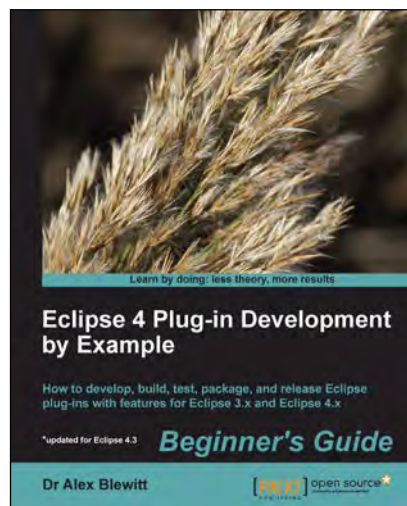


Eclipse 4 Plug-in Development by Example Beginner's Guide

Dr Alex Blewitt



Chapter No. 1 "Creating Your First Plug-in"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.1 "Creating Your First Plug-in"

A synopsis of the book's content

Information on where to buy this book

About the Author

Dr Alex Blewitt has been developing Java applications since Version 1.0 was released in 1996, and has been using the Eclipse platform since its first release as part of the IBM WebSphere Studio product suite. He even migrated some plugins from Visual Age for Java to WebSphere Studio/Eclipse as part of his PhD on Automated Verification of Design Patterns. He got involved in the open source community as a tester when Eclipse 2.1 was being released for Mac OS X, and then subsequently as an editor for EclipseZone, including being a finalist for Eclipse Ambassador in 2007.

More recently, Alex has been writing for InfoQ, covering generic Java and specifically, Eclipse and OSGi subjects. He keynoted the 2011 OSGi Community Event on the past, present, and future of OSGi. The coverage of both new releases of the Eclipse platform and its projects, as well as video interviews with some of the Eclipse project leads can be found via the InfoQ home page, for which he was nominated and won the Eclipse Top Contributor 2012 award.

For More Information:

www.packtpub.com/eclipse-4-plugin-development-by-example-beginners-guide/book

Alex currently works for an investment bank in London. He also has a number of apps on the Apple AppStore through Bandlem Limited. When he's not working on technology, and if the weather is nice, he likes to go flying from the nearby Cranfield airport.

Alex writes regularly at his blog, <http://alblue.bandlem.com>, as well as tweets regularly from Twitter and App.Net as @alblue.

For More Information:

www.packtpub.com/eclipse-4-plugin-development-by-example-beginners-guide/book

Eclipse 4 Plug-in Development by Example Beginner's Guide

This book provides a general introduction to developing plug-ins for the Eclipse platform. No prior experience, other than Java, is necessary to be able to follow the examples presented in this book. By the end of the book, you should be able to create an Eclipse plug-in from scratch, as well as be able to create an automated build of those plug-ins.

What This Book Covers

Chapter 1, Creating Your First Plug-in, provides an overview of how to download Eclipse, set it up for plug-in development, create a sample plug-in, launch, and debug it.

Chapter 2, Creating Views with SWT, provides an overview of how to build views with SWT, along with other custom SWT components such as system trays and resource management.

Chapter 3, Creating JFace Viewers, discusses creating views with JFace using TableViewers and TreeViewers, along with integration with the properties view and user interaction.

Chapter 4, Interacting with the User, discusses using commands, handlers, and menus to interact with the user, as well as the Jobs and Progress APIs.

Chapter 5, Storing Preferences and Settings, tells how to store preference information persistently, as well as displaying it via the preferences pages.

Chapter 6, Working with Resources, teaches how to load and create Resources in the workbench, as well as how to create a builder and nature for automated processing.

Chapter 7, Understanding the Eclipse 4 Model, discusses the key differences between the Eclipse 3.x and Eclipse 4.x models, as well as how to migrate existing content to the new model.

Chapter 8, Creating Features, Update Sites, Applications, and Products, tells how to take the plug-ins created so far in this book, aggregate them into features, publish to update sites, and how applications and products are used to create standalone entities.

Chapter 9, Automated Testing of Plug-ins, teaches how to write automated tests that exercise Eclipse plug-ins, including both UI and non-UI components.

Chapter 10, Automated builds with Tycho, details how to build Eclipse plug-ins, features, update sites, applications, and products automatically with Maven Tycho.

For More Information:

www.packtpub.com/eclipse-4-plugin-development-by-example-beginners-guide/book

1

Creating Your First Plug-in

Eclipse is a highly modular application, consisting of hundreds of plug-ins, and can be extended by installing additional plug-ins. Plug-ins are developed and debugged with the Plug-in Development Environment (PDE).

In this chapter, we shall:

- ◆ Set up an Eclipse environment for performing plug-in development
- ◆ Create a plug-in with the new plug-in wizard
- ◆ Launch a new Eclipse instance with the plug-in enabled
- ◆ Debug the Eclipse plug-in

Getting started

Developing plug-ins requires an Eclipse development environment. This book has been developed and tested on Juno (Eclipse 4.2) Kepler (4.3). Use the most recent version available.

Eclipse plug-ins are generally written in Java. Although it's possible to use other JVM-based languages (such as Groovy or Scala), this book will use the Java language.

There are several different packages of Eclipse available from the downloads page, each of which contains a different combination of plug-ins. This book has been tested with:

- ◆ Eclipse SDK from <http://download.eclipse.org/eclipse/downloads/>
- ◆ Eclipse Classic and Eclipse Standard from <http://www.eclipse.org/downloads/>

For More Information:

www.packtpub.com/eclipse-4-plugin-development-by-example-beginners-guide/book

These contain the necessary **Plug-in Development Environment (PDE)** feature as well as the source code, the help documentation, and other useful features. (The RCP and RAP package should not be used, as it will cause problems with exercises in the *Chapter 7, Understanding the Eclipse 4 Model*.)

It is also possible to install the Eclipse PDE feature into an existing Eclipse instance. To do this, go to the **Help** menu and select **Install New Software**, followed by choosing the **General Purpose Tools** category from the update site. The Eclipse Plug-in Development Environment feature contains everything needed to create a new plug-in.

Time for action – setting up the Eclipse SDK environment

Eclipse is a Java-based application, which needs Java installed. Eclipse is distributed as a compressed archive and doesn't require an explicit installation step.

1. To obtain Java, go to <http://java.com> and follow the instructions to download and install Java. Note that Java comes in two flavors; a 32-bit install and a 64-bit install. If the running OS is a 32-bit, install the 32-bit JDK; alternatively, if the running OS is 64-bit, install the 64-bit JDK.
2. Running `java -version` should give output like so:

```
java version "1.7.0_09"  
Java(TM) SE Runtime Environment (build 1.7.0_09-b05)  
Java HotSpot(TM) 64-Bit Server VM (build 23.5-b02, mixed mode)
```
3. Go to <http://www.eclipse.org/downloads/> and select the **Eclipse Classic** or **Eclipse Standard** distribution.
4. Download the one which matches the installed JDK. Running `java -version` should report either:
 - If it's a 32-bit JDK:

```
Java HotSpot(TM) Client VM
```
 - If it's a 64-bit JDK:

```
Java HotSpot(TM) 64-Bit Server VM
```



On Linux, Eclipse requires GTK2 to be installed. Most Linux distributions have a window manager based on GNOME that provides GTK2.x.

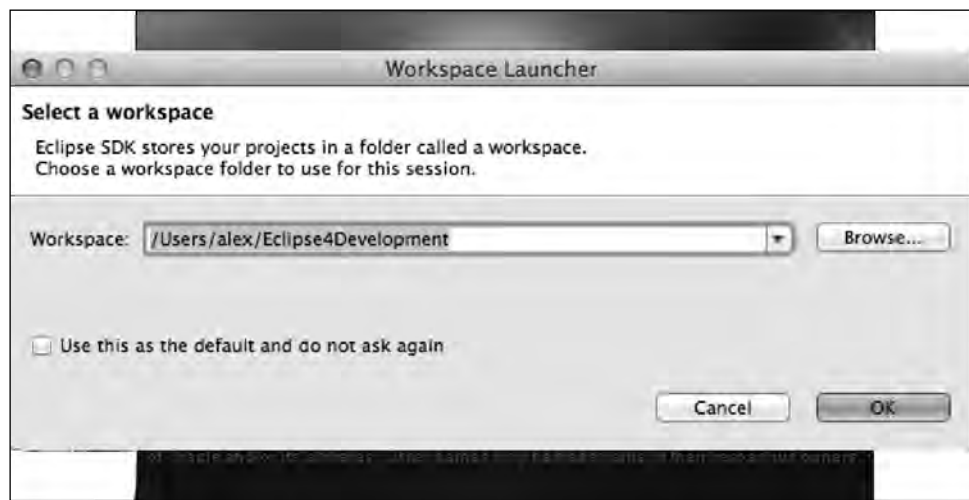
5. To install Eclipse, download and extract the contents to a suitable location. Eclipse is shipped as an archive, and needs no administrator privileges. Do not run it from a networked drive as this will cause performance problems.


