5
Building Eclipse Projects Using Ant

ONJava.com Editor's Note: This is a beta preview release from the book "Eclipse: A Java Developer's Guide" by Steve Holzner copyright O'Reilly & Associates. This text has not yet been edited and is presented in this raw form as a preview.

Eclipse is great for building your code. But for more advanced project development, there's still something missing. For example, what if you wanted to not only compile several files at once, but also wanted to copy files over to other build directories, create JAR files and Javadoc, create new directories, delete previous builds, and create deployment packages all at once?

You can do that with a build tool like Apache's Ant (http://ant.apache.org/). Ant is a Java-based build tool that can perform all these tasks, and much more. You can download Ant and run it on the command line, automating your build tasks to not only compile code, but create JAR files, move and create classes, delete and make directories, and a great deal more.

The good news here is that Ant comes built into Eclipse, ready to use. Ant is the premier build tool for Java development, and we'll get an idea why in this chapter. As your projects become more and more elaborate, Ant can automate dozens of tasks that you'd need to perform manually otherwise—when you have things set up to run with Ant, all you've got to do is to point and click to perform a complete build without having to take dozens of separate steps manually. And that can save many steps omitted in error over the development process.

The fact that Ant comes built into Eclipse means that it's easier to use for us than for all those developers who use it on the command line. To see how that works, we'll start with a quick example.

Working With Ant

To use Ant from Eclipse, create a new project, Ch05_01, and add a new class to it, Ch05_01. In this class's main method, we'll just display the message "No worries.", as you see in Example 5-1.

Example 5-1. A sample project

```
package org.eclipsebook.ch05;

public class Ch05_01 {
        public static void main(String[] args) {
                System.out.println("No worries.");
        }
}
```

