

# Debugging with the Eclipse Platform

Use Eclipse for debugging your software projects

Chris Aniszczyk, Software Engineer, IBM

Pawel Leszek ([pawel@eva-consulting.pl](mailto:pawel@eva-consulting.pl)), Independent Software Consultant, Freelance

**Summary:** Find out how to use the built-in debugging features in the Eclipse Platform for debugging software projects. Debugging is something programmers can't avoid. There are many ways to go about it, but it essentially comes down to finding the code responsible for a bug. For example, one of the most common errors in Linux® applications is known as a *segmentation fault*. This occurs when a program attempts to access memory not allocated to it and terminates with a segmentation violation. To fix this kind of error, you need to find the line of code that triggers the behavior. Once the line of code in question has been found, it is also useful to know the context in which the error occurs, and the associated values, variables, and methods. The use of a debugger makes finding this information quite simple.

**Date:** 01 May 2007

**Level:** Intermediate

**Also available in:** [Chinese](#) [Korean](#) [Japanese](#)

**Activity:** 233088 views

**Comments:** 3 ([View](#) | [Add comment](#) - [Sign in](#))



Average rating (199 votes)

[Rate this article](#)

- **Show articles and other content related to my search: eclipse debugger**

*Editor's note:* The following article, originally written by Pawel Leszek in May 2003, was updated in April 2007 by Chris Aniszczyk.

The Eclipse debugger and the Debug view

The Eclipse SDK -- particularly, the Java™ Development Tools (JDT) project -- features a built-in Java debugger that provides all standard debugging functionality, including the ability to perform step execution, to set breakpoints and values, to inspect variables and values, and to suspend and resume threads. Additionally, you can debug applications running on a remote machine. The Eclipse Platform is robust in such a way that other programming languages can use the debug facilities for their respective language runtimes. As you will see below, the same Eclipse Debug view is also available for the C/C++ programming languages.

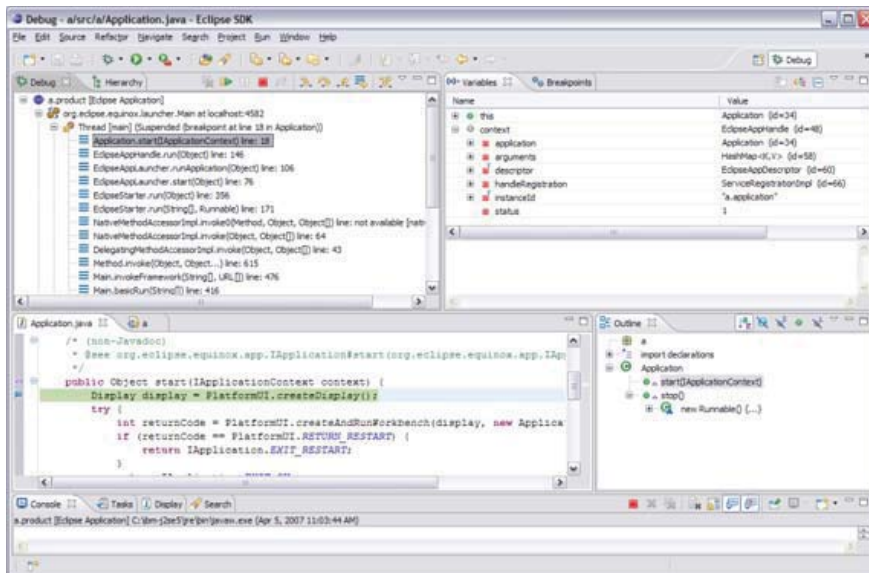
The Eclipse Platform Workbench and its tools are built around the JDT components, which provide the following features to Eclipse:

- Project management tools
- Perspectives and views
- Builder, editor, search, and build functions
- The debugger

The Eclipse debugger itself exists as a standard set of plug-ins included within Eclipse. Eclipse also has a special Debug view that allows you to manage the debugging or running of a program in the Workbench. It displays the stack frame for the suspended threads for each target you're debugging. Each thread in your program appears as a node in the tree, and the Debug view displays the process for each target you're running. If the thread is suspended, its stack frames are shown as child elements.

Before you begin using the Eclipse debugger, it is assumed that you have the appropriate Java SDK/JRE (I recommend you use Java VM V1.4) and the Eclipse Platform SDK V3.3 installed, and that both are running without problems. In general, it's a good idea to test debugging options using the Eclipse samples first. If you want to develop and debug C/C++ projects, you will also need to get and install the C/C++ Development Tools (CDT). For links to the Java SDK/JRE, the Eclipse Platform and samples, and the CDT, see [Resources](#). Figure 1 shows the general view of the Debug perspective.