Note that Eclipse needs to write to the folder from which it is extracted, so it's normal that the contents are writable afterwards. Generally, installing into /Applications or C:\Program Files, while being logged in with administrator account, is not recommended.

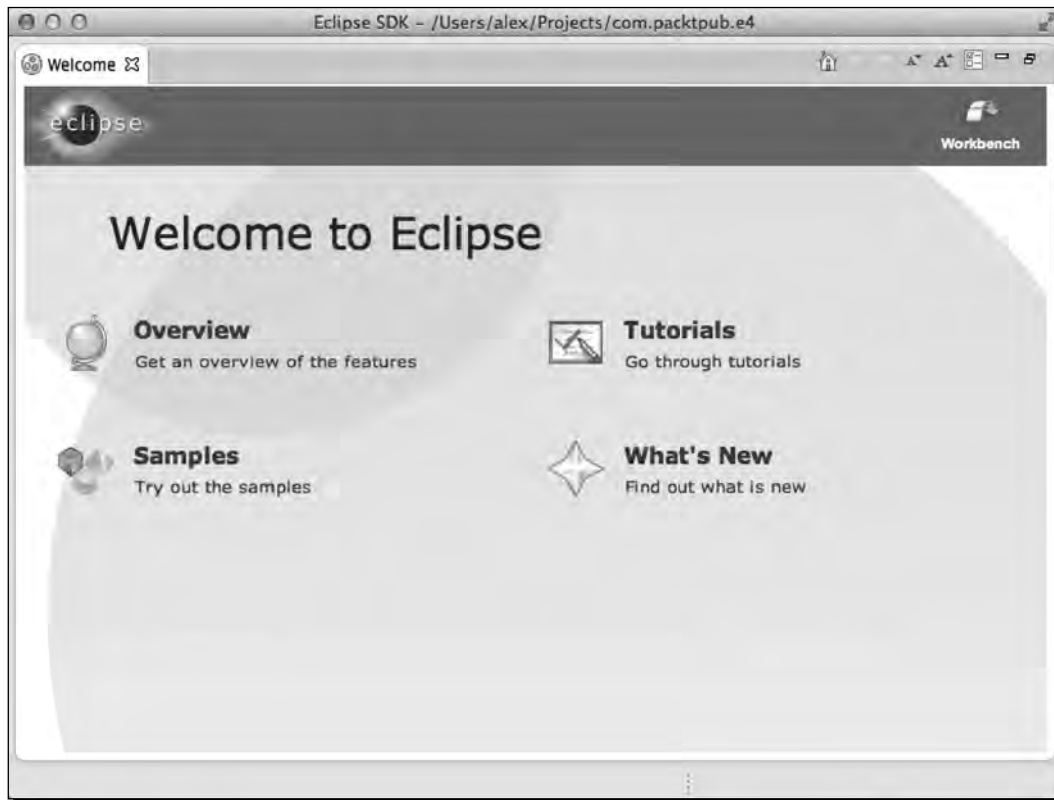
6. Run Eclipse by double-clicking on the Eclipse icon, or by running `eclipse.exe` (Windows), `eclipse` (Linux), or `Eclipse.app` (OS X).
7. Upon startup, the splash screen should be shown:



8. Choose a workspace, which is the location in which projects are be stored, and click on **OK**:



9. Close the welcome screen by clicking on the cross icon  in the tab next to the **Welcome** text. The welcome screen can be re-opened by navigating to **Help | Welcome**:



What just happened?

Eclipse needs Java to run, and so the first step involved in installing Eclipse is ensuring that an up-to-date Java installation is available. By default, Eclipse will find a copy of Java installed on the path or from one of the standard locations. It is also possible to specify a different Java version by using the `-vm` command-line argument.

If the splash screen doesn't show up, the Eclipse version may be incompatible with the JDK (for example, a 64-bit JDK with a 32-bit Eclipse, or vice-versa). Common error messages shown at the launcher may include `Unable to find companion launcher` or a cryptic message about being unable to find an SWT library.

On Windows, there is an additional `eclipse.exe` launcher, which allows log messages printed to the console to be seen. This is sometimes useful if Eclipse fails to load and no other message is displayed. Other operating systems can use the `eclipse` command. Both `eclipse.exe` and `eclipse` support the `-consolelog` argument, which can display more diagnostic information about problems while launching Eclipse.

The Eclipse workspace is a directory used for two purposes: as the default project location, and to hold the `.metadata` directory containing Eclipse settings, preferences, and other runtime information. The Eclipse runtime log is stored in the `.metadata/.log` file.

The workspace chooser dialog has an option to set the chosen value as the default workspace. It can be changed within Eclipse by navigating to **File | Switch Workspace**. It can also be overridden by specifying a different workspace location with the `-data` command-line argument.

Finally, the welcome screen is useful for first-time users, but is worth closing (rather than minimizing) once Eclipse has started.

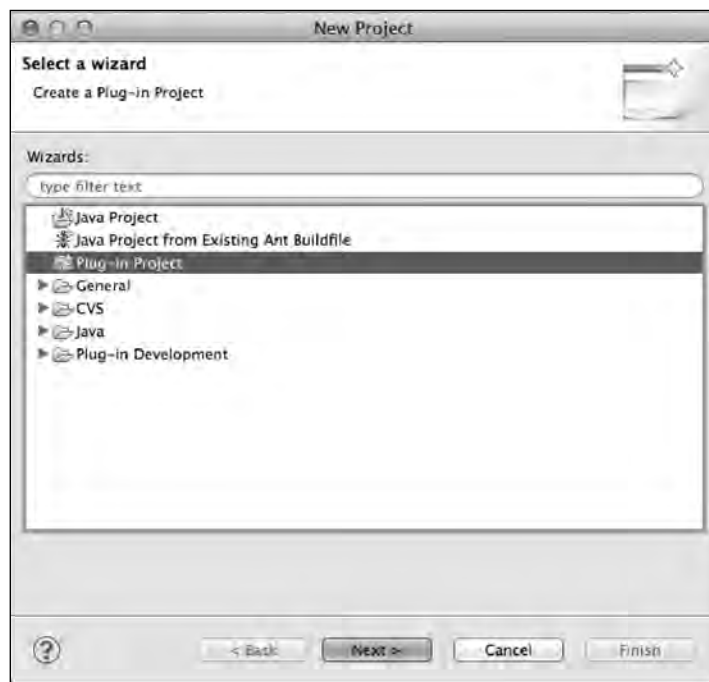
Creating your first plug-in

In this task, Eclipse's plug-in wizard will be used to create a plug-in.

Time for action – creating a plug-in

In the Plug-in Development Environment (PDE), every plug-in has its own individual project. A plug-in project is typically created with the **New Project** wizard, although it is possible to upgrade an existing Java project to a plug-in project by adding the PDE nature and the required files (by navigating to **Configure | Convert to plug-in project**).

1. To create a Hello World plugin, navigate to **File | New | Project**:



2. The project types shown may be different from this list, but should include **Plug-in Project** with Eclipse Classic. If nothing is shown under the **File | New** menu, navigate to **Window | Open Perspective | Other | Plug-in Development** first; the entries should then be seen under the **File | New** menu.
3. Choose **Plug-in Project** and click on **Next**. Fill in the dialog as follows:
 - ❑ **Project name:** `com.packtpub.e4.hello.ui`
 - ❑ Select the checkbox for **Use default location**
 - ❑ Select the checkbox for **Create a Java project**
 - ❑ **Target Eclipse Version:** 3.5 or greater
4. Click on **Next** again, and fill in the plug-in properties:
 - ❑ **ID:** `com.packtpub.e4.hello.ui`
 - ❑ **Version:** `1.0.0.qualifier`
 - ❑ **Name:** Hello
 - ❑ **Vendor:** PacktPub

- ❑ **Execution Environment:** Use the default (for example: JavaSE-1.6 or JavaSE-1.7)
 - ❑ Select the checkbox for **Generate an Activator**
 - ❑ **Activator:** `com.packtpub.e4.hello.ui.Activator`
 - ❑ Select the checkbox for **This plug-in will make contributions to the UI**
 - ❑ **Rich client application:** No
5. Click on **Next** and a set of templates will be provided:
 - ❑ Select the checkbox for **Create a plug-in using one of the templates**
 - ❑ Choose the **Hello World Command** template
 6. Click on **Next** to customize the sample, including:
 - ❑ The Java package name, which defaults to the project's name
 - ❑ The handler class name, which is the code that gets invoked for the action
 - ❑ The message box text, which is the message supplied
 7. Finally, click on **Finish** and the project will be generated.
 8. If a dialog asks, click on **Yes** to show the plug-in development perspective.

