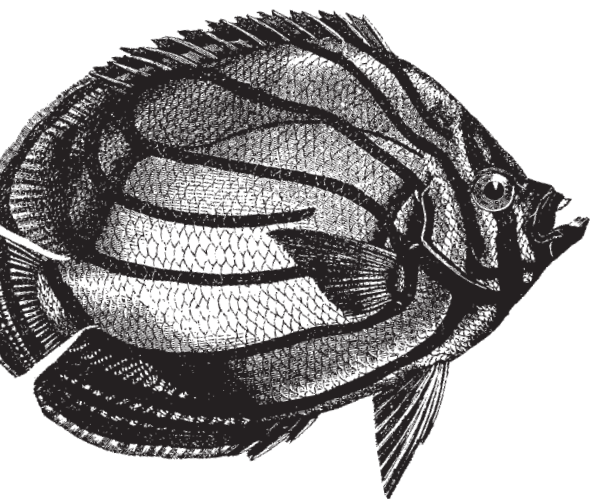


Using The Full-Featured IDE

Eclipse IDE

Pocket Guide



O'REILLY®

Ed Burnette

Tips and Tricks

The Eclipse IDE has an incredibly rich set of features, but many of them are hidden from view. With a little digging, you can discover its secrets and get the most out of the environment. This part of the book gets you started with several useful but less visible features.

Code Assist

The Java editor is always paying attention to what you type, ready to offer helpful suggestions through a feature called *code assist* (also called *content assist*). To use it, go to the Java editor in the Hello example and start a new statement in the `main()` method. Begin typing the following:

System.

Pause after the period. A code assist window (similar to that shown in Figure 23) will appear. The window shows you all the valid possibilities at this point. Type the letter `o` and the choices will narrow down to `out`. Press Enter to accept this choice. Given the long names most Java programs use, this can be a real time-saver.

Besides reducing typing, code assist is especially handy when you are exploring unfamiliar territory—for example, making calls to a library you haven't used before. Code assist is activated automatically by certain keystrokes—like the period in the previous example—but you can also invoke it at any time by pressing Ctrl+Space (Edit → Content Assist). This feature

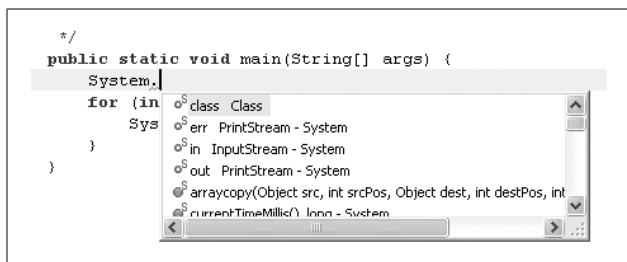


Figure 23. Code assist tells you what comes next and displays any Javadoc (if the source is available).

is fully configurable in the Java editor preferences (Window → Preferences → Java → Editor).

Templates

Eclipse provides a shorthand way of entering text called *templates*. For example, in the Java editor, if you type **for** and press Ctrl+Space, the code assist window will pop up as before, but this time it will display a few templates that start with the word “for” (see Figure 24).

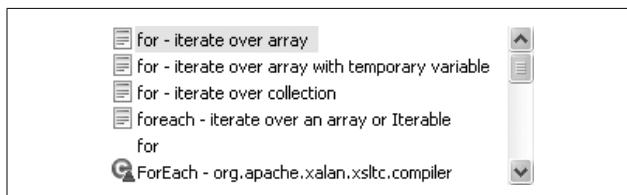


Figure 24. Editor templates are shorthand for entering boilerplate text (e.g., for loops).

Selecting the first one will cause code similar to this to appear in the editor:

```
for (int i = 0; i < array.length; i++) {
}
```

The cursor highlights the first variable `i`. If you start typing, all three occurrences of that variable will be modified. Pressing Tab will cause the variable `array` to be selected; pressing Tab again will put the cursor on the blank line between the braces so you can supply the body of the loop.

TIP

If you try this, you may see different variable names. Eclipse guesses which variables to use based on the surrounding code.

For a list of all predefined templates, and to create your own or export them to an XML file, see Window → Preferences → Java → Editor → Templates.