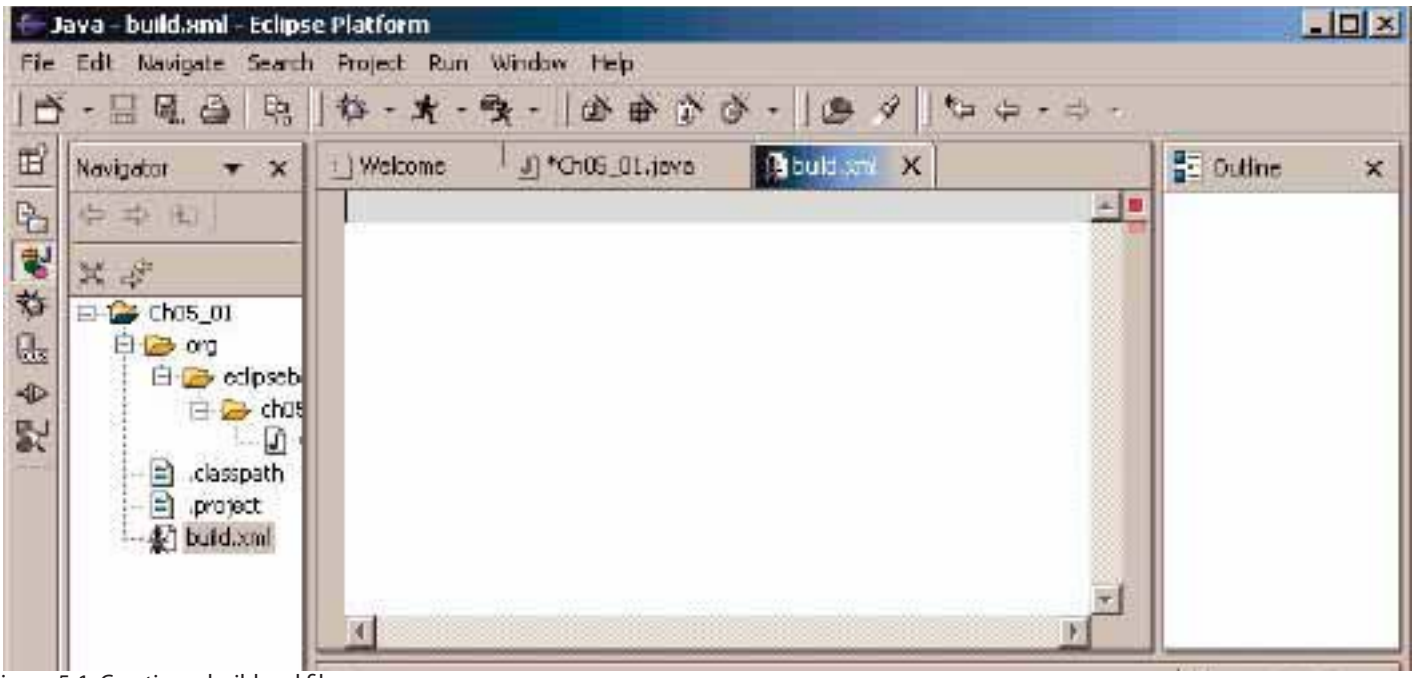

Figure 5-1. Creating a build.xml file

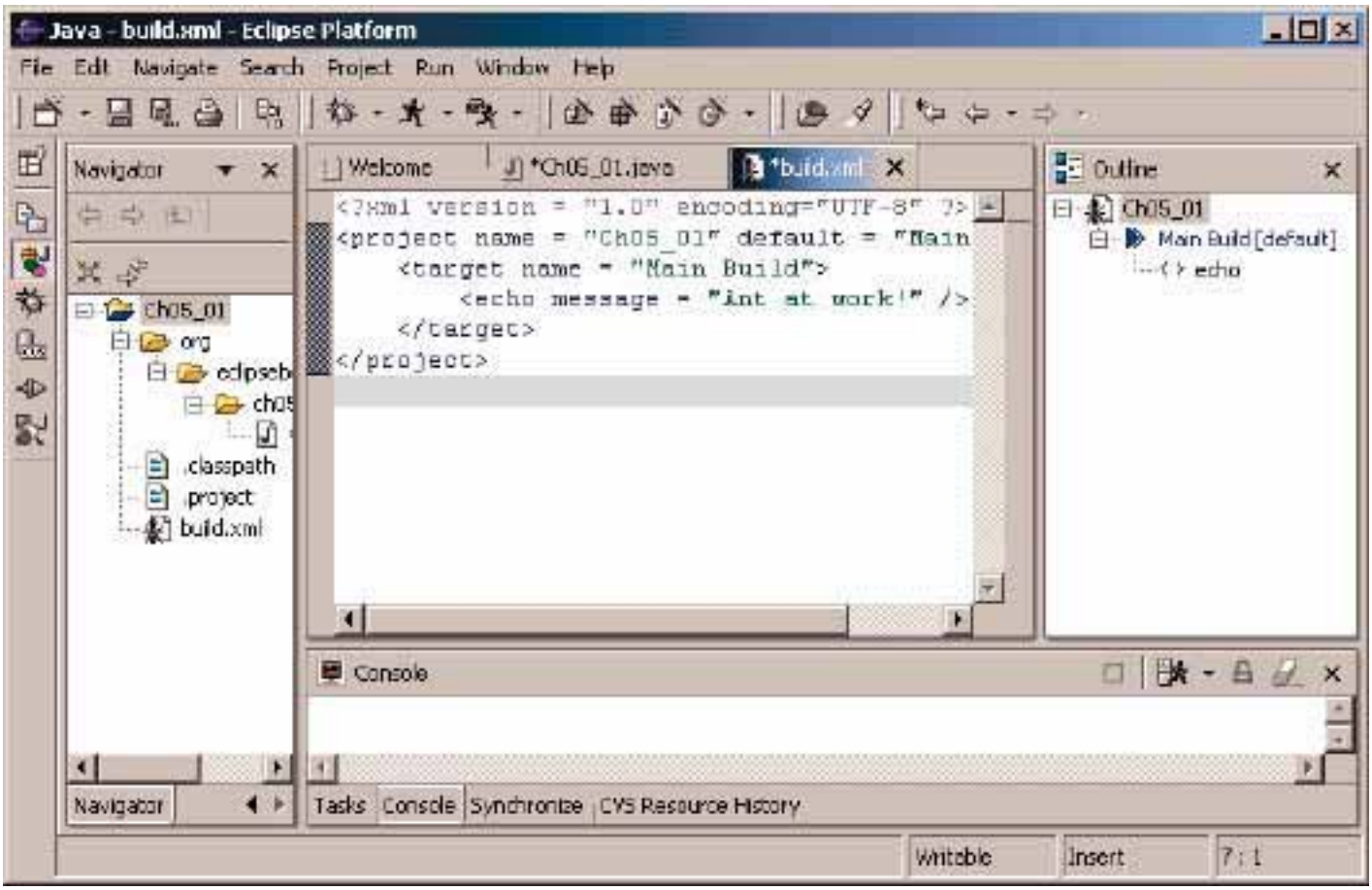

Figure 5-2. Entering XML in build.xml

To work with Ant, we'll need an Ant build file, which is named build.xml by default. To create that file in this project, right-click the project and select Newγ File. Enter the name of the new file, build.xml, in the File name box and click finish. You use the XML in this file to tell Ant how to build your project. Although Eclipse can automate the connection to Ant, Ant needs this XML build file to understand what you want it to do, which means that we will have to master the syntax in this file ourselves in this chapter.

Eclipse recognizes that build.xml is an Ant build file, and marks it with an ant icon, as you see at left in Figure 5-1.

In this case, enter this simple XML into build.xml—all we're going to do here is to have Ant echo a message, "Ant at work!", to the console:

```
<?xml version = "1.0" encoding="UTF-8" ?>
<project name = "Ch05_01" default = "Main Build">
  <target name = "Main Build">
    <echo message = "Ant at work!" />
  </target>
</project>
```

This XML makes the target which we've named "Main Build" into the default target. An Ant target specifies a set of tasks you want to have run, similar to a method in Java. The default target is that which is executed when no specific target is supplied to Ant. In this case, we're just going to have this default target echo a message to the console, nothing more; nothing's going to be compiled.

