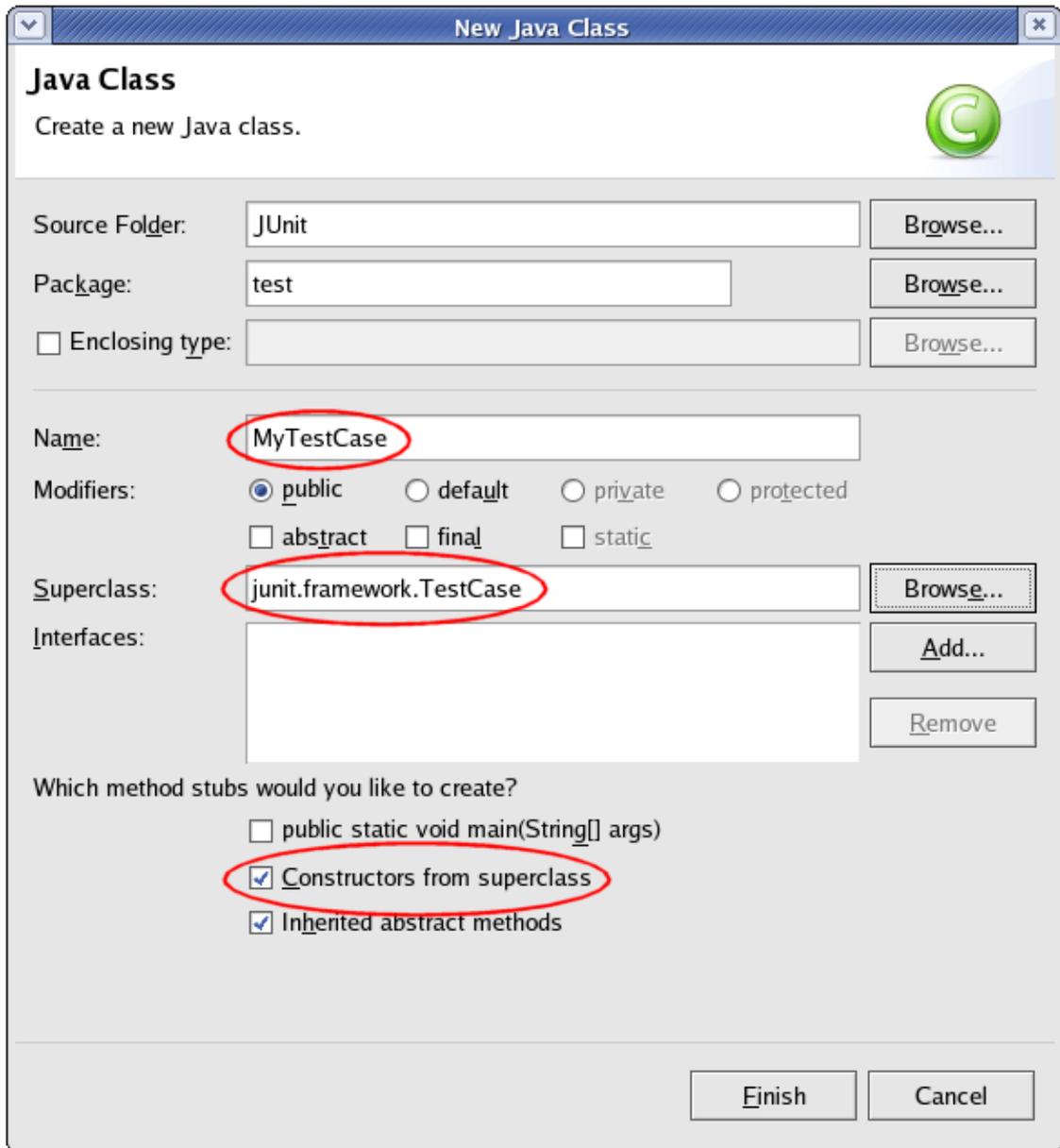
Constructors from superclass

Inherited abstract methods

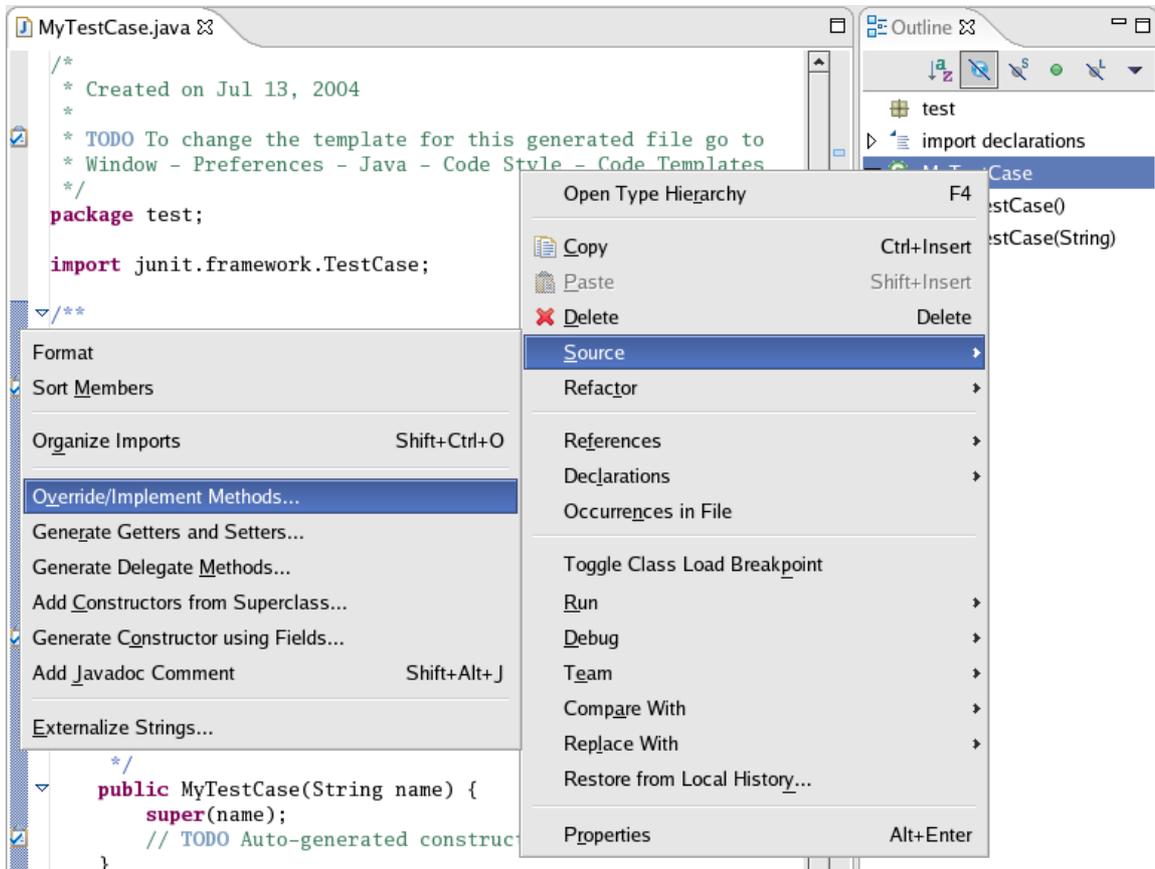
5. Click the **Browse** button next to the **Superclass** field.
6. In the **Choose a type** field in the **Superclass Selection** dialog, type **Test** to narrow the list of available superclasses.



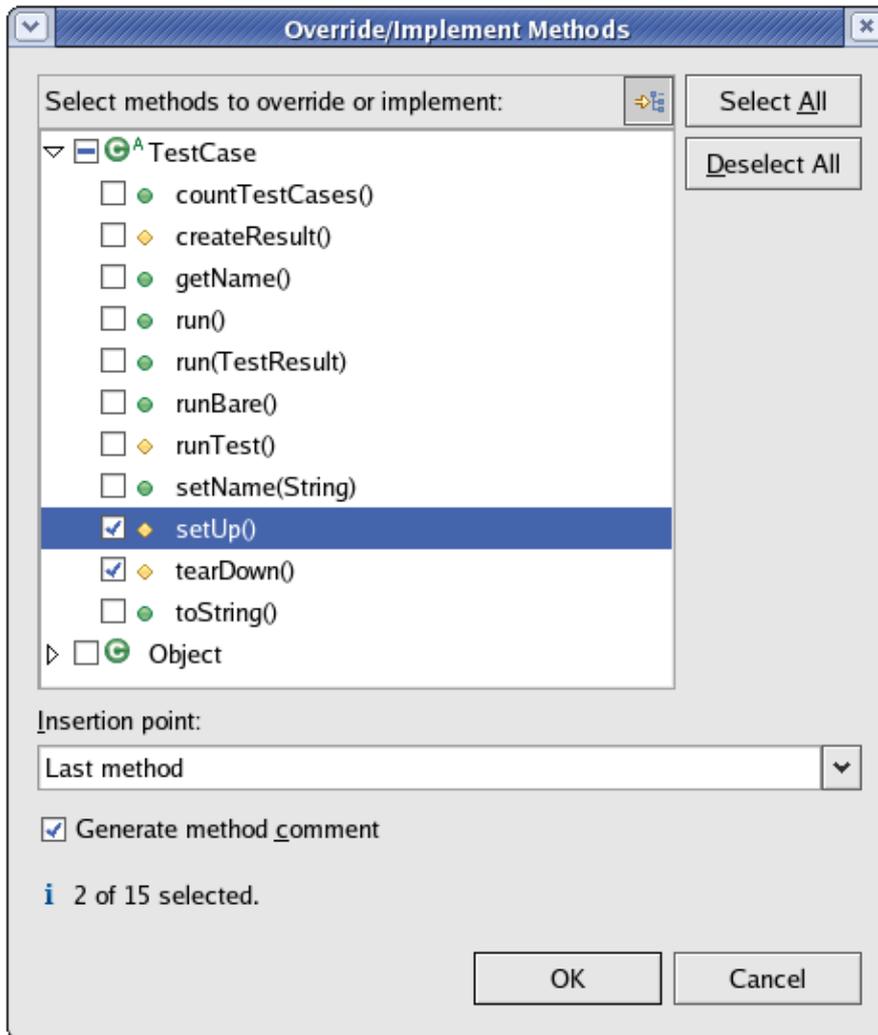
7. Select the **Test**Case class and click **OK**.
8. Select the checkbox for *Constructors from superclass*.
9. Click **Finish** to create the new class.



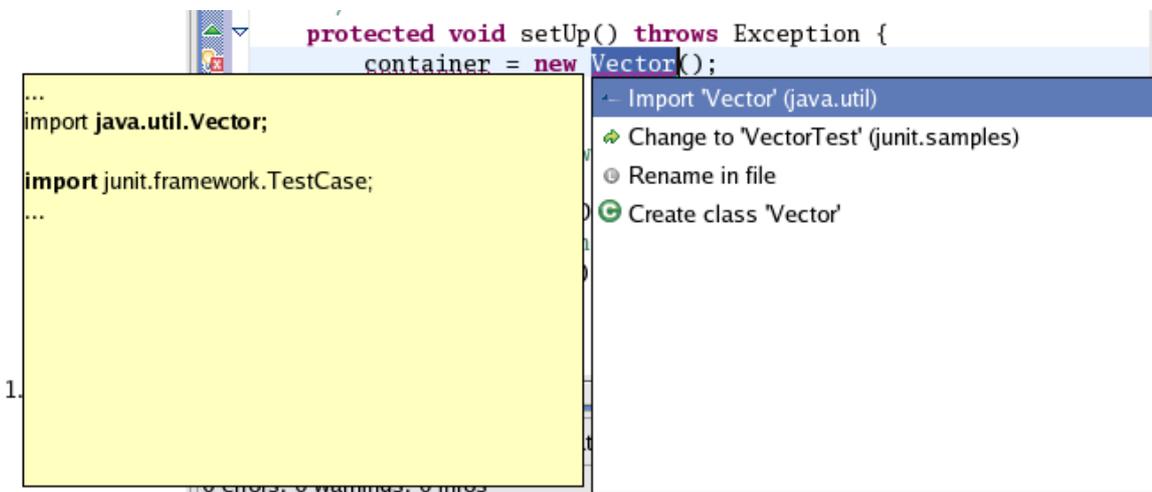
10. The new file is opened in the editor. It contains the new class, the constructor and comments. You can select options for the creation and configuration of generated comments in the Java preferences (*Window > Preferences > Java > Code Style > Code Templates*).
11. In the **Outline** view, select the new class **MyTestCase**, right-click, and select **Source > Override/Implement Methods** from the context menu.



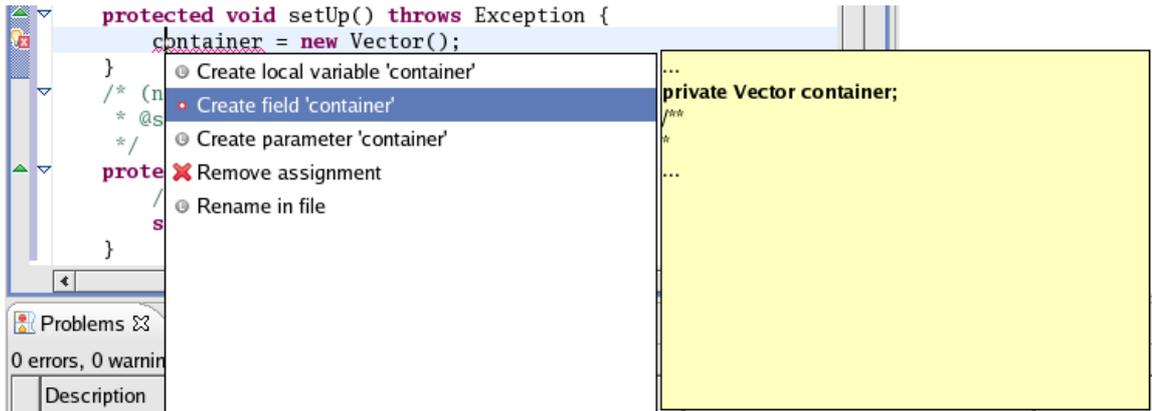
12. In the **Override Methods** set up() and tearDown() and click **OK**. Two methods are added to the class.



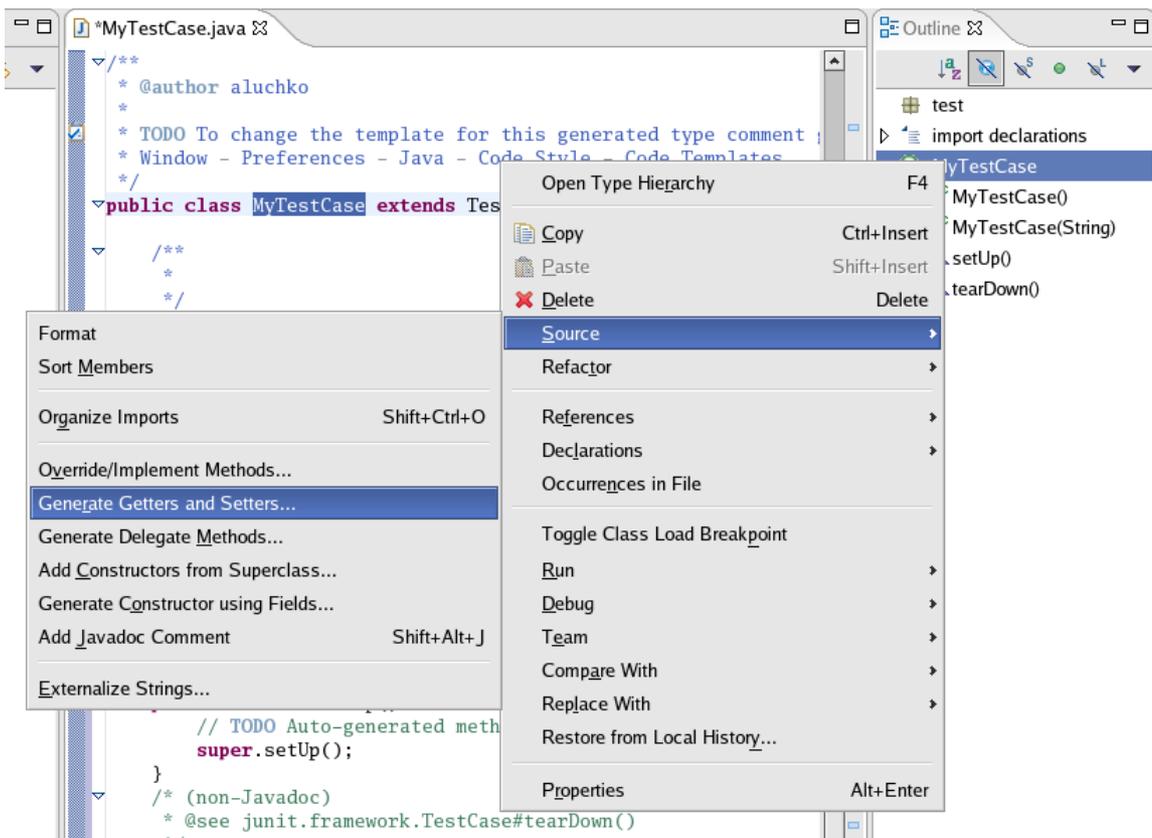
13. Change the body of **setUp()** to `container= new Vector() ;`
14. **container** and **Vector** are underlined with a problem highlight line as they cannot be resolved. A light bulb appears on the marker bar. Set the cursor inside **Vector** and press [Ctrl]+[1] (or use *Edit > Quick Fix* from the menu bar). Choose *Import 'Vector' (java.util)*. This adds the missing import declaration.



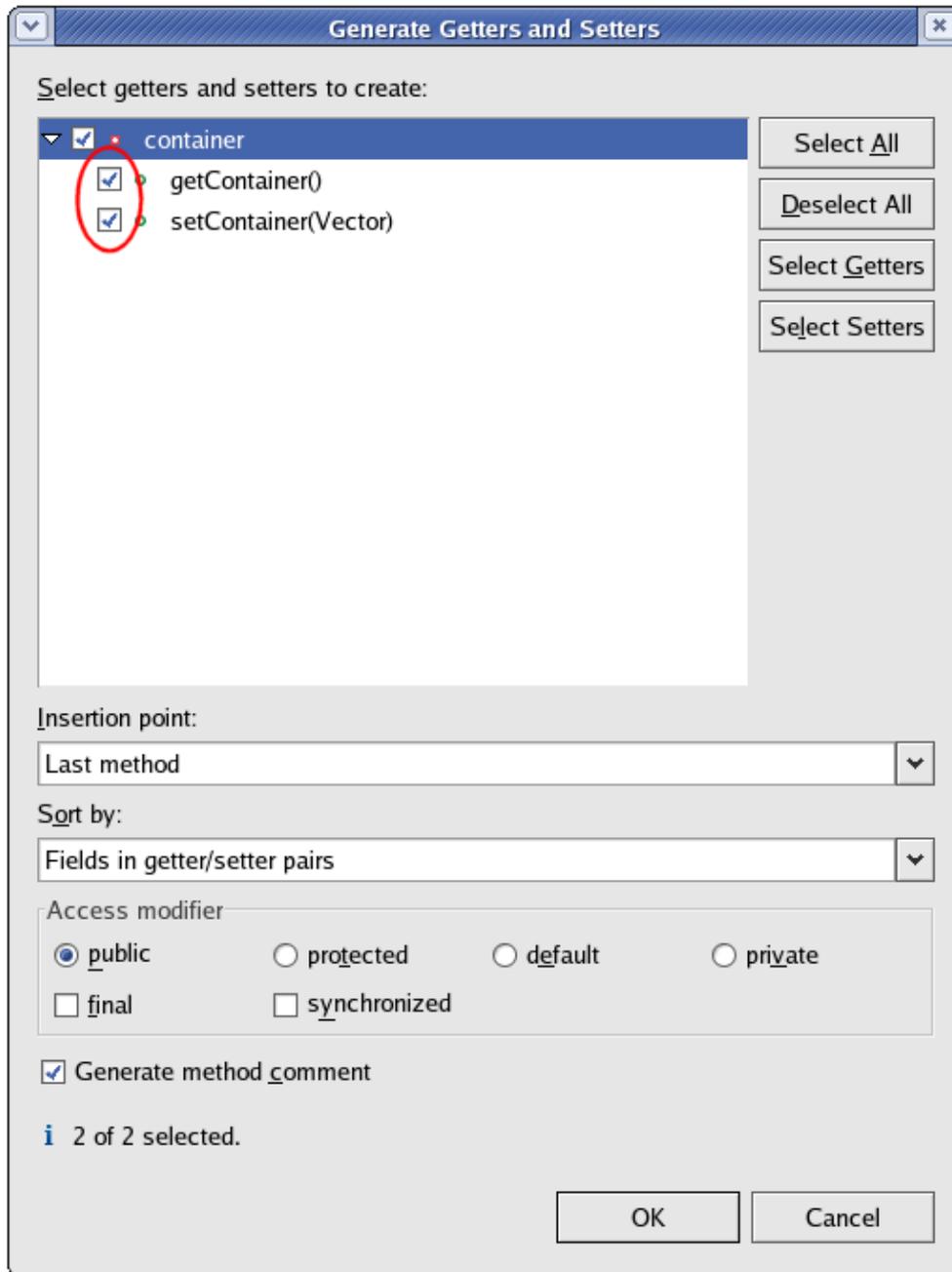
Set the cursor inside **container** and press [Ctrl]+[1]. Choose **Create field 'container'** to add the new field.



- In the **Outline** view, select the class **MyTestCase**, right-click, and select **Source > Generate Getter and Setter** from the context menu.



- The **Generate Getter and Setter** dialog suggests that you create the methods `getContainer` and `setContainer`. Select both and click **OK**. A getter and setter method for the field `container` are added.

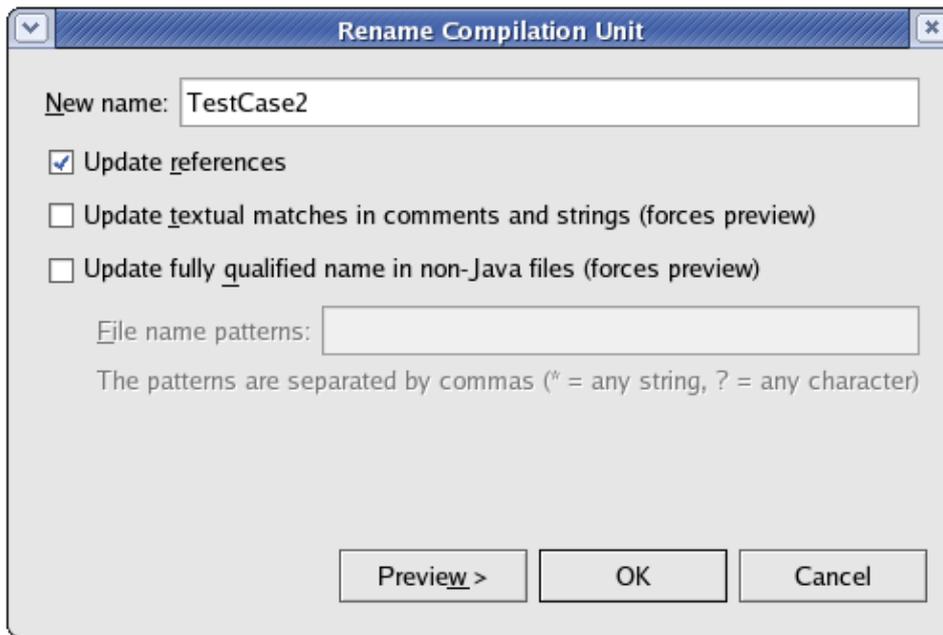


17. Save the file.
18. The formatting of generated code can be configured in *Window > Preferences > Java > Code Style > Code Formatter*. If you use a prefix or suffix for field names (for example, fContainer), you can specify this in *Window > Preferences > Java > Code Style > Fields* so that generated getters and setters will suggest method names without the prefix or suffix.

Renaming Java elements

In this section, you will rename a Java element using refactoring. Refactoring actions change the structure of your code without changing its semantic behavior.

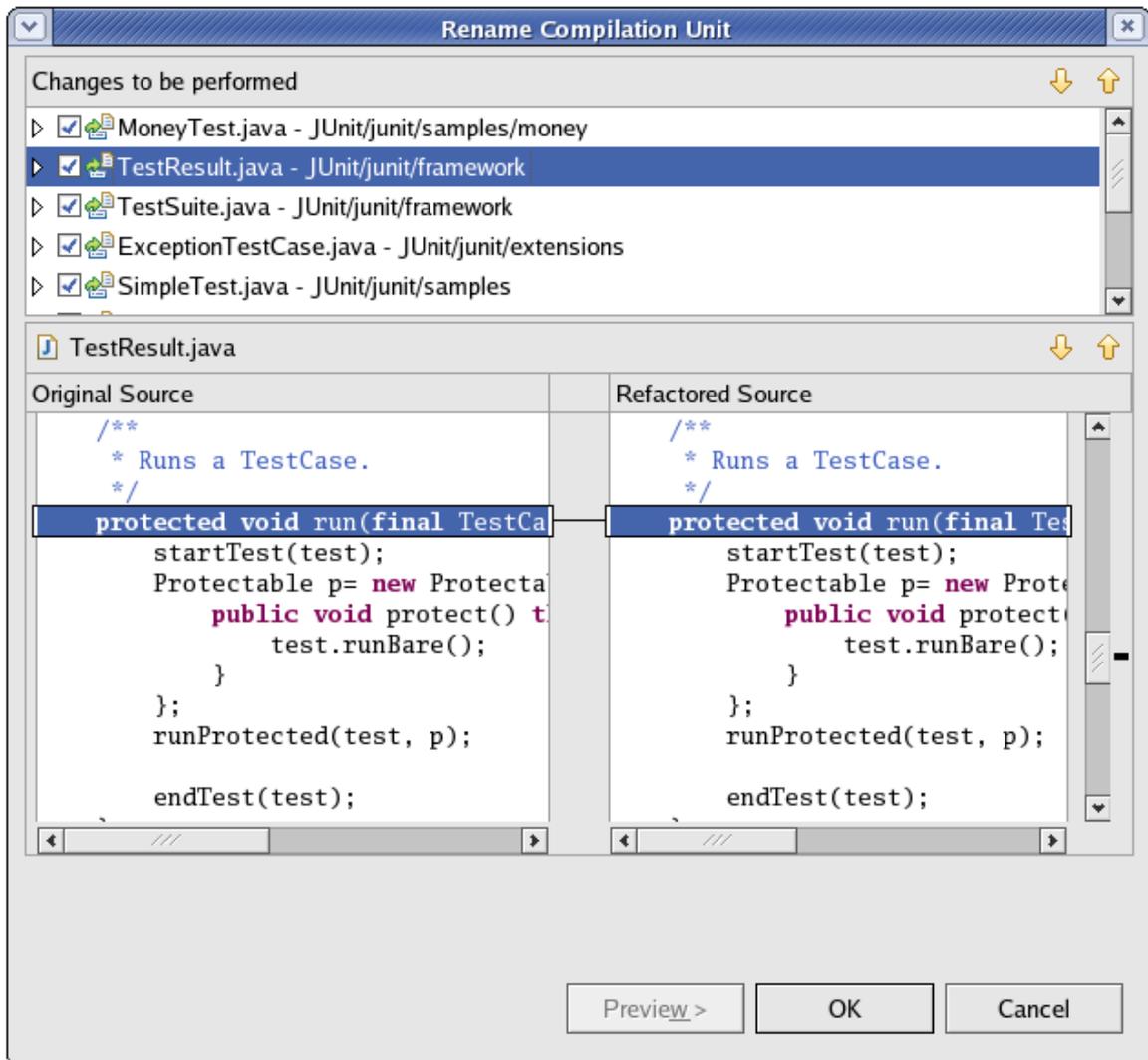
1. In the Package Explorer view, select `junit.framework.TestCase.java`, right-click, and select **Refactor** > **Rename** from its context menu.
2. In the *Enter New Name* field on the **Rename Compilation Unit** page, type "TestCase2".



3. To preview the changes that will be made as a result of renaming the class, press **Preview >**.

The workbench analyzes the proposed change and presents you with a preview of the changes that would take place if you rename this resource.

Because renaming a compilation unit will affect the import statements in other compilation units, there are other compilation units affected by the change. These are shown in a list of changes in the preview pane.



4. On the **Refactoring preview** page, you can scroll through the proposed changes and select or deselect changes, if necessary. You will typically accept all of the proposed changes.
5. Click **OK** to accept all proposed changes.

You have seen that a refactoring action can cause many changes in different compilation units. These changes can be undone as a group: in the menu bar, select *Refactor > Undo Rename TestCase.java to TestCase2.java*.



The refactoring changes are undone, and the workbench returns to its previous state. You can undo refactoring actions right up until you change and save a compilation unit, at which time the refactoring undo buffer is cleared.

Moving Java elements

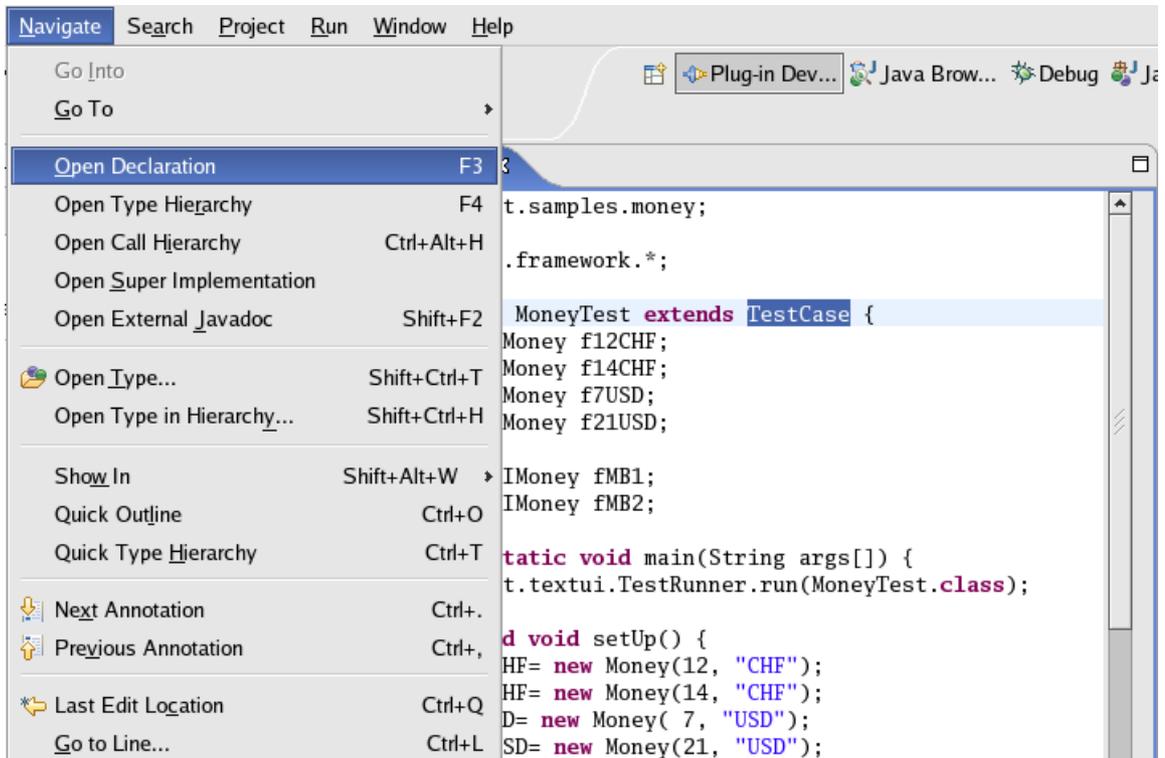
In this section you will move a resource between Java packages.

1. In the Package Explorer view, select and right-click the MyTestCase.java file, then select **Refactor** > **Move**.
2. In the Move dialog, expand the hierarchy to browse the possible new locations for the resource. Select the junit.samples package, then click OK. The class is moved and its package declaration is updated to the new location.



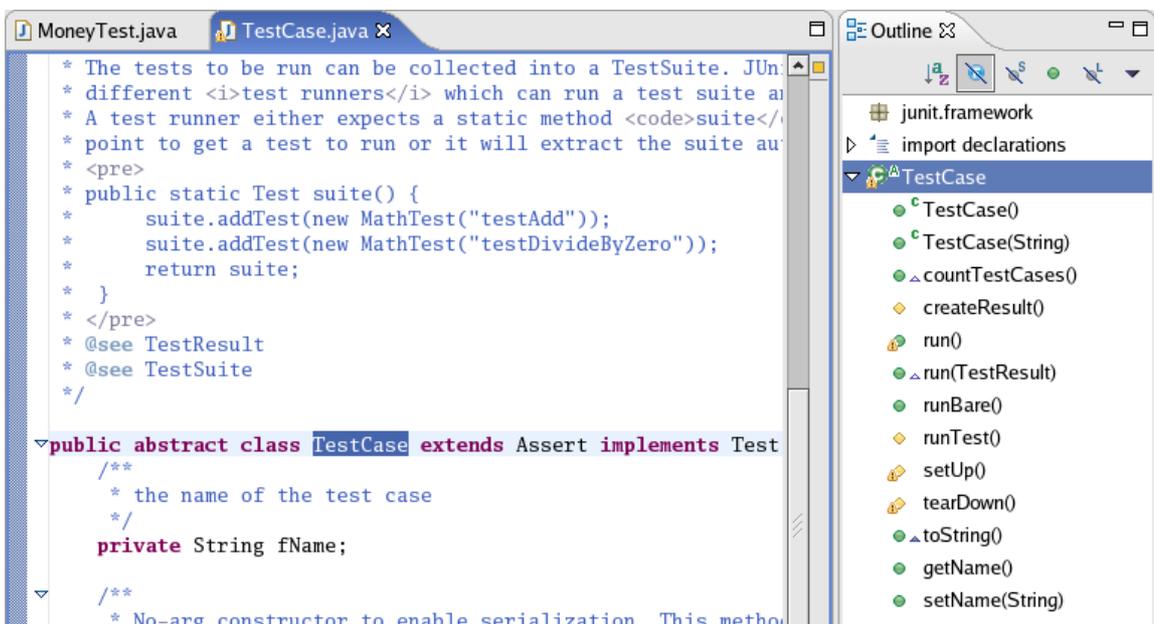
Navigate to a Java element's declaration

1. Open the `junit.samples.money.MoneyTest.java` file in the Java editor.
2. On the first line of the `MoneyTest` class declaration, select the superclass `TestCase` and click **Navigate > Open Declaration**.