What just happened?

Creating a plug-in project is the first step towards creating a plug-in for Eclipse. The **New Plug-in Project** wizard was used with one of the sample templates to create a project.

Plug-ins are typically named in reverse domain name format, so these examples will be prefixed with `com.packtpub.e4`. This helps to distinguish between many plug-ins; the stock Eclipse SDK comes with more than 440 individual plug-ins, for example, the Eclipse-developed ones start with `org.eclipse`.



Conventionally, plug-ins which create additions to (or require) the use of the UI have `.ui` in the name. This helps to distinguish from those that don't, which can often be used headlessly. Of the 440+ plug-ins that make up the Eclipse SDK, 120 of those are UI related and the rest are headless.

The project contains a number of files which are automatically generated, based on the content filled in the wizard. The key files in an Eclipse plug-in are:

- ◆ **META-INF/MANIFEST.MF:** The OSGi manifest describes the plug-in's dependencies, version, and name. Double-clicking it will open a custom editor, which shows the information entered in the wizards; or it can be opened in a standard text editor.

The Manifest follows standard Java conventions; continuations are represented by a new line followed by a single space character, and the file must end with a new line. (For example, the maximum line length is 72 characters, although many ignore this.)

- ◆ `plugin.xml`: The `plugin.xml` file declares what extensions this plug-in provides to the Eclipse runtime. Not all plug-ins need a `plugin.xml` file; headless (non-UI) plug-ins often don't need to have one. Extension points will be covered in more detail later, but the sample project creates an extension for the commands, handlers, bindings, and menus extension points. (If the older Hello World template was chosen, present on 3.7 and older, only the `actionSets` extension will be used.)

Text labels for the commands, actions, or menus are represented declaratively in the `plugin.xml` file, rather than programmatically; this allows Eclipse to show the menu before needing to load or execute any code.



This is one of the reasons Eclipse starts so quickly; by not needing to load or execute classes, it can scale by showing what's needed at the time, and then load the class on demand when the user invokes the action. Java Swing's Actions provides labels and tool tips programmatically, which can result in a slower initialization of the user interface.

- ◆ `build.properties`: This file is used by PDE at development time and at build time. Generally it can be ignored, but if resources are added that need to be made available to the plug-in (such as images, properties files, HTML content, and so on), an entry must be added here as otherwise it won't be found. Generally, the easiest way to do this is by going to the **Build** tab of the `build.properties` file, which will give a tree-like view of the project's contents.

This file is an archaic hangover from the days of Ant builds, and is generally useless when using more up-to-date builds such as Maven Tycho, which will be covered in *Chapter 10, Automated Builds with Tycho*.

Pop quiz – Eclipse workspaces and plug-ins


- Q1. What is an Eclipse workspace?
- Q2. What is the naming convention for Eclipse plug-in projects?
- Q3. What are the names of the three key files in an Eclipse plug-in?

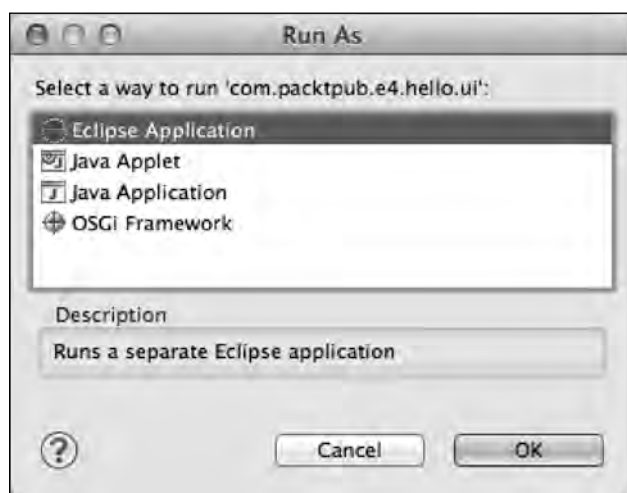
Running plug-ins

To test an Eclipse plug-in, Eclipse is used to run or debug a new Eclipse instance with the plug-in installed.

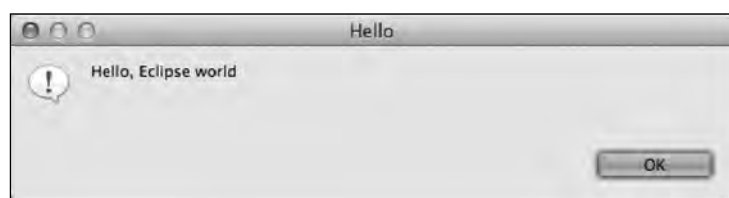
Time for action – launching Eclipse from within Eclipse

Eclipse can launch a new Eclipse application by clicking on the Run button, or via the **Run** menu.

1. Select the plug-in project in the workspace.
2. Click on the Run button  to launch the project. The first time this happens, a dialog will be shown; subsequent launches will remember the previous type:



3. Choose the **Eclipse Application** type and click on **OK**, and a new Eclipse instance will be launched.
4. Close the welcome page in the launched application, if shown.
5. Click on the Hello World icon in the menu bar, or navigate to **Sample Menu | Sample Command**, and the dialog box created via the wizard will be shown:




For More Information:

www.packtpub.com/eclipse-4-plugin-development-by-example-beginners-guide/book

6. Quit the test Eclipse instance by closing the window, or via the usual keyboard shortcuts or menus (*Cmd + Q* on OS X, *Alt + F4* on Windows).

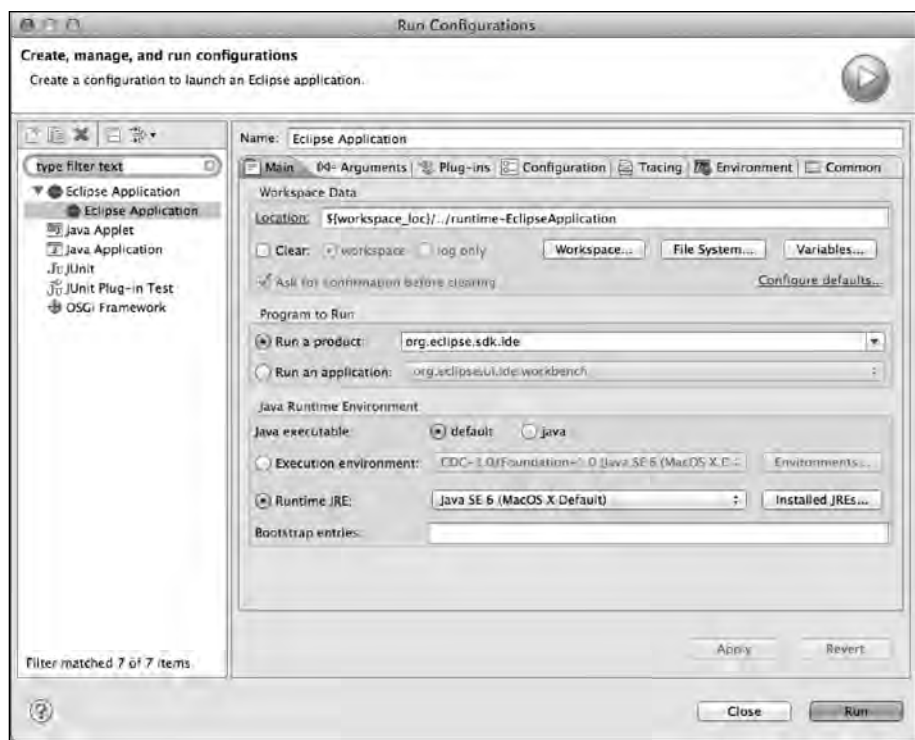
What just happened?

When clicking Run  in the toolbar (or via the **Run | Run As | Eclipse Application** menu) a launch configuration is created, which includes any plug-ins open in the workspace. A second copy of Eclipse—with its own temporary workspace—will enable the plug-in to be tested and verified so that it works as expected.



The Run operation is intelligent, in that it launches an application based on what is selected in the workspace. If a plug-in is selected, it will offer the opportunity to run as an Eclipse application; if a Java project with a class with a `main` method, it will run it as a standard Java application; and if it has tests then it will offer to run the test launcher instead.

However, the Run operation can also be counter-intuitive; if clicked a second time, and in a different project context, something other than the expected launch might be run.

A list of the available launch configurations can be seen by going to the Run menu, or by going to the drop-down list to the right of the Run icon. The **Run | Run Configurations** menu shows all the available types, including any previously run:




By default, the runtime workspace is kept between runs. The launch configuration for an Eclipse application has options that can be customized; in the previous screenshot, the **Workspace Data** section in the **Main** tab shows where the runtime workspace is stored, and an option is shown that allows the workspace to be cleared (with or without confirmation) between runs.