You can see this new XML in the Ant editor in Figure 5-2. Note that the Ant editor uses syntax highlighting, just as the JDT editor does, and that you can see an outline of the build.xml document in the Outline view at right.

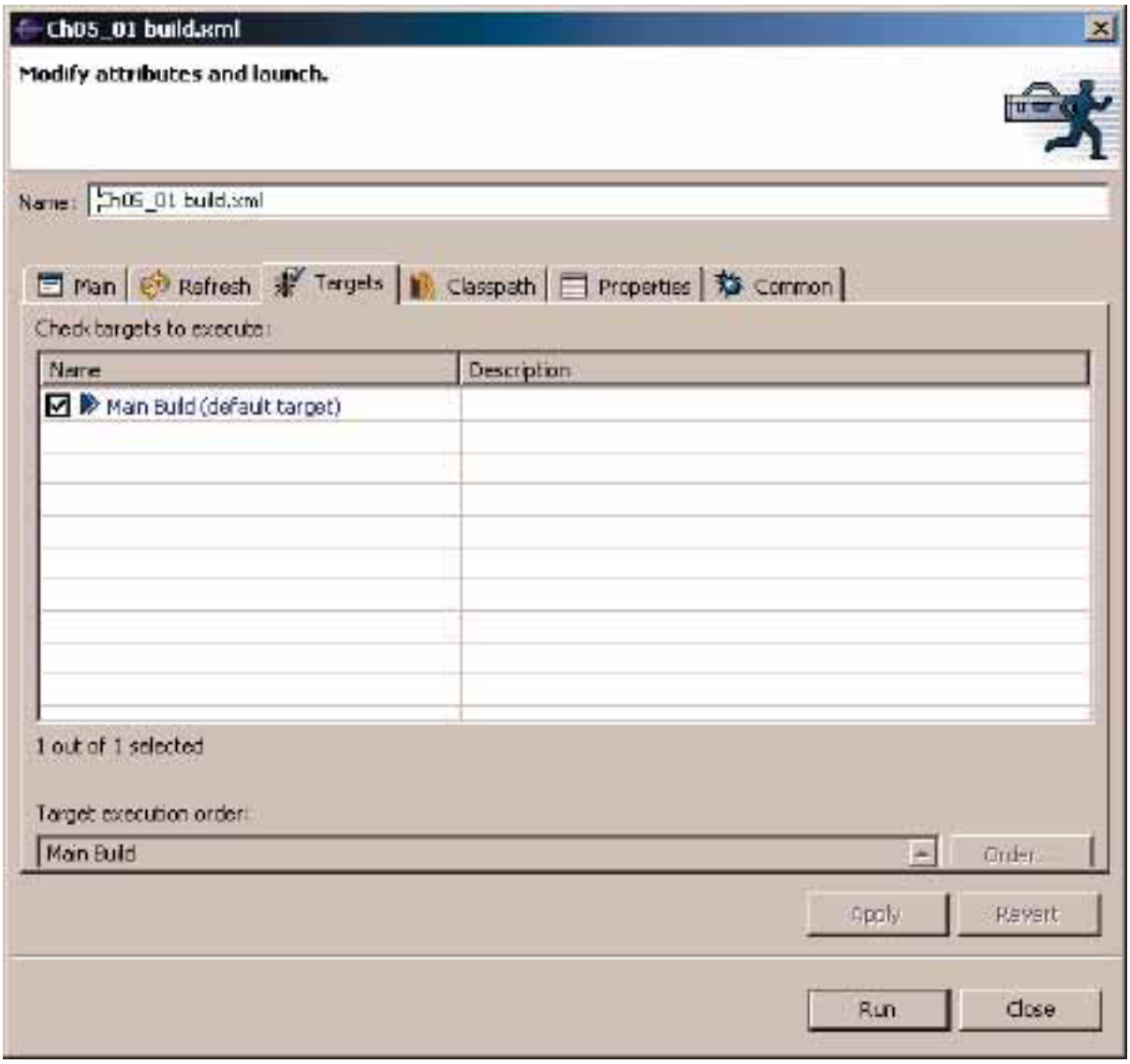

Figure 5-3. Running Ant

Save build.xml and right-click it, selecting the Run Ant item. This opens the Ch05_01 build.xml dialog you see in Figure 5-3. You can see that the default target, Main Build, is already selected.

Click the Run button in this dialog to run Ant. When you do, you'll see something like this in the Console view--note that our message was echoed here:

```
Buildfile: D:\eclipse211\eclipse\workspace\Ch05_01\build.xml

Main Build:
     [echo] Ant at work!
BUILD SUCCESSFUL
Total time: 430 milliseconds
```

That's what we wanted this example to do, and it did it. That's a quick example to show how to interact with Ant from Eclipse, but all it does is to display the message "Ant at work!". It doesn't compile anything, it doesn't create any JAR files, but it does give us a start on using Ant in Eclipse. We'll get more advanced in our next example.