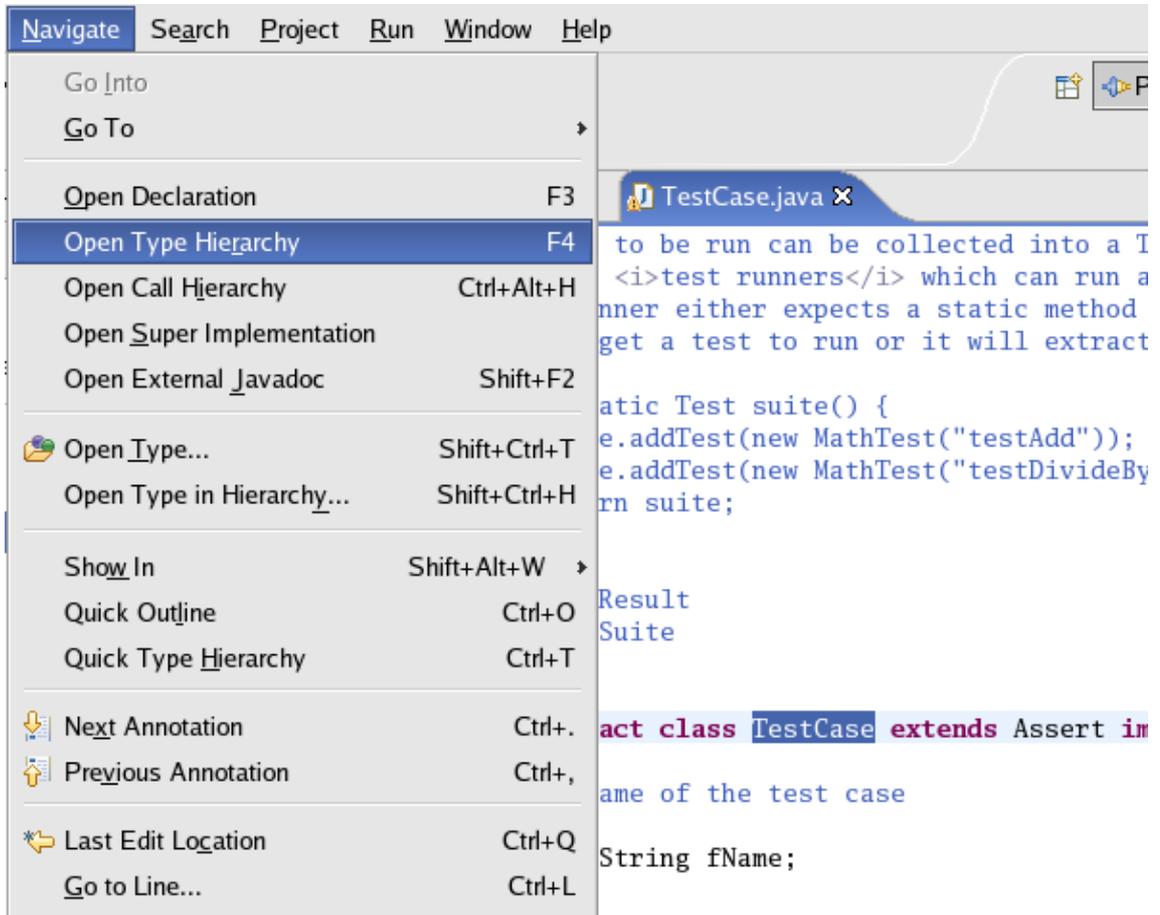


The TestCase class opens in the editor area and is also represented in the **Outline** view.

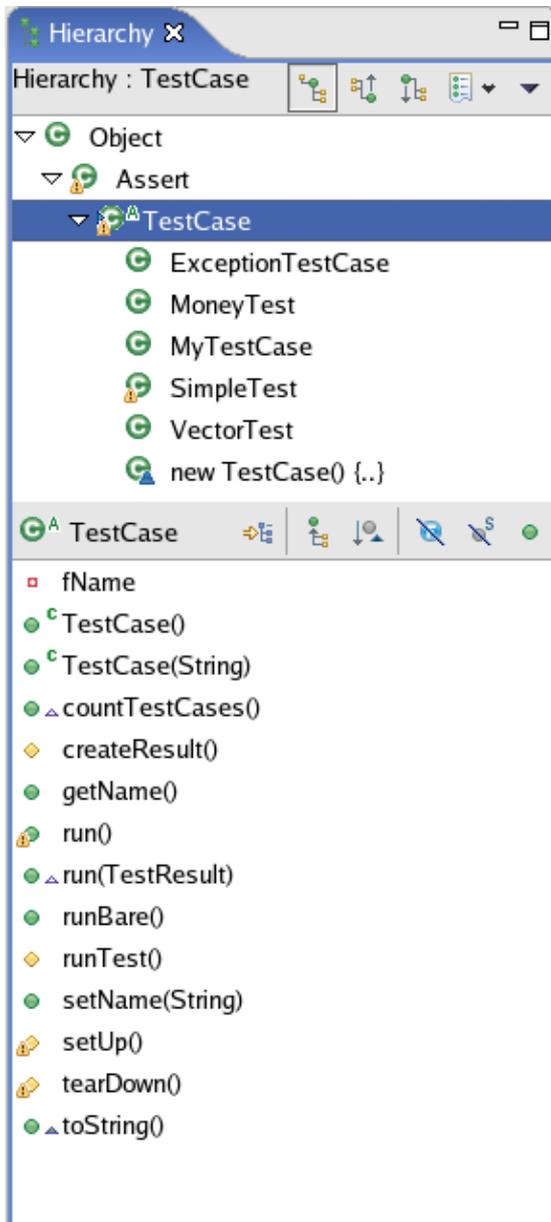
Note: This command also works on methods and fields.



3. With the TestCase.java editor open and the class declaration selected, click **Navigate > Open Type Hierarchy**.



4. The Hierarchy view opens with the TestCase class displayed.

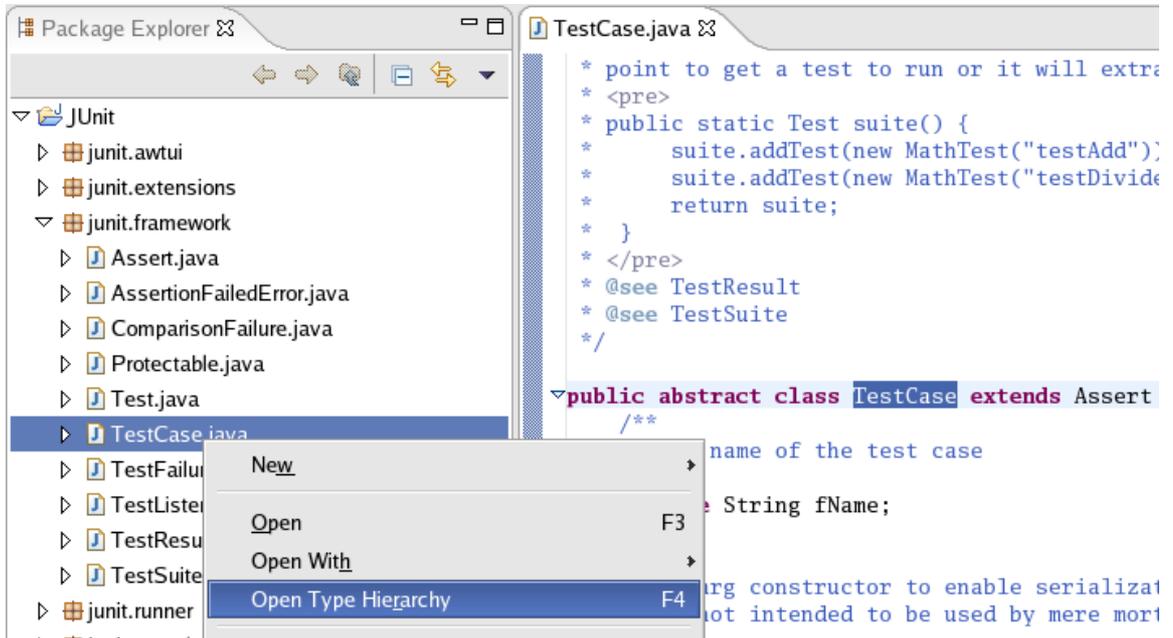


Note: You can also open editors on types and methods in the **Hierarchy** view.

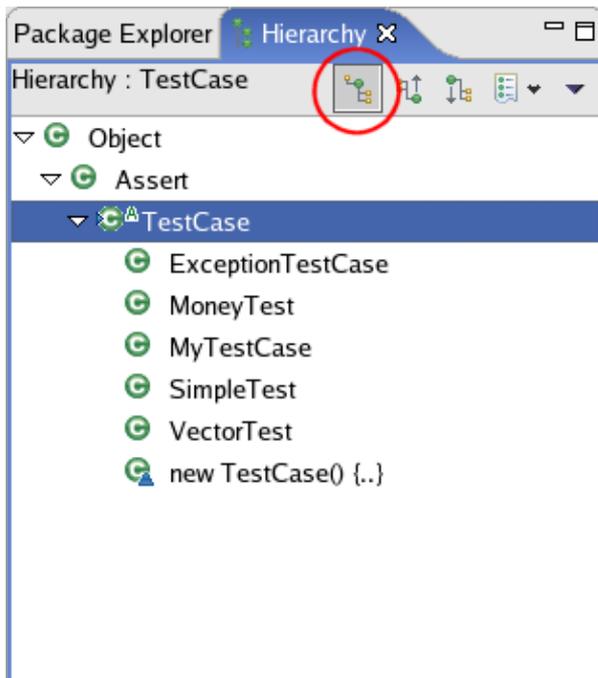
Viewing the type hierarchy

In this section, you will learn about using the **Hierarchy** view by viewing classes and members in a variety of different ways.

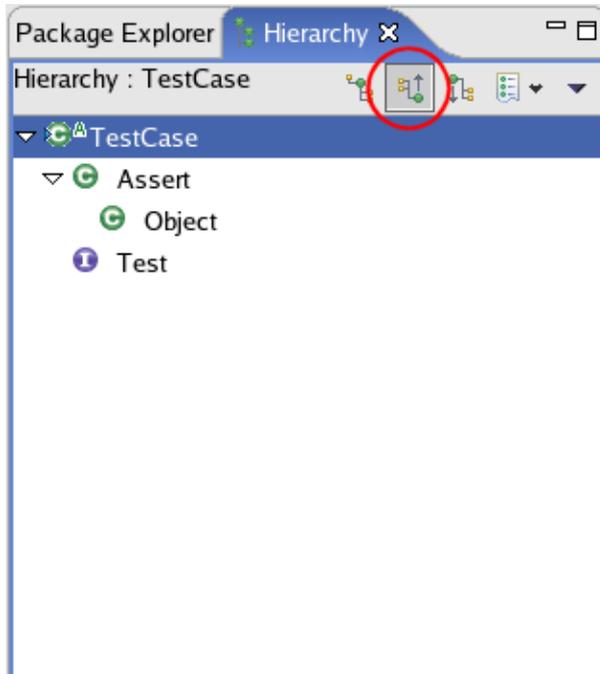
1. In the **Package Explorer** view, select **junit.framework.TestCase.java**, right-click, and select **Open Type Hierarchy** from its context menu. You can also open type **Hierarchy** view:
 - ◆ From the menu bar by selecting **Navigate > Open Type Hierarchy**.
 - ◆ From the keyboard by pressing F4 after selecting **TestCase.java**.



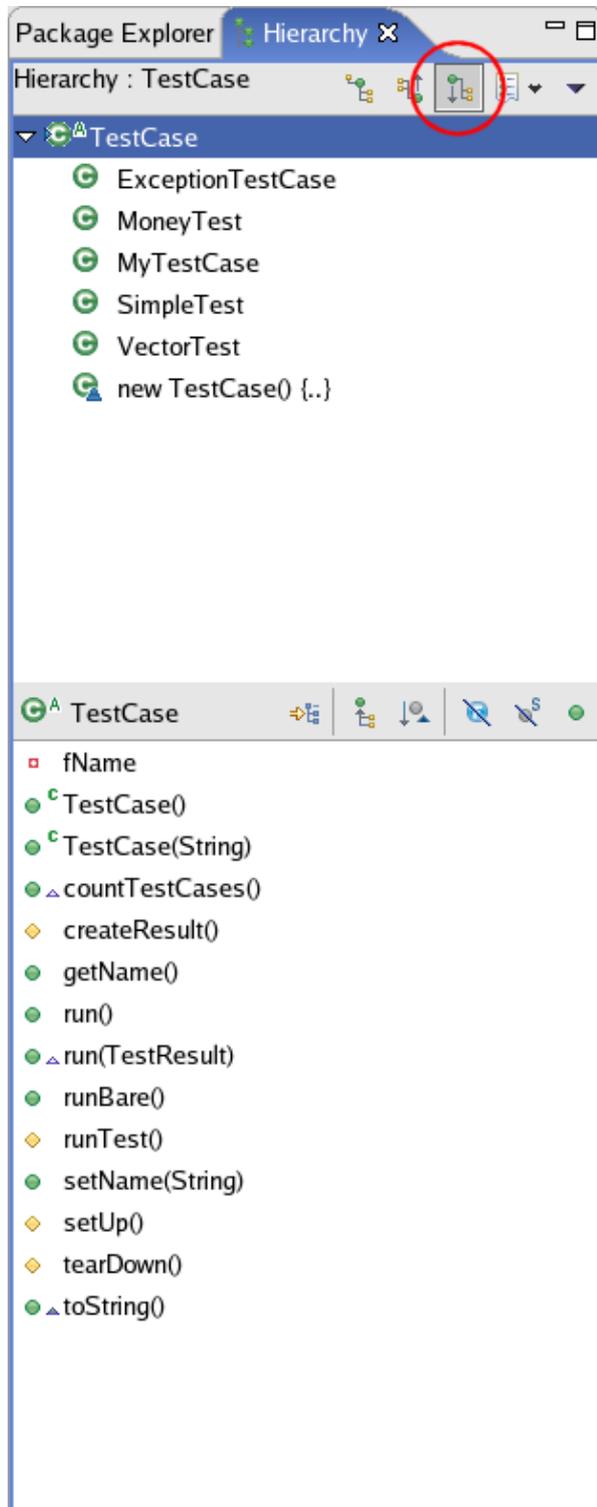
2. The buttons in the view tool bar control which part of the hierarchy is shown. Click *Show the Type Hierarchy* to see the class hierarchy, including the base classes and subclasses. The small arrow on the left side of the type icon of TestSuite indicates that the hierarchy was opened on this type.



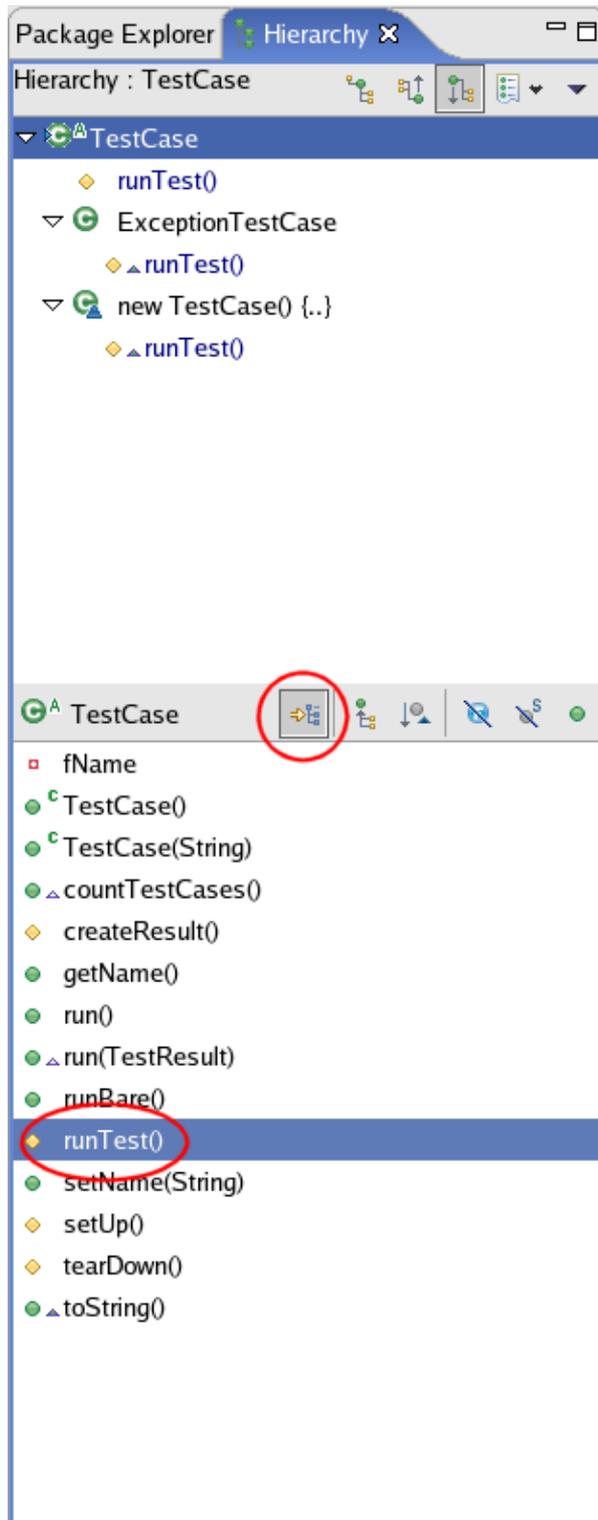
3. Click *Show the Supertype Hierarchy* to see a hierarchy showing the type's parent elements including implemented interfaces. This view shows the results of going up the type hierarchy.



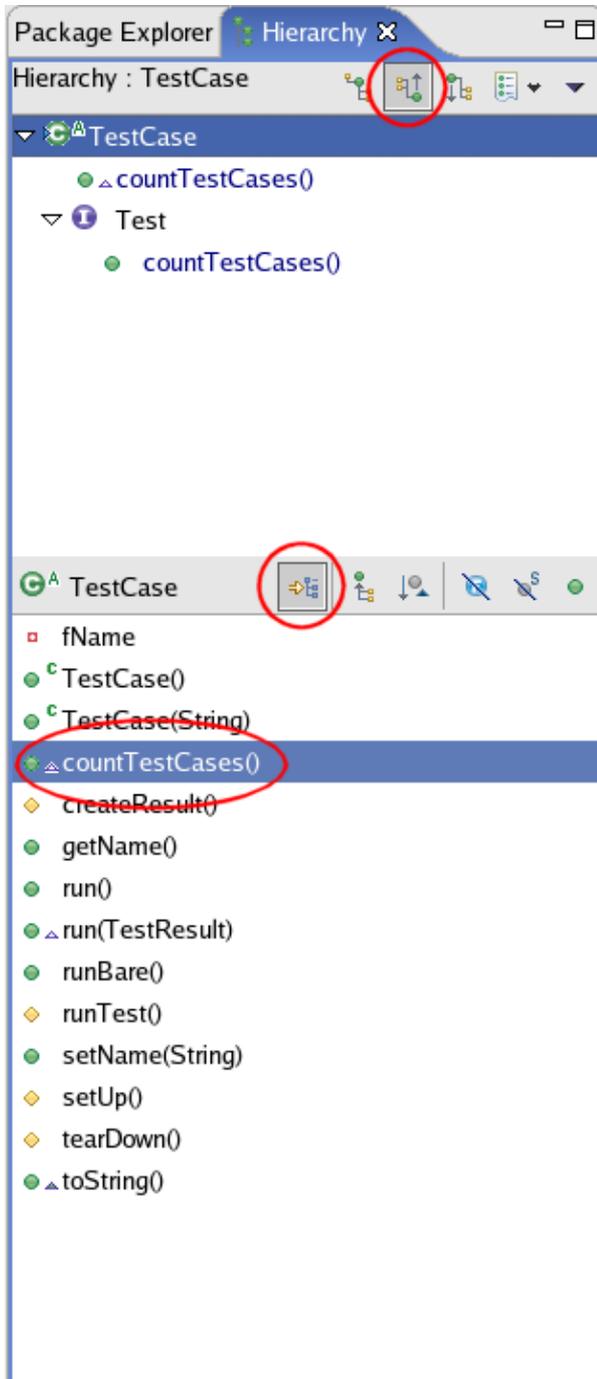
- In this "reversed hierarchy" view, you can see that TestCase implements the Test interface.
4. Click **Show the Subtype Hierarchy** in the view toolbar.



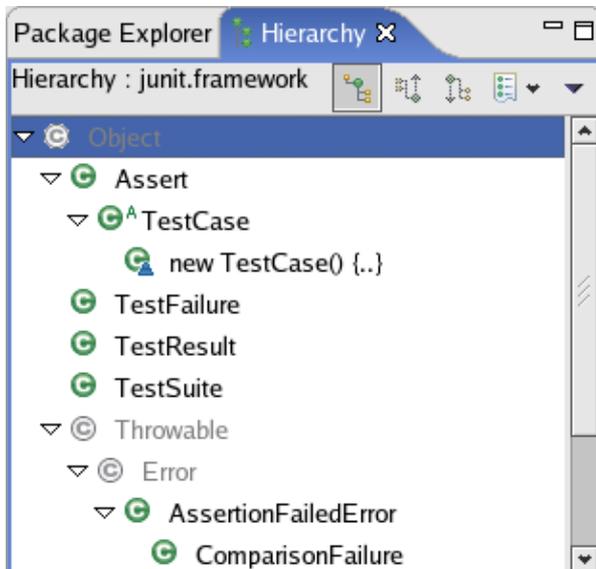
5. Click the *Lock View and Show Members in Hierarchy* button in the toolbar of the member pane, then select the **runTest()** method in the member pane. The view now shows all the types implementing runTest().



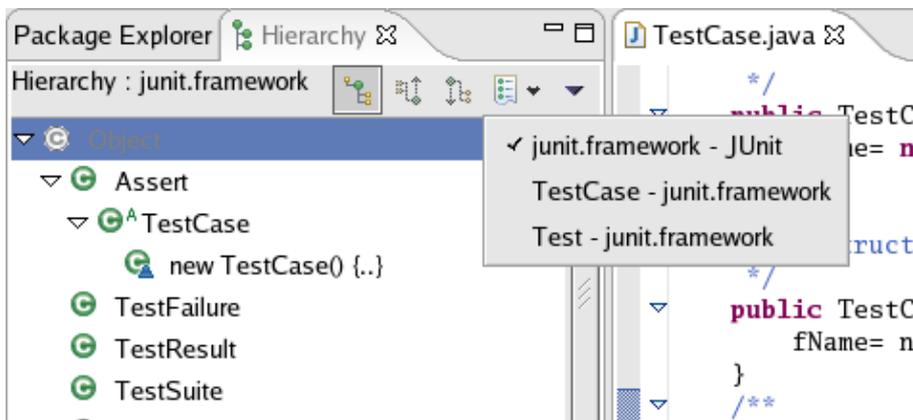
6. In the **Hierarchy** view, click **Show the Supertype Hierarchy**. Then on the member pane, select `countTestCases()` to display the places where this method is declared.



7. In the **Hierarchy** view select the *Test* element, right-click, and select **Focus On 'Test'** from its context menu. *Test* is presented in the **Hierarchy** view.
8. In the **Package Explorer** view, select the package **junit.framework**, right-click, and select **Open Type Hierarchy** from its context menu. A hierarchy opens containing all classes of the package. For completion of the tree, the hierarchy also shows some classes from other packages. These types are shown by a type icon with a white fill.



9. Use *Previous Type Hierarchies* to go back to a previously opened element. Click on the arrow next to the button to see a list of elements or click on the button to edit the history list.



10. From the menu bar, select *Window > Preferences*. Go to *Java*, select *Open a new Type Hierarchy Perspective*, then click *OK*.
11. In the **Hierarchy** view, select the **Test** element again, and activate *Open Type Hierarchy* from the **Navigate** menu bar. The resource containing the selected type is shown in a new perspective (the **Java Type Hierarchy** perspective), and its source is shown in the Java editor.

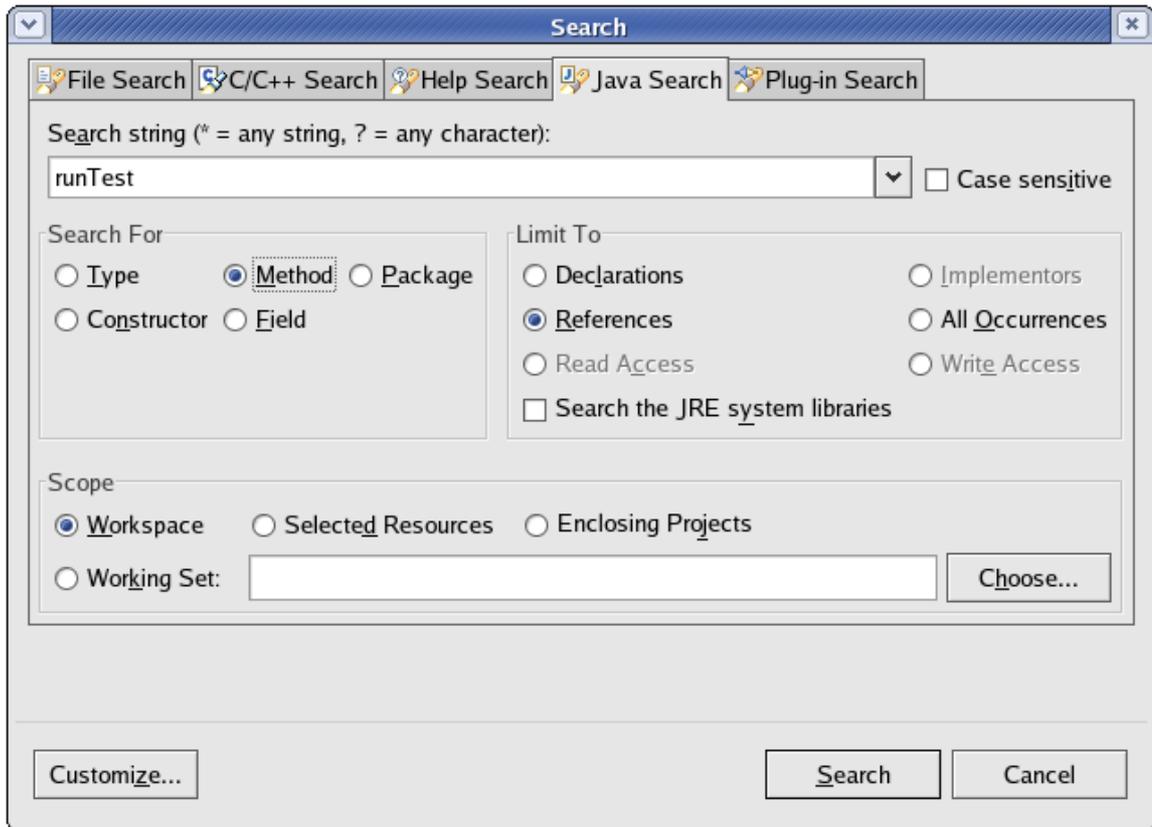
Searching the workbench

In the Search dialog, you can perform file, text or Java searches. Java searches operate on the structure of the code. With File Search you can search files by name and/or text content. Indexing makes Java searches faster, while searches on text content will allow you to find matches inside comments and strings.

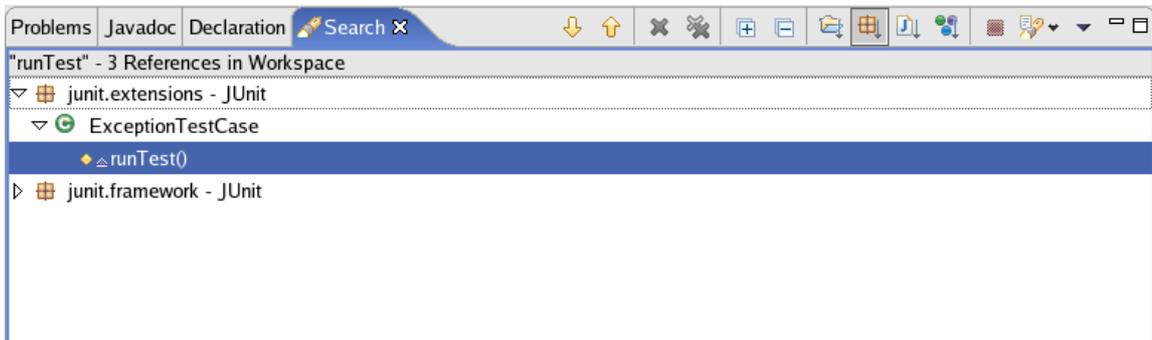
Performing a JDT search

1. In the Java perspective, click the Search button in the workbench toolbar. Alternatively, you can go to the menu bar and select *Search > Java*.
2. If it is not already selected, select the Java Search tab.

3. In the *Search string* field, type `runTest`. In the Search For area, select *Method*, and in the Limit To area, select *References*. Verify that the Scope is set to *Workspace*.

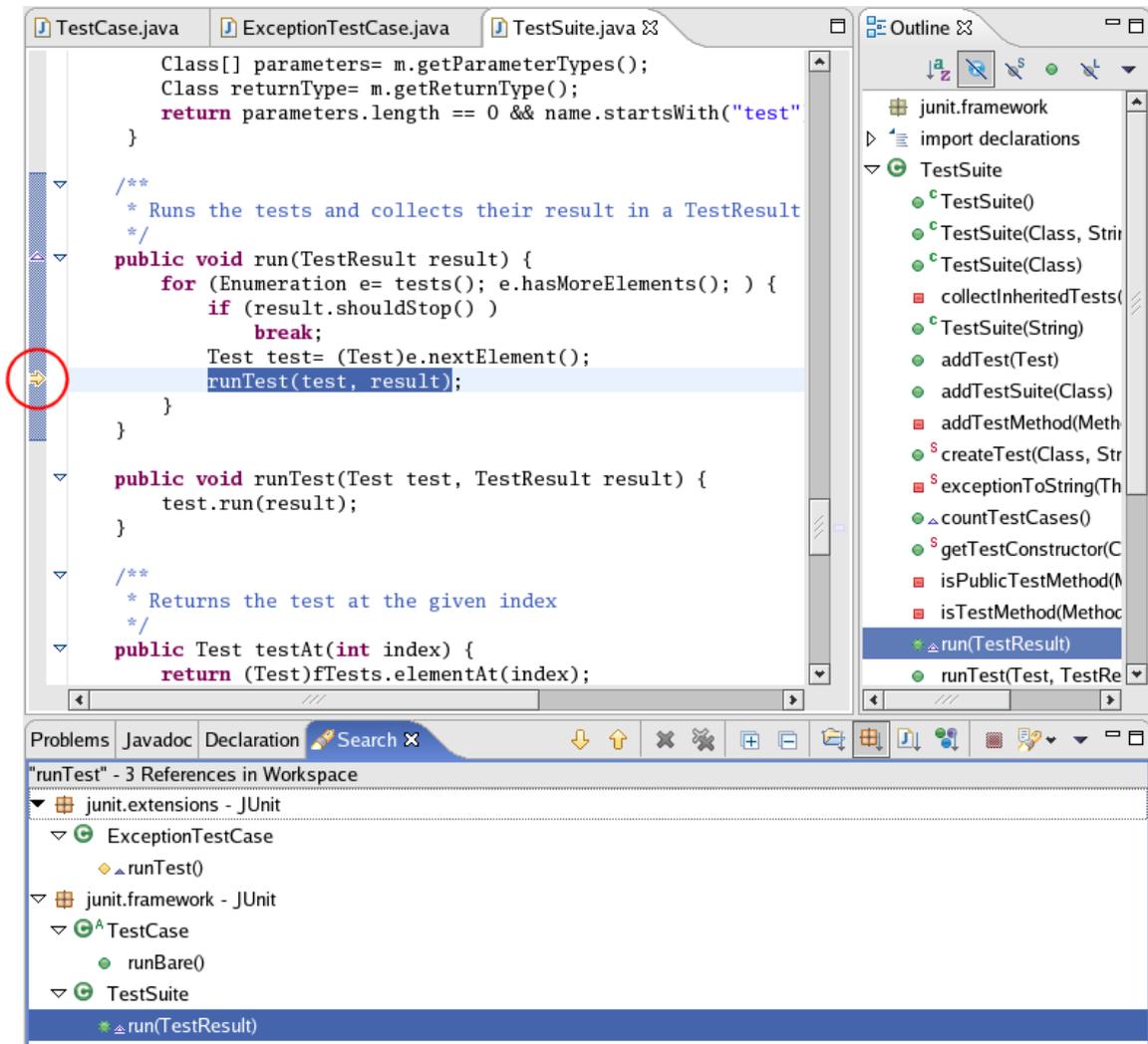


4. Click Search. While searching, you can click *Cancel* at any time to stop the search; partial results will be shown.
5. The Search view shows the search results.



Use the *Show Next Match*  and *Show Previous Match*  buttons to navigate to each match. If the file in which the match was found is not currently open, it is opened in an editor.