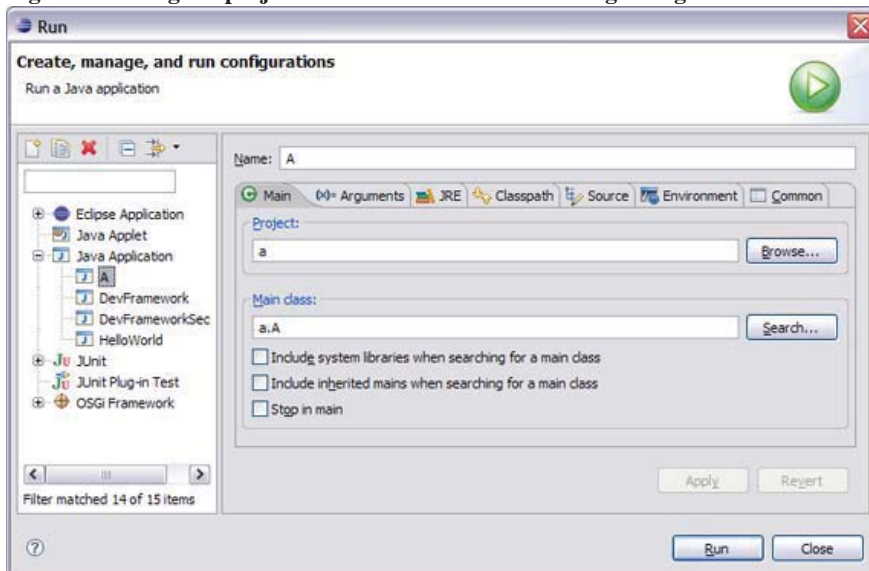
**Figure 1. General view of Eclipse Debug perspective**



## Debugging Java language programs

Before you are able to debug your project, the code needs to compile and run cleanly. You first need to create a run configuration for your application and make sure it starts properly. Next, you need to set up the debug configuration in a similar way using the **Run > Debug** menu. You also need to select the class to be used as the main Java class by the debugger (see Figure 2). You can have as many debug configurations for a single project as you wish. When the debugger is started (from **Run > Debug**), it is opened in a new window, and you are ready to start debugging.

**Figure 2. Setting the project's main Java class in the debug configuration**



Next, we discuss some of the common debugging practices in Eclipse.

## Setting breakpoints

When you launch your application for debugging, Eclipse switches to the Debug perspective automatically. Undoubtedly, the most common debugging procedure is to set breakpoints that will allow the inspection of variables and the values inside conditional statements or loops. To set breakpoints in the Package Explorer view of the Java perspective, double-click the selected source code file to open it in an editor. Walk through the code and place your cursor on the marker bar (along the left edge of the editor area) on the line with the suspected code. Double-click to set the breakpoint.

**Figure 3. Two breakpoints markers visible in the left margin of the editor**

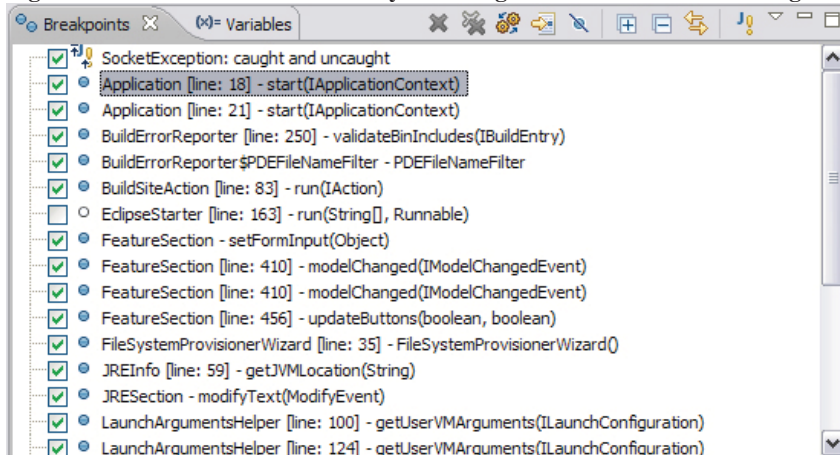
```

/* (non-Javadoc)
 * @see org.eclipse.equinox.app.IApplication#start(org.eclipse.equinox.app.IApplicatio
 */
public Object start(IApplicationContext context) {
    Display display = PlatformUI.createDisplay();
    try {
        int returnCode = PlatformUI.createAndRunWorkbench(display, new ApplicationWork
        if (returnCode == PlatformUI.RETURN_RESTART) {
            return IApplication.EXIT_RESTART;
        }
    }
    return IApplication.EXIT_OK;
}

```

Now start the debugging session from the **Run > Debug** menu. It is important not to put multiple statements on a single line because you cannot step over or set line breakpoints on more than one statement on the same line.