JARing Your Output

Here's another example; in this case, we'll build an Eclipse project and store the resulting class in a JAR file. You won't need to be an Ant professional to follow along, because we're interested in looking at Ant from an Eclipse point of view, not in the details of Ant per se. This example is designed to give you the basics of creating a working Ant build file in Eclipse; if you want more details on Ant itself, take a look at the manual at http://ant.apache.org/manual/index.html.

Our goal here is to create a new Java project in Eclipse, use Ant to compile it, and store the resulting .class file in a JAR file. To follow along, create a new Java project, Ch05_02. To emulate a somewhat real-world project, we're going to store the example's source code in a directory named src, and its output in a directory named bin. You can set those directories up when you create the project in the third pane of the New Java Project dialog by clicking the Source tab, clicking the Add Folder button, followed by the Create New Folder button to open the New Folder dialog. Enter the name src in the Folder name box and click OK twice. Eclipse will ask if you want to remove the project as source folder and update the build output folder to Ch05_02/bin. Click Yes, then click Finish to create the new project, which will be complete with src and bin folders.

Next, add a new class, Ch05_02, in a package named org.eclipsebook.ch05, to the project. Add code to the main method in this example to display the message "This code was built using Ant", as you can see in Example 5-2.

Example 5-2. A sample project

```
package org.eclipsebook.ch05;

public class Ch05_02 {

   public static void main(String[] args) {
      System.out.println("This code was built using Ant.");
   }
}
```

Finally, add build.xml to the project by right-clicking the project in the Package Explorer and selecting New> File. Type build.xml in the File name box and click Finish, which creates the file and opens it in the Ant editor. We'll start writing build.xml with the standard XML declaration and a <project> element that identifies the Main Build task as the default:

```
<?xml version="1.0" encoding = "UTF-8"?>
<project name="Ch05_01" default="Main Build" basedir=".">
    .
    .
    .
</project>
```

Next we'll create the properties corresponding to the directories we'll use--src, bin, a directory for the JAR file, jardir (we'll create a lib directory under the bin directory to store JAR files), and the JAR file itself, jarfile (we'll call this file Ch05_02.jar). Setting up properties this way lets you access these directory names later in the build file. We'll also set the build.compiler property to the adapter for the JDT compiler, org.eclipse.jdt.core.JDTCompilerAdapter, so Ant will use that compiler:

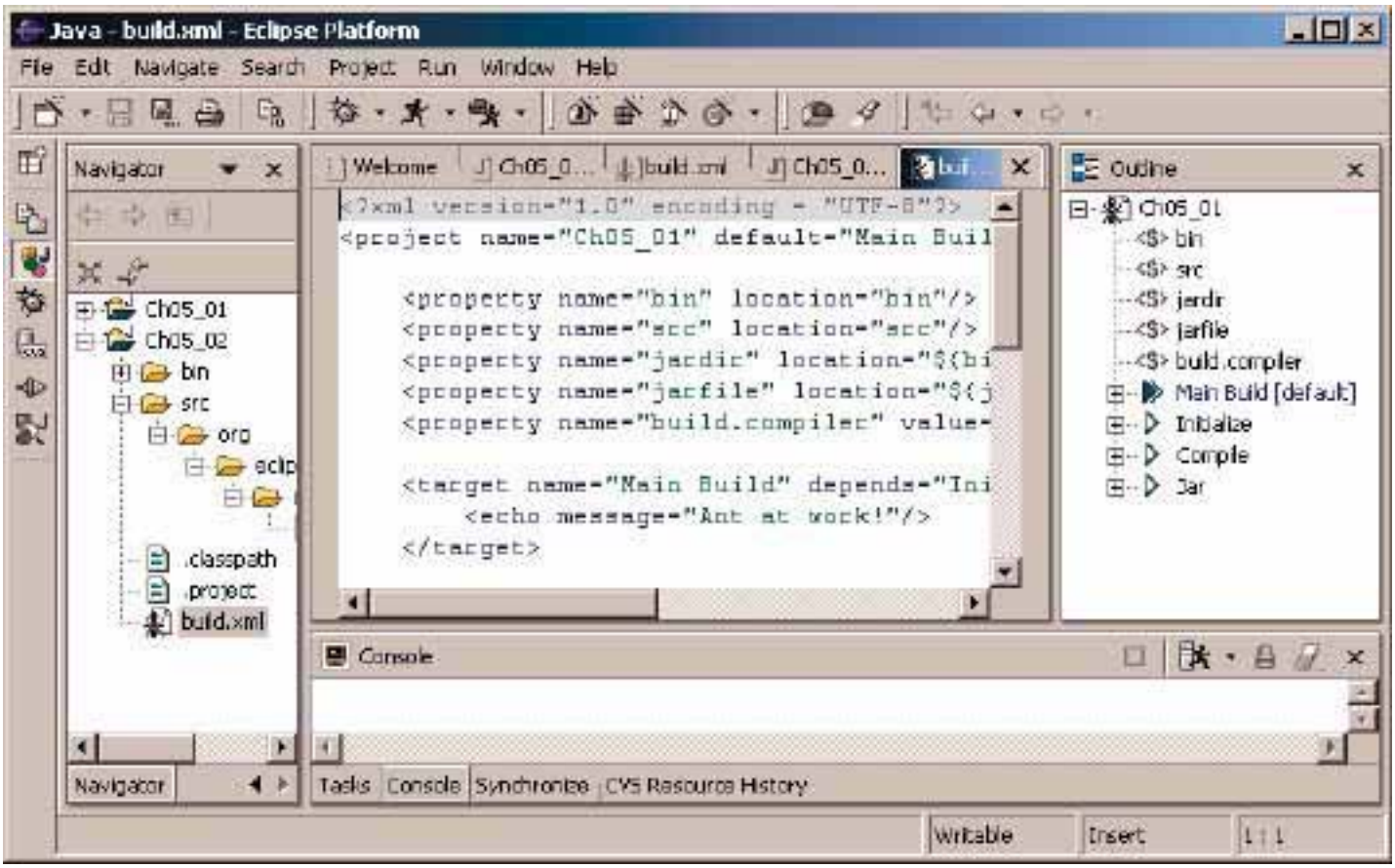

Figure 5-4. Our new build.xml

```
<?xml version="1.0" encoding = "UTF-8"?>
<project name="Ch05_01" default="Main Build" basedir=".">
   <property name="bin" location="bin"/>
   <property name="src" location="src"/>
   <property name="jardir" location="${bin}/lib"/>
   <property name="jarfile" location="${jardir}/Ch05_02.jar"/>
    <property name="build.compiler"
      value="org.eclipse.jdt.core.JDTCompilerAdapter"/>
      . . .
</project>
```

Now we'll create the main task, Main Build. We'll use three stages in this Ant file--an initialization stage, a compile stage, and a JAR-creation stage, each with its own task--Initialize, Compile, and Jar. To make sure that all those tasks are performed, we'll make the main, default task dependent on them using the depends attribute. Then the main task only has to echo a message to the console indicating that Ant is at work--Ant will take care of the details of running each needed task:

```
<?xml version="1.0" encoding = "UTF-8"?>
<project name="Ch05_01" default="Main Build" basedir=".">
   <property name="bin" location="bin"/>
   <property name="src" location="src"/>
   <property name="jardir" location="${bin}/lib"/>
   <property name="jarfile" location="${jardir}/Ch05_02.jar"/>
    <property name="build.compiler"
      value="org.eclipse.jdt.core.JDTCompilerAdapter"/>
   <target name="Main Build" depends="Initialize, Compile, Jar">
      <echo message="Ant at work!"/>
   </target>
      .. .
</project>
```