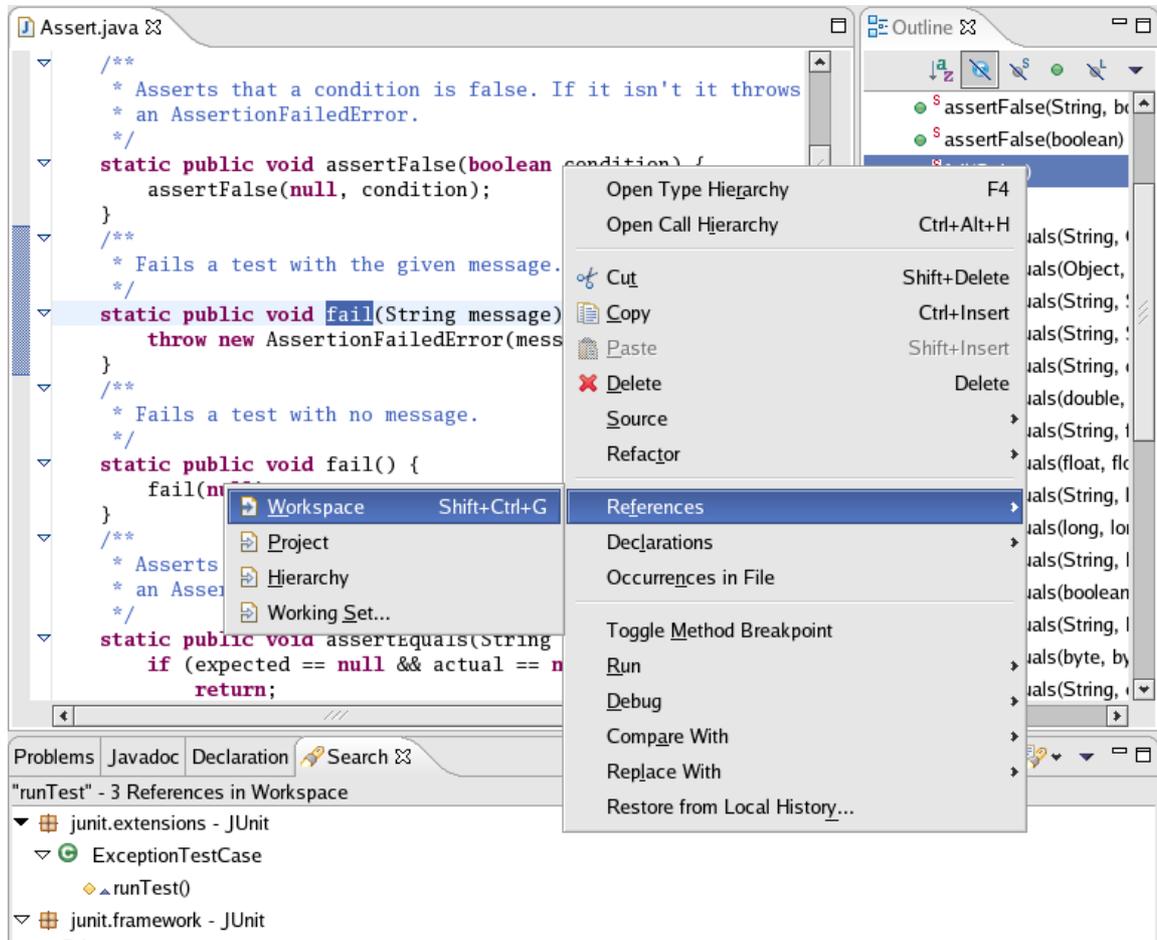
Note: When you navigate to a search match using the **Search** view buttons, the file opens in the editor at the position of the match. Search matches are tagged with a search marker in the marker bar.



Searching from a Java view

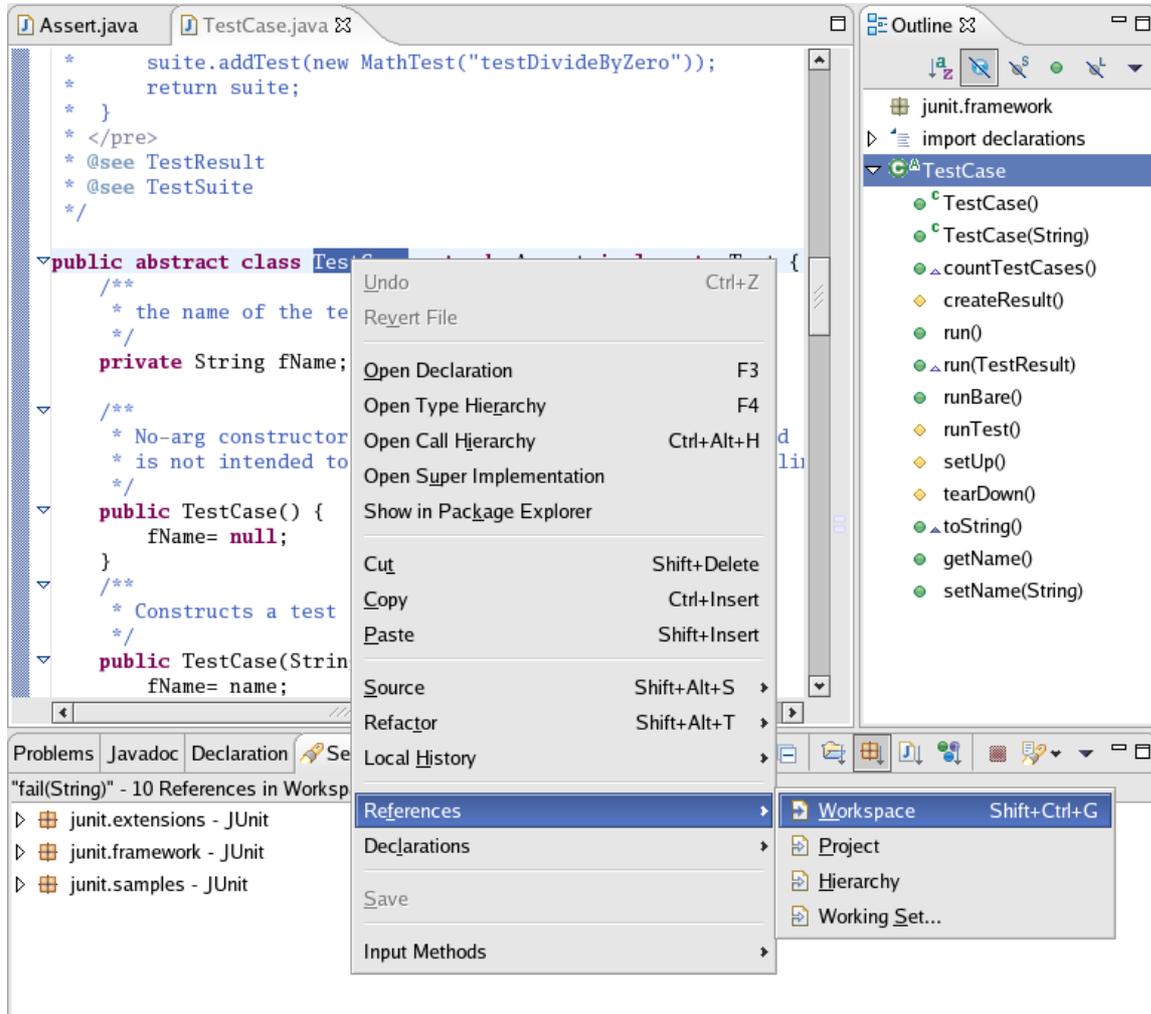
Java searches can be performed from several views, including the **Outline** view, **Hierarchy** view, and the **Package Explorer** view.

1. In the **Package Explorer** view, select and right-click `junit.framework > Assert.java`, then select **Open** to open it in an editor.
2. In the **Outline** view, select and right-click the `fail(String)` method then select **References > Workspace**.



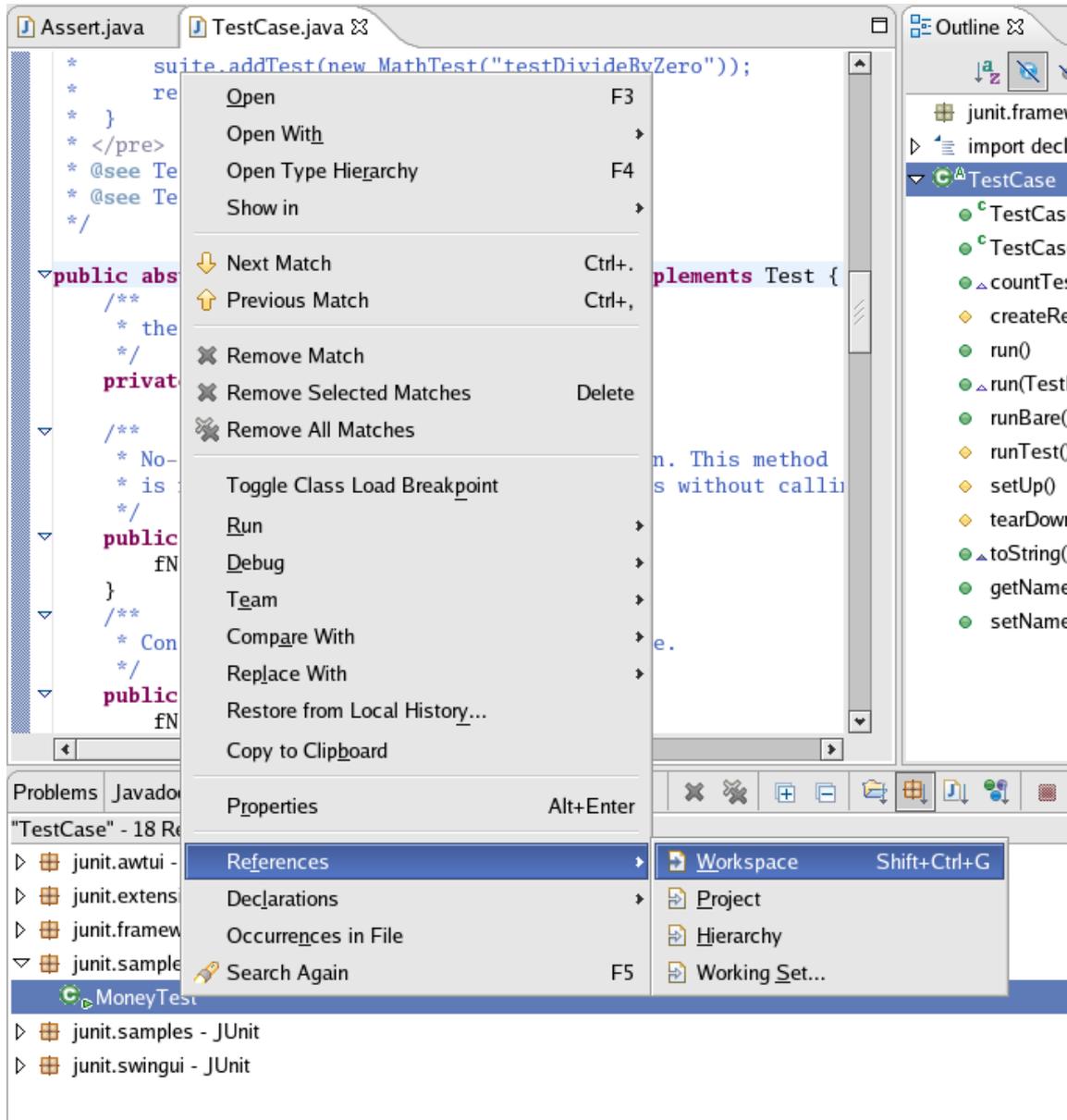
Searching from an editor

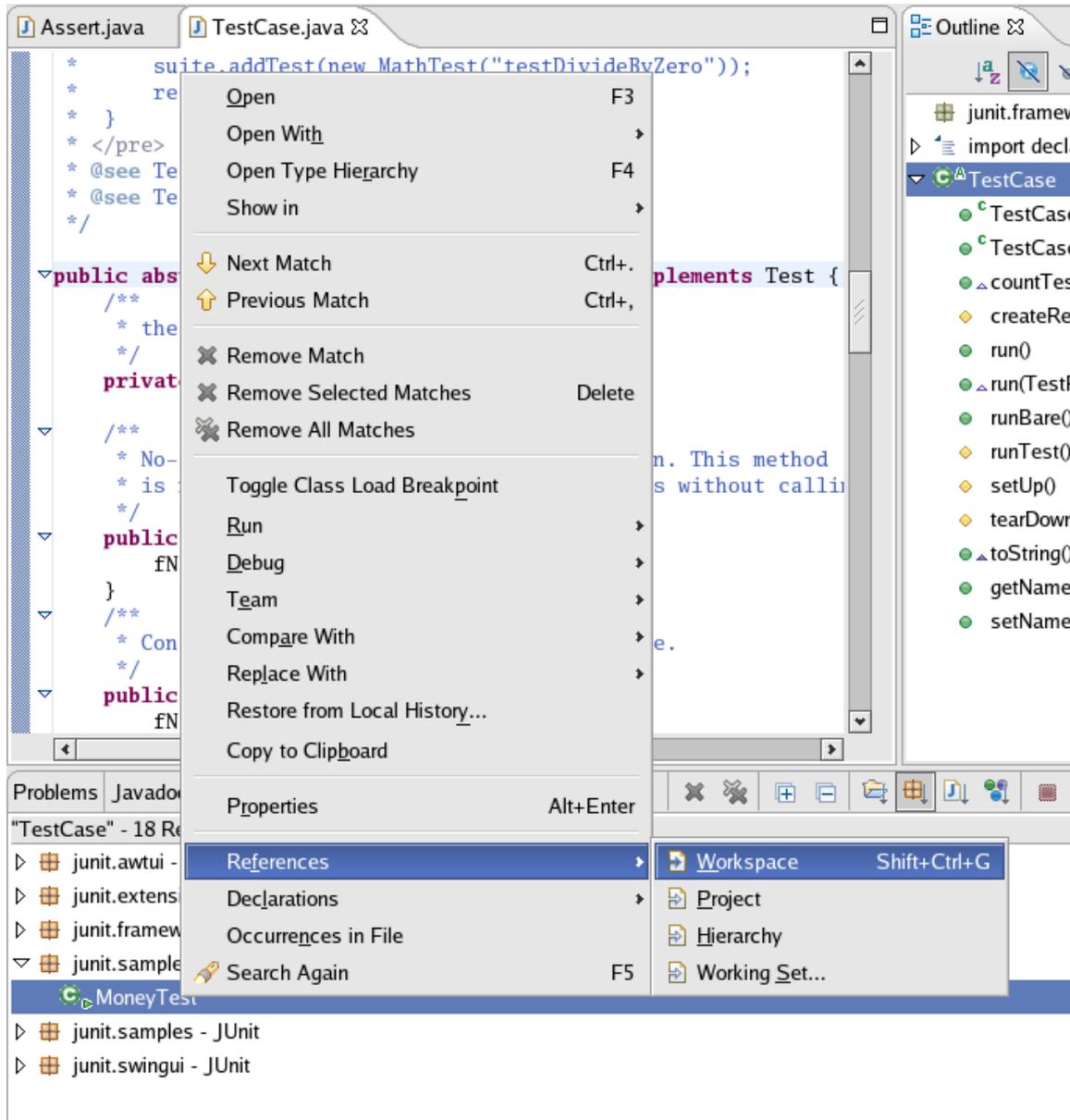
From the **Package Explorer** view, open **junit.framework > TestCase.java**. In the editor, select the class name **TestCase** and from the context menu, select **Search > References > Workspace**.



Continuing a search from the Search view

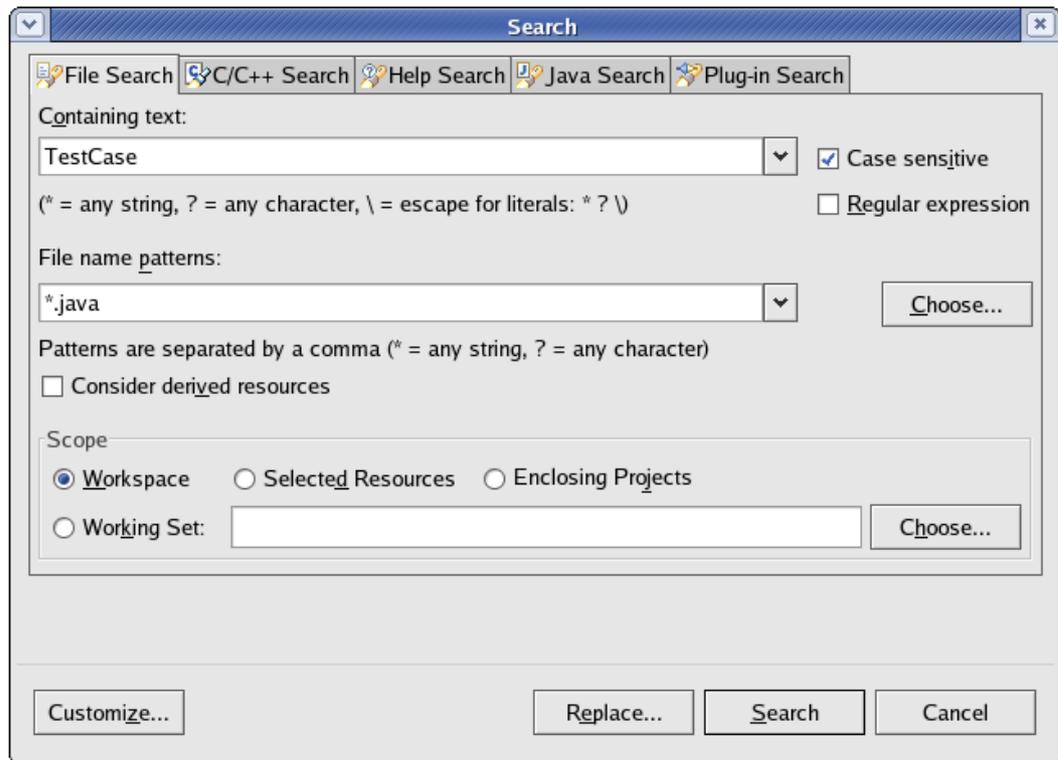
The **Search Results** view shows the results for the `TestCase` search. Select a search result and right-click on it to open the context menu. You can continue searching the selected element's references and declarations.





Performing a file search

1. In the **Java** perspective, click the Search button in the workbench toolbar or from the menu bar select Search > File.
2. If it is not already selected, select the **File Search** tab.
3. Set up the search parameters:
 - a. In the **Containing text** field, type: **TestCase**
 - b. In the **File name patterns** field, make sure that is set to *.java.
 - c. The **Scope** should be set to Workspace.

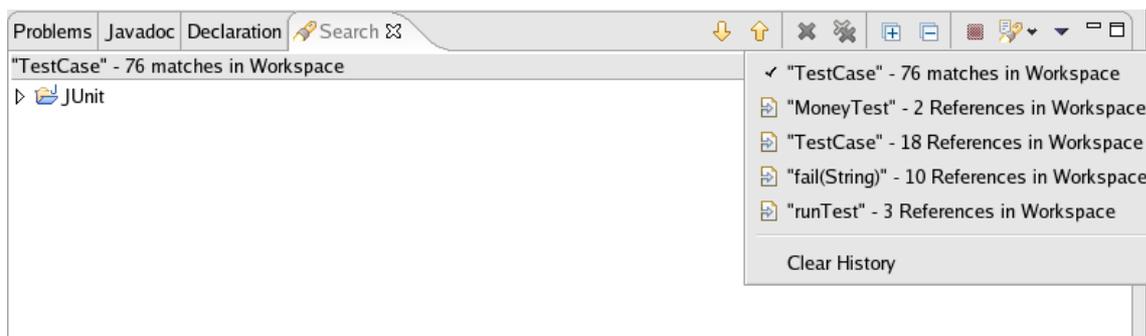


4. Click Search.

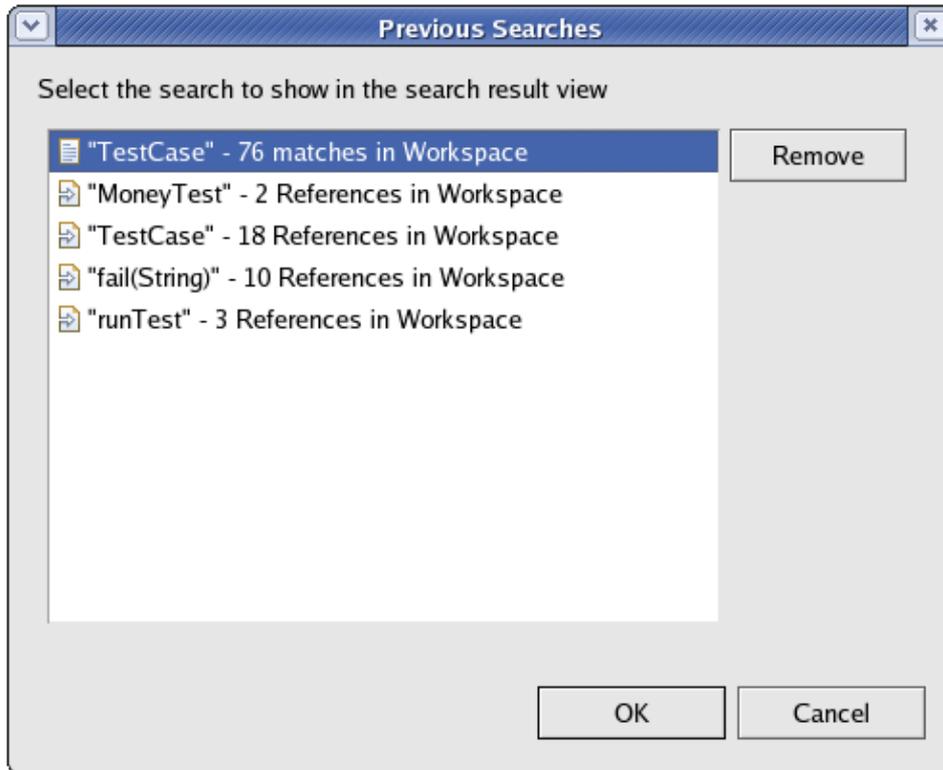
Note: To find all files with a given file name pattern, leave the **Containing Text** field empty.

Viewing previous search results

In the **Search Results** view, click the arrow next to the *Previous Search Results* button to see a menu containing the list of the most recent searches. The list can be cleared by invoking *Clear History* from this menu.



The *Previous Search Results* button will display a dialog with the list of all previous searches from the current session.

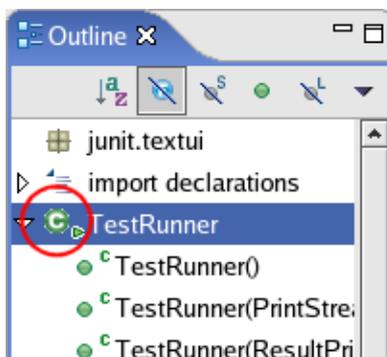


Select any one of these previous searches either in the menu or in the dialog to review the results of the selected search.

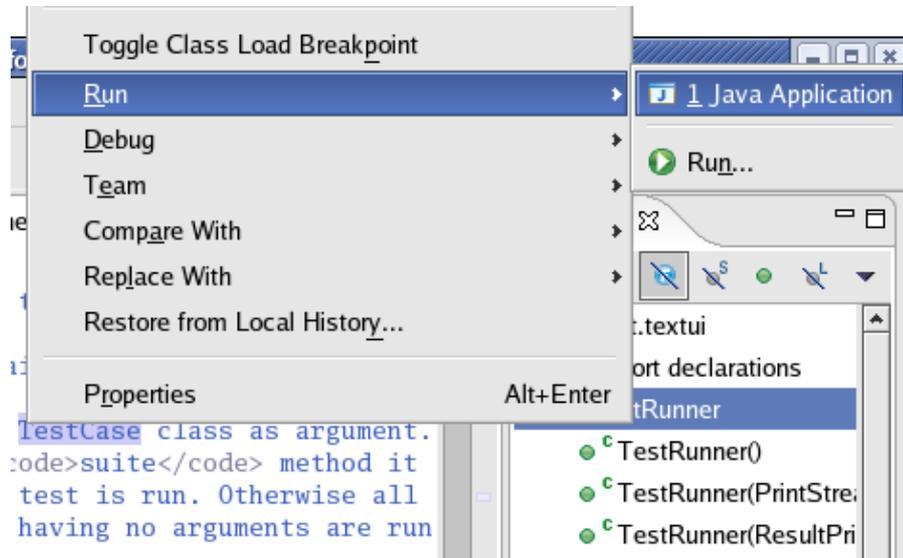
Running your programs

In this section you will learn more about running Java programs in the workbench.

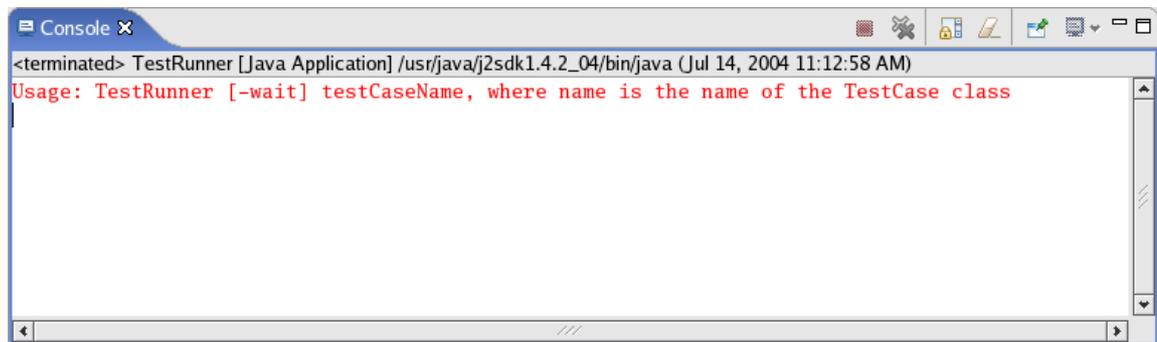
1. In the **Package Explorer** view, select `junit.textui > TestRunner.java`, right-click it and select **Open** to open it in an editor.
2. In the **Outline** view, notice that the `TestRunner` class has an icon that indicates that this class defines a `main` method.



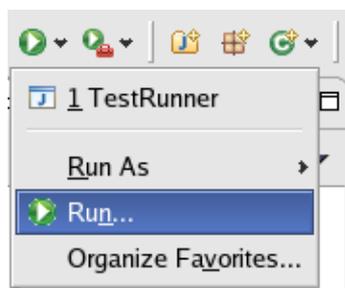
3. Using the drop-down **Run** button in the toolbar, select **Java Application** from the cascading **Run As** menu. This launches the class in the active editor, or the selected class in the **Navigator**, as a local Java application.



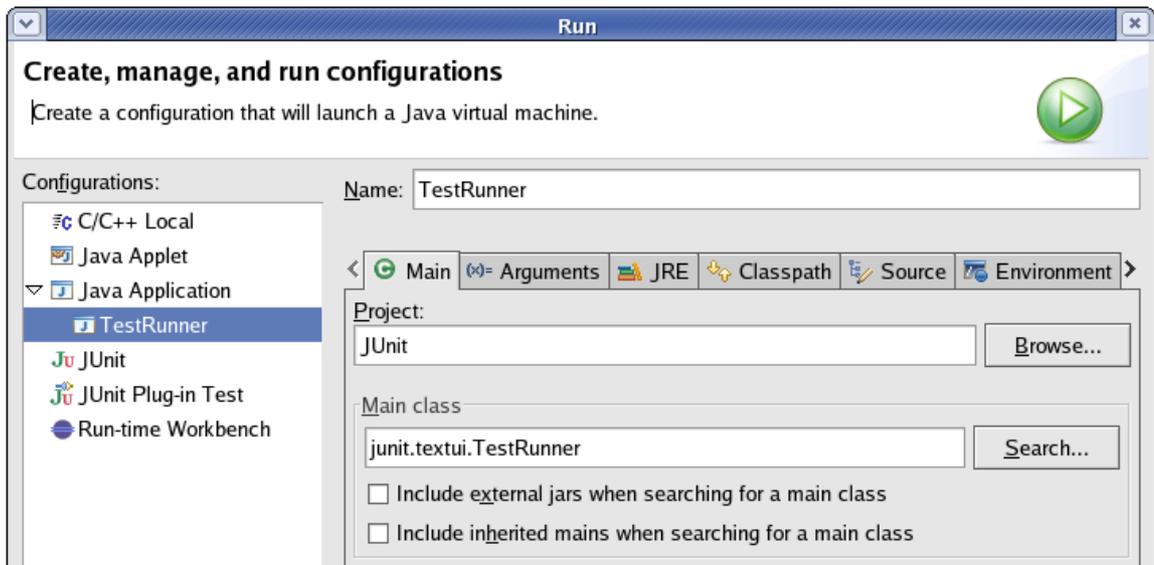
Notice that the program has run and the following message appears in the Console view telling you that the program needs an execution argument.



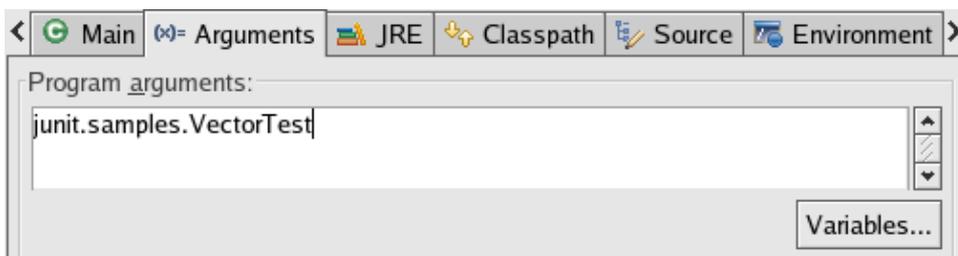
- Using the drop-down **Run** menu in the toolbar, select **Run**.



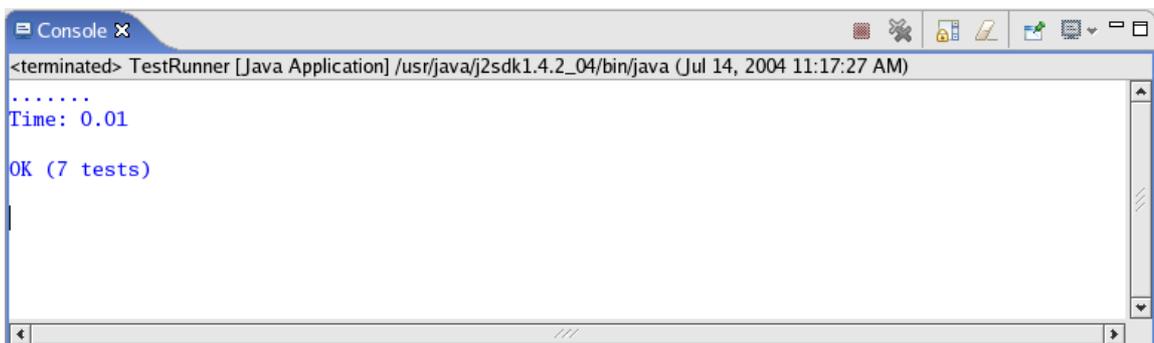
The **Launch Configurations** dialog opens and the TestRunner launch configuration is selected. When you ran the program using the Java Application shortcut, a launch configuration was automatically created with default settings to launch the selected class.



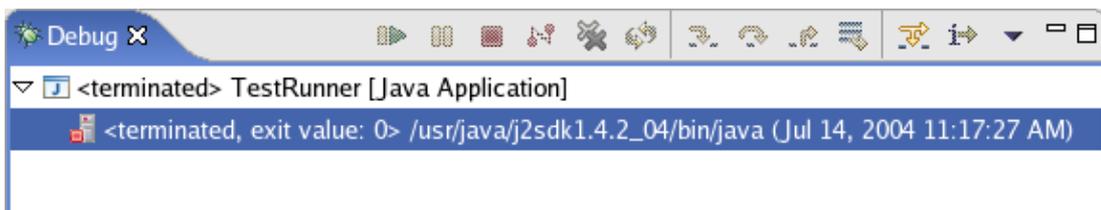
5. Select the Arguments tab and type `junit.samples.VectorTest` in the *Program* arguments area.



6. Click **Run**. This time the program runs correctly and the Console view indicates the number of tests that were run.

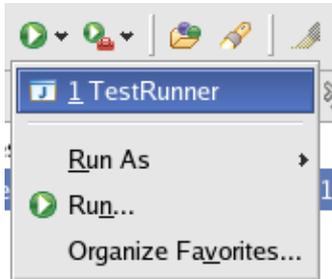


7. Switch to the **Debug** perspective. In the **Debug** view, notice that a launch was registered each time the program was run.