**Figure 4. View indicates the currently executing line with arrow in left margin**



There is also a convenient Breakpoints view to manage all your breakpoints.

**Figure 5. Breakpoints view**

```

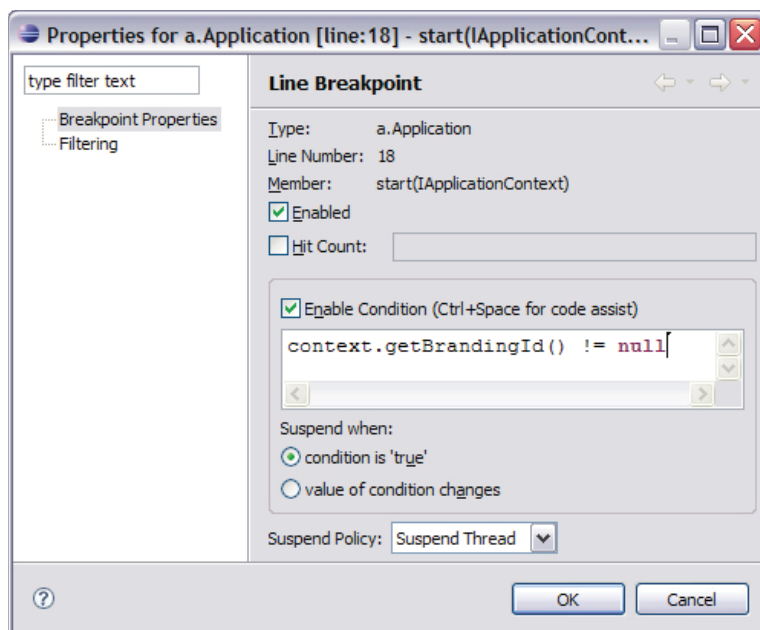
/* (non-Javadoc)
 * @see org.eclipse.equinox.app.IApplication#start(org.eclipse.equinox.app.IApplicatio
 */
public Object start(IApplicationContext context) {
    Display display = PlatformUI.createDisplay();
    try {
        int returnCode = PlatformUI.createAndRunWorkbench(display,
        if (returnCode == PlatformUI.RETURN_RESTART) {
            return IApplication.EXIT_RESTART;
        }
    }
}

```

### Conditional breakpoints

Once you know where an error occurs, you will want to see what the program is doing right before it crashes. One way to do this is to step through every statement in the program, one at a time, until you reach the point of concern. Sometimes it's better to just run a section of code and stop execution at that point so you can examine data at that location. It's possible to declare conditional breakpoints triggered whenever the value of an expression changes (see Figure 6). In addition, code assist is available when typing in the conditional expression.

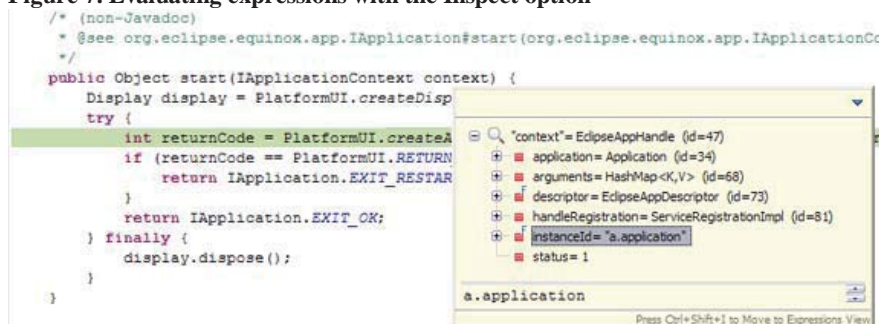
**Figure 6. Setting the conditional breakpoint trigger**



### Evaluating expressions

To evaluate expressions in the editor in the Debug perspective, select the entire line where the breakpoint is set, and from the context menu, select the Inspect option via **Ctrl+Shift+I** or right-clicking on the variable you're interested in (see Figure 7). The expression is evaluated in the context of the current stack frame, and the results are displayed in the Expressions view of Display window.