Launch configurations can be deleted by clicking on the red Delete icon  on the top-left, and new launch configurations can be created by clicking on the New icon . Each launch configuration has a type:

- ◆ **Eclipse Application**
- ◆ **Java Applet**
- ◆ **Java Application**
- ◆ **JUnit**
- ◆ **JUnit Plug-in Test**
- ◆ **OSGi Framework**

The launch configuration can be thought of as a precanned script, which can launch different types of programs. Additional tabs are used to customize the launch, such as the environment variables, system properties, or command-line arguments. The type of the launch configuration specifies what parameters are required, and how the launch is executed.

When a program is launched with the Run icon, changes to the project's source code do not take effect. However, as we'll see in the next section, if it's launched with the Debug icon, changes can take effect.

If the test Eclipse is hanging or otherwise unresponsive, in the host Eclipse instance the **Console** view (shown with the **Window | View | Show View | Other | General | Console** menu) can be used to stop () the test Eclipse instance.

Pop quiz – launching Eclipse

Q1. What are the two ways of terminating a launched Eclipse instance?

Q2. What are launch configurations?

Q3. How do you create and delete launch configurations?

Have a go hero – modifying the plug-in

Now that you've got the Eclipse plug-in running, try the following:

- ◆ Change the message of the label and title of the dialog box to something else
- ◆ Invoke the action by using the keyboard shortcut (defined in `plugin.xml`)
- ◆ Change the tool tip of the action to a different message
- ◆ Switch the action icon to a different graphic (note that if you use a different filename, remember to update `build.properties`).

Debugging a plug-in

Since it's rare that everything works first time, it's often necessary to develop iteratively, adding progressively more functionality each time. Secondly, sometimes it's necessary to find out what's going on under the covers when trying to fix a bug, particularly if it's hard to track down exceptions such as `NullPointerException`.


Fortunately, Eclipse comes with excellent debugging support, which can be used for debugging both standalone Java applications as well as Eclipse plug-ins.


Time for action – debugging a plug-in

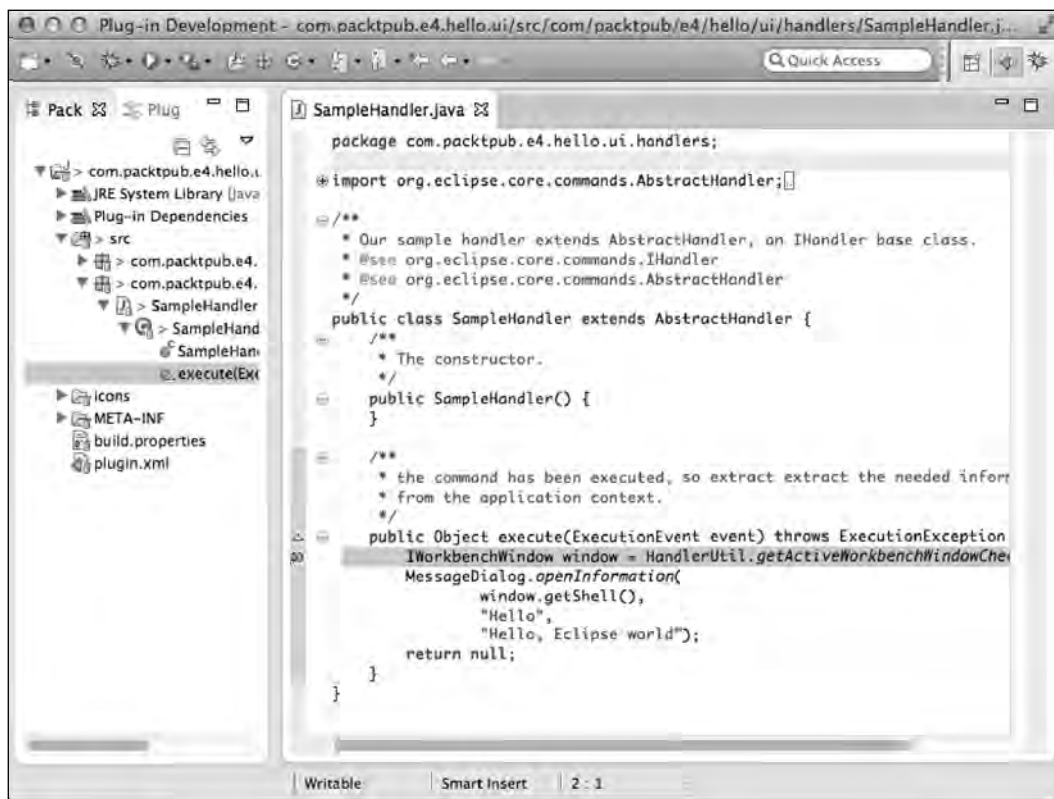
Debugging an Eclipse plug-in is almost the same as running an Eclipse plug-in, except that breakpoints can be used, and the state of the program can be updated, variables, and minor changes to the code can be done. Rather than debugging plug-ins individually, the entire Eclipse launch configuration is started in debug mode. That way, all the plug-ins can be debugged at the same time.


Although run mode is slightly faster, the added flexibility of being able to make changes makes debug mode much more attractive to use as a default.

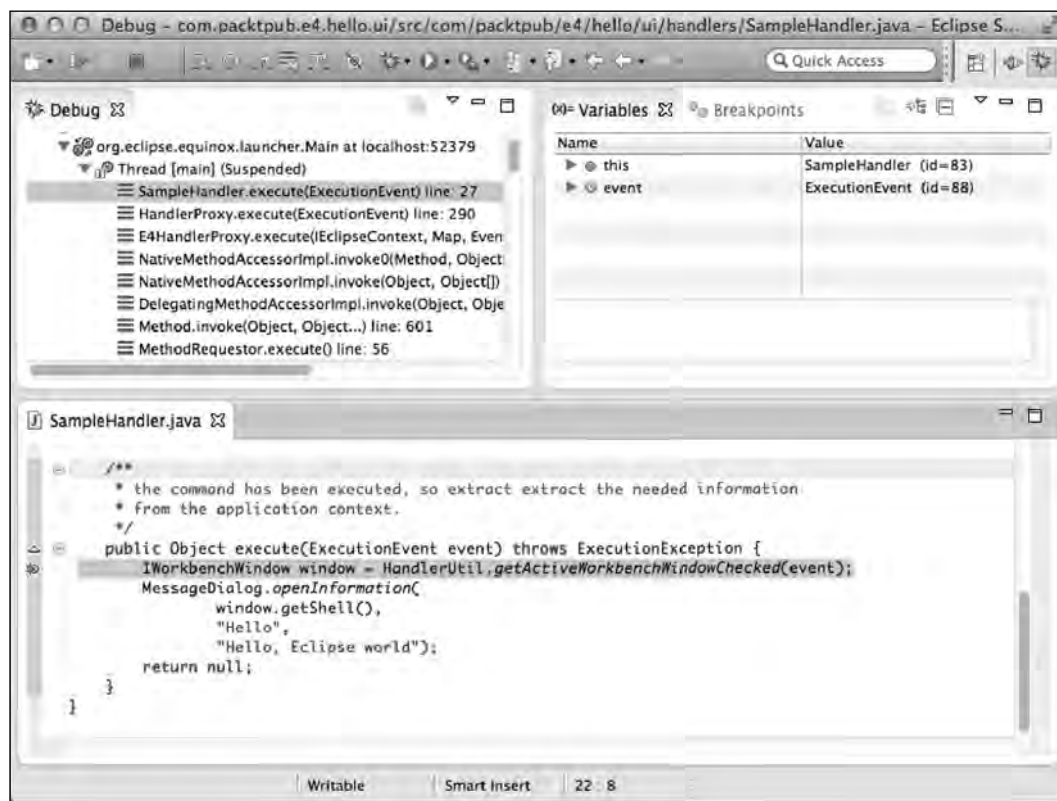
Start the test Eclipse, by navigating to the **Debug | Debug As | Eclipse Application** menu, or by clicking on Debug () in the toolbar.


1. Click on the Hello World icon () in the test Eclipse to display the dialog, as before, and click on **OK** to dismiss it.