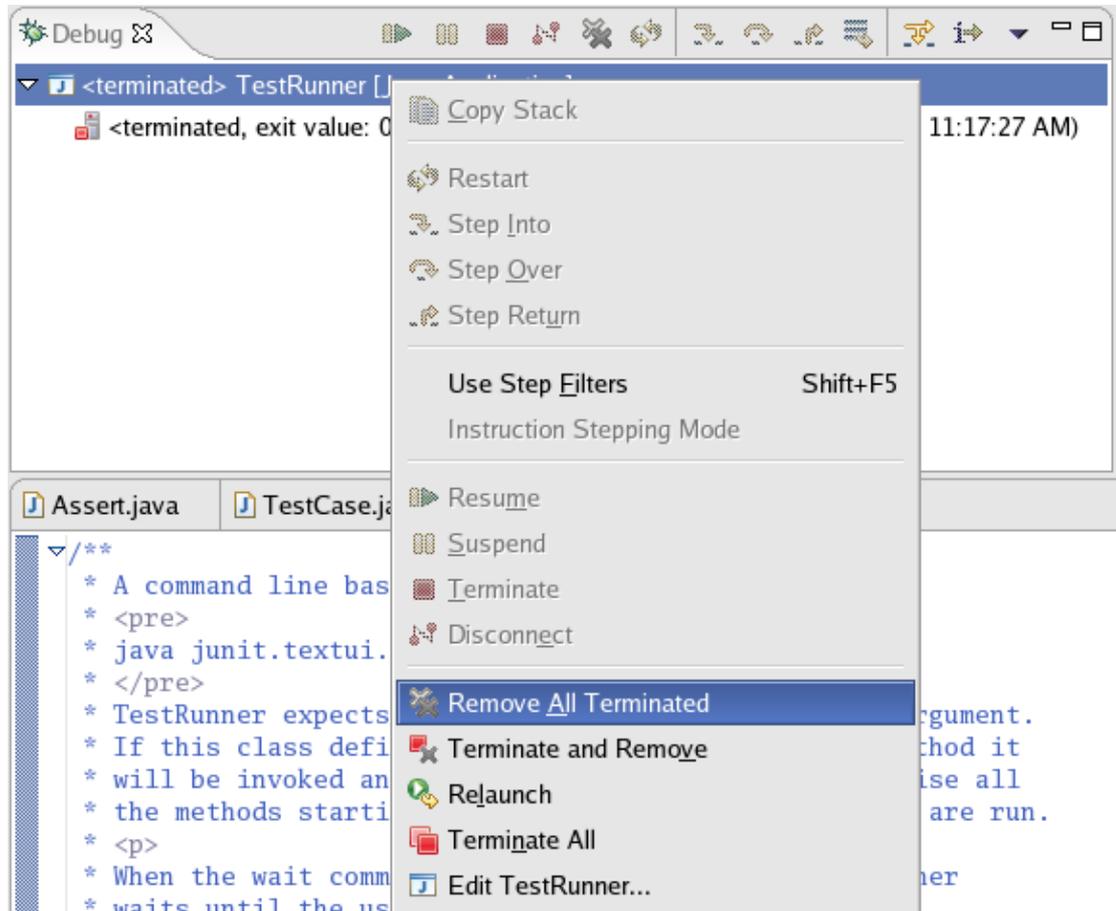


Note: You can relaunch any of these processes by selecting **Relaunch** from its context menu.

8. Select the drop-down menu from the **Run** button in the workbench toolbar. This list contains the previously launched configurations. Configurations can be relaunched by selecting them in this history list as well.



9. From the context menu in the **Debug** view (or the equivalent toolbar button), select **Remove All Terminated** to clear the view of terminated launches.



Debugging your programs

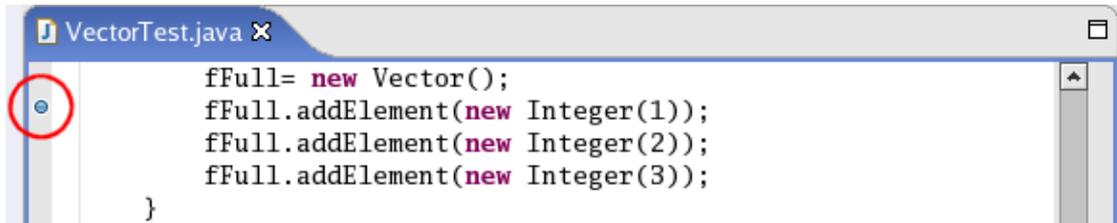
In this section, you will debug a Java program.

1. In the Package Explorer view in the **Java** perspective, select **junit.samples > VectorTest.java**, right-click it and select Open to open it in an editor.

2. Place your cursor on the marker bar (along the left edge of the editor area) on the following line in the setUp() method:

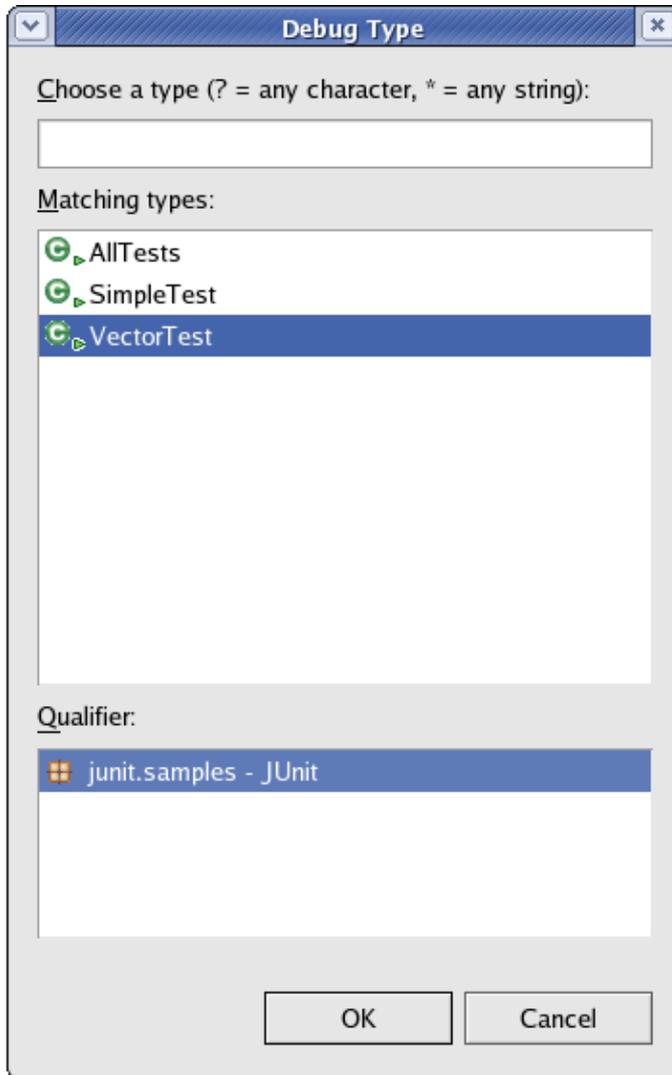
```
fFull.addElement (new Integer(1));
```

and double-click to set a breakpoint.



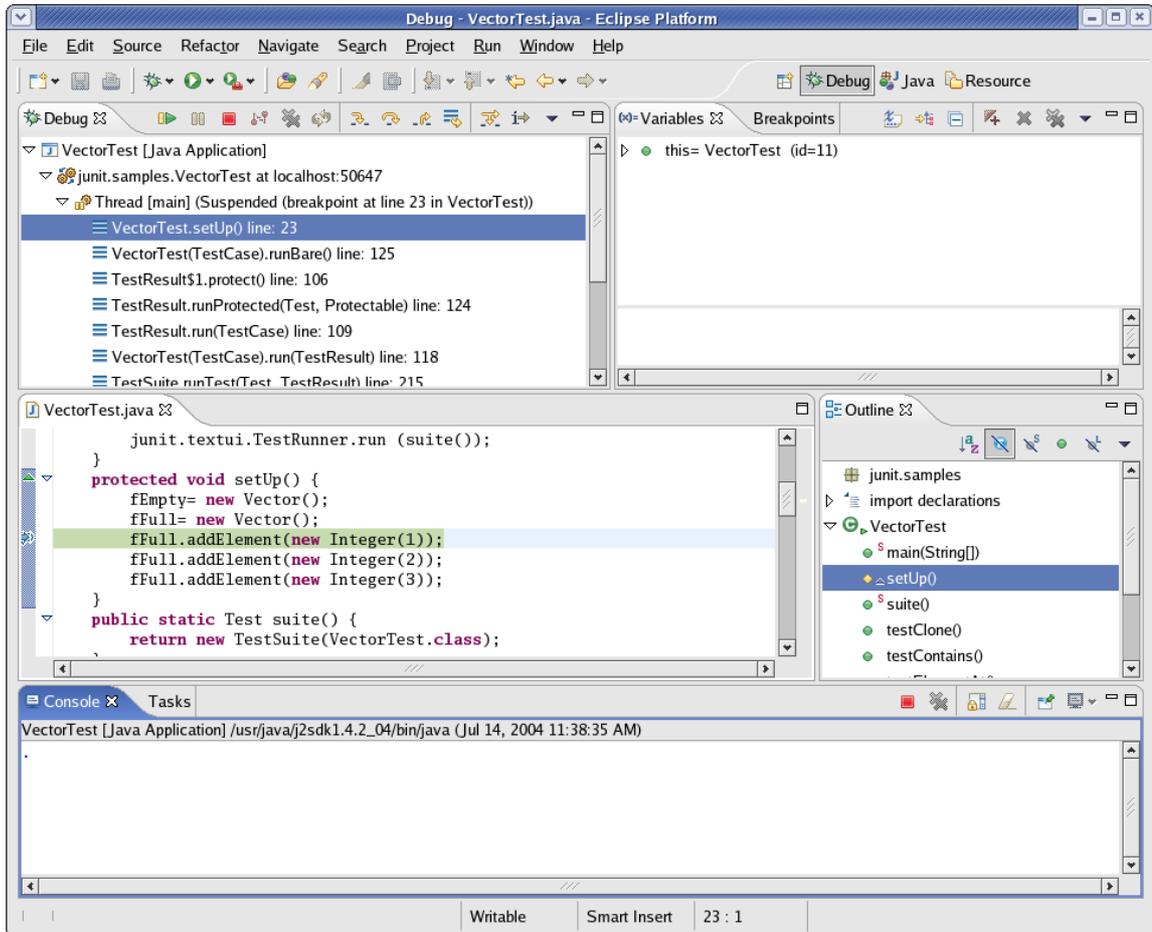
Note: The breakpoint is blue with no checkmark overlay as the breakpoint is not yet installed, meaning that the containing class has not yet been loaded by the Java VM.

3. In the **Package Explorer** view, select the junit.samples package. From the menu bar, select **Run > Debug As > Java Application**. When you run a program from a package, you are prompted to choose a type from all classes in the package that define a main method.
4. Select the **VectorTest – junit.samples – /JUnit** item in the dialog, then click OK.



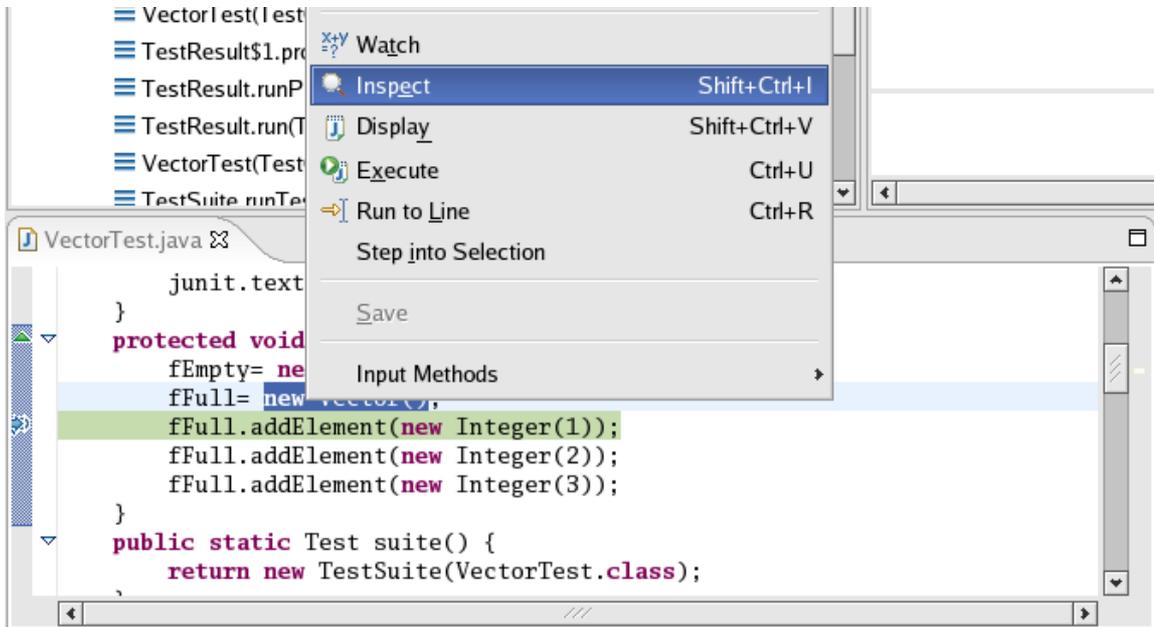
5. As soon as the breakpoint is hit, the **Debug** perspective opens, and execution is suspended. Notice that the process is still active (not terminated) in the Debug view. Other threads might still be running.

Eclipse Tutorials



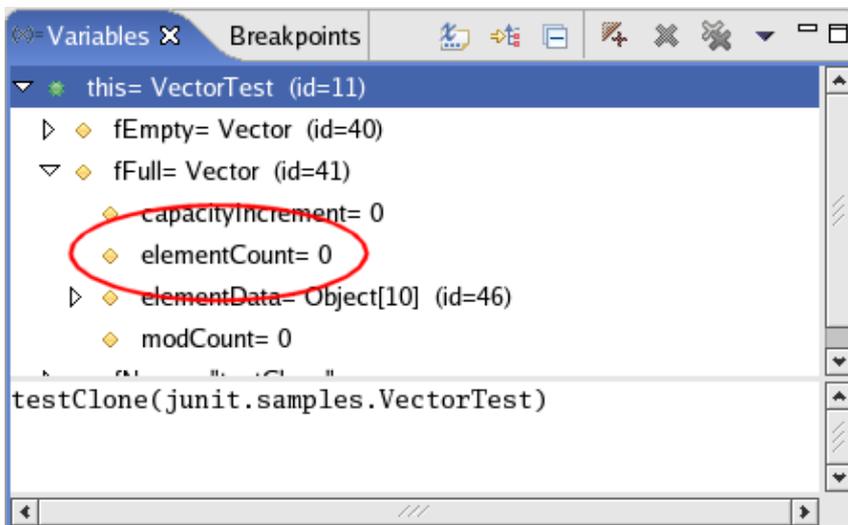
Note: The breakpoint has an overlay icon of a checkmark because it is now installed (VectorTest has been loaded in the Java VM).

6. In the editor in the **Debug** perspective, select the entire line where the breakpoint is set, and from its context menu, select **Inspect**.



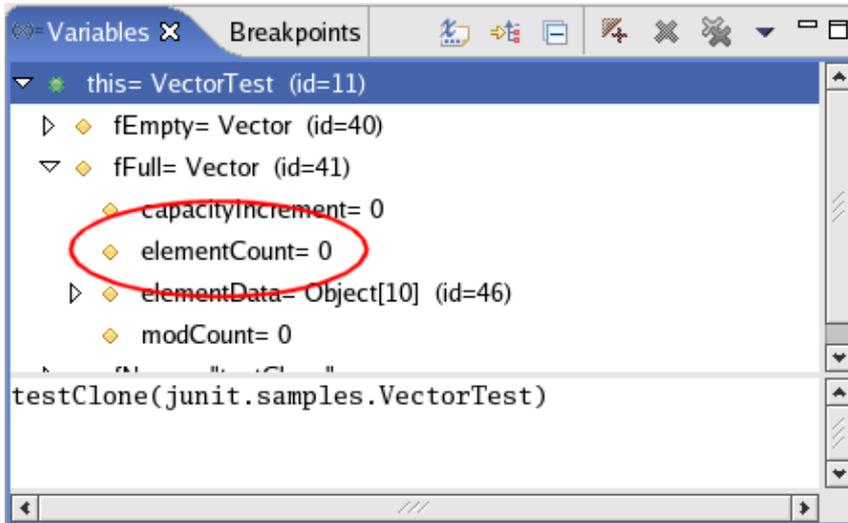
The expression is evaluated in the context of the current stack frame, and a pop-up displays the results. You can send a result to the Expressions view from the pop-up.

7. To delete an expression after working with it, select the expression in the **Expressions** view, right-click on it, and select **Remove**.
8. The **Variables** view displays the values of the variables in the selected stack frame. Expand the **fFull** tree in the **Variables** view until you can see **elementCount**.



9. Watch the variables (for example, **elementCount**) in the **Variables** view as you step through **VectorTest** in the **Debug** view.

Click the **Step Over** button to step over the highlighted line of code. Execution will continue at the next line in the same method (or, if you are at the end of a method, it will continue in the method from which the current method was called).



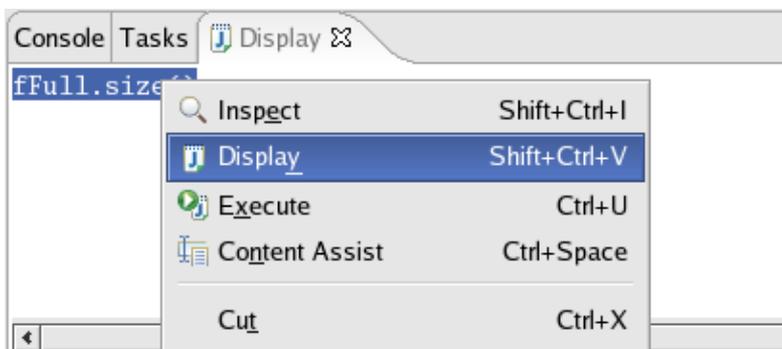
10. Try some other step buttons (Step Into, Step Return) to step through the code. Note the differences in stepping techniques.
11. You can end a debugging session by allowing the program to run to completion or by terminating it.
 - ◆ You can continue to step over the code with the Step buttons until the program completes.
 - ◆ You can click the Resume button to allow the program to run until the next breakpoint is encountered or until the program is completed.
 - ◆ You can right-click on the program's process in the Debug view and select Terminate from the context menu to terminate the program.

Evaluate expressions

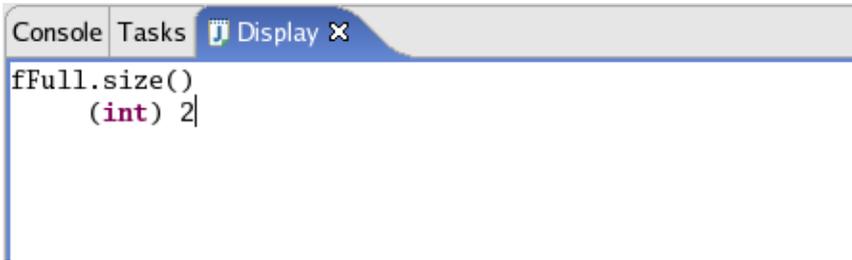
In this section, you will evaluate expressions in the context of your running Java program.

1. Debug `junit.samples.VectorTest.java` to the breakpoint in the `setUp()` method and select **Step Over** twice to populate `fFull`. (See [Debugging your Programs](#) for full details.)
2. Click **Window > Show View > Display** and type the following line in the Display view:


```
fFull.size()
```
3. Select the text you just typed, right-click, and from its context menu, select **Display**. (You can also choose **Display Result of Evaluating Selected Text** from the Display view toolbar.)



4. The expression is evaluated and the result is displayed in the Display view.

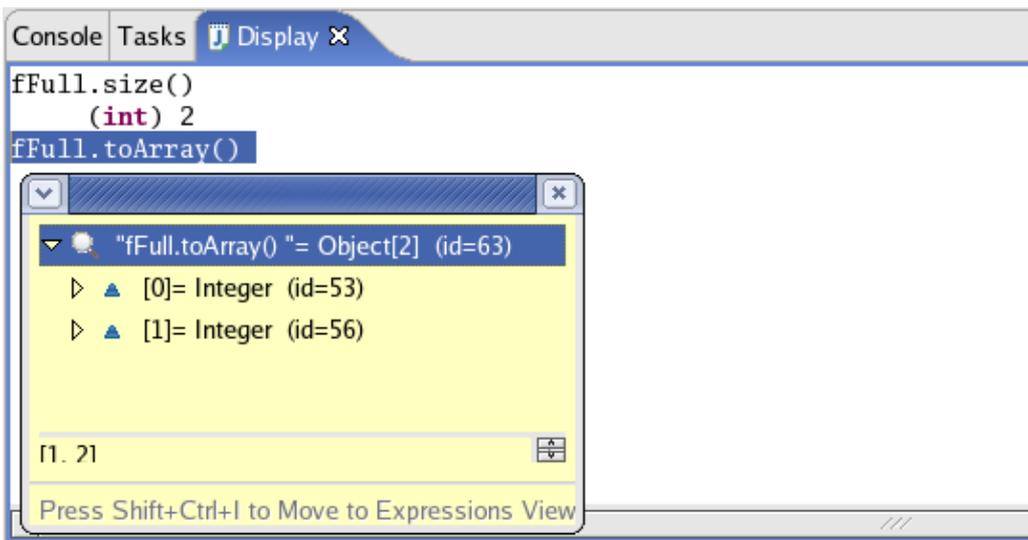


5. On a new line in the Display view, type the following line:

```
fFull.toArray()
```

6. Select this line, right-click, and select **Inspect** from the context menu. (You can also choose **Inspect Result of Evaluating Selected Text** from the Display view toolbar.)

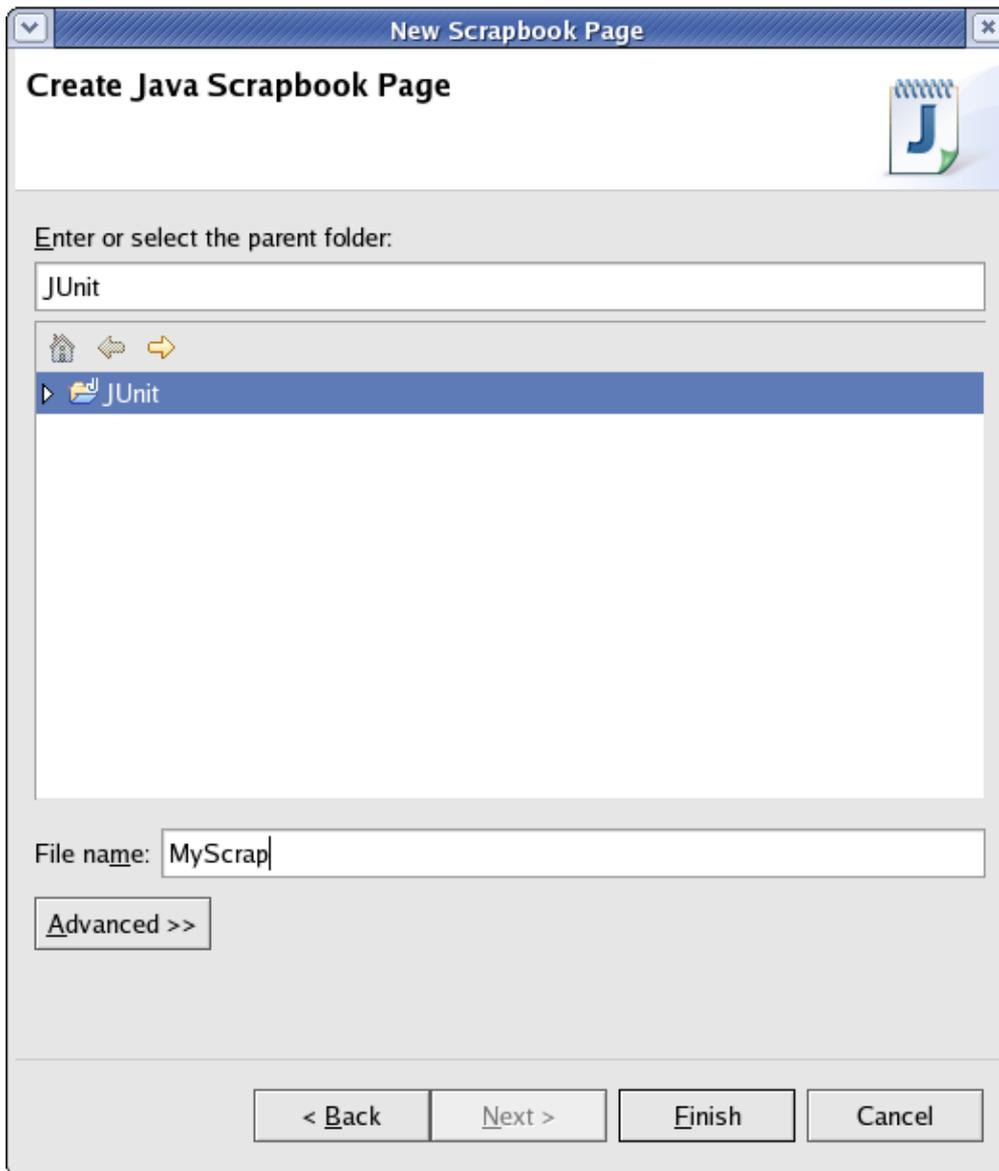
7. A pop-up opens with the value of the evaluated expression.



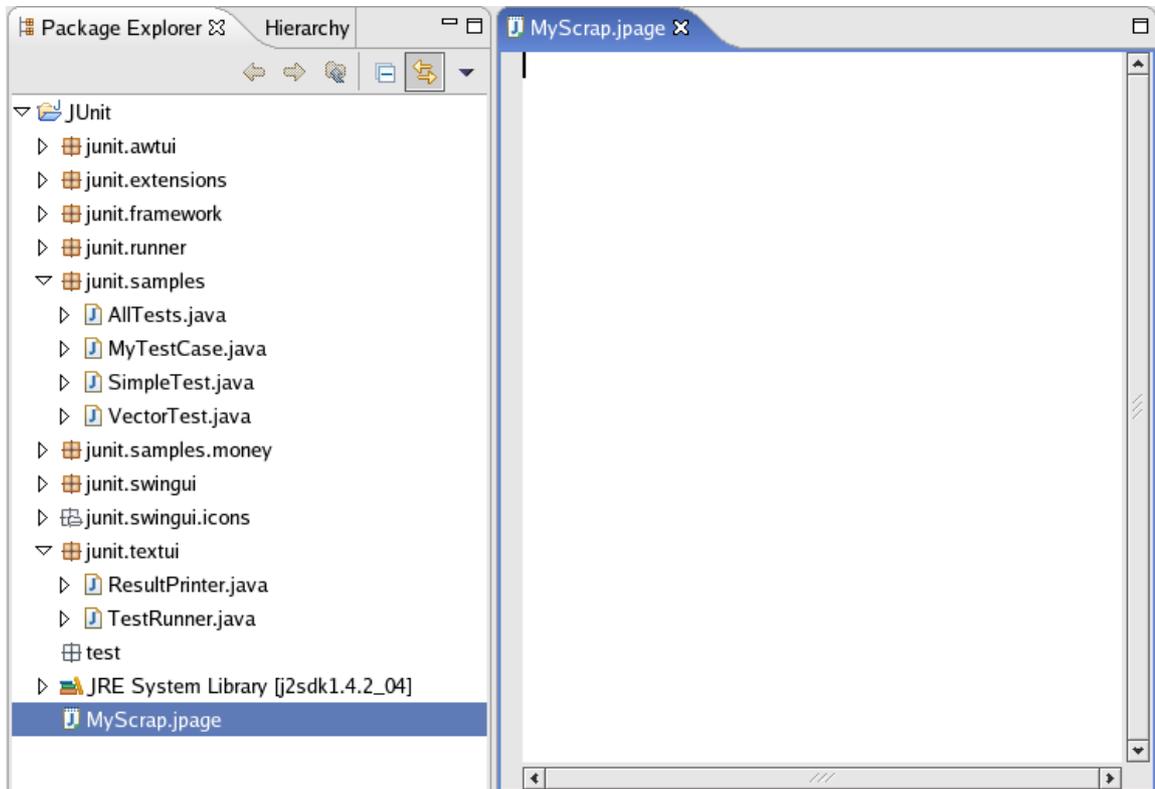
Evaluating snippets

In this section, you will evaluate Java expressions using the Java scrapbook. Java scrapbook pages allow you to experiment with Java code fragments before putting them in your program.

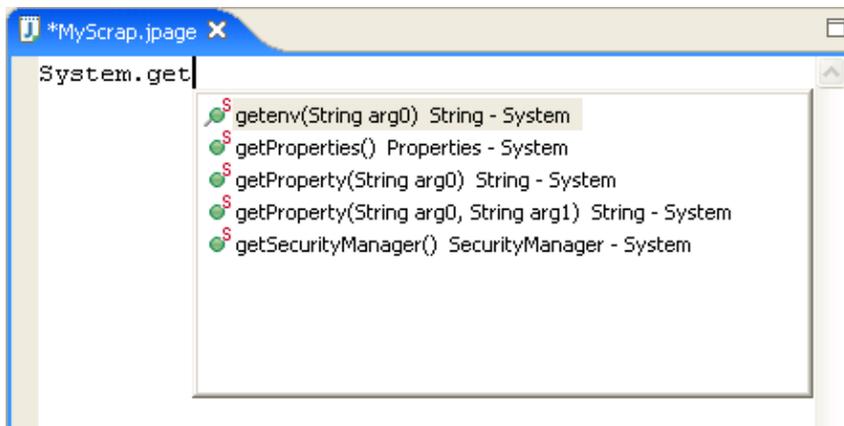
1. Click **New > Other > Java > Java Run/Debug > Scrapbook Page**. You are prompted for a folder destination for the page.



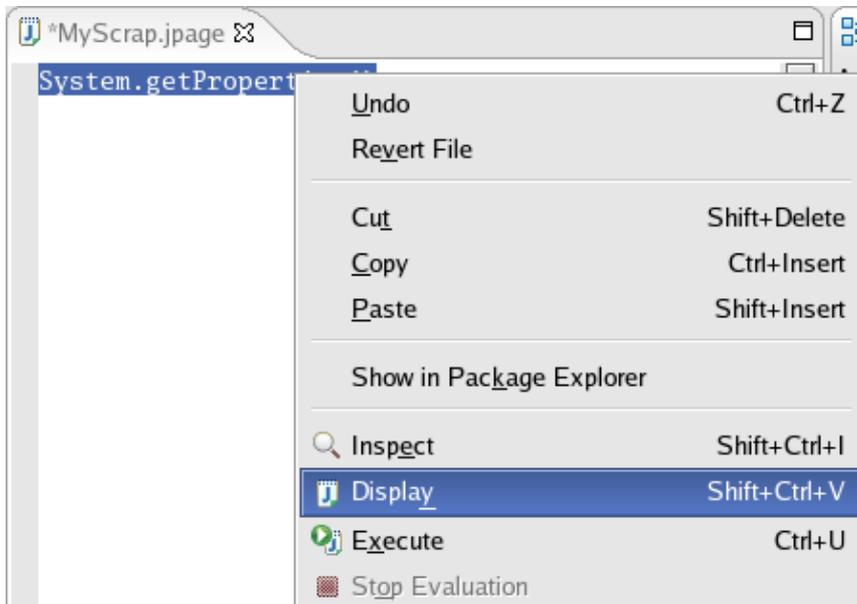
2. In the *Enter or select the folder* field, type or browse to select the `JUnit` project root directory.
3. In the *File name* field, type **MyScrap**.
4. Click **Finish** when you are done. A scrapbook page resource is created for you with the `.jpage` file extension. (The `.jpage` file extension is added automatically if you do not enter it yourself.) The scrapbook page opens automatically in an editor.



5. In the editor, type `System.get` and then use content assist [`Ctrl`]+[`Space`] to complete the snippet as `System.getProperties()`.



6. Select the entire line you just typed and select **Display** from the context menu. You can also select **Display Result of Evaluating Selected Text** from the toolbar.

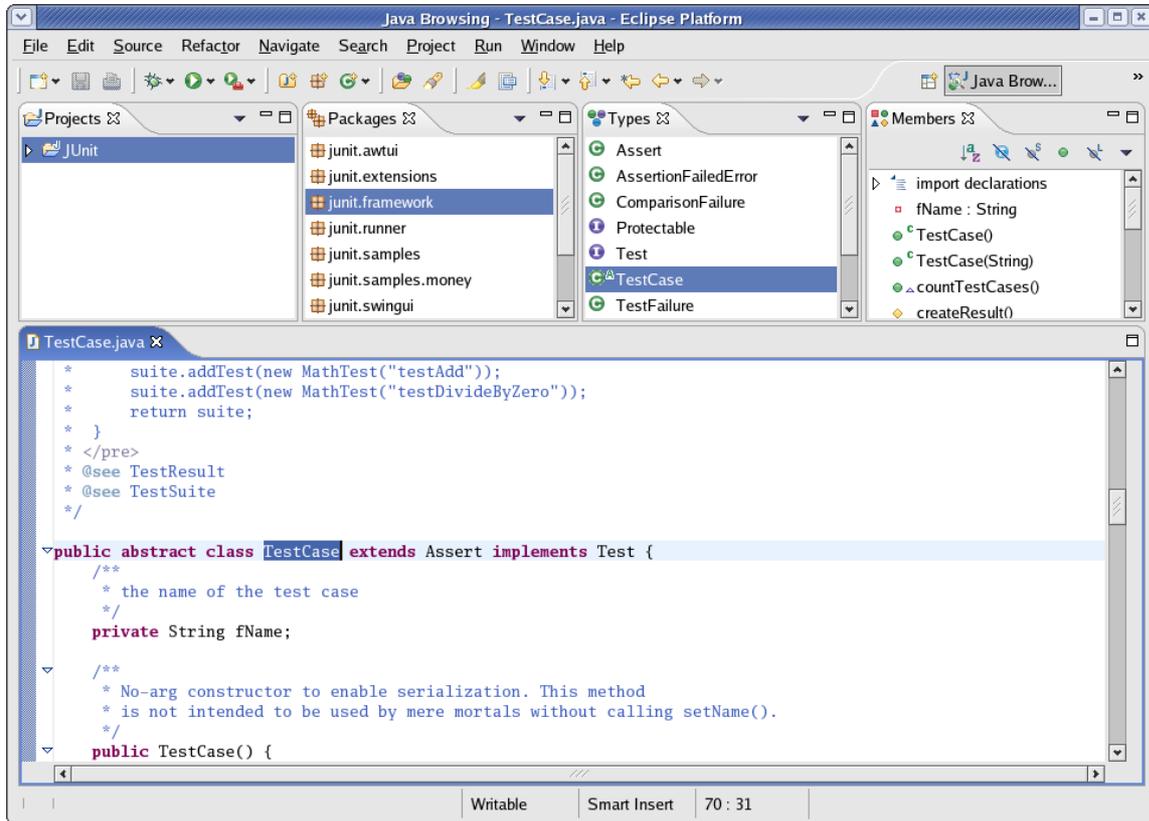


7. When the evaluation completes, the result of the evaluation is displayed and highlighted in the scrapbook page.
8. You can inspect the result of an evaluation by selecting text and choosing *Inspect* from the context menu (or selecting *Inspect Result of Evaluating Selected Text* from the toolbar.)
9. When you are finished evaluating code snippets in the scrapbook page, you can close the editor. If you want to keep the snippets for future use, save the changes in the page.