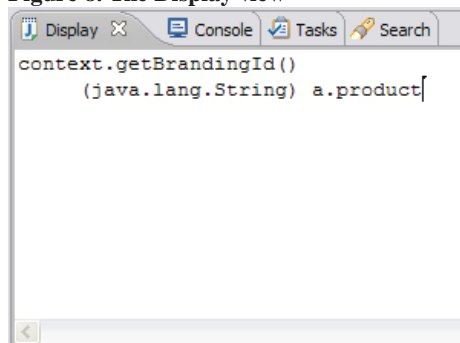
**Figure 7. Evaluating expressions with the Inspect option**



### Scrapbooking your live code

The Display view allows you to manipulate live code in a scrapbook type fashion (see Figure 8). To manipulate a variable, simply type the name of the variable in the Display view, and you'll be greeted with a familiar content assist.

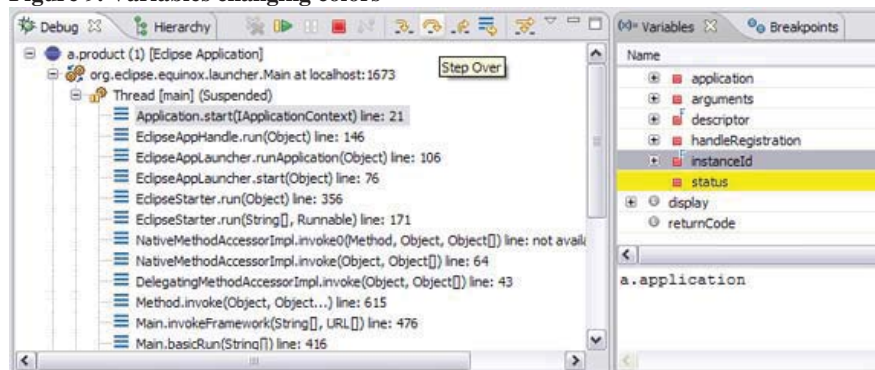
**Figure 8. The Display view**



When the debugger stops at a breakpoint, you can continue the debugger session by selecting the Step Over option from the Debug view toolbar

(see Figure 9). This steps over the highlighted line of code and continues execution at the next line in the same method (or it will continue in the method from which the current method was called). The variables that are changed as a result of the last step are highlighted in a color (default is yellow). Colors can be changed in the debug preference pages.

**Figure 9. Variables changing colors**



To suspend the execution of threads in the Debug view, select a running thread and click **Suspend** in the Debug view toolbar. The current call stack for the thread is displayed, and the current line of execution is highlighted in the editor in the Debug perspective. When a thread is suspended, the cursor is placed over a variable in the Java editor, and the value of that variable is displayed in a small hovering window. Also, the top stack frame of the thread is automatically selected, and the visible variables in that stack frame are displayed in the Variables view. You can examine the appropriate variable in the Variables view by clicking its name.

#### Hotswap Bug Fixing: On-the-fly code fixing

If you are running Java Virtual Machine (JVM) V1.4 or higher, Eclipse supports a feature called Hotswap Bug Fixing (not available in JVM V1.3 or lower). It allows the changing of source code during a debugger session, which is better than exiting the application, changing the code, recompiling, then starting another debugging session. To use this function, simply change the code in the editor and resume debugging. This feature became available because JVM V1.4 is compatible with the Java Platform Debugger Architecture (JPDA). JPDA implements the ability to substitute modified code in a running application. This is, of course, particularly useful when it takes a long time to start your application or to get to the point where it fails.

If the program has not fully executed when you are done debugging, select the Terminate option from the context menu in the Debug view. A common mistake is to use Debug or Run instead of Resume while you're in a debugger session. This will launch another debugger session rather than continuing the current one.