2. In the host Eclipse, open the `SampleHandler` class and go to the `execute()` method.
3. Add a breakpoint by double-clicking in the vertical ruler (the gray/blue bar on the left of the editor), or by pressing `Ctrl + Shift + B` (or `Cmd + Shift + B` on OS X). A blue dot  representing the breakpoint will appear in the ruler:




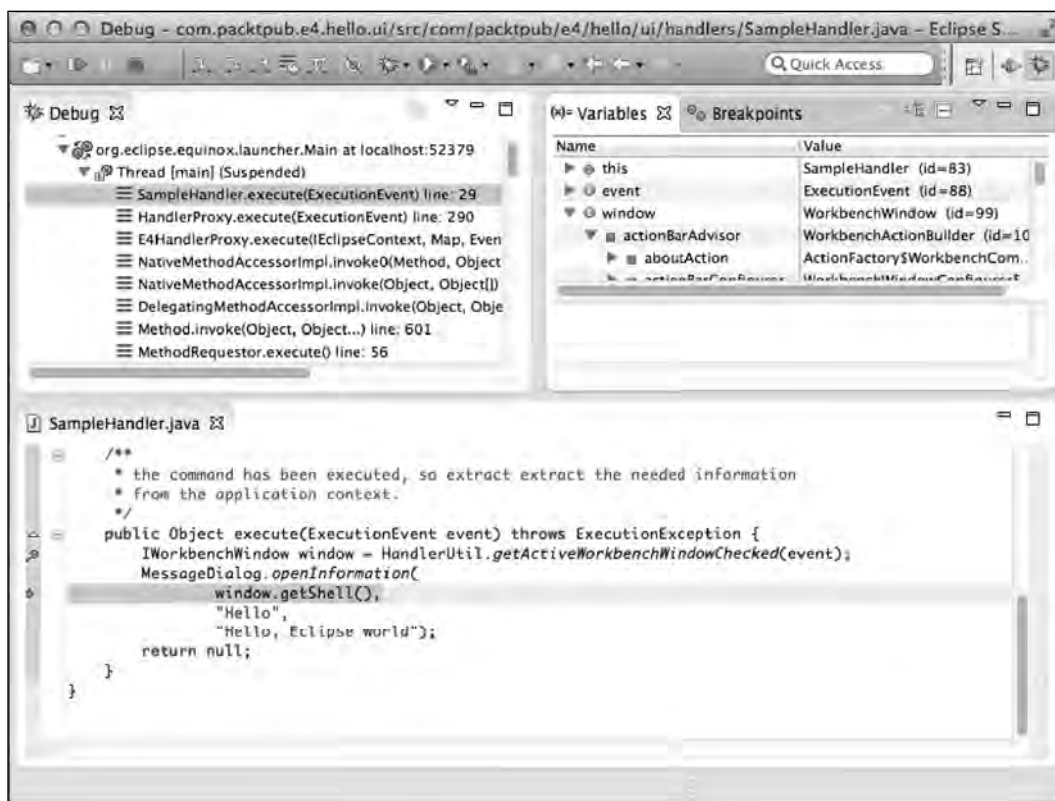
4. Click on the Hello World icon  in the test Eclipse to display the dialog, and the debugger will pause the thread at the breakpoint in the host Eclipse:



 The debugger perspective will open whenever a breakpoint is triggered and the program will be paused. While it is paused, the test Eclipse is unresponsive. Any clicks on the test Eclipse application will be ignored, and it will show a busy cursor.

5. On the top-right, variables that are active in the line of code are shown. In this case, it's just the implicit variables (via `this`), any local variables (none, yet) as well as the parameter (in this case, `event`).

6. Click on Step Over  or press *F6*, and window will be added to the list of available variables:





7. When ready to continue, click on Resume  or press *F8* to keep running.

What just happened?



The built-in Eclipse debugger was used to launch Eclipse in debug mode. By triggering an action which led to a breakpoint, the debugger was revealed allowing the local variables to be introspected.

When in the debugger, there are several options available for stepping through the code:

- ◆ Step Over  – allows stepping over line-by-line in the method
- ◆ Step Into  – follow the method calls recursively as execution unfolds





There is also a **Run | Step into Selection** menu item that does not have a toolbar icon. It can be invoked with *Ctrl + F5* (*Alt + F5* on OS X), and is used to step into a specific expression.

- ◆ Step Return  – jump to the end of a method
- ◆ Drop to Frame  – return to a stack frame in the thread to re-run an operation

Time for action – updating code in debugger

When an Eclipse instance is launched in run mode, changes made to the source code aren't reflected in the running instance. However, debug mode allows changes made to the source to be reflected in the running test Eclipse instance.

1. Launch the test Eclipse in debug mode by clicking on the Debug  icon.
2. Click on the Hello World icon  in the test Eclipse to display the dialog, as before, and click on **OK** to dismiss it. It may be necessary to remove or resume the breakpoint in the host Eclipse instance to allow execution to continue.
3. In the host Eclipse, open the `SampleHandler` class and go to the `execute()` method.
4. Change the title of the dialog to `Hello again, Eclipse world` and save the file. Provided that **Project | Build Automatically** is enabled, the change will be recompiled.
5. Click on the Hello World icon in the test Eclipse instance again. The new message should be shown.

What just happened?

By default, Eclipse ships with **Project | Build Automatically** enabled. Whenever changes are made to Java files, they are recompiled along with their dependencies if necessary.

When a Java program is launched in run mode, it will load classes in on-demand and then keep using that definition until the JVM shuts down. Even if the classes are changed, the JVM won't notice that they have been updated, and so no differences will be seen in the running application.

However, when a Java program is launched in debug mode, whenever changes to classes are made, it will update the running JVM with the new code if possible. The limits to what can be replaced are controlled by the JVM through the JVMTI and whether, for example, the virtual machine's `canUnrestrictedlyRedefineClasses()` call returns `true`. Generally, updating an existing method and adding a new method or field will work, but changes to interfaces and super classes may not be. (Refer to http://en.wikipedia.org/wiki/Java_Virtual_Machine_Tools_Interface for more information.)



The ex-Sun Hotspot JVM cannot replace classes if methods are added or interfaces are updated. Some JVMs have additional capabilities which can substitute more code on demand. With the merging of JRockit and Hotspot over time, more may be replaceable at runtime than before; for everything else, there's JRebel.

Other JVMs, such as IBM's, can deal with a wider range of replacements.



Note that there are some changes which won't be picked up; for example, new extensions added to the `plugin.xml` file. In order to see these changes, it is possible to start and stop the plug-in through the command-line OSGi console, or restart Eclipse inside or outside the host Eclipse to see the change.

Debugging with step filters

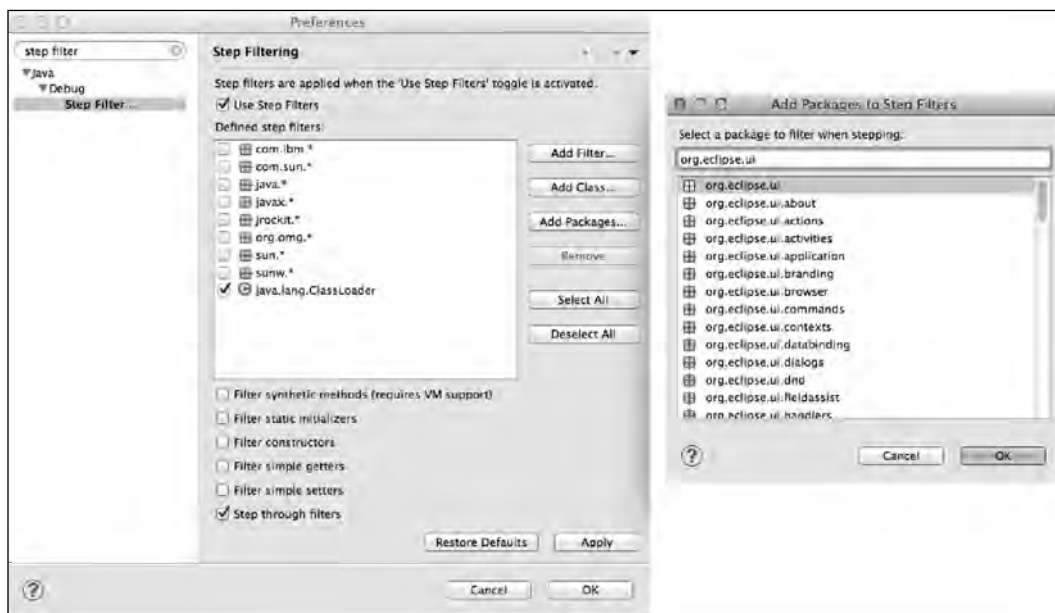
When debugging using Step Into, the code will frequently go into Java internals, such as the implementation of Java collections classes or other internal JVM classes. These don't usually add value, but fortunately Eclipse has a way of ignoring uninteresting classes.




Time for action – setting up step filtering

Step filtering allows for uninteresting packages and classes to be ignored during step debugging.

1. Run the test Eclipse application in debug mode.
2. Ensure a breakpoint is set at the start of the `SampleHandler` class's `execute()` method.
3. Click on the Hello World icon, and the debugger should open at the first line as before.
4. Click on Step Into  five or six times. At each point, the code will jump into the next method in the expression; first through various methods in `HandlerUtil` and then into `ExecutionEvent`.
5. Click on Resume  to continue.
6. Open **Preferences**, and then navigate to **Java | Debug | Step Filtering**.
7. Check the **Use Step Filters** option.