Using the Java browsing perspective

In this section you will use the Java browsing perspective to browse and manipulate your code. *Browsing Java elements with the Package Explorer* gives an overview of using the **Package Explorer** to browse elements. In contrast to the **Package Explorer**, which organizes all Java elements in a tree, consisting of projects, packages, compilation units, types, etc., the browsing perspective uses distinct views to present the same information. Selecting an element in one view, will show its content in another view.

To open a browsing perspective, activate *Window > Open Perspective > Java Browsing* from within the Java perspective or use the context menu of the *Open a Perspective* toolbar button.



The views of the perspective are connected to each other in the following ways:

- Selecting an element in the **Projects** view shows its packages in the **Packages** view.
- The **Types** view shows the types contained in the package selected in the **Packages** view.
- The **Members** view shows the members of a selected type. Functionally, the **Members** view is comparable to the **Outline** view used in the normal Java perspective.
- Selecting an element in the **Members** view reveals the element in the editor. If there is no editor open for the element, double-clicking on the element opens a corresponding editor.

All four views are by default linked to the active editor. This means that the views will adjust their content and their selection according to the file presented in the active editor. The following steps illustrate this behavior:

1. Select `junit.extensions` in the **Packages** view.
2. Open type `TestSetup` in the editor by double-clicking it in the **Types** view.
3. Now give back focus to the editor opened on file `TestCase.java` by clicking on the editor tab. The **Packages**, **Types**, and **Members** view adjust their content and selections to reflect the active editor. The **Packages** view's selection is set to `junit.framework` and the **Types** view shows the content of the `junit.framework` packages. In addition, the type `TestCase` is selected.

Functionally, the Java browsing perspective is fully comparable to the Java perspective. The context menus for projects, packages, and types as well as the global menu and tool bar are the same. Therefore activating these functions is analogous to activating them in the Java perspective.

Writing and running JUnit tests

In this section, you will be using the **JUnit** testing framework to write and run tests. To get started with JUnit, you can refer to the *JUnit Cookbook* at <http://junit.sourceforge.net/doc/cookbook/cookbook.htm>.

Writing Tests

Before you can write JUnit tests, you have to add the *junit.jar* library to your build class path. The Eclipse installation includes JUnit in the org.junit plug-in.

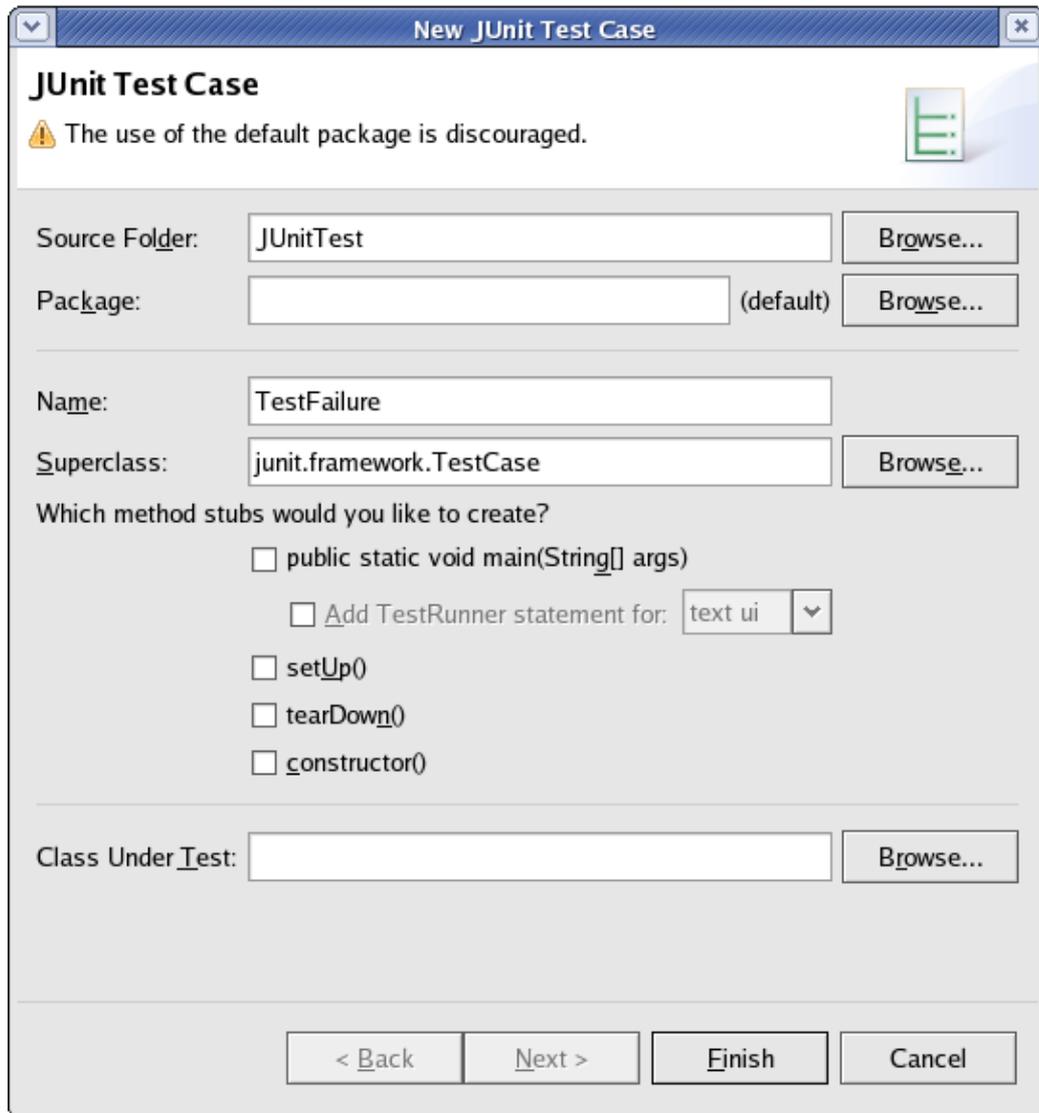
The JUnit package is installed in the /usr/share/eclipse/plugins/org.junit_3.8.1/ directory.

1. Create a Java project "JUnitTest"
2. Open the project's build path property page (on the project's context menu choose *Properties > Java Build Path*).
3. Switch to the **Libraries** tab.
4. Add the junit.jar contained in org.junit in the plug-ins directory as an external JAR to your project.

Optionally, if you want to browse the JUnit source, then attach the junitsrc.zip to the junit.jar. The source zip is located in the org.eclipse.jdt.source plug-in in src/org.junit_3.8.1.

Now that the JUnitTest project has access to the JUnit classes, you can write your first test. You implement the test in a subclass of *TestCase*. You can do so either using the standard Class wizard or the specialized *TestCase* wizard:

1. Open the New wizard (*File > New > JUnit Test Case*).
2. Enter "TestFailure" as the name of your test class:



3. Click **Finish** to create the test class.

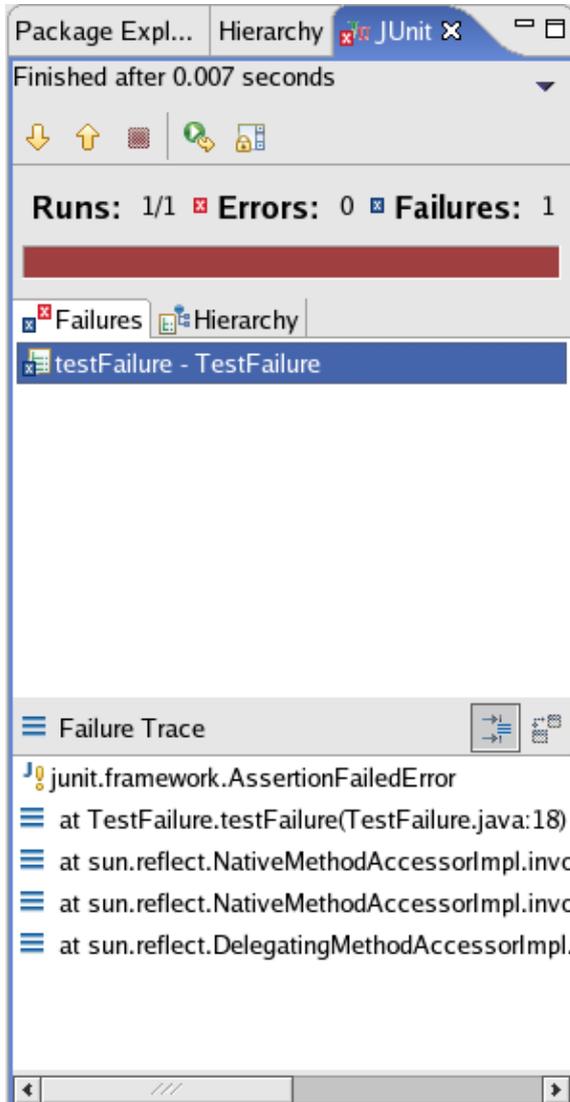
Add a test method that fails to the class `TestFailure`. A quick way to enter a test method is with the test template. To do so, type "test" followed by [Ctrl]+[Space] to activate code assist and select the "test" template. Change the name of the created method to `testFailure` and invoke the `fail()` method. Also change the visibility modifier such that the test method is public.

```
public void testFailure() {
    fail();
}
```

Now you are ready to run your first test.

Running Tests

To run `TestFailure`, click **Run > Run as > JUnit Test**. You can inspect the test results in the **JUnit** view. This view shows you the test run progress and status:



The view is shown in the current perspective whenever you start a test run. A convenient arrangement for the **JUnit** view is to dock it as a fast view. The **JUnit** view has two tabs: one shows you a list of failures and the other shows you the full test suite as a tree. You can navigate from a failure to the corresponding source by double clicking the corresponding line in the failure trace.

Dock the **JUnit** view as a fast view, remove the `fail()` statement in the method `testFailure()` so that the test passes, and rerun the test again. You can rerun a test either by clicking the **Rerun** button in the view's tool bar or you can re-run the program that was last launched by clicking the **Run** drop down. This time the test should succeed. Because the test was successful, the **JUnit** view does not pop up, but the success indicator shows on the **JUnit** view and the status line shows the test result. As a reminder to rerun your tests, the view icon is decorated by a "*" whenever you change the workspace contents after a run.



– A successful test run



– A successful test run, but the workspace contents has changed since the last test run.

In addition to running a test case as described above you can also:

Run all tests inside a project, source folder, or package:

Select a project, package or source folder and run all the included tests with **Run as > JUnit Test**. This command finds all tests inside a project, source folder or package and executes them.

Run a single test method:

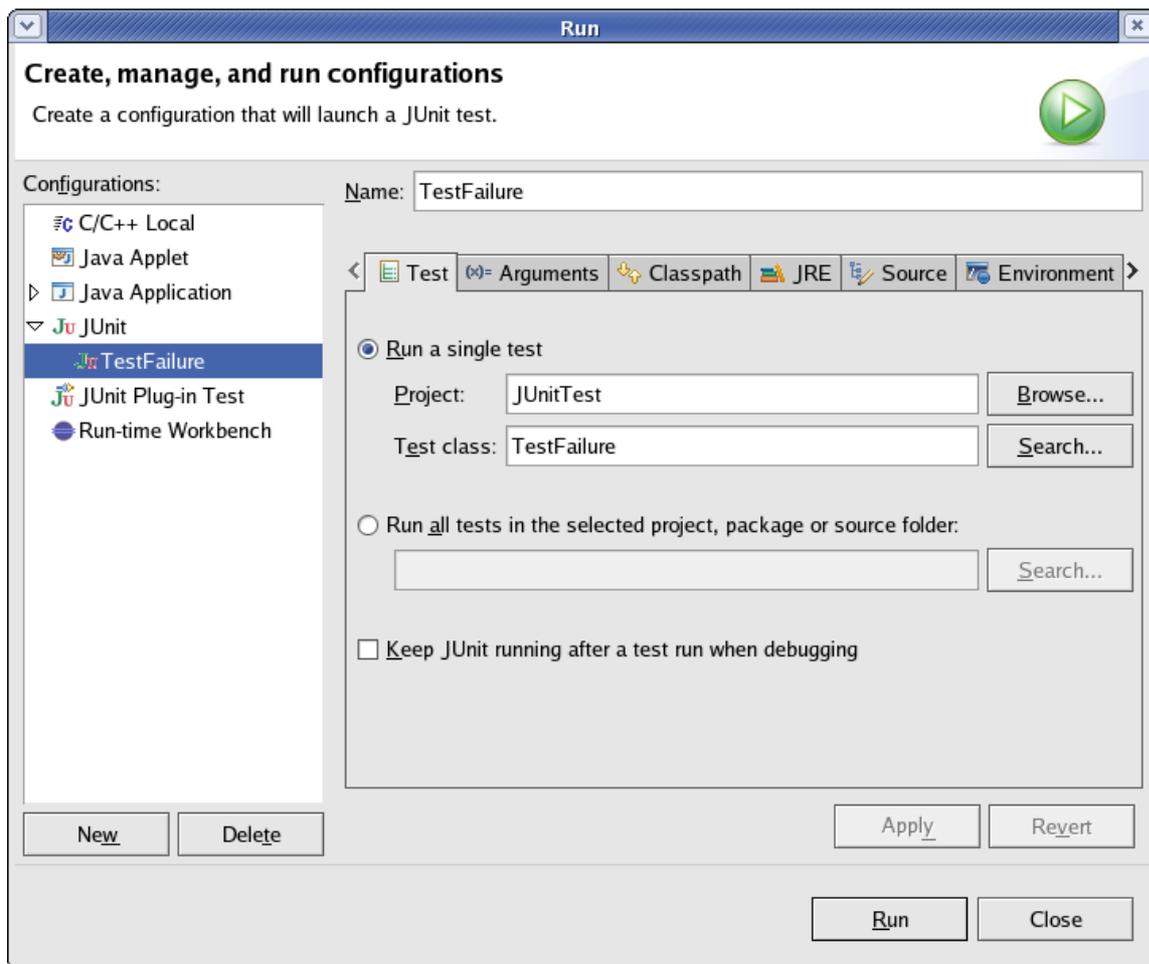
Select a test method in the **Outline** or **Package Explorer** and with **Run as > JUnit Test**. The selected test method is run.

Rerun a single test:

Select a test in the **JUnit** view, right-click, and select **Rerun** from the context menu.

Customizing a Test Configuration

When you want to pass parameters or customize the settings for a test run you open the **Launch Configuration** dialog. Click **Run > Run** :



In this dialog you can specify the test to be run, its arguments, its run-time class path, and the Java run-time environment.

Debugging a Test Failure

In the case of a test failure, you can follow these steps to debug it:

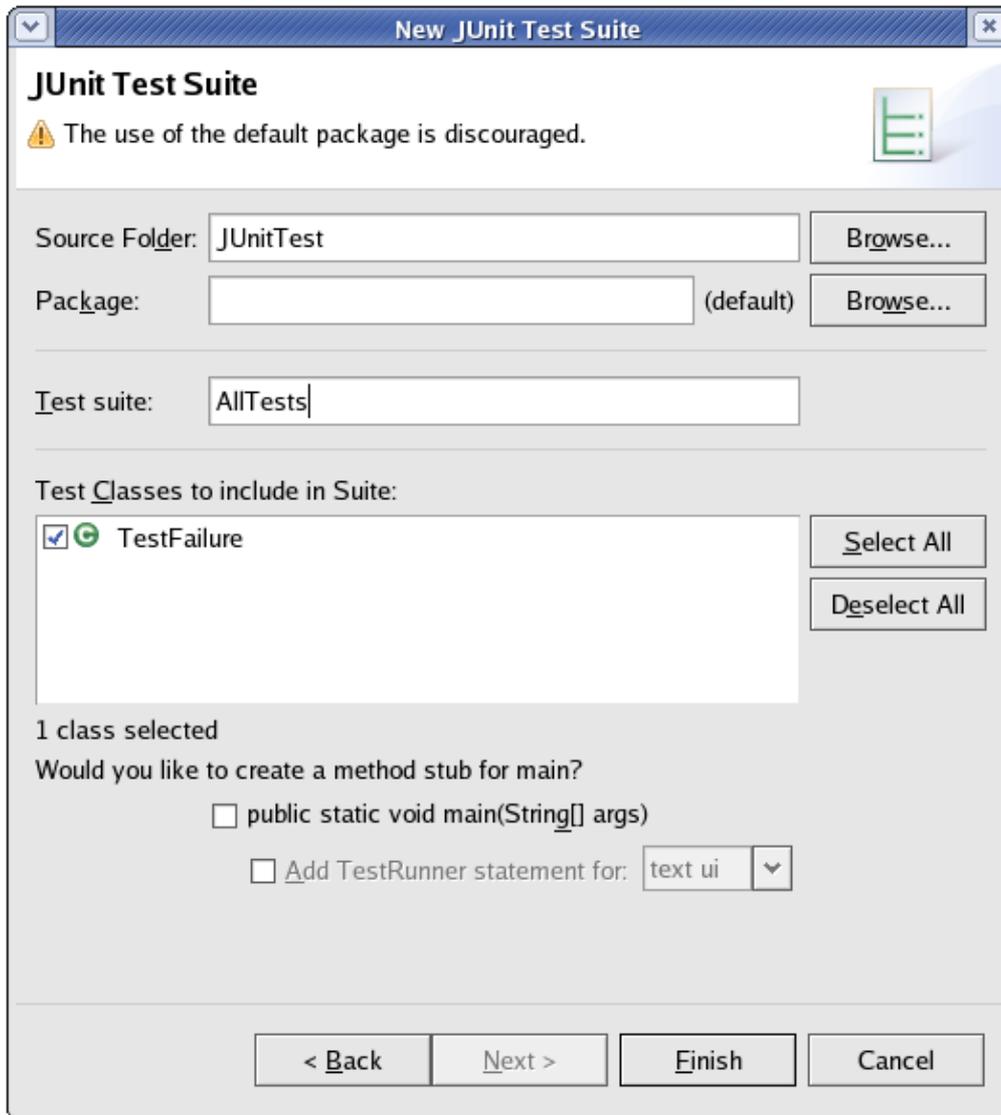
1. Double-click the failure entry from the stack trace in the **JUnit** view to open the corresponding file in the editor.
2. Set a breakpoint at the beginning of the test method.
3. Select the test case and execute **Debug As>JUnit Test** from the **Debug** drop down.

A JUnit launch configuration has a "keep alive" option. If your Java virtual machine supports "hot code replacement", you can fix the code and rerun the test without restarting the full test run. To enable this option, select the *Keep JUnit running after a test run when debugging* checkbox in the JUnit launch configuration.

Creating a Test Suite

The JUnit **TestSuite** wizard helps you with the creation of a test suite. You can select the set of classes that should belong to a suite.

1. Click **File > New**.
2. Select **Java > JUnit > JUnit Test Suite** and click *Next*.
3. Enter a name for your test suite class (the convention is to use "AllTests", which appears by default).



4. Select the classes that should be included in the suite. There is currently only a single test class, but you can add to the suite later.

You can add or remove test classes from the test suite in two ways:

- Manually, by editing the test suite file
- By re-running the wizard and selecting the new set of test classes.

Note: The wizard puts two markers, `// $JUnit-BEGIN$` and `// $JUnit-END$`, into the created Test suite class, which allows the wizard to update existing test suite classes. Editing code between the markers is not recommended.

Project configuration tutorial

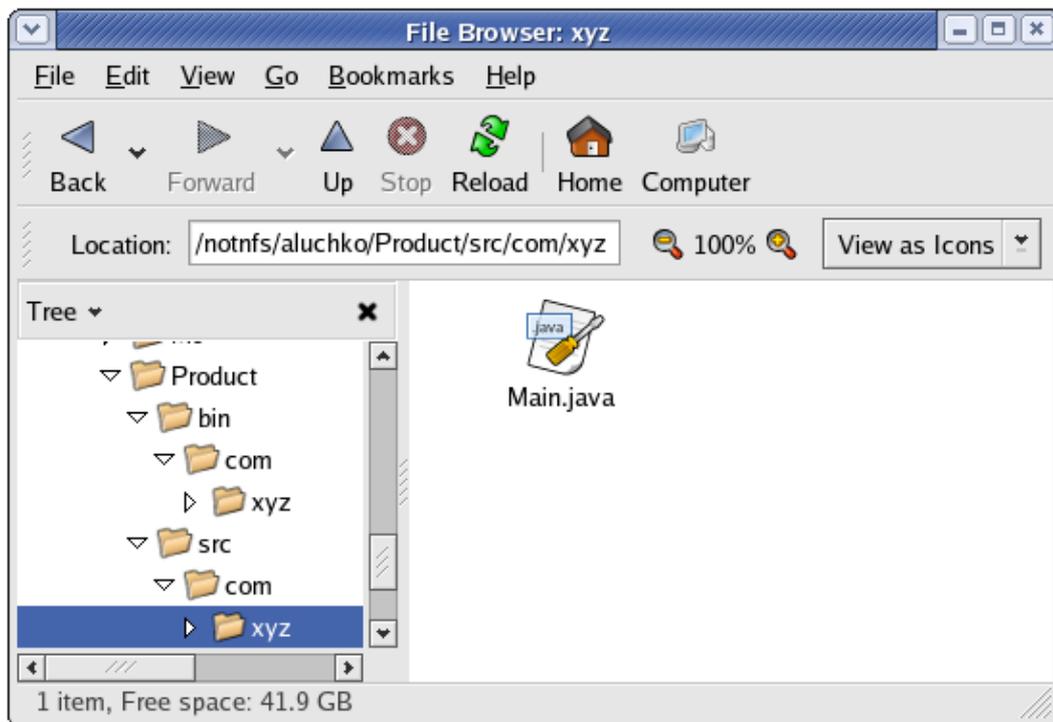
In this section, you will create and configure a new Java project to use source folders and to match some existing layout on the file system. Some typical layouts have been identified. Choose the sub-section that matches your layout.

Detecting existing layout

Layout on file system

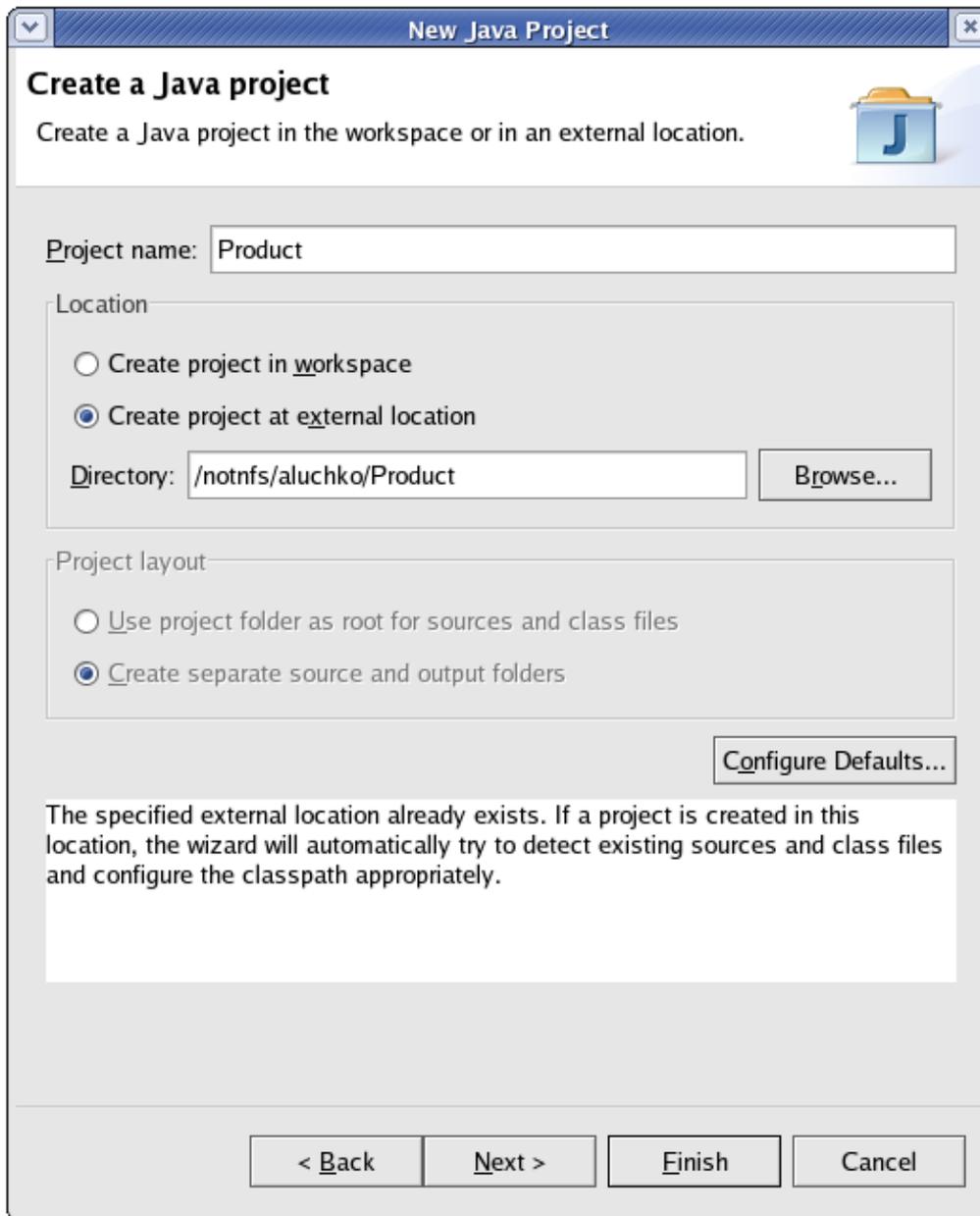
It is necessary to create the following file structure before proceeding further.

- The source files for a product are laid out in one directory "src".
- The class files are in another directory "bin".



Steps for defining a corresponding project

1. Click **File > New > Project** to open the **New Project** wizard.
2. Select **Java project** in the list of wizards and click **Next**.
3. Type "Product" in the **Project name** field.
4. In the **Project layout** group, change the selection to **Create separate source and output folders**.
5. In the **Location** group, change the selection to **Create project at external location**.
6. Click **Browse** and choose the "Product" directory.



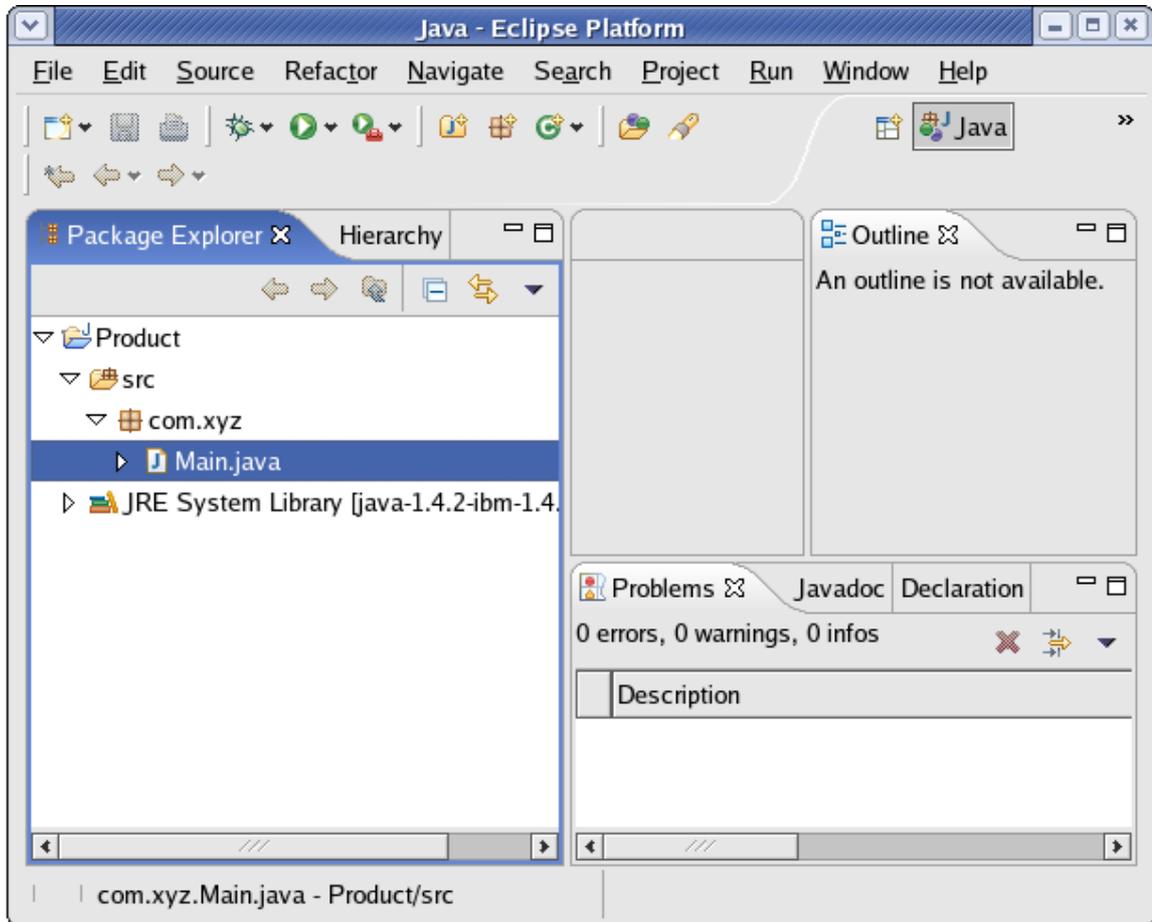
7. Click *Next*.
8. Ensure that the source and output folders are detected. Product/src/com/xyz is sometimes detected instead of Product/src. If this happens, you need to remove Product/src/com/xyz and add Product/src.



Warning: If the preference *Window > Preferences > Java > Compiler > Build Path > Scrub output folder when cleaning projects* is checked, clicking *Finish* will scrub the "bin" directory in the file system before generating the class files.

9. Click *Finish*.

You now have a Java project with a "src" folder that contains the sources of the "Product" directory.

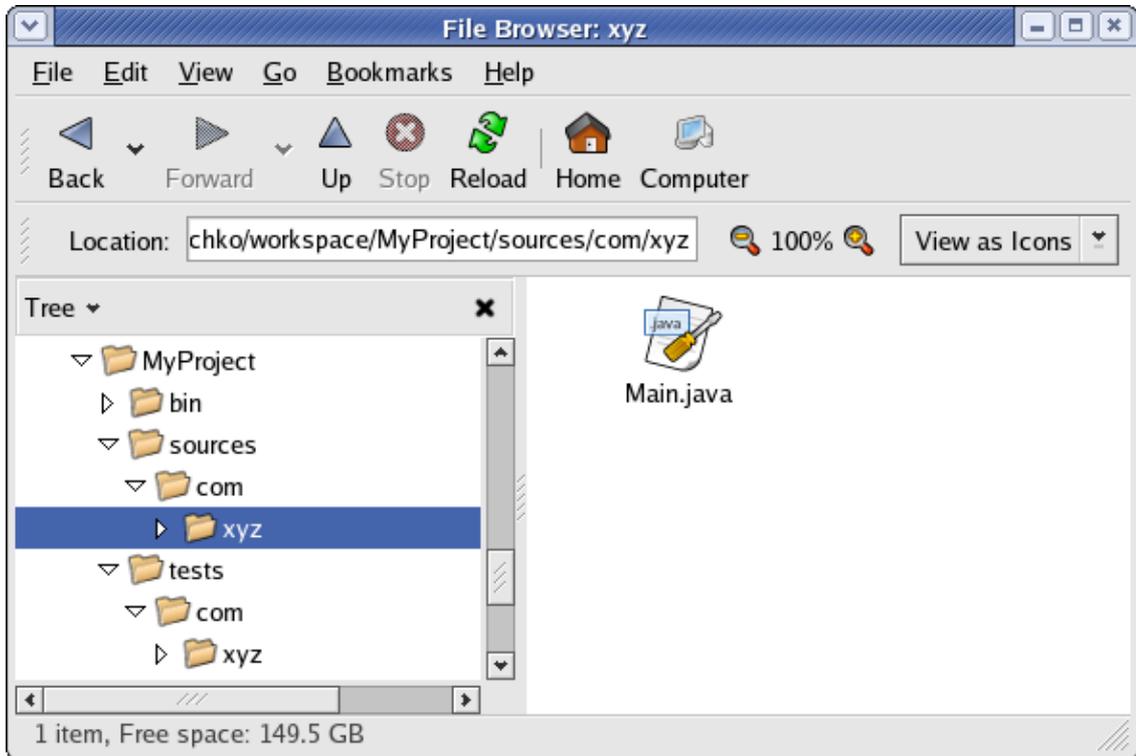


Note: This solution creates a ".project" file and a ".classpath" file in the "Product" directory. If you do not wish to have these files in the "Product" directory, you should use linked folders as shown in the [Sibling products in a common source tree](#) section.