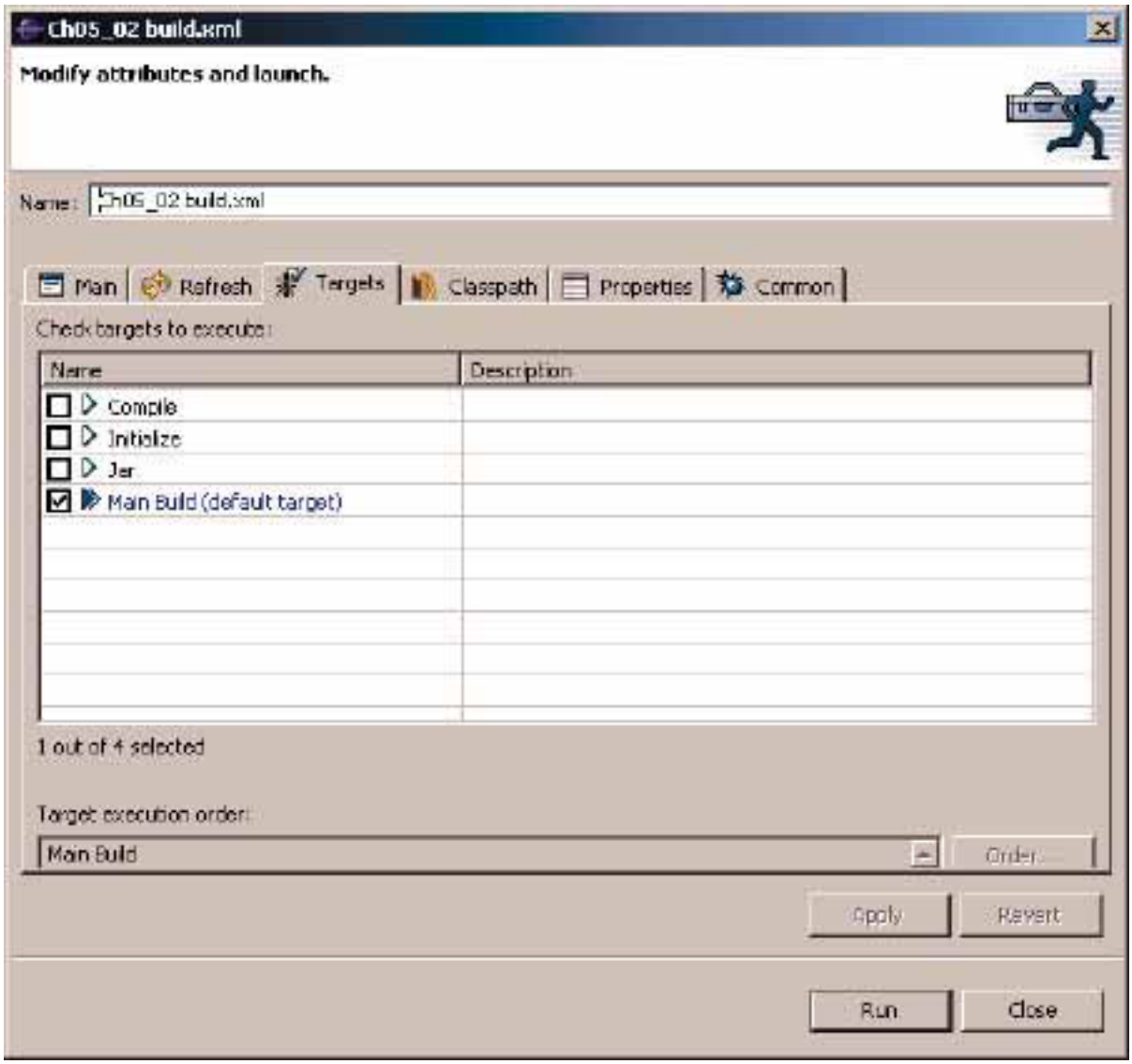

Figure 5-5. Selecting which target to run

The Initialize task will delete everything in the output ${bin} and ${jardir} directories, and then recreate those directories:

```
<target name="Initialize">
  <delete dir="${bin}"/>
  <delete dir="${jardir}"/>
  <mkdir dir="${bin}"/>
  <mkdir dir="${jardir}"/>
</target>
```

The Compile task will compile the source files in ${src} (which is just Ch05_02.java) and put the resulting .class file into ${bin}:

```
<target name="Compile" depends="Initialize">
  <javac srcdir="${src}"
    destdir="${bin}">
  </javac>
</target>
```

Finally, the Jar task will compress Ch05_02.class into a JAR file and store that file as ${jarfile}--note that this task depends on the Initialize and Compile tasks:

```
<target name="Jar" depends="Initialize, Compile">
  <jar destfile="${jarfile}" basedir="${bin}"/>
</target>
```

That completes build.xml; you can see the whole file in Example 5-3.

Example 5-3. A sample Ant build file

```
<?xml version="1.0" encoding = "UTF-8"?>
<project name="Ch05_01" default="Main Build" basedir=".">

  <property name="bin" location="bin"/>
  <property name="src" location="src"/>
  <property name="jardir" location="${bin}/lib"/>
  <property name="jarfile" location="${jardir}/ch05_01.jar"/>
   <property name="build.compiler"
    value="org.eclipse.jdt.core.JDTCompilerAdapter"/>

  <target name="Main Build" depends="Initialize, Compile, Jar">
    <echo message="Ant at work!"/>
  </target>

  <target name="Initialize">
    <delete dir="${bin}"/>
    <delete dir="${jardir}"/>
    <mkdir dir="${bin}"/>
    <mkdir dir="${jardir}"/>
  </target>

  <target name="Compile" depends="Initialize">
    <javac srcdir="${src}"
      destdir="${bin}">
    </javac>
  </target>

  <target name="Jar" depends="Initialize, Compile">
    <jar destfile="${jarfile}" basedir="${bin}"/>
  </target>

</project>
```

> Here's another tip—Eclipse can generate Ant scripts for you under certain circumstances. If your project already has an XML-based manifest file, as the plug-in projects we're going to create in Chapters 11 and 12 will have (the plug-in manifest file is named plugin.xml), all you have to do is right-click the manifest file and select the Create Ant Build File item.

When you enter this XML into build.xml, you can see its properties and tasks in the outline view, as you see in Figure 5-4.

To build the project, right-click build.xml in the Package Explorer and select Run Ant. This opens the Ch05_02 build.xml dialog you see in Figure 5-5. You can see the various Ant targets we've set up here, which you can build independently. The default target, Main Build, is already selected, so just click Run now to build the project.

You can see the results in Figure 5-6--the build was successful, as you see in the Console view.