8. Click on **Add Package** and enter `org.eclipse.ui`, followed by clicking on **OK**.



9. Click on the Hello World icon again.
10. Click on Step Into  as before. This time, the debugger goes straight to `getApplicationContext()` in the `ExecutionEvent` class.
11. Click on the Resume icon  to continue.
12. To make debugging more efficient by skipping accessors, go back into the **Step Filters** preference and select the **Filter Simple Getters** from the **Step Filters** preference's page.
13. Click on the Hello World icon again.
14. Click on Step Into  as before.
15. Instead of going into the `getApplicationContext()` method, execution will drop through to the `ExecutionContext` class's `getVariable()` method instead.

What just happened?

The **Step Filters** preferences allows uninteresting packages to be skipped, at least from the point of debugging. Typically, JVM internal classes (such as those beginning with `sun` or `sunw`) are not helpful when debugging and can easily be ignored. This also avoids debugging through the class loader, as it loads classes on demand.

Typically, it makes sense to enable all the default packages in the Step Filters dialog, as it's pretty rare to need to debug any of the JVM libraries (internal or public interfaces). This means when stepping through the code, if a common method such as `List.toString()` is called, debugging won't step through the internal implementation.

It also makes sense to filter out simple setters and getters (those that just set a variable, or those that just return a variable). If the method is more complex (like the `getVariable()` method discussed previously), it will still stop in the debugger.

Constructors and static initializers can also be specifically filtered.

Using different breakpoint types

Although it's possible to place a breakpoint anywhere in a method, a special breakpoint type exists, which can fire on method entry, exit, or both. Breakpoints can also be customized to only fire in certain situations or when certain conditions are met.

Time for action – breaking at method entry and exit

Method breakpoints allow the user to see when a method is entered or exited.

1. Open the `SampleHandler` class, and go to the `execute()` method.
2. Double-click in the vertical ruler at the method signature, or select **Toggle Method Breakpoint** from the method in one of the **Outline**, **Package Explorer**, or **Members** views.
3. The breakpoint should be shown on the line `public Object execute(...)`
`throws ExecutionException {`.
4. Open the breakpoint properties by right-clicking on the breakpoint or via the **Breakpoints** view, which is shown in the debug perspective. Set the breakpoint to trigger at the method entry and method exit.
5. Click the Hello World icon again.
6. When the debugger stops at the method entry, click on the Resume icon.
7. When the debugger stops at the method exit, click on the Resume icon.


What just happened?

The breakpoint triggers at the time the method enters and subsequently when the method's `return` statement is reached.

Note that the exit is only triggered if the method returns normally; if an exception is raised which causes the method to return, this is not treated as a normal method exit, and so the breakpoint won't fire.

Other than the breakpoint type, there's not a significant difference between creating a breakpoint on method entry and creating one on the first statement of the method. Both give the ability to introspect the parameters and do further debugging prior to any statements in the method itself are called.

The method exit breakpoint, on the other hand, will only trigger once the `return` statement is about to leave the method. Thus, any expression in the method's return value will have been evaluated prior to the exit breakpoint firing. Compare and contrast this to the line breakpoint, which will wait to evaluate the argument of the `return` statement.

Note that Eclipse's Step Return  icon has the same effect; this will run until the method's `return` statement is about to be executed. However, to find when a method returns, using a method exit breakpoint is far faster than stopping at a specific line and then clicking on Step Return.

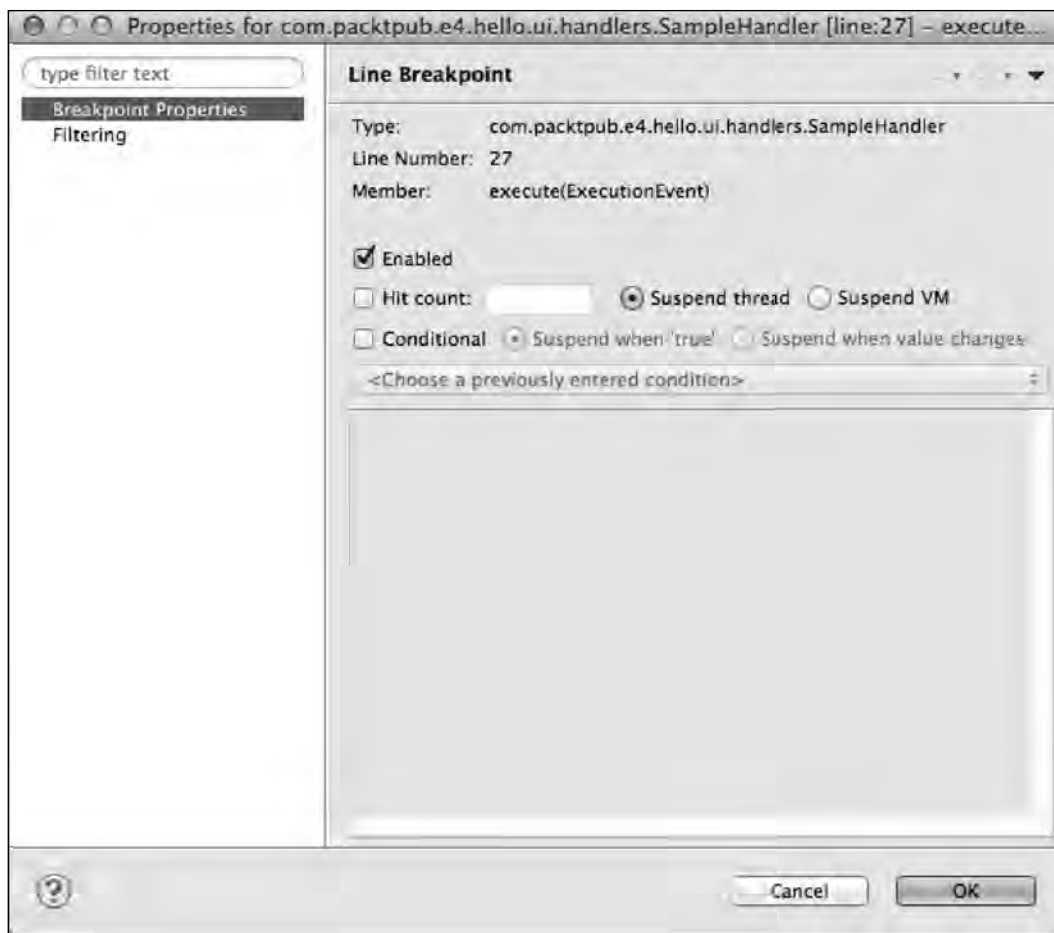
Using conditional breakpoints

Breakpoints are useful, since they can be invoked on every occasion that a line of code is triggered. However, sometimes they need to break for specific actions only, such as when a particular option is set, or when a value has been incorrectly initialized. Fortunately, this can be done with **conditional breakpoints**.

Time for action – setting a conditional breakpoint

Normally, breakpoints fire on each invocation. It is possible to configure breakpoints such that they fire when certain conditions are met.

1. Go to the `execute()` method of the `SampleHandler` class.
2. Clear any existing breakpoints by double-clicking them or using **Delete all breakpoints** from the **Breakpoints** view.
3. Add a breakpoint to the first line of the `execute()` method body.
4. Right-click on the breakpoint and select the **Breakpoint Properties** menu. (It can also be shown by *Ctrl* + double-clicking, or *Cmd* + double-clicking on OS X on the breakpoint icon itself.)




5. Set the **Hit Count** field to 3, and click on **OK**.
6. Click on the Hello World icon button three times. On the third click, the debugger will open up at that line of code.
7. Open the breakpoint properties, deselect **Hit Count** and select the **Enabled** and **Conditional** options. Put the following code into the conditional trigger field:

```
((org.eclipse.swt.widgets.Event)event.trigger).stateMask == 65536
```
8. Click on the Hello World icon and the breakpoint will not fire.
9. Hold down *Alt*, click on the Hello World icon, and the debugger will open. (65536 is the value of `SWT.MOD3`, which is the *Alt* key.)

What just happened?

When a breakpoint is created, it is enabled by default. A breakpoint can be temporarily disabled, which has the effect of removing it from the flow of execution. Disabled breakpoints can be trivially re-enabled on a per-breakpoint basis, or from the **Breakpoints** view. Quite often it's useful to have a set of breakpoints defined in the codebase, but not necessarily have them all enabled at once.

It is also possible to temporarily disable all breakpoints using the **Skip All Breakpoints** setting, which can be changed from the corresponding item in the **Run** menu (when the debug perspective is enabled), or the corresponding icon  in the **Breakpoints** view. When this is toggled, no breakpoints will be fired.

Conditional breakpoints must use a Boolean expression, rather than a statement or a set of statements. Sometimes this is constraining; if that's the case, having a utility class with a static method allows more complex code paths (with the caveat that all interesting data must be passed in as method arguments).