Organizing sources

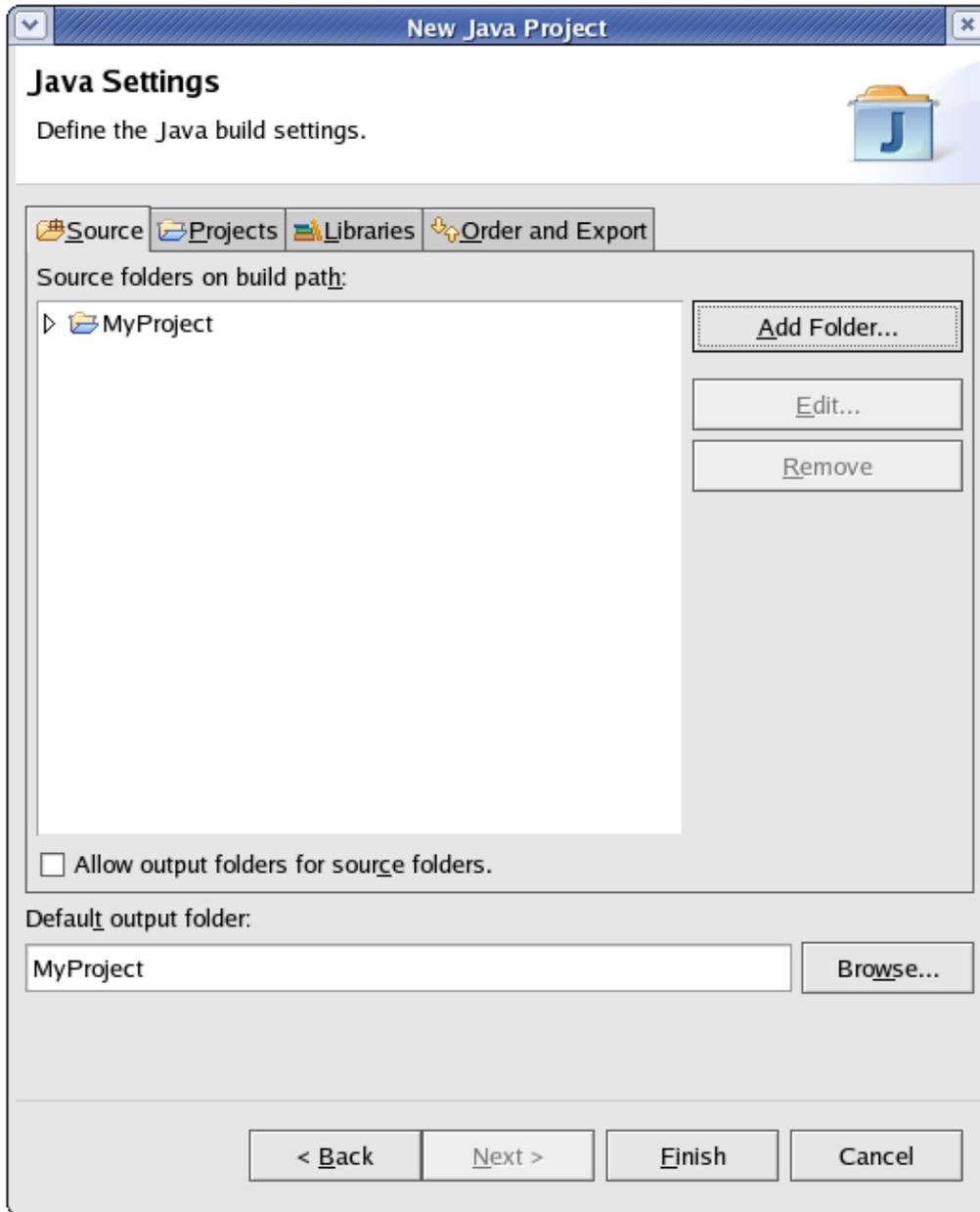
Layout on file system

- In this section, you will create a new Java project and organize your sources in separate folders. This will prepare you for handling more complex layouts.
- Let's assume you want to put your sources in one folder and your tests in another folder to achieve a layout similar to the following:



Steps for defining a corresponding project

1. Open a Java perspective, select the menu item **File > New > Project...** to open the **New Project** wizard.
2. Select **Java project** in the list of wizards and click **Next**.
3. On the next page, type "MyProject" in the **Project name** field. Click **Next**.

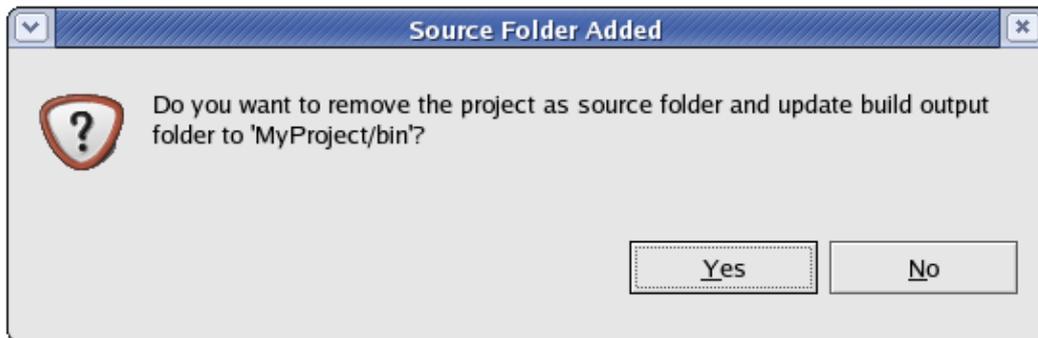


4. Select the "MyProject" source folder and click **Add Folder...**
5. In the **New Source Folder** dialog, type "sources" in the **Folder name** field.



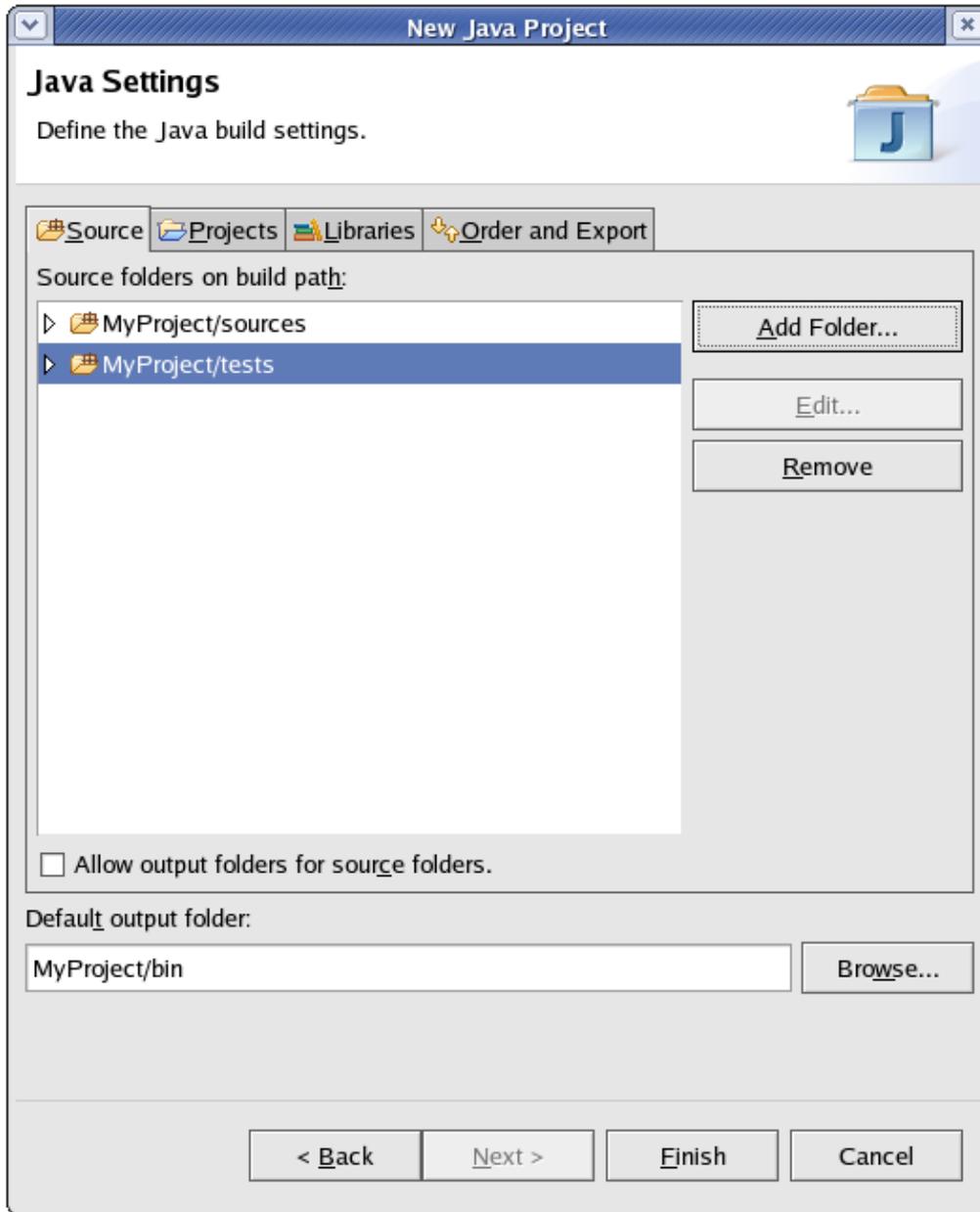
6. Click **OK** to close the dialog.

7. Click *Yes* in confirmation dialog to have "MyProject/bin" as your default output folder.

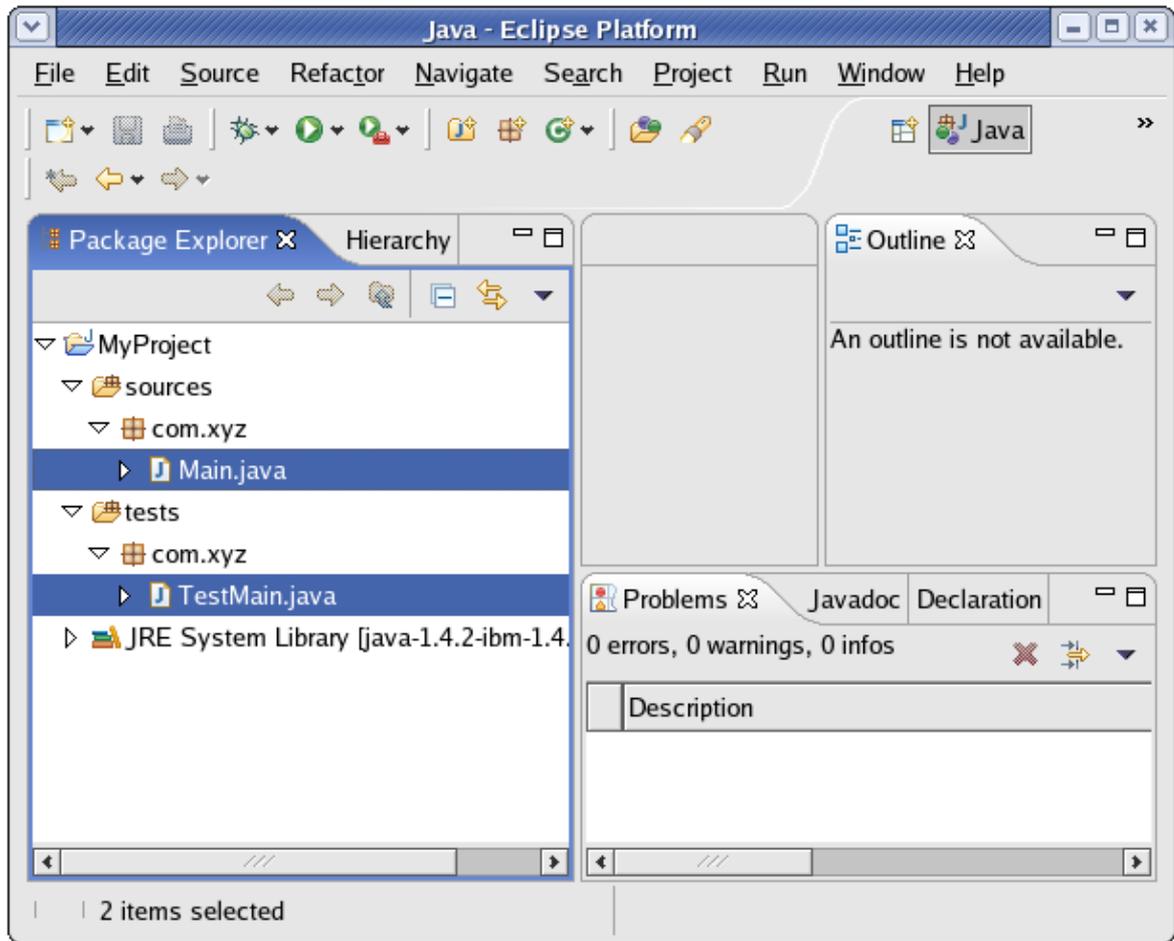


8. Click again on *Add Folder...*
9. In *Source Folder Selection*, click *Create New Folder...*
10. In the *New Folder* dialog, type "tests" in the *Folder name* field.
11. Click *OK* twice to close the two dialogs.

Your project setup now looks as follows:



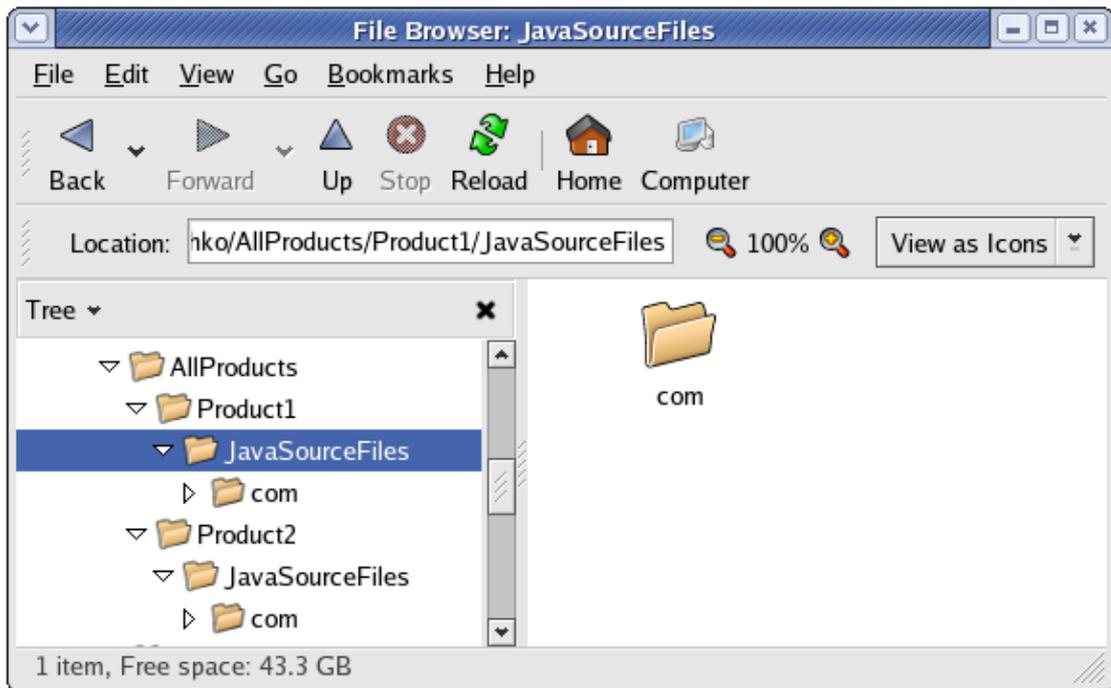
12. Click **Finish**.
13. You now have a Java project with a "sources" and a "tests" folders. You can start adding classes and packages to these folders or you can copy them using drag and drop.



Sibling products in a common source tree

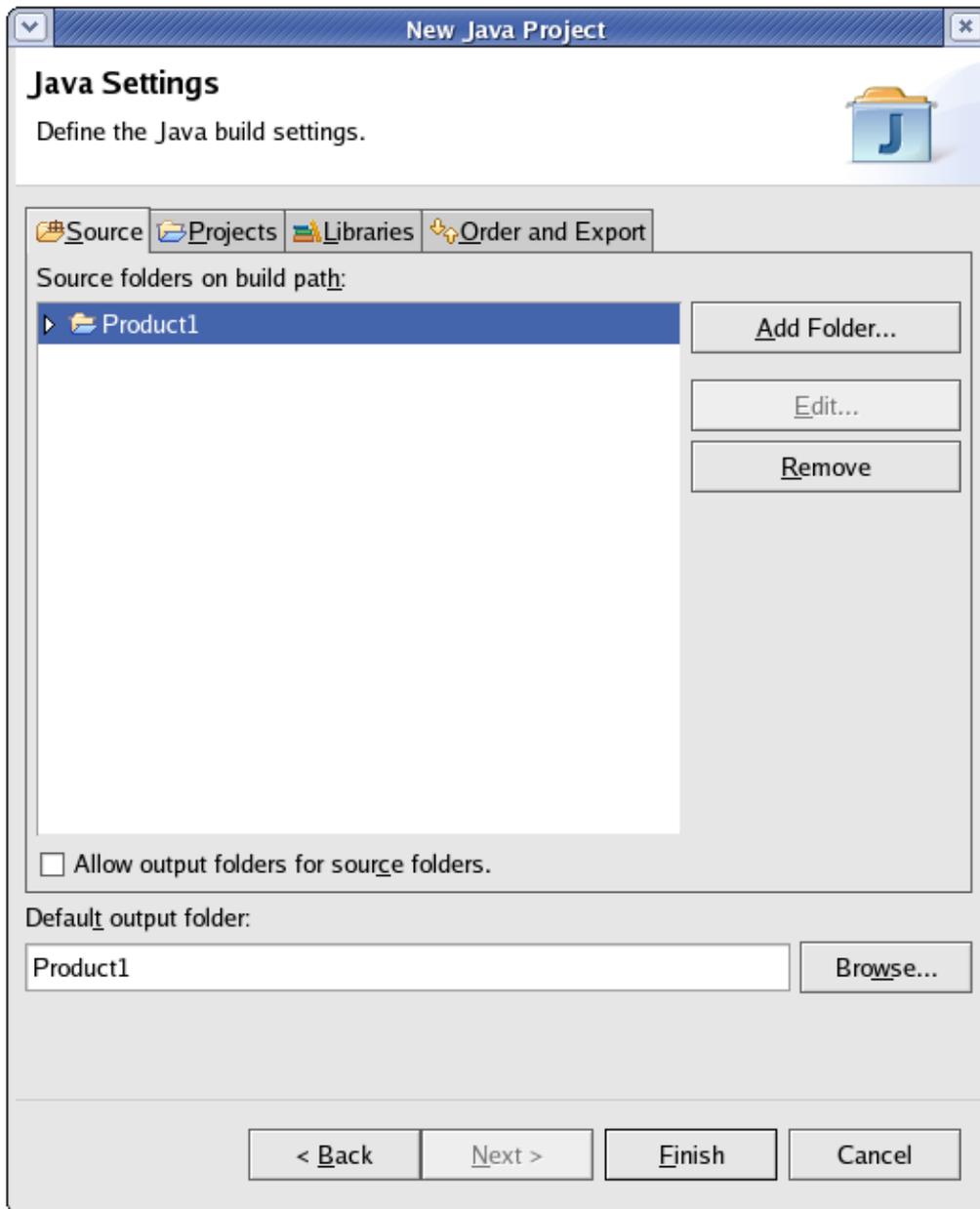
Layout on the file system

- The source files for products should already be laid out in one big directory that is version and configuration managed outside Eclipse.
- The source directory contains two siblings directories "Product1" and "Product2".

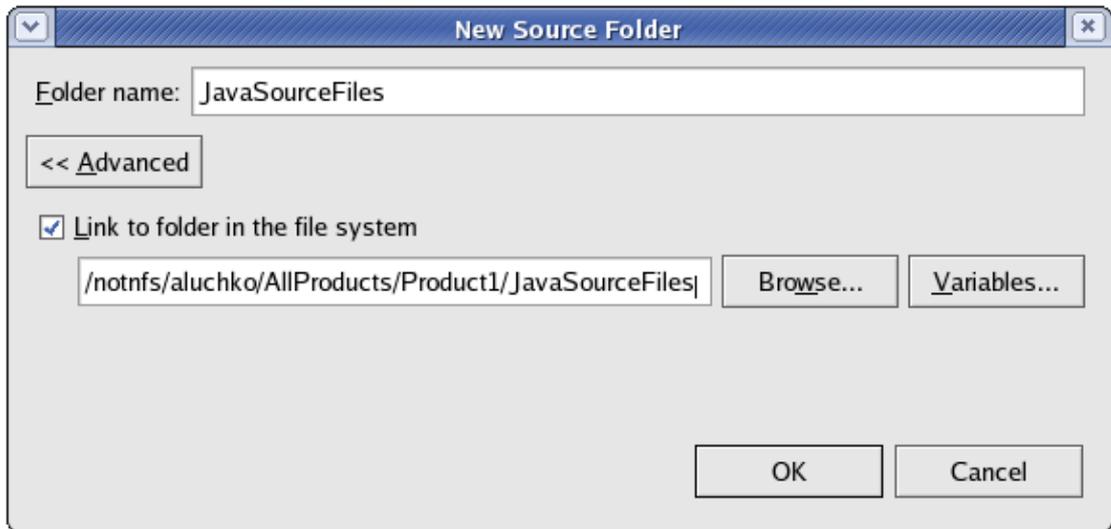


Steps for defining corresponding projects

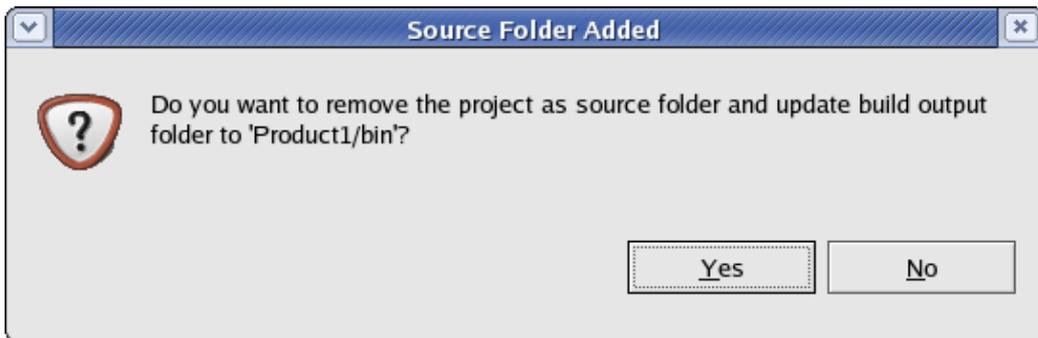
1. Open a Java perspective, then select the menu item **File > New > Project...** to open the **New Project** wizard.
2. Select **Java project** in the list of wizards and click **Next**.
3. Type "Product1" in the **Project name** field. Click **Next**.
4. Select "Product1" source folder and click **Add Folder...**



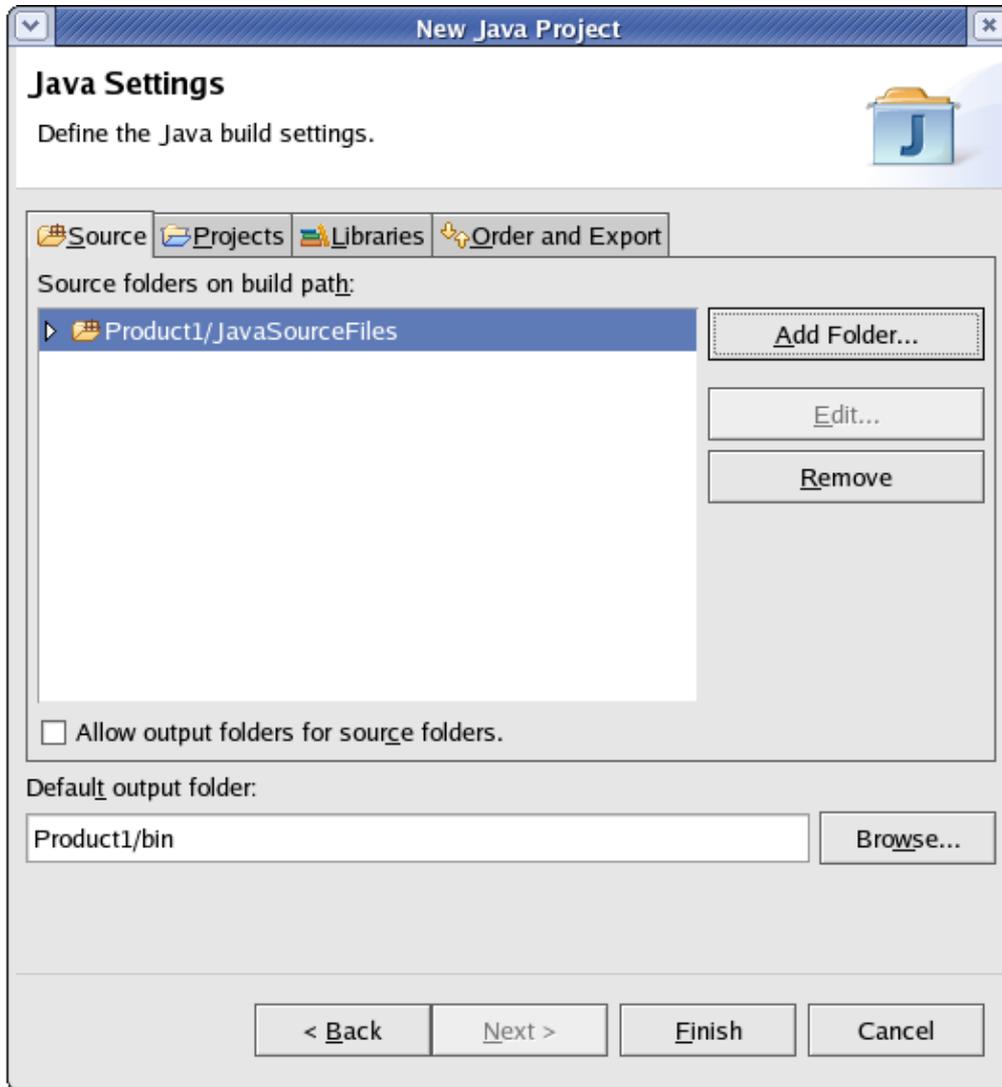
5. In the *New Source Folder* dialog, type "JavaSourceFiles" in the *Folder name* field, then click *Advanced*.
6. Check *Link to folder in the file system*, then click *Browse....* and choose the "JavaSourceFiles" directory in "Product1".



7. Click **OK** to close the dialog.
8. Click **Yes** in confirmation dialog to have "Product1/bin" as default output folder.

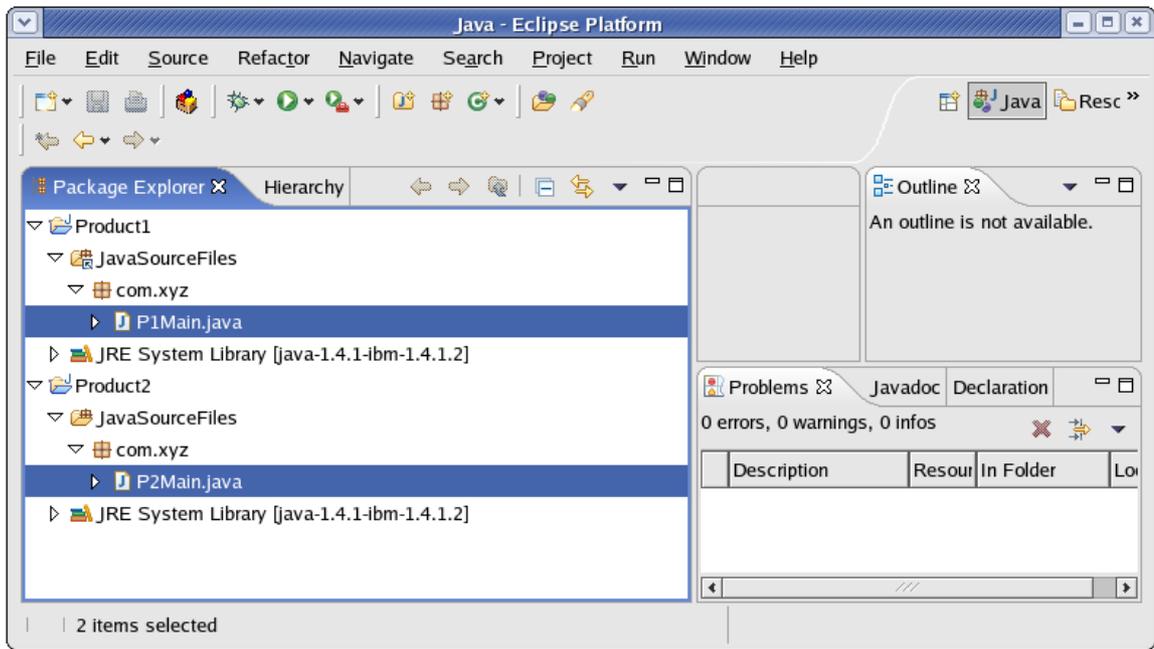


Your project source setup now looks as follows:



9. Click **Finish**.
10. Repeat these steps for "Product2".

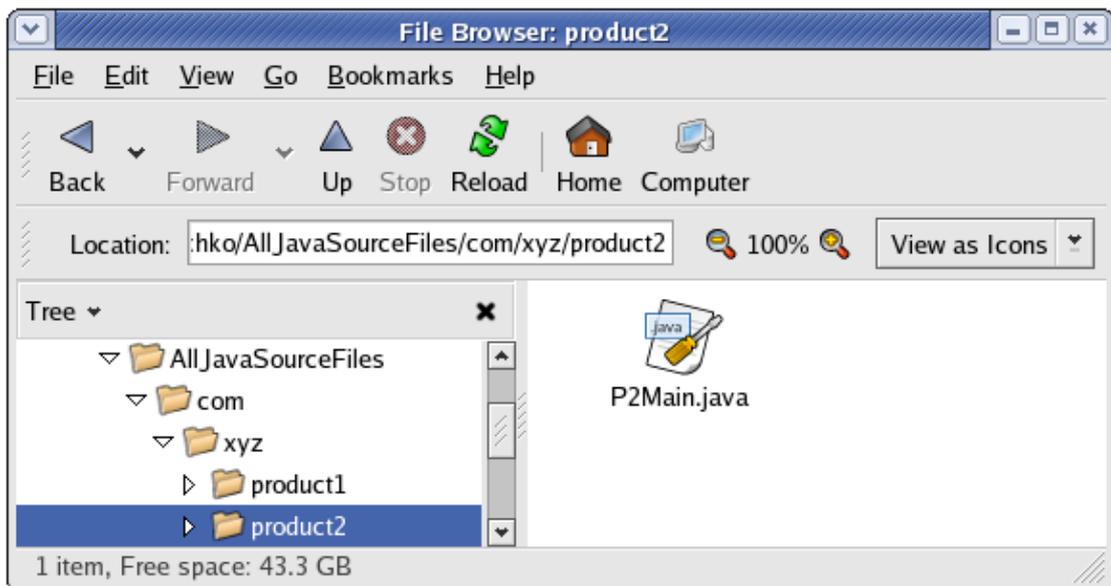
You now have two Java projects which respectively contain the sources of "Product1" and "Product2".



Overlapping products in a common source tree

Layout on file system

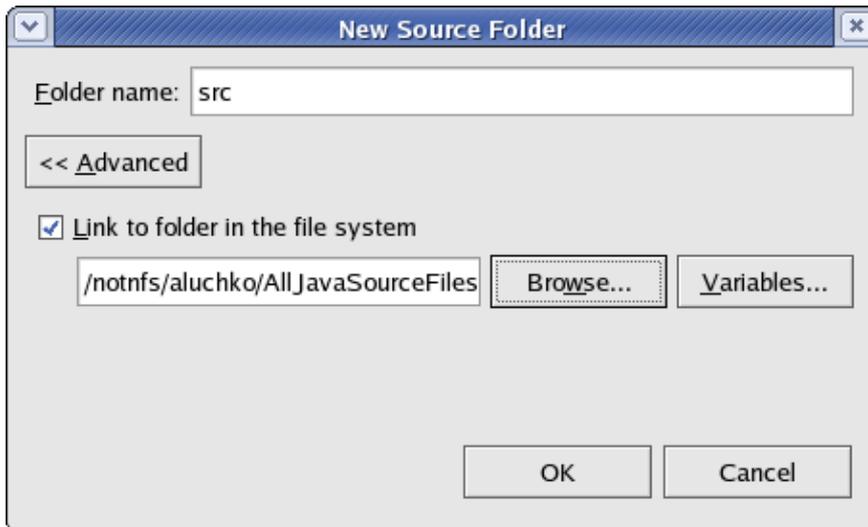
- Set up the Java source files for the products so they are all already held in a single main directory.
- Products are separated into two siblings packages "product1" and "product2".



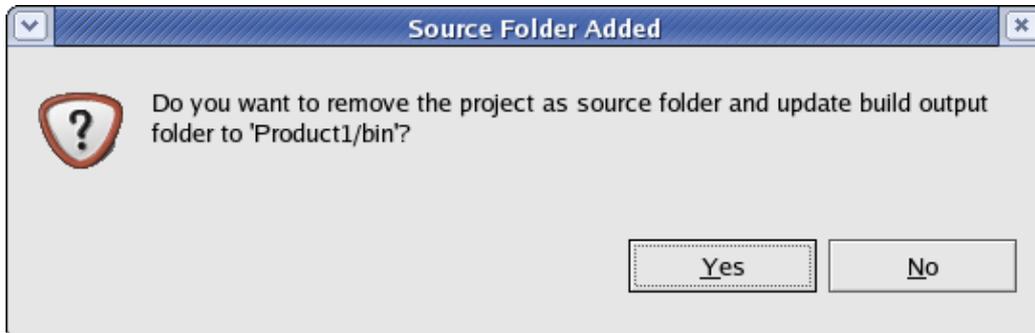
Steps for defining corresponding "Product1" and "Product2" projects

1. Open a Java perspective, select the menu item **File > New > Project...** to open the *New Project* wizard.

2. Select **Java project** in the list of wizards and click **Next**.
3. Type "Product1" in the **Project name** field. Click **Next**.
4. Select the "Product1" source folder and click **Add Folder...**
5. In the **New Source Folder** dialog, type "src" in the **Folder name** field, then click **Advanced**.
6. Check **Link to folder in the file system**, then click **Browse...** and choose the "AllJavaSourceFiles" directory.