---

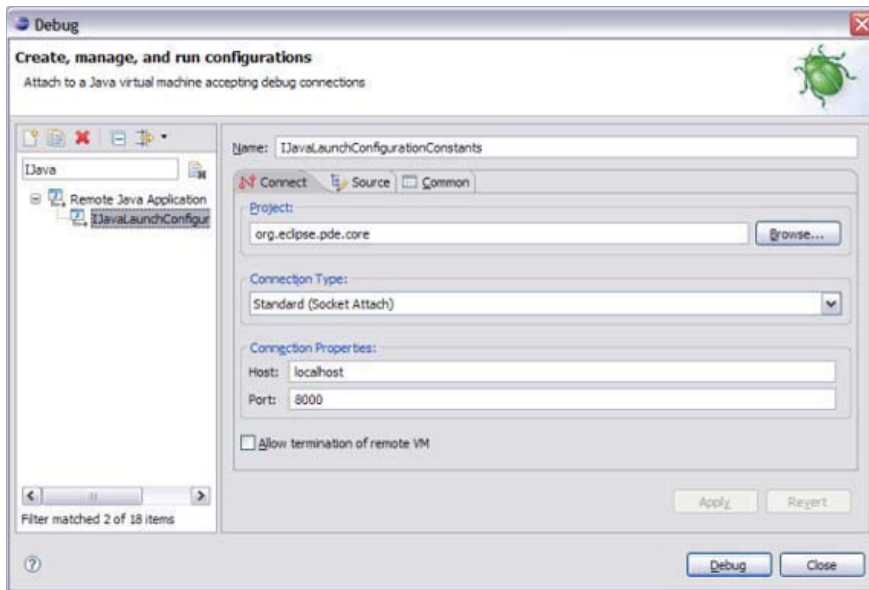
#### Remote debugging

The Eclipse debugger offers an interesting option for debugging remote applications. It can connect to a remote VM running a Java application and attach itself to the application. Working with a remote debugging session is largely similar to local debugging. However, a remote debugging configuration requires different settings in the **Run > Debug** window. You need to first select the **Remote Java Application** entry in the left-hand view, then click **New**. A new remote launch configuration is created, and three tabs are shown: Connect, Source, and Common.

In the Project field of the Connect tab, select which project to use as a reference for the launch (for source code lookup). In the Host field of the Connect tab, type the IP address or domain name of the remote host where the Java program is running. In the Port field of the Connect tab, type the port where the remote VM is accepting connections. Generally, this port is specified when the remote VM is launched. Select Allow termination of remote VM option when you want the debugger to determine whether the Terminate command is available in a remote session. Select this option if you want to be able to terminate the VM to which you are connecting. Now when you select the Debug option, the debugger attempts to connect to a remote VM at the specified address and port, and the result is displayed in the Debug view.

If the launcher is unable to connect to a VM at the specified address, an error message appears. In general, the availability of remote debugging functionality strictly depends on the Java VM that runs on the remote host. Figure 10 shows the setting of connection properties for a remote debugging session.

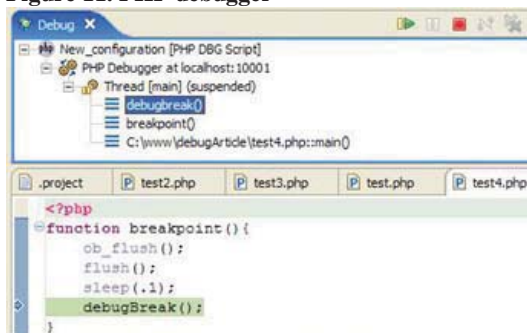
**Figure 10. Setting connection properties for a remote debugging session**



## Debugging other languages

Although the Java language is the most widely used for Eclipse, it is an extensible platform that can support many others. Eclipse supports C/C++ through the C/C++ Development Tools (CDT) project. The CDT extends the standard Eclipse Debug view with functions for debugging C/C++ code, and the CDT Debug view allows you to manage the debugging of C/C++ projects in the workbench. The CDT doesn't include its internal debugger, but it offers a front end to GNU GDB debugger, which must be available locally. There are other projects like the PHP Development Tools (PDT) that offer their respective debuggers (see Figure 11).

**Figure 11. PHP debugger**



## Conclusion

The Eclipse Platform provides a built-in Java language debugger with standard debugging functionality, including the ability to perform step execution, to set breakpoints and values, to inspect variables and values, and to suspend and resume threads. It can also be used to debug applications running on a remote machine. The Eclipse Platform is mainly a Java development environment, but the same Eclipse Debug view is also available for the C/C++, PHP and many more programming languages.

## Acknowledgements

Thanks to Tyler Anderson for creating Figure 11.

## Resources

### Learn

- Visit [Eclipse.org](http://Eclipse.org) for comprehensive information on the program and how to use it.
- Visit [PlanetEclipse.org](http://PlanetEclipse.org) to stay in tune with Eclipse community happenings.