Using exceptional breakpoints

Sometimes when debugging a program, an exception occurs. Typically this isn't known about until it happens, when an exception message is printed or displayed to the user via some kind of dialog box.

Time for action – catching exceptions

Although it's trivial to put a breakpoint in the `catch` block, this is merely the location where the failure was ultimately caught, not where it was caused. The place where it was caught can often be in a completely different plug-in from where it was raised, and depending on the amount of information encoded within the exception (particularly if it has been transliterated into a different exception type), it may hide the original source of the problem. Fortunately, Eclipse can handle such cases with a Java Exception breakpoint.

1. Introduce a bug into the `SampleHandler` class's `execute()` method, by adding the following just before the `MessageDialog.openInformation()` call:

```
window = null;
```




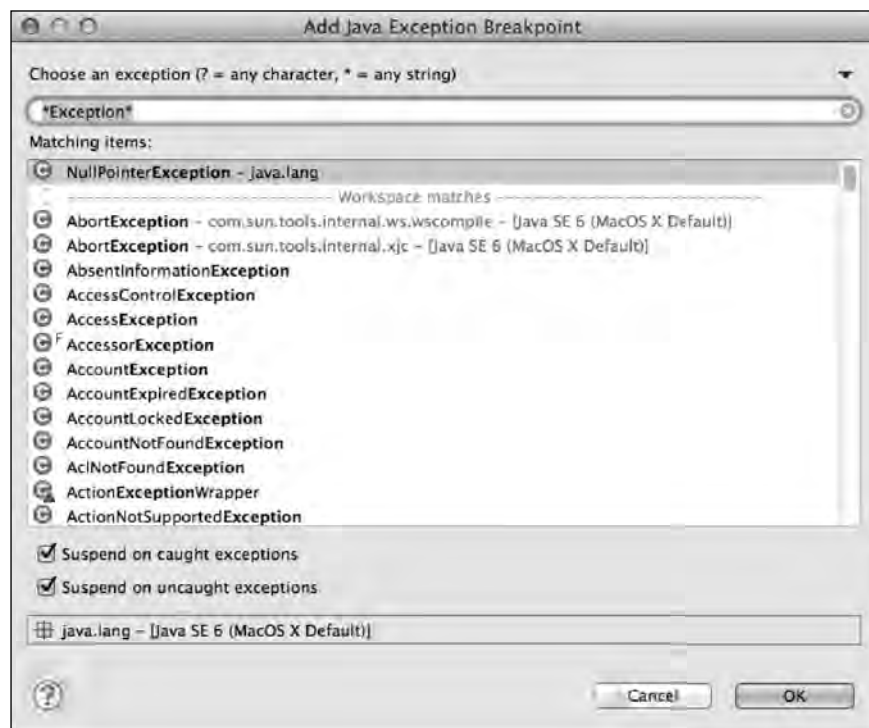
Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

2. Click on the Hello World icon.
3. Nothing will appear to happen in the target Eclipse, but in the **Console** view of the host Eclipse instance, the following error message should be seen:

```
Caused by: java.lang.NullPointerException
    at com.packtpub.e4.hello.ui.handlers.SampleHandler.execute(SampleHandler.java:30)
    at org.eclipse.ui.internal.handlers.HandlerProxy.execute(HandlerProxy.java:293)
    at org.eclipse.ui.internal.handlers.E4HandlerProxy.execute(E4HandlerProxy.java:76)
```

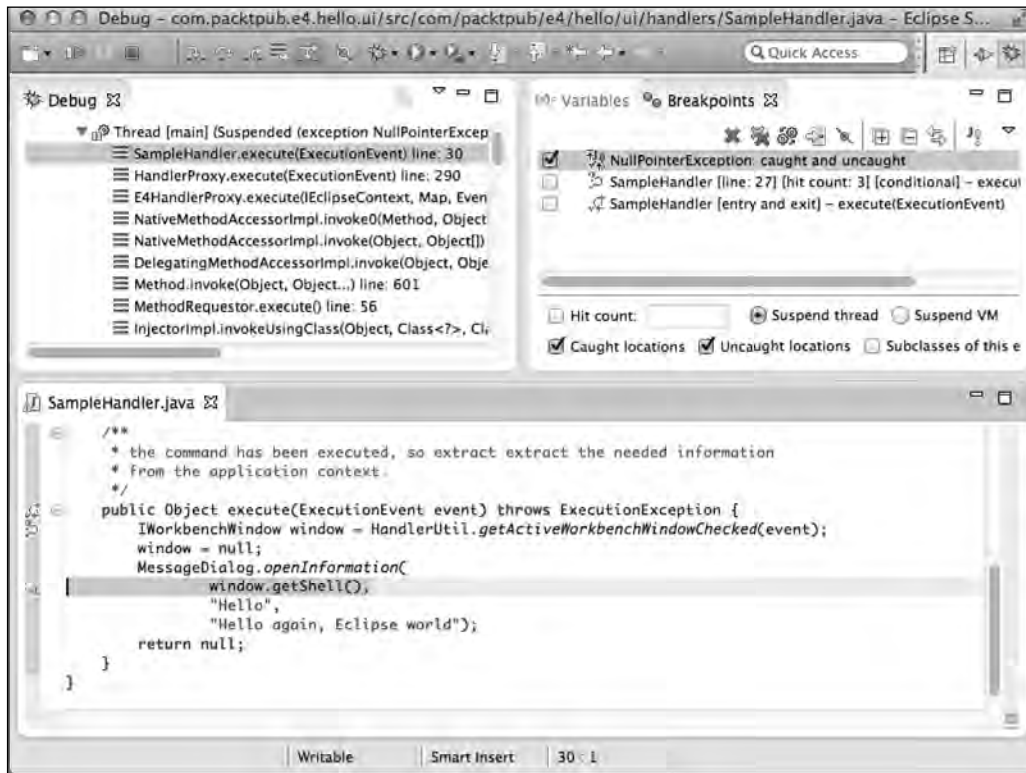
4. Create a Java Exception breakpoint  in the **Breakpoints** view of the debug perspective. The exception dialog will be shown as follows:



For More Information:

www.packtpub.com/eclipse-4-plugin-development-by-example-beginners-guide/book

5. Enter `NullPointerException` into the search dialog and click on **OK**.
6. Click on the Hello World icon and the debugger will stop at the line the exception is thrown, instead of where it is caught:



What just happened?

The Java Exception breakpoint stops when an exception is thrown, not when it is caught. The dialog asks for a single exception class to catch and by default, the wizard has been prefilled with any class whose name includes `*Exception*`. However, any name (or filter) can be typed into the search box, including abbreviations such as `FNFE` for `FileNotFoundException`. Wildcard patterns can also be used, which allows searching for `Nu*Ex` or `*Unknown*`.

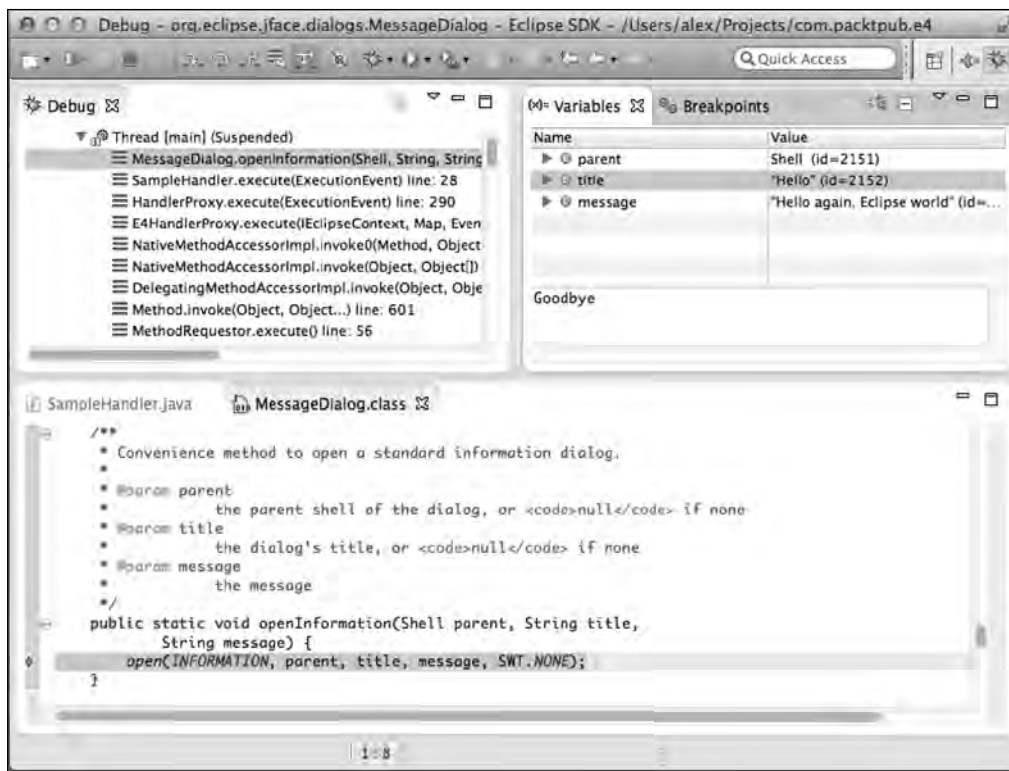
By default, the exception breakpoint corresponds to instances of that specific class. This is useful (and quick) for exceptions such as `NullPointerException`, but not so useful for ones with an extensive class hierarchy such as `IOException`. In this case, there is a checkbox visible in the **Breakpoint** properties window and the bottom of the **Breakpoints** view, which allows the selection of all subclasses of that exception, not just of the specific class itself.