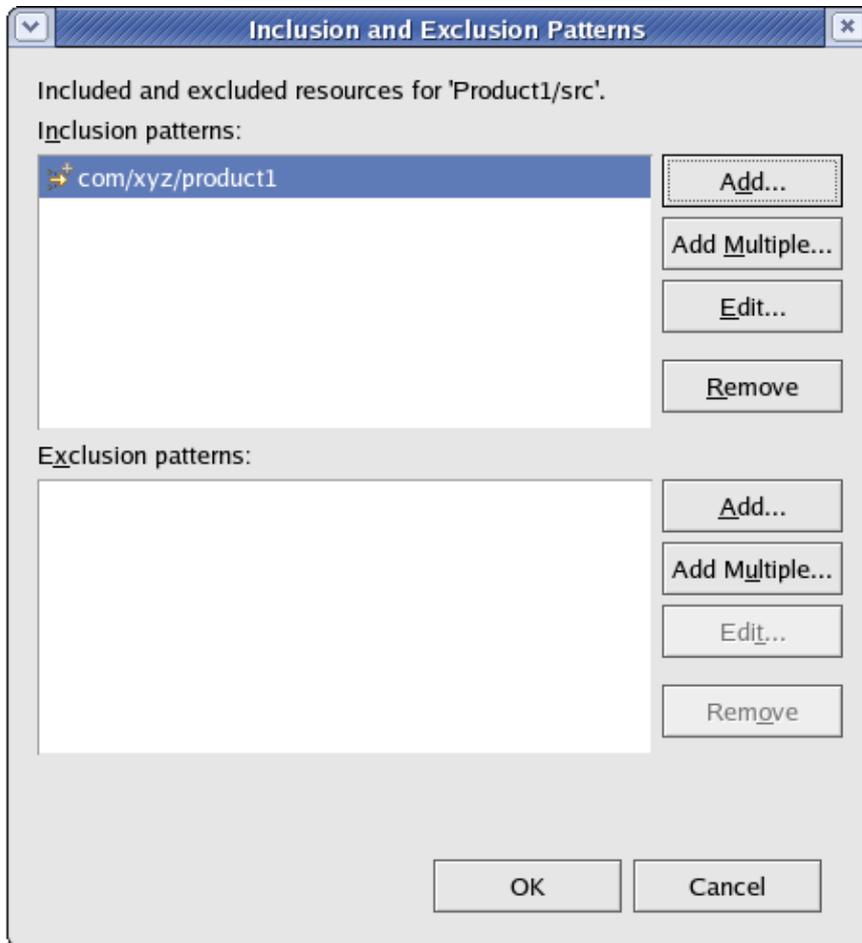


7. Click **OK** to close the dialog.
8. Click **Yes** in the confirmation dialog to have "Product1/bin" as the default output folder.



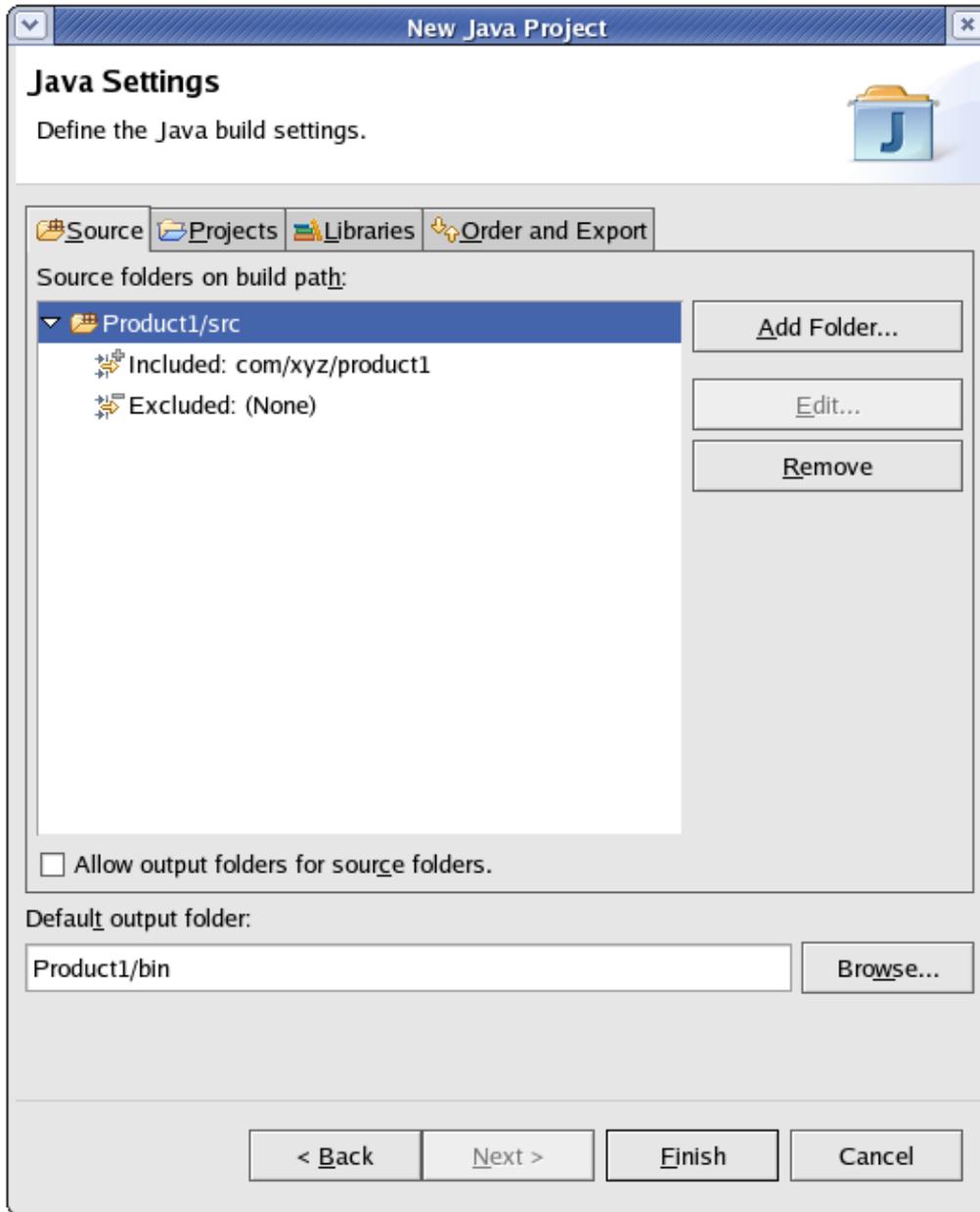
9. Expand the "src" source folder, select **Included**, and click **Edit...**
10. In the **Inclusion patterns** section of **Inclusion and Exclusion Patterns** dialog, click **Add...**
11. Type or browse to "com/xyz/product1/" in the **Add Inclusion Pattern** dialog, then click **OK** to validate and close the dialog.

The **Inclusion and Exclusion Patterns** dialog now looks as follows:

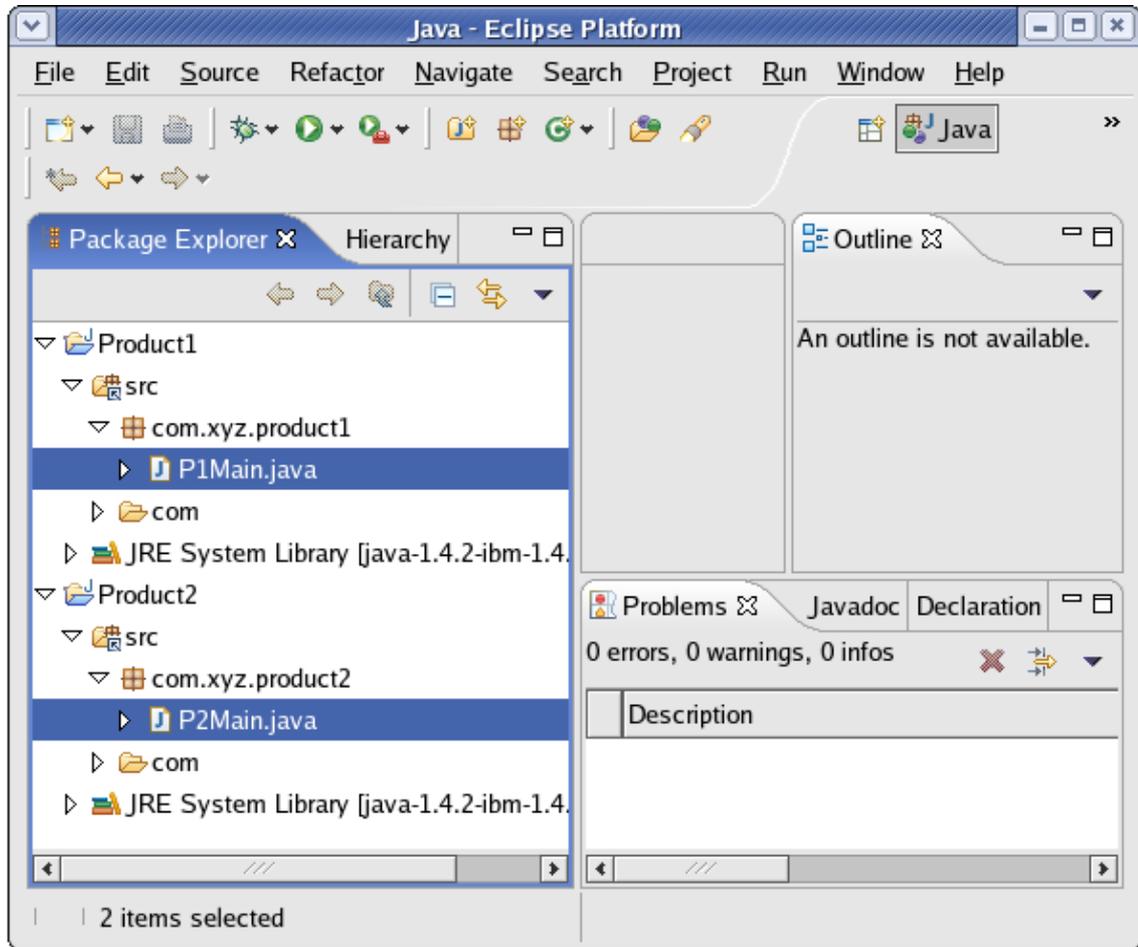


12. Click **OK** to validate and close the dialog.

Your project source setup now looks as follows:



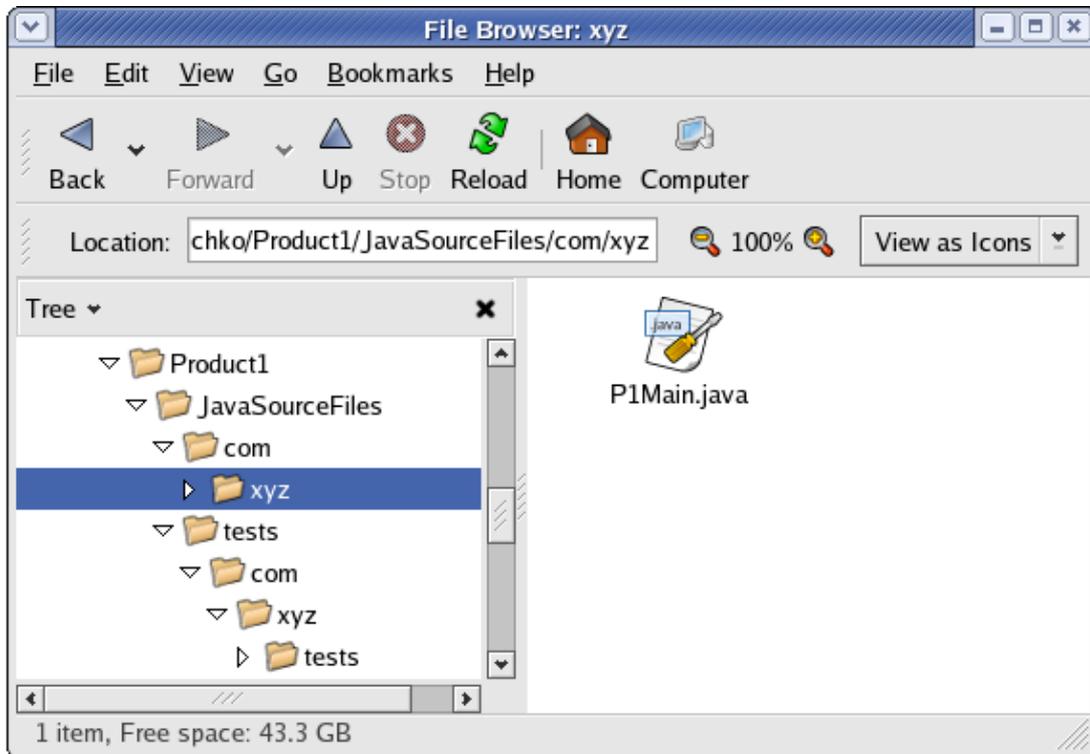
13. Click **Finish**.
14. Repeat these steps for "Product2", including "com/xyz/product2/" instead.
15. You now have two Java projects that respectively contain the sources of "product1", "product2".



Product with nested tests

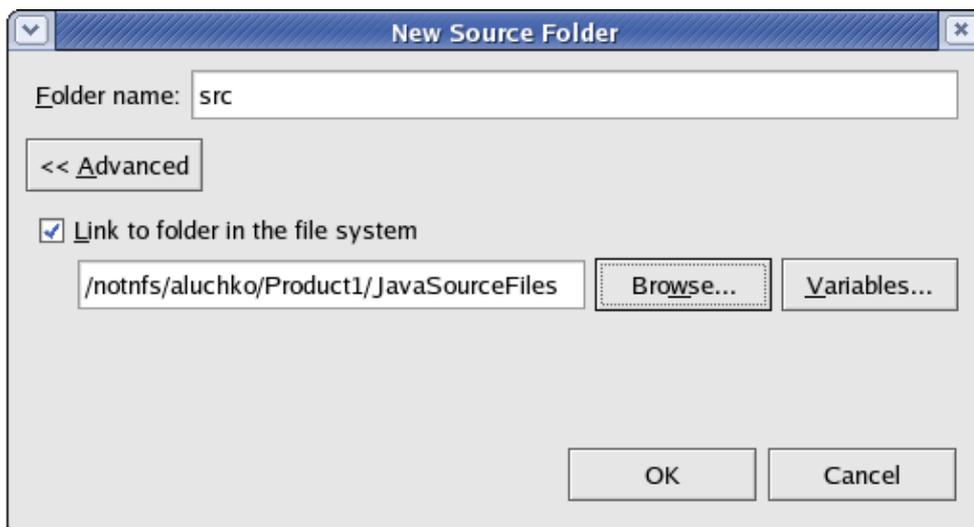
Layout on file system

- Have the Java source files for a product already laid out in a package directory.
- Source files of tests are laid out in a nested package directory.



Steps for defining a corresponding project

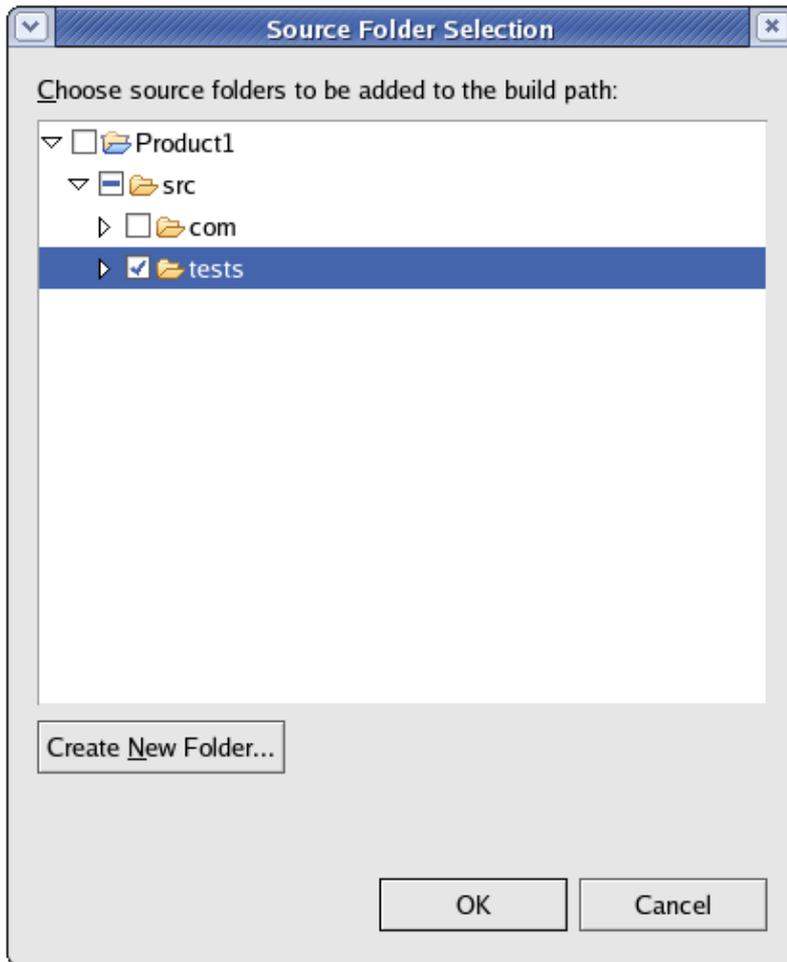
1. Open a Java perspective, select the menu item **File > New > Project...** to open the *New Project* wizard.
2. Select **Java project** in the list of wizards and click *Next*.
3. On the next page, type "Product1" in the **Project name** field. Click *Next*.
4. On the next page, Select "Product1" source folder and click *Add Folder...*
5. In the **New Source Folder** dialog, type "src" in the **Folder name** field, then click *Advanced*.
6. Check **Link to folder in the file system**, then click *Browse...* and choose the "JavaSourceFiles" directory.



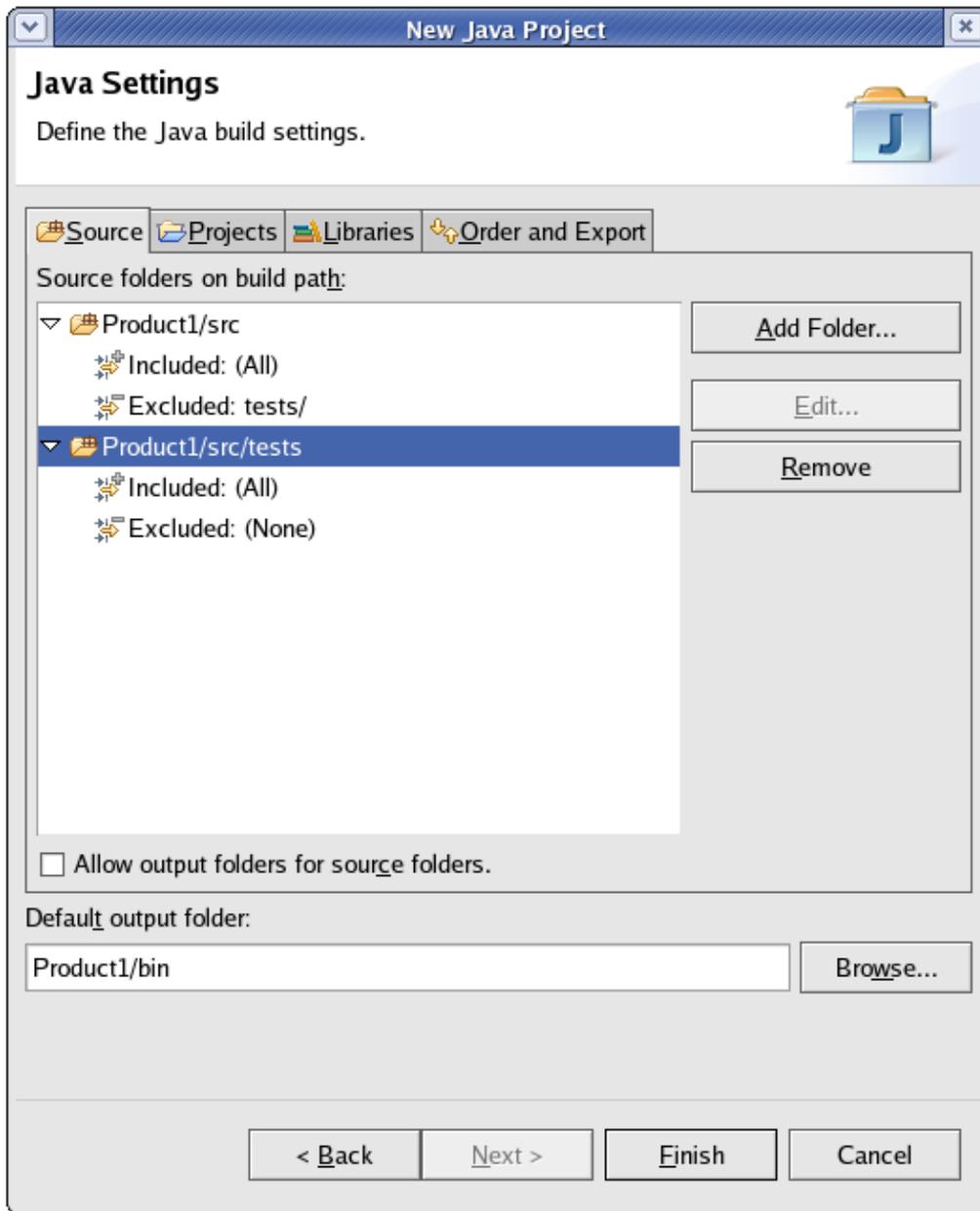
7. Click **OK** to close the dialog.
8. Click **Yes** in the confirmation dialog to have "Product1/bin" as the default output folder.



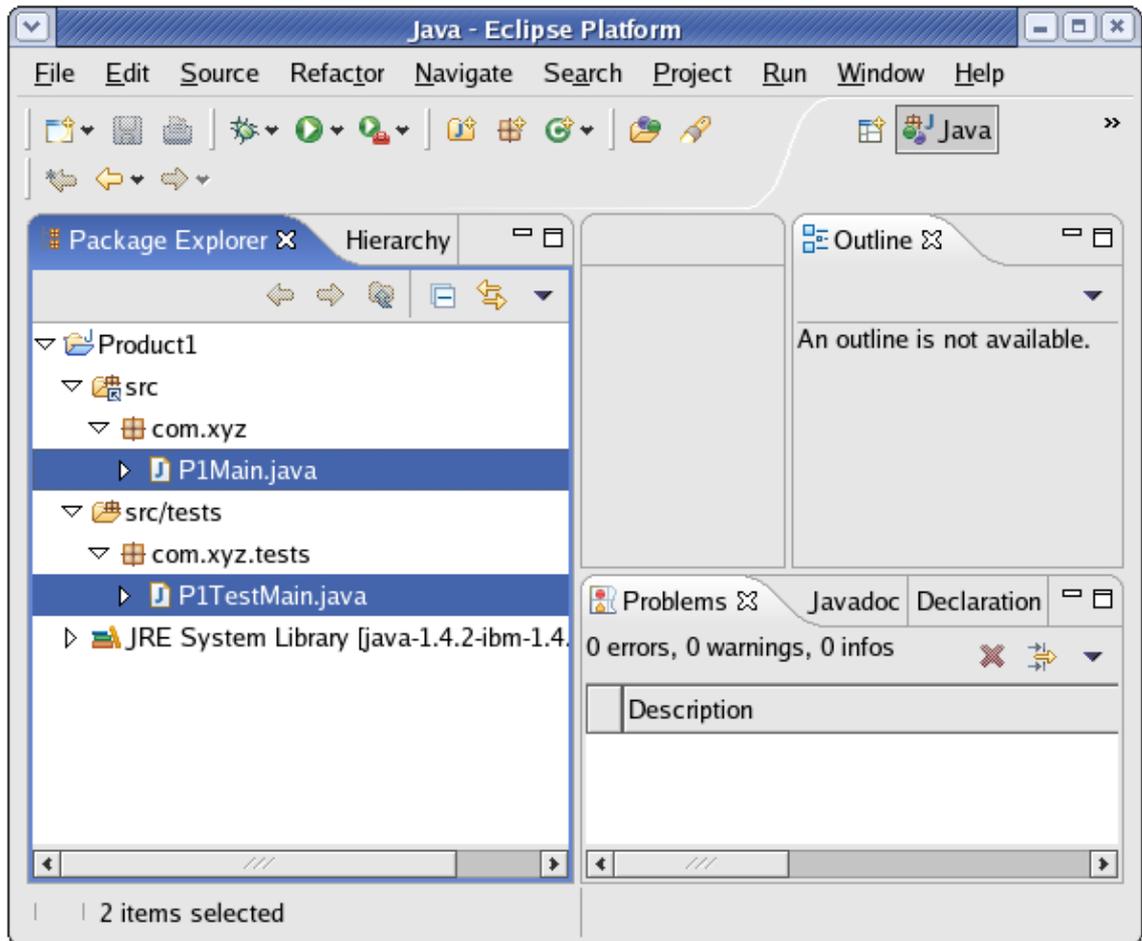
9. Click **Add Folder...**
10. Expand "Product1", then "src" and select "tests".



11. Click **OK**. An information dialog appears saying that exclusion filters have been added. Click **OK**.
12. Your project source setup now looks as follows:



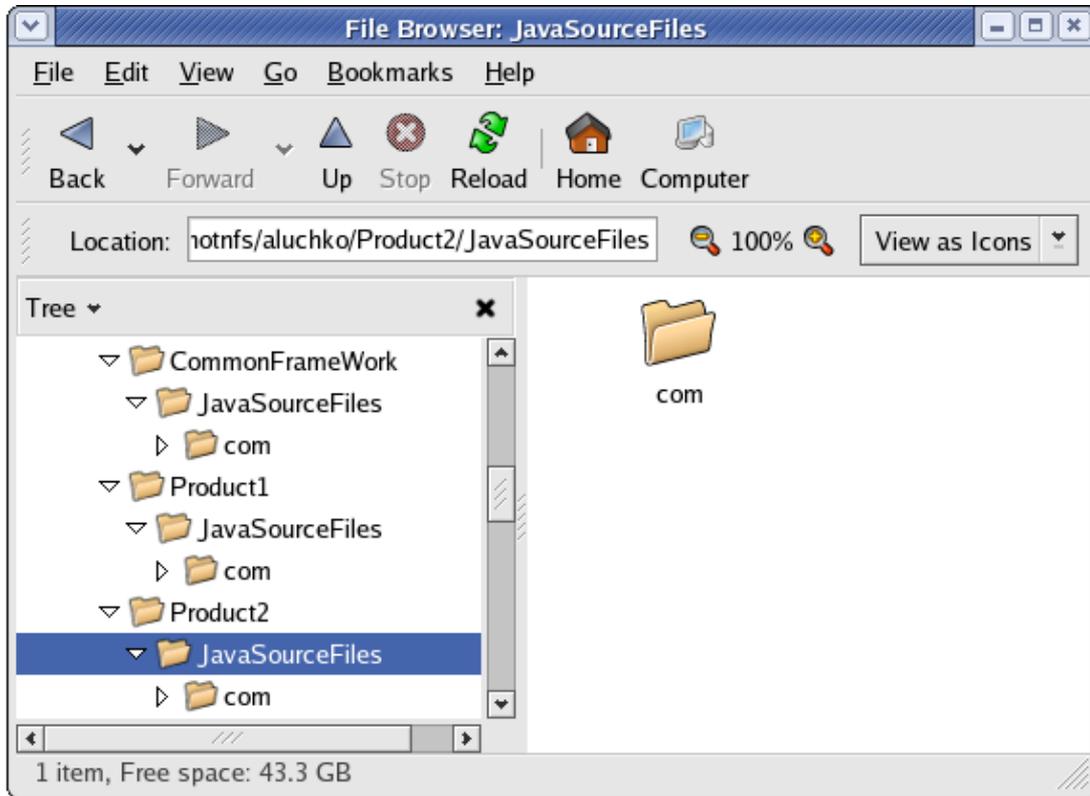
13. Click **Finish**.
14. You now have a Java project with a "src" folder and a "tests" folder that contain respectively the "JavaSourceFiles" directory and the "tests" directory.



Products sharing a common source framework

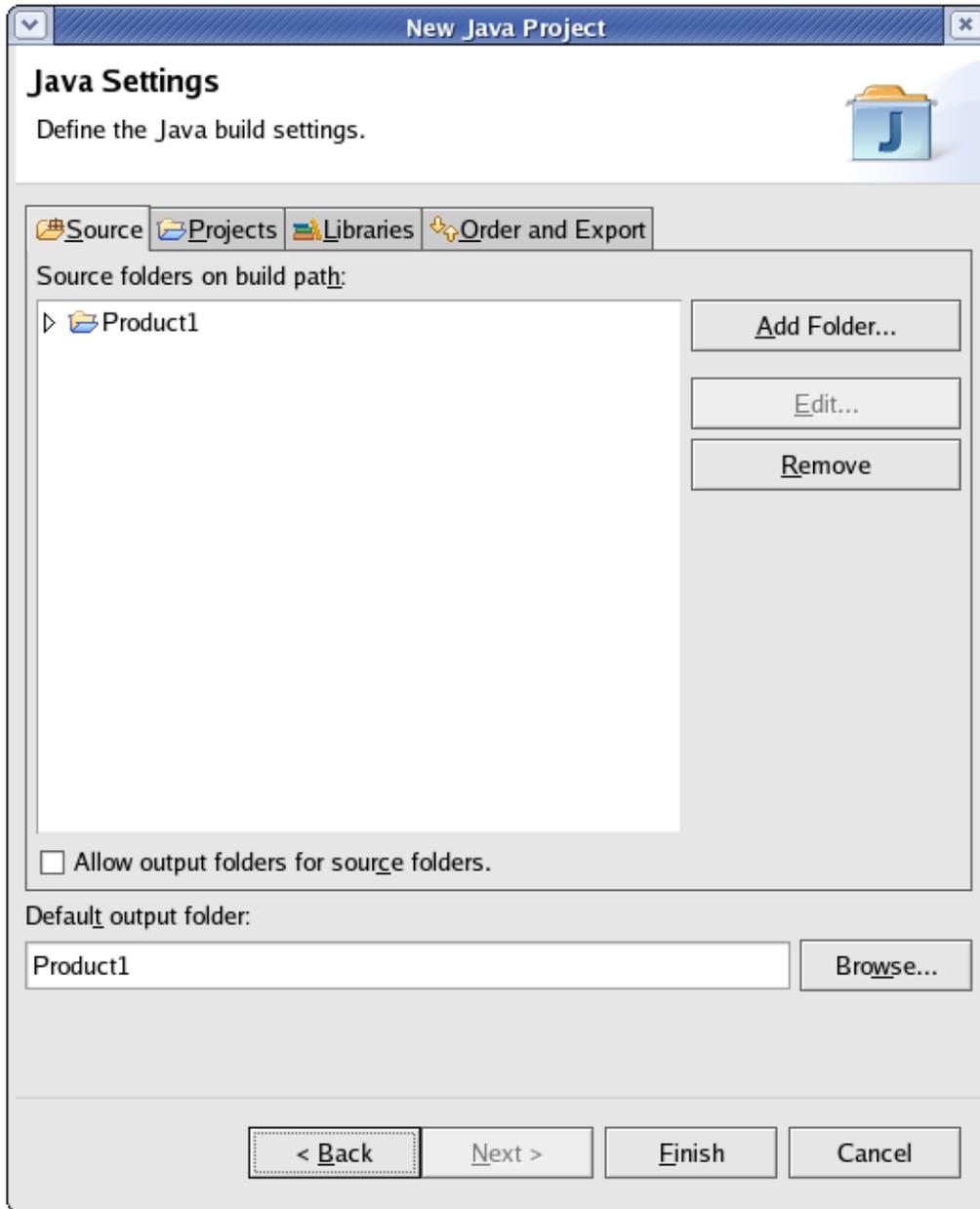
Layout on file system

- The Java source files for two products require a common framework.
- Set up the projects and common framework in separate directories with their own source and output folders.