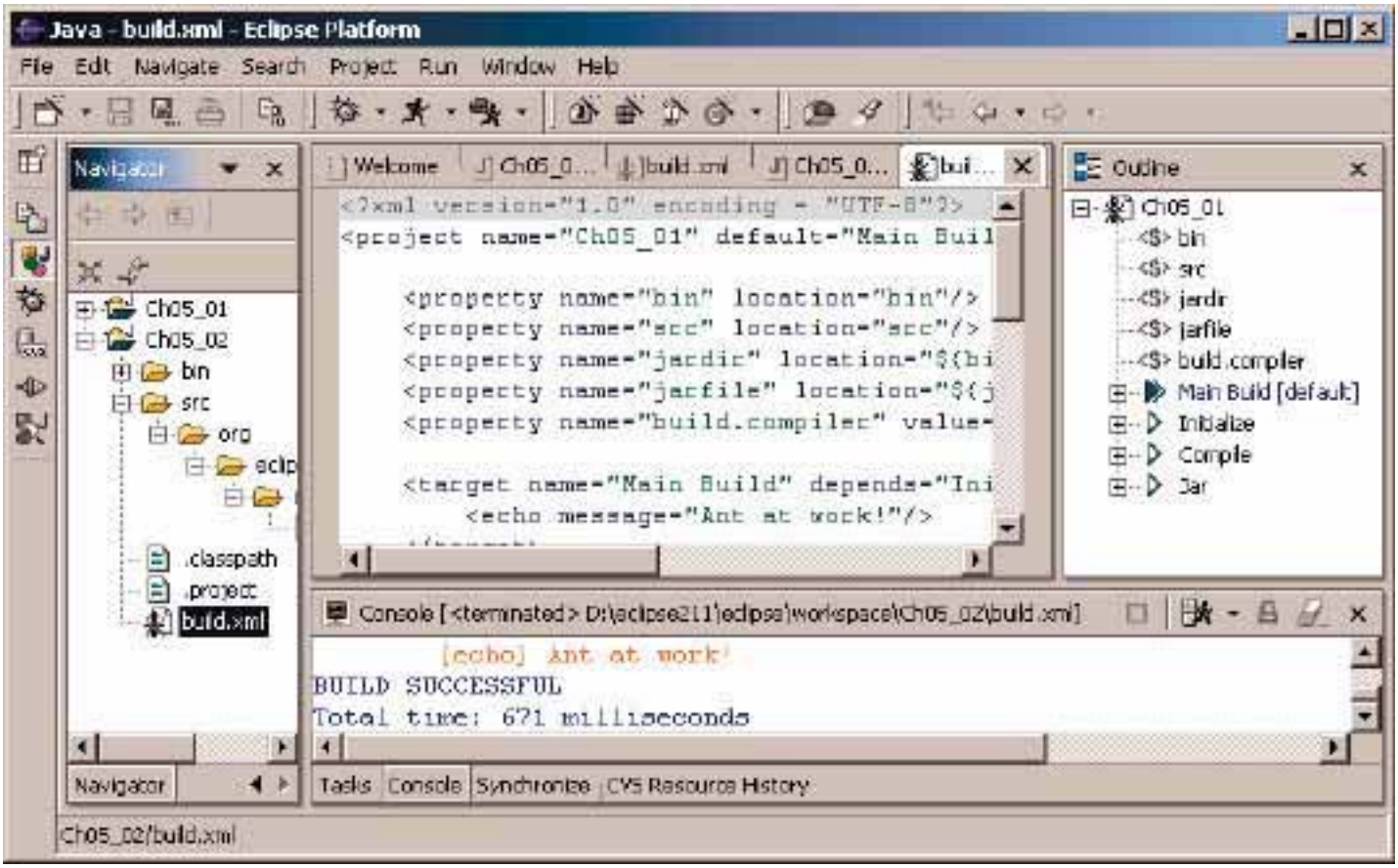

Figure 5-6. A successful build

Here's the complete text that appears in the Console view—you can see the results of each task as it runs:

```
Buildfile: D:\eclipse211\eclipse\workspace\Ch05_02\build.xml

Initialize:
    [delete] Deleting directory D:\eclipse211\eclipse\workspace\Ch05_02\bin
     [mkdir] Created dir: D:\eclipse211\eclipse\workspace\Ch05_02\bin
     [mkdir] Created dir: D:\eclipse211\eclipse\workspace\Ch05_02\bin\lib

Compile:
     [javac] Compiling 1 source file to D:\eclipse211\eclipse\workspace\Ch05_02\bin
     [javac] D:\eclipse211\eclipse\workspace\Ch05_02\src\org\eclipsebook\ch05\Ch05_02.java
     [javac] Compiled 20 lines in 210 ms (95.2 lines/s)
     [javac] 1 .class file generated

Jar:
      [jar] Building jar: D:\eclipse211\eclipse\workspace\Ch05_02\bin\lib\ch05_01.jar

Main Build:
     [echo] Ant at work!
BUILD SUCCESSFUL
Total time: 1 second
```

And that's it—the project was built and Ch05_02.jar was created in the bin/lib directory. Not bad—as you can see, Ant lets you go far beyond the normal Eclipse build process to copy files, create directories, create JAR files, and so on.

Configuring Ant in Eclipse

Eclipse also lets you configure its internal verison of Ant. To configure how Ant will run, select Windowγ Preferences, followed by the Ant item, as shown in Figure 5-7. In Eclipse, you don't need to name your build file build.xml; Ant will try to guess which file is the build file (the build file does need to be an XML file, with a name that has the extension .xml). You can help Eclipse out by giving an alternate name, or a list of names, in this dialog.