Automatic Typing

Closely related to code assist is a feature called *automatic typing*. If you're following along with the earlier example shown in Figure 23, the text cursor should be positioned after `System.out`. Type `.println(` (that is, period, `println`, opening parenthesis). The Java editor will type the closing parenthesis for you automatically. Now, type a double quote, and the closing quote appears. Type in some text and then press the Tab key. Tab advances to the next valid place for input, which is after the closing quote. Hit Tab again, and the cursor advances to the end. Type a semicolon to finish the statement.

TIP

Code assist and automatic typing take a little getting used to. At first you may be tempted to turn them off, but I suggest you give it time and try to learn to work with them. After a while, you'll wonder how you ever got by without the extra support.

Refactoring

Refactoring means transforming code without changing its functionality. Consider renaming, which is the simplest form of refactoring. If you rename a local variable from *rose* to *daisy*, it would smell as sweet.

Much has been written on refactoring, such as *Refactoring: Improving the Design of Existing Code* (Addison Wesley). Before Eclipse and similar tools were available, programmers had to do refactoring manually or with simple text substitutions. For example, in the *vi* editor, running the command `:1,$s/rose/daisy/g` will replace “rose” with “daisy” everywhere in the current file.

If you’ve ever tried this, you know it’s usually a bad idea. Your simple search-and-replace operation can change more than just the variable you intended, even with a clever substitution string. Plus, if you need to change multiple files, you’ll have to go to a scripting language such as Perl.

Here’s how it works in Eclipse. To rename a symbol (i.e., a class, method, variable, etc.), select it in the editor and press `Alt+Shift+R` (Refactor → Rename). Type in the new name and press `Enter` to perform the change. Done!

If you like, you can select the Preview button before performing the changes; this will show you what the modified source will look like (see Figure 25). You can also undo the refactoring (`Ctrl+Z` or `Edit` → `Undo`) if you change your mind.

Here’s another handy refactoring supported by Eclipse: to move a class from one package to another, simply go to the Package Explorer view and drag the file to where you want it. Eclipse will take care of changing the package statement in the file and in all the other class files that refer to it. Neat, huh?

Eclipse implements over a dozen different types of refactorings, and more are being added all the time. See the Java Development User Guide (`Window` → `Help Contents` → `Java`

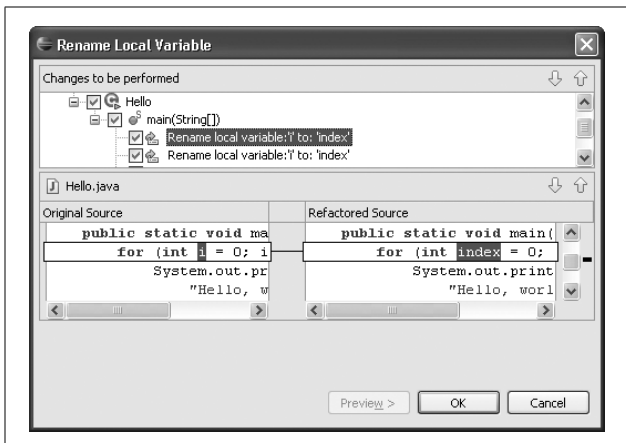


Figure 25. You can preview the changes that any of Eclipse’s refactorings would make.

Development User Guide) under Reference → Refactoring for more information.

Hover Help

You’ve seen that code assist is a good way to explore an unfamiliar API. Another useful tool is *hover help*. To use hover help, simply move the mouse cursor over a symbol you want to know more about and pause for a moment. For example, try hovering over `println` in `System.out.println`. A little pop-up window will appear, giving you a short description of the method.

For best results, you need access to the source code of the symbol you are examining. For Java library methods, the source comes with the JDK (J2SE SDK) package. Eclipse can usually figure out how to find this source code on its own, but see Window → Preferences → Java → Installed JREs to configure the JDK’s location.

If you are using code from a third-party JAR file, the source is often provided in a separate file or a subdirectory. You can tell Eclipse about this location by right-clicking on the JAR file in the Package Explorer and selecting Properties → Java Source Attachment.