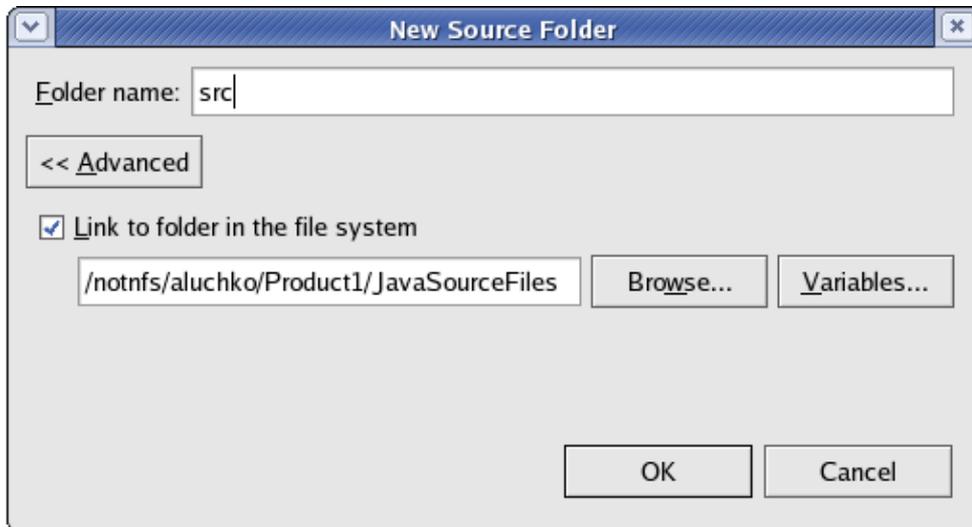


Steps for defining corresponding projects

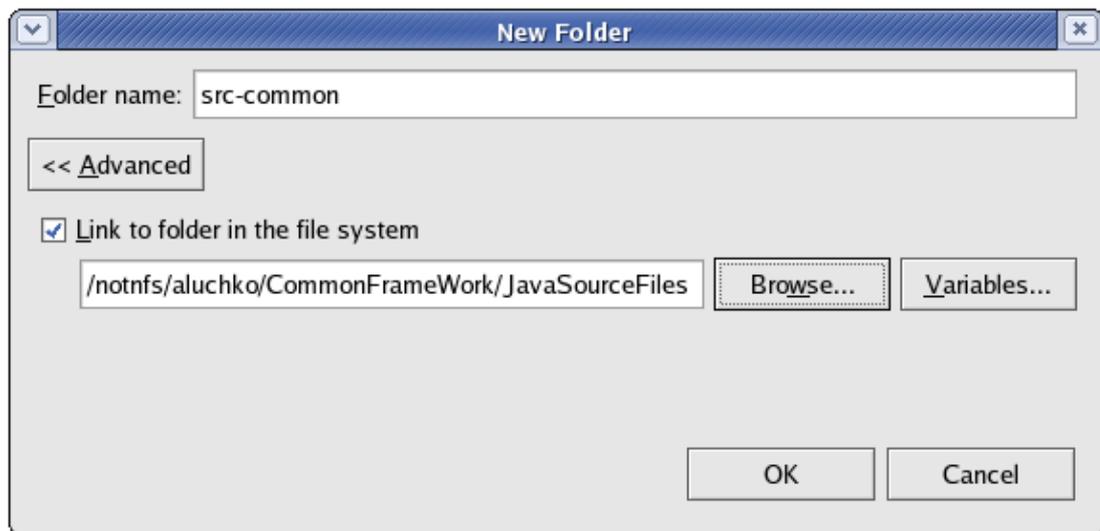
1. Open the Java perspective, then click **File > New > Project...** to open the **New Project** wizard.
2. Select **Java project** in the list of wizards and click **Next**.
3. On the next page, type "Product1" in the **Project name** field. Click **Next**.



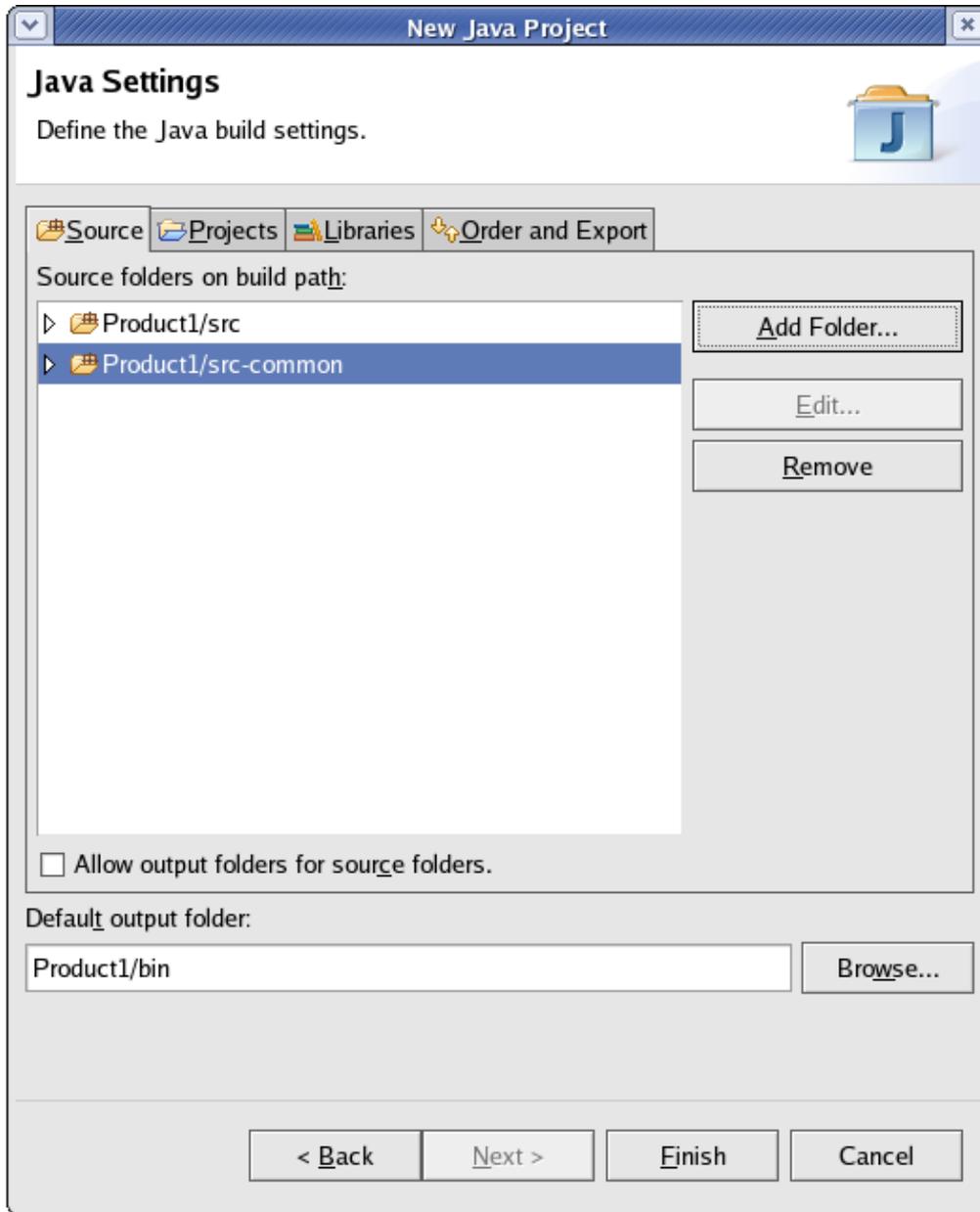
4. Select the "Product1" source folder and click **Add Folder...**
5. In the **New Source Folder** dialog, type "src" in the **Folder name** field, then click **Advanced**.
6. Check **Link to folder in the file system**, then click **Browse....** and choose the "JavaSourceFiles" directory in "Product1".



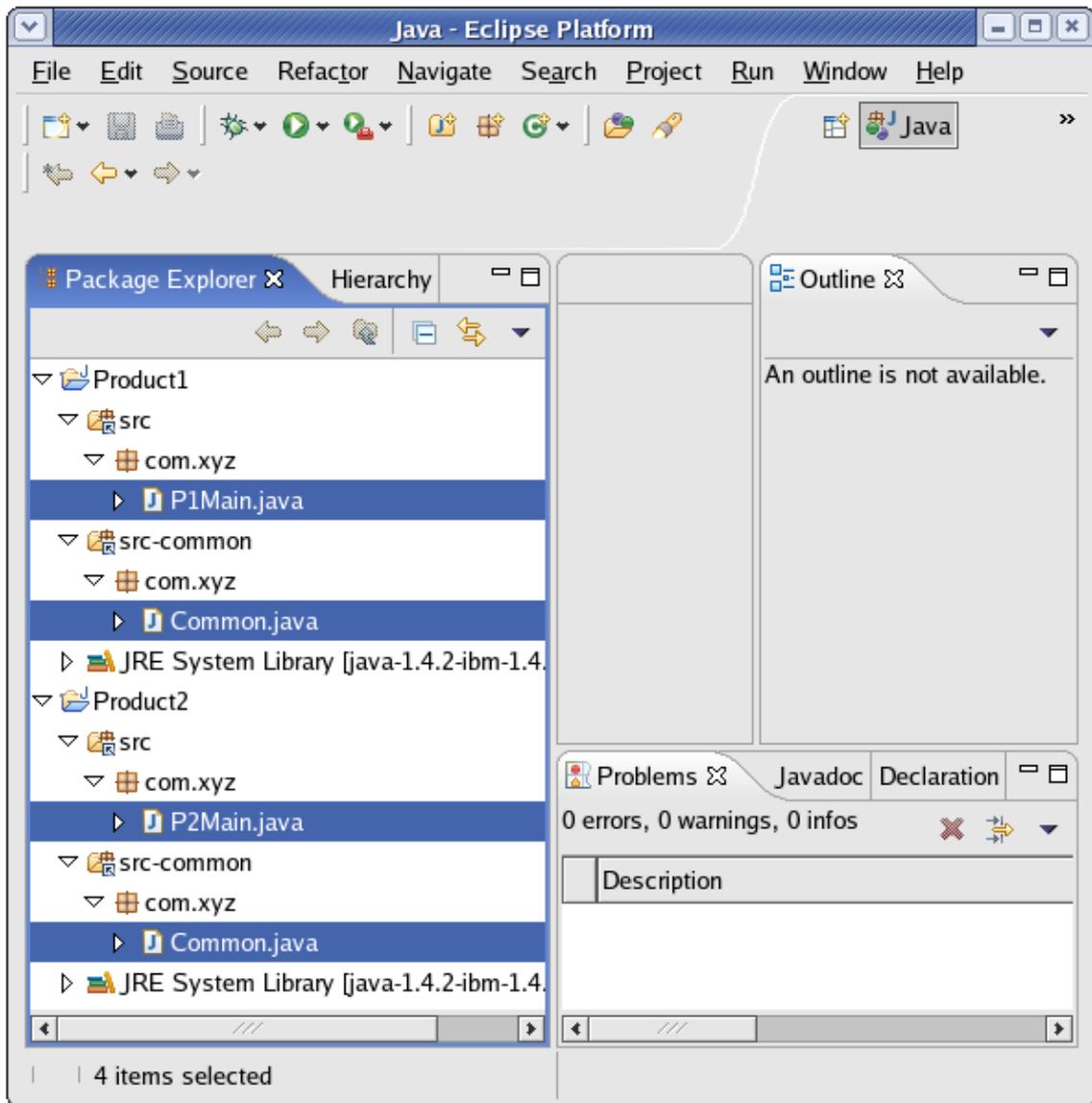
7. Click **OK** to close the dialog.
8. Click again on **Add Folder...**
9. In the **New Folder** dialog, type "src-common" in the **Folder name** field, then click **Advanced**.
10. Check **Link to folder in the file system**, then click **Browse...** and choose the "JavaSourceFiles" directory in "CommonFramework".



11. Click **OK** to close the dialog.
12. Check that "Product1/bin" is in the **Default output folder** field.



13. Click **Finish**.
14. Repeat these steps for "Product2".
15. You now have two Java projects which respectively contain the sources of "Product1" and "Product2" and which are using the sources of "CommonFramework".

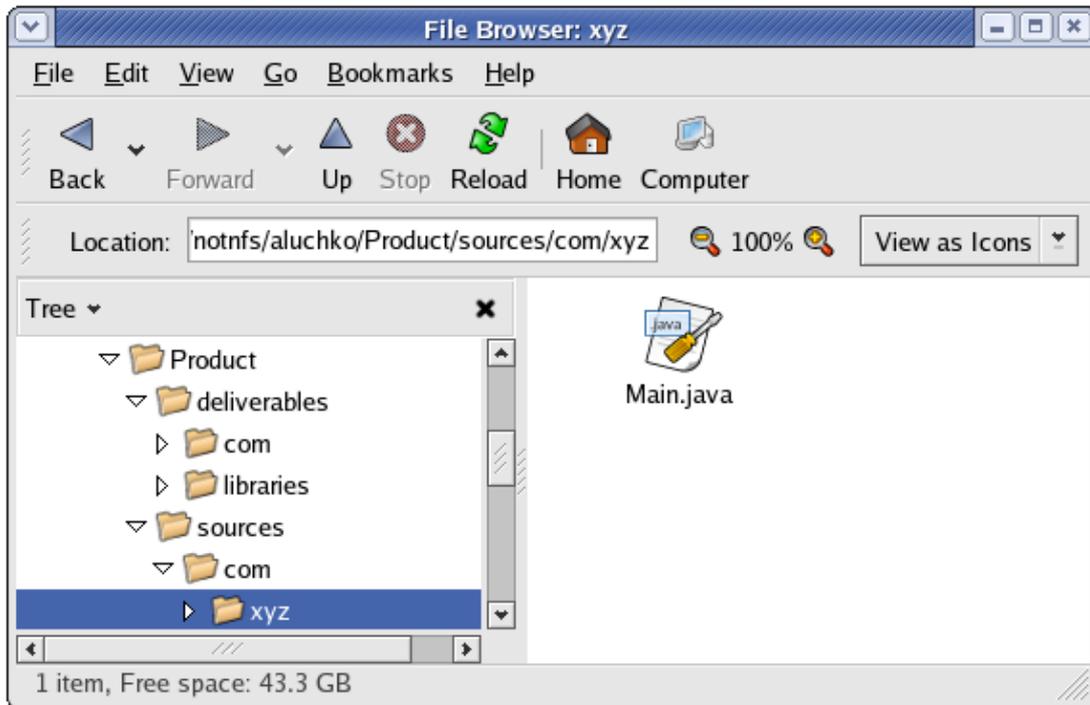


Note: Files in "src-common" are shared, so editing "Common.java" in "Product1" will modify "Common.java" in "Product2". However they are compiled in the context of their respective projects. Two "Common.class" files will be generated; one for each project. If the two projects have different compiler options, then different errors could be reported on each "Common.java" file.

Product nesting resources in an output directory

Layout of the file system

- The Java source files for a product are laid out in a "sources" directory.
- Java class files are laid out in a "deliverables" directory.
- The project uses some libraries located in the "deliverables/libraries" directory:



Defining a corresponding project

To define a corresponding project:

1. From the Java perspective, click **File > New > Project**.
2. From the **New Project** wizard, select **Java project** and click Next.
3. On the next page, type **Product** in the **Project name** field. Click Next.
4. On the next page, select the "Product" source folder and click Remove.

Type **/Product/deliverables** in **Default output folder** field.

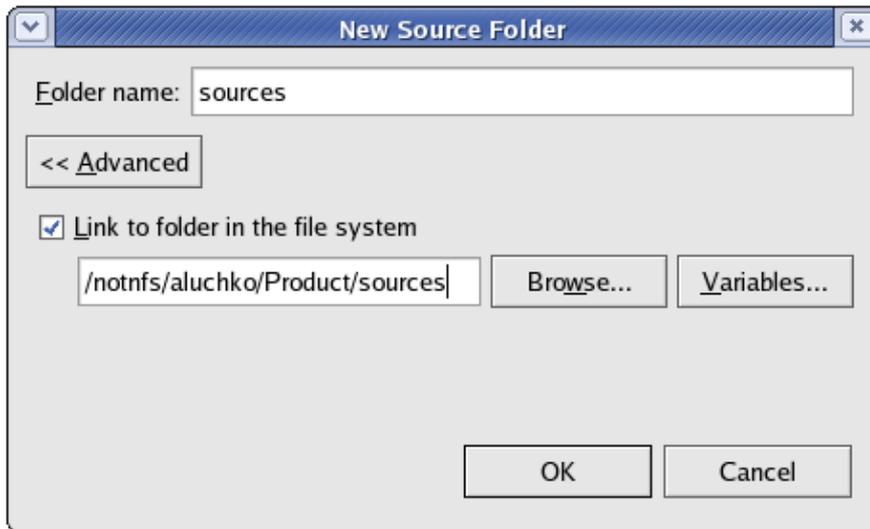
Click Add Folder.

5. In the **New Source Folder** dialog, type **sources** in the **Folder name** field.

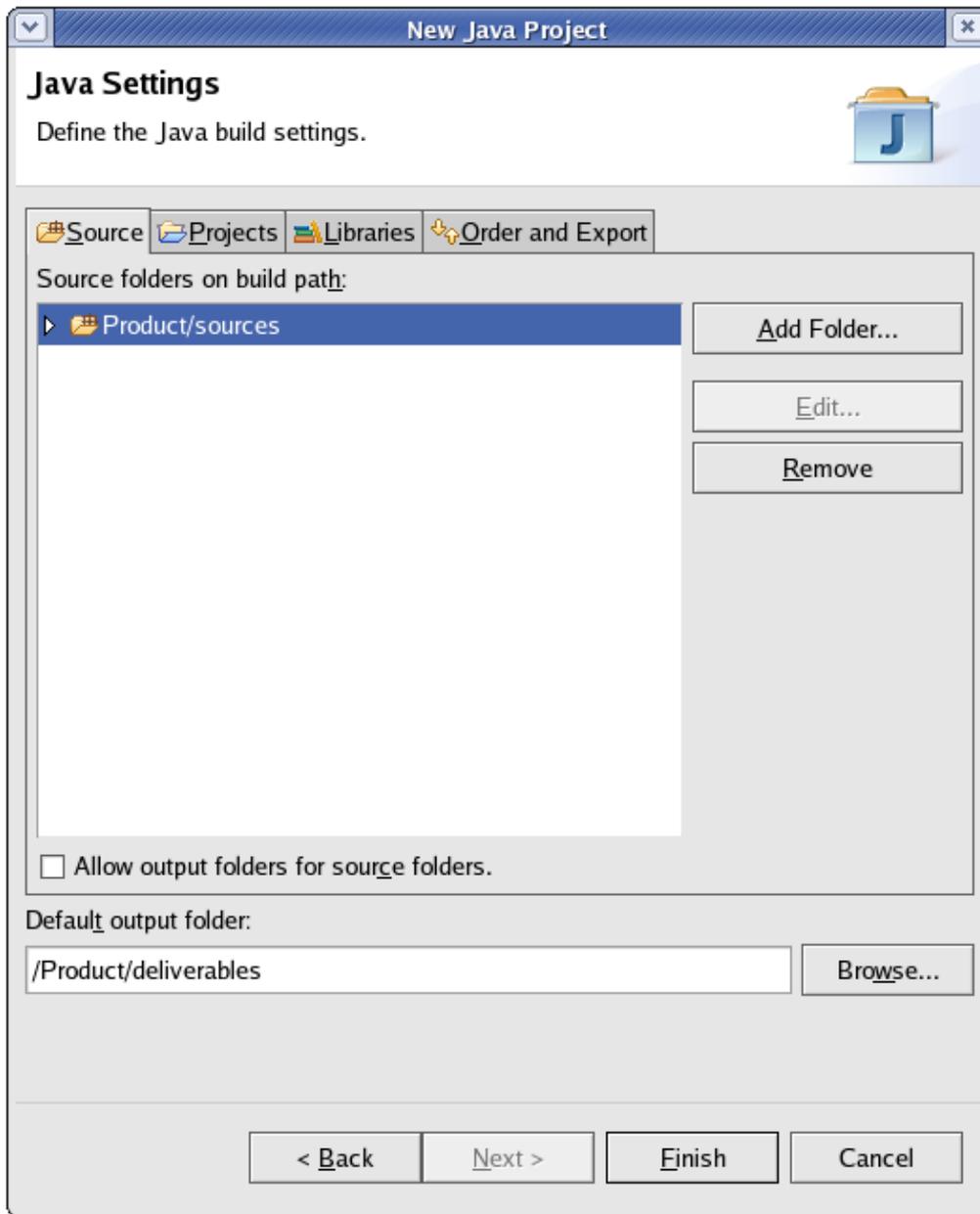
Click Advanced.

6. Check **Link to folder in the file system**.

Click Browse and choose the "sources" directory in "Product".



7. Click OK to close the dialog.
8. Click again on Add Folder.



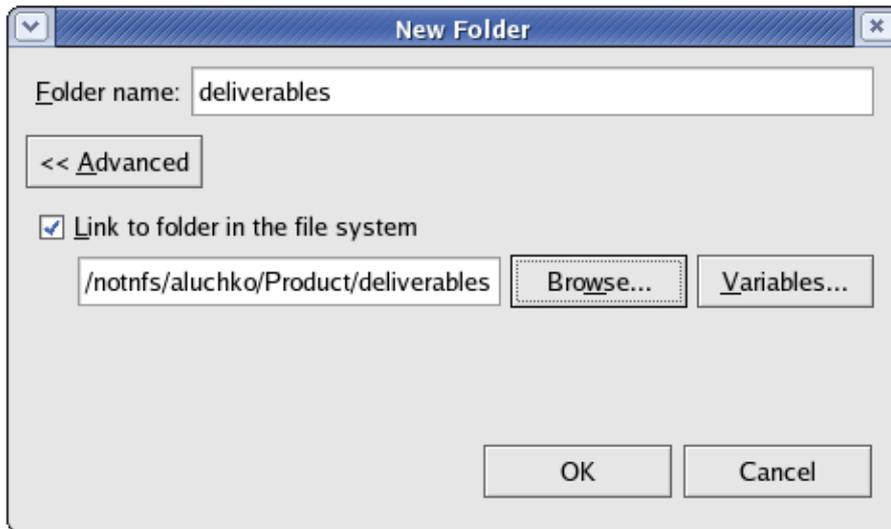
9. In the *Source Folder Selection* dialog, click Create New Folder.

Type **deliverables** in the *Folder name* field.

Click Advanced.

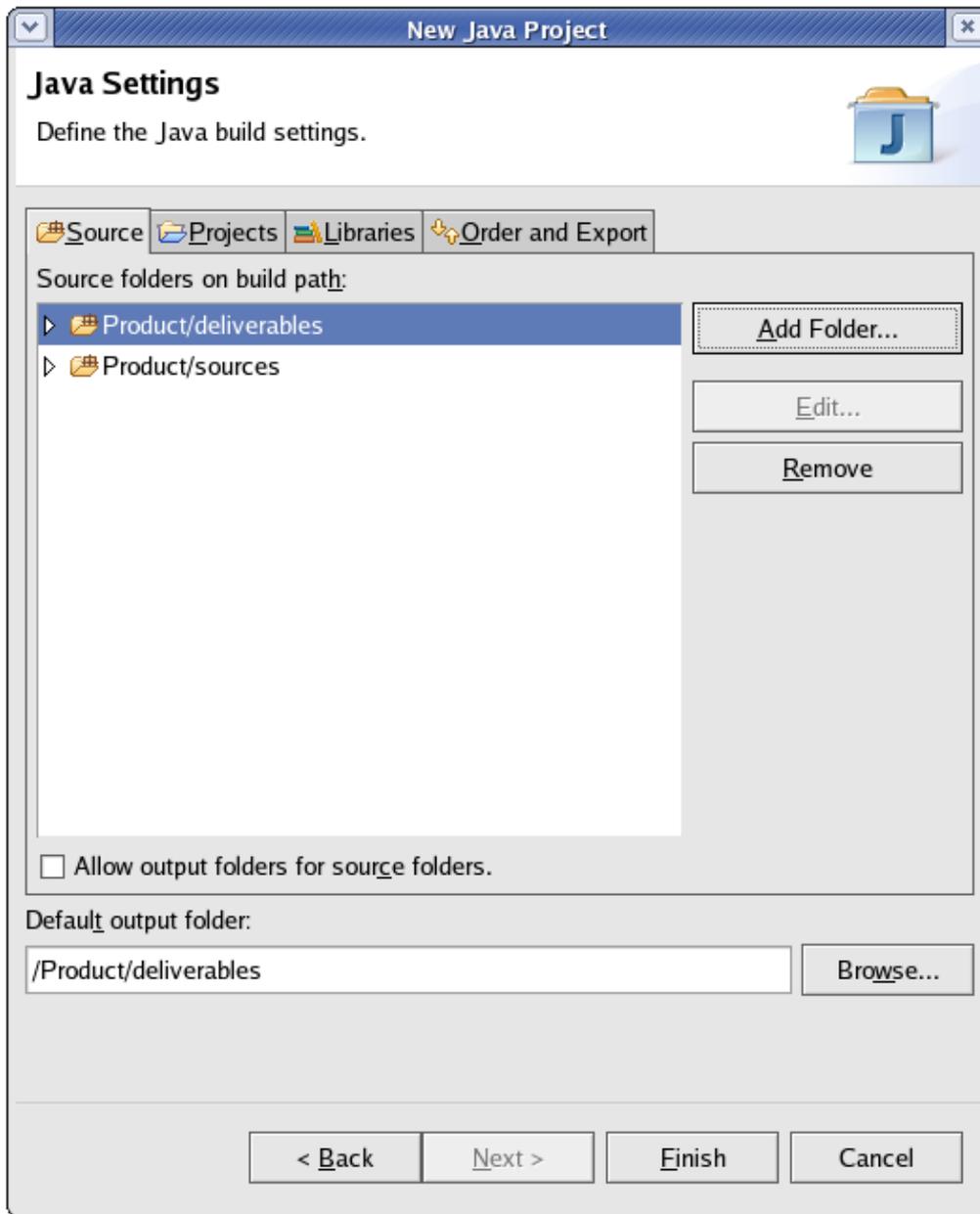
10. Check *Link to folder in the file system*.

Click Browse and choose the "deliverables" directory in "Product".



11. Click OK twice to close the two dialogs.

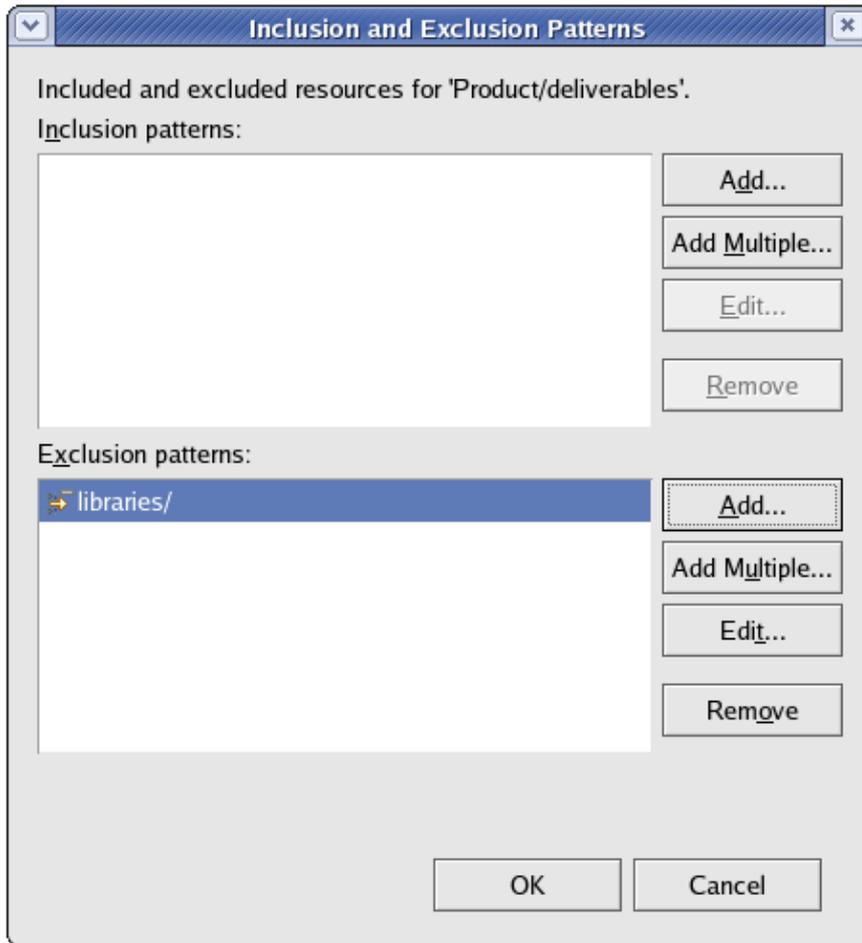
You project setup now looks as follows:



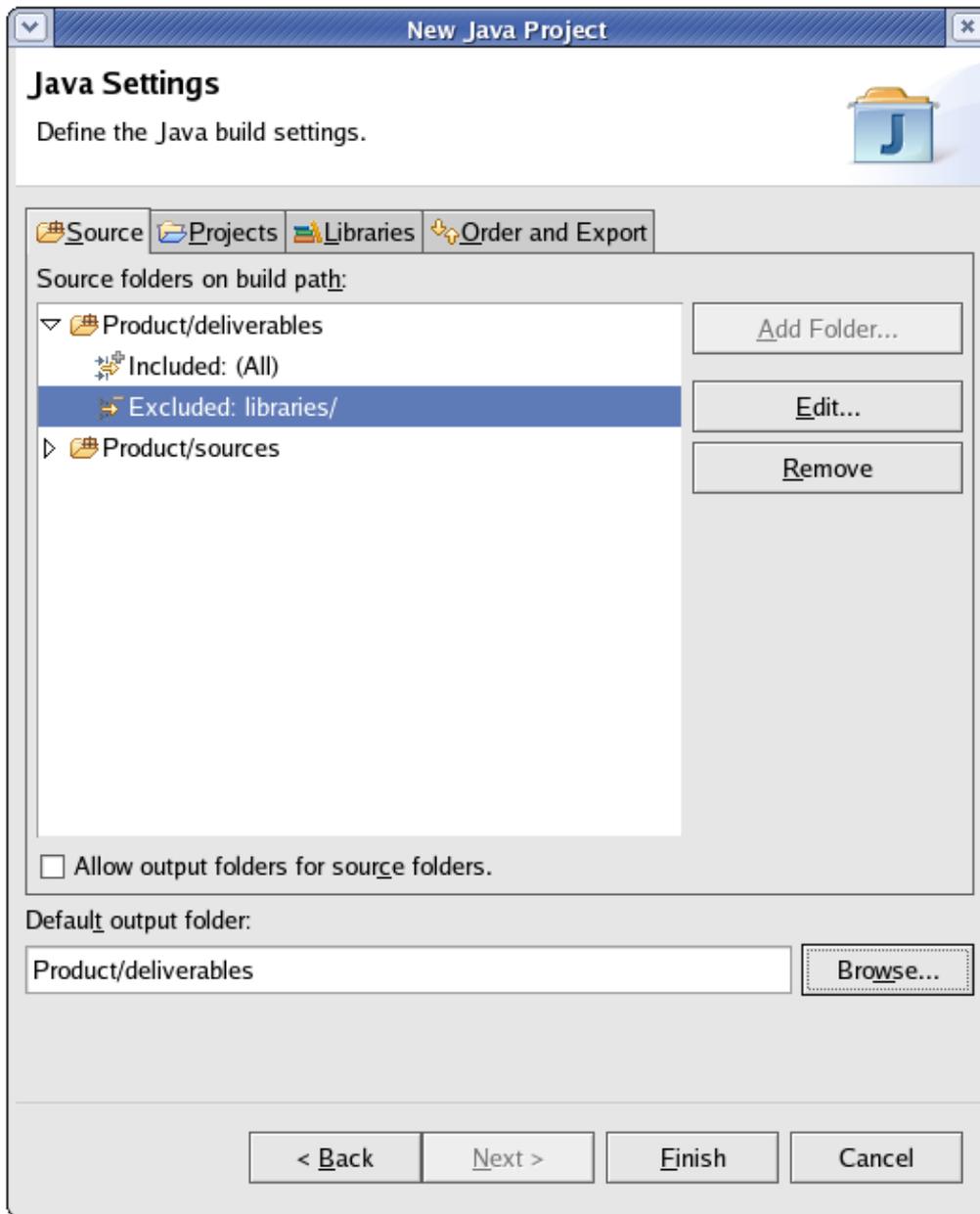
12. Expand "Product/deliverables" and select *Excluded*.

Click Edit and click Add in the *Exclusion patterns* part of the *Inclusion and Exclusion Patterns* dialog.

Type `libraries/` in *Add Exclusion Pattern* dialog and click OK to validate and close the dialog.



13. Click OK to close the dialog.
14. Your project source now looks as follows:

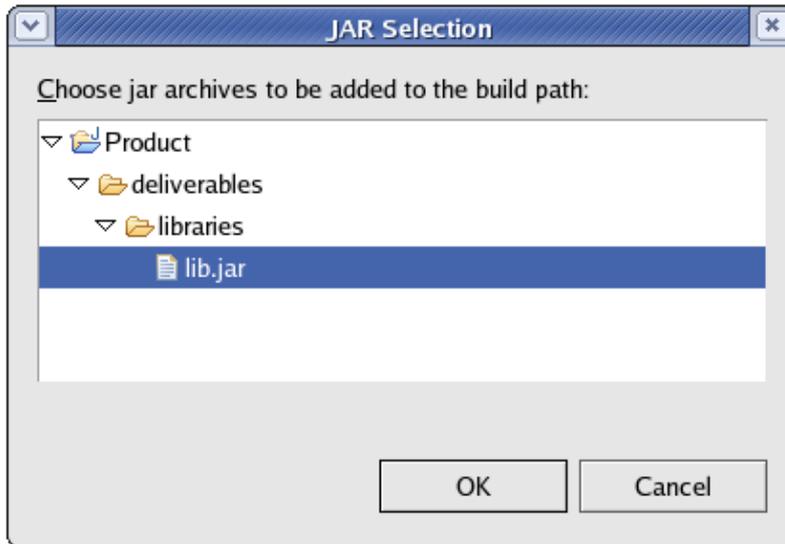


15. Click Finish.

16. Right-click on the project and select Properties from the context menu.

Select **Java Build Path** page and go to the **Libraries** tab.

17. Click Add JARs and expand "Product" hierarchy to select JAR files in the "libraries" directory.



18. Click OK.

You now have a Java project with a "sources" folder and an output folder that contains nested library resources.