There are also two other checkboxes, which say whether the debugger should stop when the exception is caught or uncaught. Both of these are selected by default; if both are deselected, the breakpoint effectively becomes disabled. Caught means that the exception is thrown in a corresponding `try/catch` block, and Uncaught means that the exception is thrown without a `try/catch` block (thus, bubbles up to the method's caller).

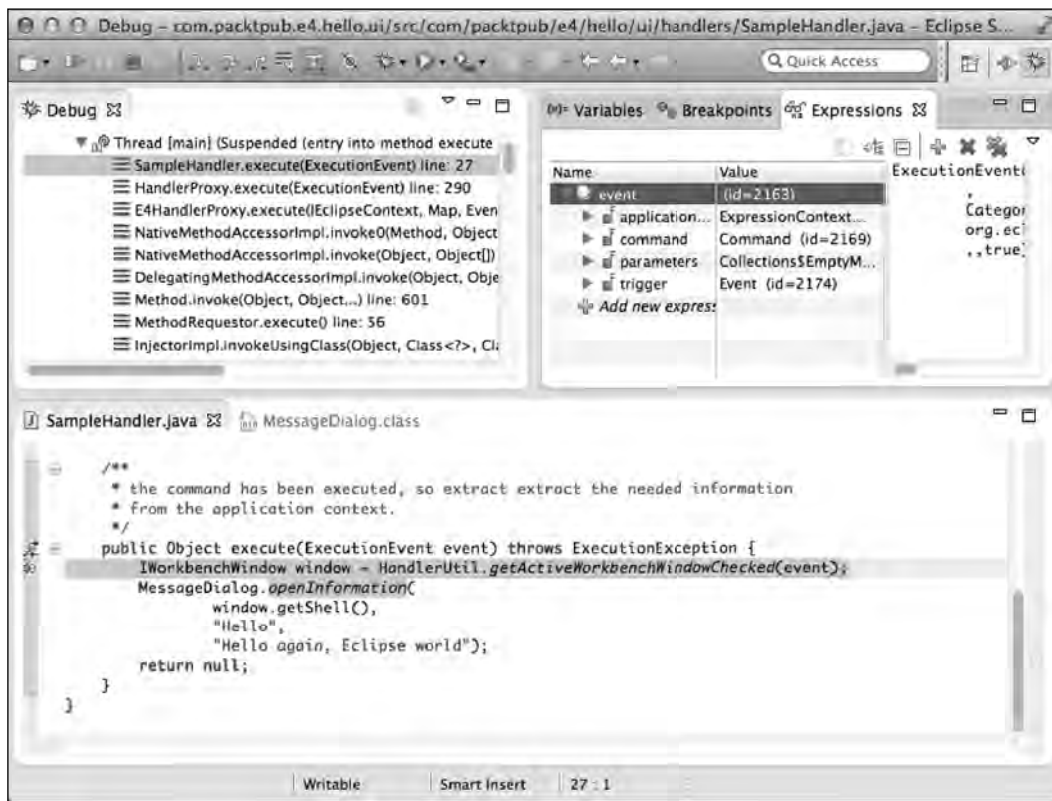
Time for action – using watch variables and expressions

Finally, it's worth seeing what the **Variables** view can do.

1. Create a breakpoint at the start of the `execute()` method.
2. Click on the Hello World icon again.
3. Highlight the `openInformation()` call and navigate to **Run | Step Into Selection**.
4. Select the `title` in the the **Variables** view.
5. Modify where it says Hello in the bottom half of the **Variables** view and change it to Goodbye:



6. Save the value using *Ctrl + S* (or *Cmd + S* on OS X).
7. Click on the Resume icon and the newly updated title can be seen in the dialog.
8. Click on the Hello World icon again.
9. With the debugger stopped in the `execute()` method, highlight `event` in the **Variables** view.
10. Right-click on the value and choose **Inspect** (*Ctrl + Shift + I* or *Cmd + Shift + I* on an OS X) and the value is opened in the **Expressions** view:



11. Click on the **Add new expression** option in the bottom of the **Variables** view.
12. Add new `java.util.Date()` and the right-hand side will show the current time.
13. Right-click on the new `java.util.Date()` and choose **Re-evaluate Watch Expression**. The right-hand pane shows the new value.
14. Step through the code line by line, and notice that the watch expression is reevaluated after each step.

- 15.** Disable the watch expression by right-clicking on it and choosing **Disable**.
- 16.** Step through the code line by line and the watch expression will not be updated.

What just happened?

The Eclipse debugger has many powerful features, and the ability to inspect (and change) the state of the program is one of the more important ones.

Watch expressions, when combined with conditional breakpoints, can be used to find out when data becomes corrupted or used to show the state of a particular object's value.

Expressions can also be evaluated based on objects in the **Variables** view, and the code completion is available to select methods, with the result being shown with **Display**.

Pop quiz – debugging

- Q1. How can an Eclipse plug-in be launched in debug mode?
- Q2. How can certain packages be avoided when debugging?
- Q3. What are the different types of breakpoints that can be set?
- Q4. How can a loop be debugged that only exhibits a bug after 256 iterations?
- Q5. How can a breakpoint be set on a method when its argument is null?
- Q6. What does inspecting an object do?
- Q7. How can the value of an expression be calculated?

Have a go hero – working with breakpoints

Using the conditional breakpoint to stop at a certain method is fine if the data is simple, but sometimes there needs to be more than one expression. To implement additional functionality, the breakpoint can be delegated to a `breakpoint()` method in a `Utility` class. The following steps will help you to work with breakpoints:

1. Create a `Utility` class with a static method `breakpoint()`, this will return a `true` value if the breakpoint should stop and `false` otherwise:

```
public class Utility {
    public static boolean breakpoint() {
        System.out.println("Breakpoint");
        return false;
    }
}
```


2. Create a conditional breakpoint in the `execute()` method, which calls `Utility.breakpoint()`.
3. Click on the Hello World icon again and the message will be printed to the host Eclipse's **Console** view. The breakpoint will not stop.
4. Modify the `breakpoint()` method to return `true` instead of `false`. Run the action again. The debugger will stop.
5. Modify the `breakpoint()` method to take the message as an argument, along with a Boolean value that is returned to say if the breakpoint should stop.
6. Set up a conditional breakpoint with the following code:

```
Utility.breakpoint(  
    ((org.eclipse.swt.widgets.Event)event.trigger).stateMask  
    != 0, "Breakpoint")
```

7. Modify the `breakpoint()` method to take a `varargs Object` array, and use that in conjunction with the message to use `String.format()` for the resulting message:

```
Utility.breakpoint( ((org.eclipse.swt.widgets.Event)event.  
trigger).stateMask  
    != 0, "Breakpoint" %s %h", event,  
    new java.util.Date().getTime()))
```

Summary

In this chapter, we covered how to get started with Eclipse plug-in development. From downloading the right Eclipse package (from a bewildering array of choices) to getting started with a wizard-generated plug-in, you should now have the tools to follow through with the remainder of the chapters in this book.

Specifically, we covered:

- ◆ The Eclipse SDK (also known as Eclipse Classic) has the necessary Plug-in Development Environment to get you started
- ◆ The plug-in creation wizard can be used to create a plug-in project, optionally using one of the example templates
- ◆ Testing an Eclipse plug-in launches a second copy of Eclipse with the plug-in installed and available for use
- ◆ Launching Eclipse in debug mode allows you to update code and stop execution at breakpoints defined via the editor

Now that we've learned about how to get started with Eclipse plug-ins, we're ready to look at creating plug-ins which contribute to the IDE; starting with **SWT** and **Views**, which is the topic of the next chapter.

Where to buy this book

You can buy Eclipse 4 Plug-in Development by Example Beginner's Guide from the Packt Publishing website: <http://www.packtpub.com/eclipse-4-plugin-development-by-example-beginners-guide/book>.

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our [shipping policy](#).

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.



www.PacktPub.com

For More Information:

www.packtpub.com/eclipse-4-plugin-development-by-example-beginners-guide/book