If you don't have the source code, but you have the API documentation (Javadoc) in HTML form, select the symbol you want information on and press Shift+F2 (Navigate → Open External Javadoc). To make this work, you have to configure the Javadoc URL in the properties for the JAR file: right-click on the JAR file and select Properties → Javadoc Location.

Hyperlinks

Did you know there is a web browser built into the Java editor? Well, there is—sort of. The editor lets you navigate around your program as if it were a web site. Hold down the Ctrl key and move your mouse through your source code. An underline will appear to indicate hyperlinked symbols. You can leave the mouse cursor over the symbol to see its definition, or click on it to open the declaration in the editor.

Like a browser, Eclipse maintains a history of all the pages you've visited. Use the Back command (⬅; Alt+Left; or Navigate → Left) to go to the previous location, and use Forward (➡; Alt+Right; or Navigate → Right) to go to the next one.

Quick Fixes

Whenever you make a syntax error in your program, Eclipse's background compiler detects it immediately and draws an error indicator (affectionately known as the *red squiggle*) under the offending code. In addition to simply detecting the problem, Eclipse can usually offer an automatic program correction, called a *quick fix*.

For example, try misspelling the `System.out` method `println` as **printline**. Press `Ctrl+1` (Edit → Quick Fix) to see several possible fixes. One of them will be `Change to println(..)`. Press the down arrow to see a preview of each proposed change; press `Enter` to accept the one you want.

The Quick Fix command can also make suggestions for small source transformations on lines that don't have errors. For example, if you have code like this:

```
if (!(hail || thunder))
```

and you select the text `!(hail || thunder)` and press `Ctrl+1`, Eclipse will suggest some possible transformations, such as “Push negation down.” Choosing that particular option would change the code to:

```
if (!hail && !thunder)
```

Searching

The Eclipse IDE provides dozens of different ways to locate things. Eclipse breaks these up into two major categories:

Find

Look for something in the current file.

Search

Look for something in multiple files.

The Find command (`Ctrl+F` or Edit → Find/Replace) is just a run-of-the-mill text locator like you would see in any editor. You can look for plain strings or full regular expressions, and you can optionally substitute the text you find with other text. The shortcut to find the next occurrence is `Ctrl+K`.

A handy variant on Find is *incremental find*, a feature borrowed from the Emacs editor. Press `Ctrl+J` (Edit → Incremental Find Next) and start typing the text you're looking for. The selection will move to the next occurrence as you type.

Searches are much more interesting. To start with, Eclipse supports locating strings and regular expressions in many files at once. You can search the entire workspace, just the current project, or any subset (called a *working set*) that you define. To do this kind of search, select Search → File....

Eclipse can also do a full language-aware search. Since Eclipse has its own built-in Java compiler, it understands the difference between, say, a method named `fact` and a field named `fact`, or even between two methods that have the same names but take different parameters, such as `fact(int)` and `fact(double)`. This kind of search is available by selecting Search → Java....

These searches and more are accessible through the Search dialog (🔍; Ctrl+H; or Search → Search). The most common variations also have direct menus or shortcuts of their own. For example, to find all references to a symbol, select the symbol and press Ctrl+Shift+G (or Search → References → Workspace). To find the symbol's declaration, press Ctrl+G (Search → Declarations → Workspace). To find only those places where the symbol is modified, try Search → Write Access → Workspace.

TIP

Current versions of Eclipse don't allow you to perform searches on arbitrary files in the filesystem, but you can use an advanced option under File → New → Folder to link outside directories into your workspace and then search them.

All search results will appear, naturally enough, in the Search view. See Part VII for more details on that view.

Scrapbook Pages

A *scrapbook page* is a way to create and test snippets of code without all the trappings of normal Java code. In some ways, it's like working in a scripting language, but you have the full expressiveness of Java in addition to being able to make calls into any of your code or any of the system libraries.

To create a scrapbook page, select File → New → Other... → Java → Java Run/Debug → Scrapbook Page. Enter the name of the page—for example, **test**—and click Finish (or just press Enter). A new editor page will open for *test.jpage*.

In the blank scrapbook page, try typing in an expression like **123/456**, press Ctrl+A to select the expression, and press Ctrl+Shift+D (Run → Display) to run it and display the result. (The answer in this case is (int) 0 because both numbers are integers and the result was truncated.) Note that the result is selected, so you can copy it quickly (or press Backspace to remove it from the page).

Next, try entering **Math.PI** and displaying its result. This works because the scrapbook page already has all the system libraries imported, including the Math class. If you need a particular import, you can bring up the context menu and select Set Imports....

Let's try something a little more complicated. Type in this snippet of code:

```
double d = 3.14;  
System.out.println(d);
```

Now select the snippet and press Ctrl+U (Run → Execute) to execute it. The output will appear in the Console window. Execute is exactly like Display except that Execute doesn't show the return value (if any).

You can execute loops or even call methods in your regular programs from the scrapbook page. This is useful for trying out new ideas or just for simple debugging.

Java Build Path

If you've done any Java programming before, you're familiar with the Java classpath—a list of directories and JAR files containing Java classes that make up the program. Usually this is controlled by an environment variable (CLASSPATH) or a command-line option (-cp).

In Eclipse, classpath details are a little more complicated. The first thing to realize is that Eclipse doesn't use the CLASSPATH environment variable. It understands and controls the location of all classes itself. Additionally, Eclipse makes a distinction between runtime and build (compile) time. In Eclipse terminology, *classpath* refers only to the runtime class list, while *build path* refers to the compile-time list. These two paths may be different, but, by default, they will both be set to the list you specify in the build path.

To see the build path, right-click on your project and select Properties → Java Build Path. A dialog will appear, with the tabs described in Table 6.

Table 6. Java Build Path tabs

Tab name	Description
Source	Tell the Java compiler where your source code is located. Each source directory is the root of a package tree. You can also control where generated output files (such as .class files) go.
Projects	Make the current project depend on other projects. Classes in the other projects will be recognized at build time and runtime. The other projects do not have to be built into a JAR file before referring to them in Eclipse; this cuts down on development time.
Libraries	Pull in code that is not in Eclipse projects, such as JAR files. See Table 7 for the kinds of locations you can access.
Order and Export	If other projects are dependent on this one, expose (or don't expose) symbols in the current project to the other projects.

In addition to going through the Java Build Path dialog, you can right-click on directories and JAR files in the Package

Explorer view and select commands under the Build Path menu to add and remove items from the build path.

The Libraries tab is very flexible about the locations it allows you to specify for JARs and class files. Other features in Eclipse use similar lists, so if you understand this tab, it will help you understand those features as well. Table 7 explains the buttons on the Libraries tab.

Table 7. JAR and class locations in the Java Build Path

Button name	Description
Add JARs...	Specify JAR files in the workspace (this project or other projects).
Add External JARs...	Specify full pathnames for JAR files outside the workspace (not recommended for team projects).
Add Variable...	Use a symbolic variable name (like JRE_LIB or ECLIPSE_HOME) to refer to a JAR file outside the workspace.
Add Library...	Refer to a directory outside the workspace containing several JAR files.
Add Class Folder...	Refer to a workspace directory containing individual class files.

Launch Configurations

How do you specify command-line parameters to your program or change the Java VM options that are used to invoke your program? Every time you select Run As → Java Application on a new class that has a `main()` method, Eclipse creates a launch configuration for you. A *launch configuration* is the set of all the options used to run your program.

To change those options, select Run → Run... and locate your configuration in the dialog. Click on the configuration to see all the options in a series of tabbed pages on the right-hand side of the window (the tabs are described in Table 8). You can also create new configurations in this dialog.

Table 8. Launch configuration tabs

Tab name	Description
Main	Specify the project and the name of the Main class.
Arguments	Set the program arguments, the Java VM arguments, and the working directory in which to start the program.
JRE	Specify the version of Java used to run the program (this can be different than the one used to compile it).
Classpath	Set the list of JARs and classes available at runtime.
Source	Locate the source code inside or outside the workspace.
Environment	Pass environment variables to the program.
Common	Miscellaneous options.

Many more features of Eclipse are waiting to be discovered, and new ones are added in each release. The “Tips and Tricks” section of the online help (Help → Tips and Tricks) is a good place to look for the kinds of little nuggets that can save you time or let you do something new. You can also find a useful command and keyboard shortcut listing in the Appendix.