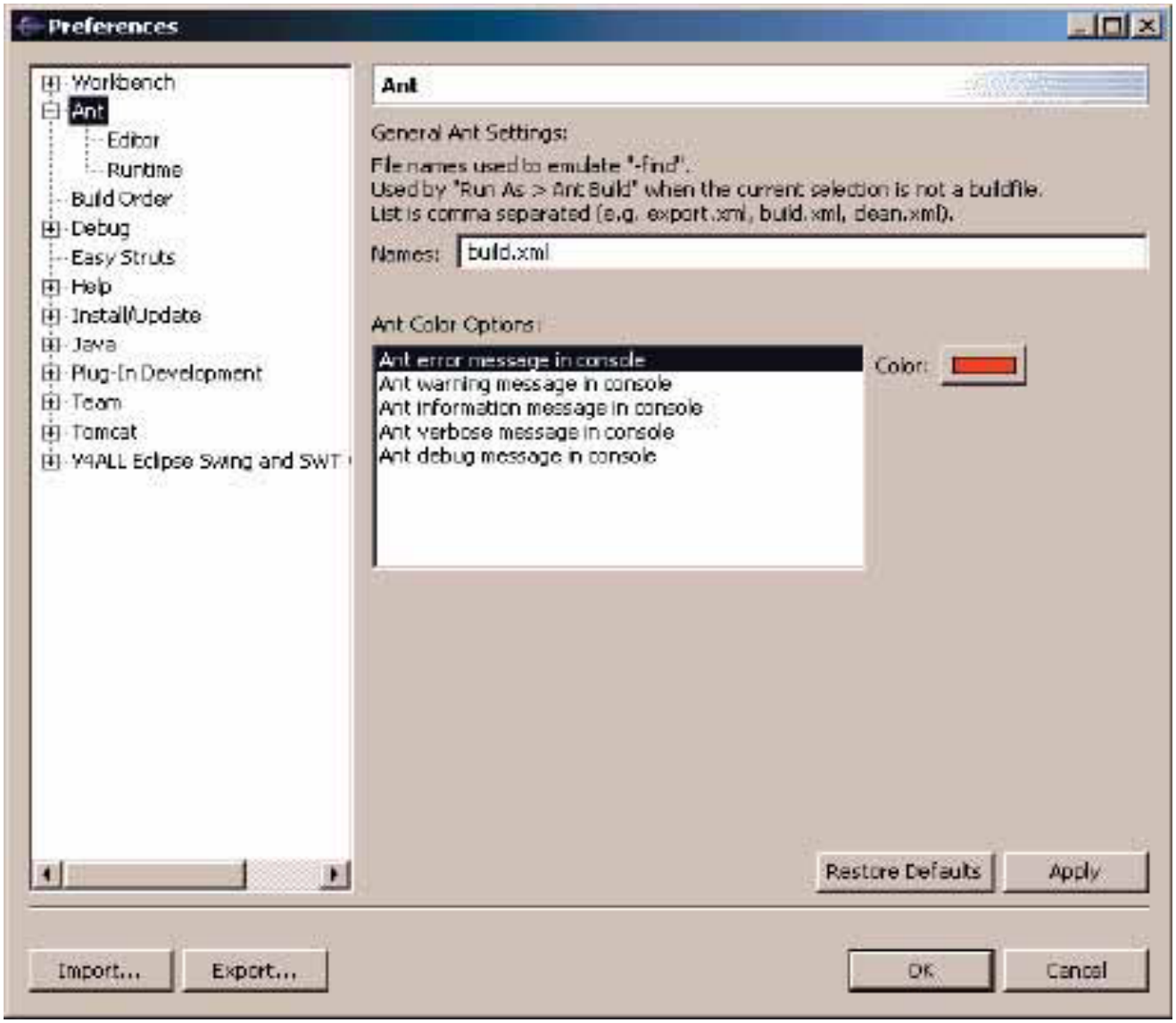

Figure 5-7. Configuring the build file

You can also set Ant runtime options by selecting the Runtime node, as shown in Figure 5-8. For example, to use a more recent version of Ant in Eclipse (Eclipse 2.1.1 comes with Ant 1.5.3, but the latest version of Ant as of this writing is 1.5.4, and version 1.6 is out in beta—you can get alternate Ant versions directly from Apache at **http://ant.apache.org**), select the Runtime item and change the JAR entries you see in Figure 5.7 to the new versions of ant.jar and optional.jar. Note that you can also set Ant variables like ANT_HOME in this dialog.

Figure 5-8. Configuring Ant in Eclipse

You can add new Ant tasks and types with the Tasks and Types tabs, which means that those tasks and types will be available to build files without having to use Ant taskdef or typedef elements. Eclipse also lets you set global Ant properties if you select the Properties tab in this dialog. To add a new global property, click the Add button in the Properties tab, and enter a name and value for the new property.

You also have limited control over the Ant editor's options using the preferences dialog and selecting the Antγ Editor item. The Ant editor is actually little more than a simple XML editor, but you can specify such items as the colors used in syntax highlighting, or whether or not the editor shows an overview ruler, as you see in Figure 5-9.

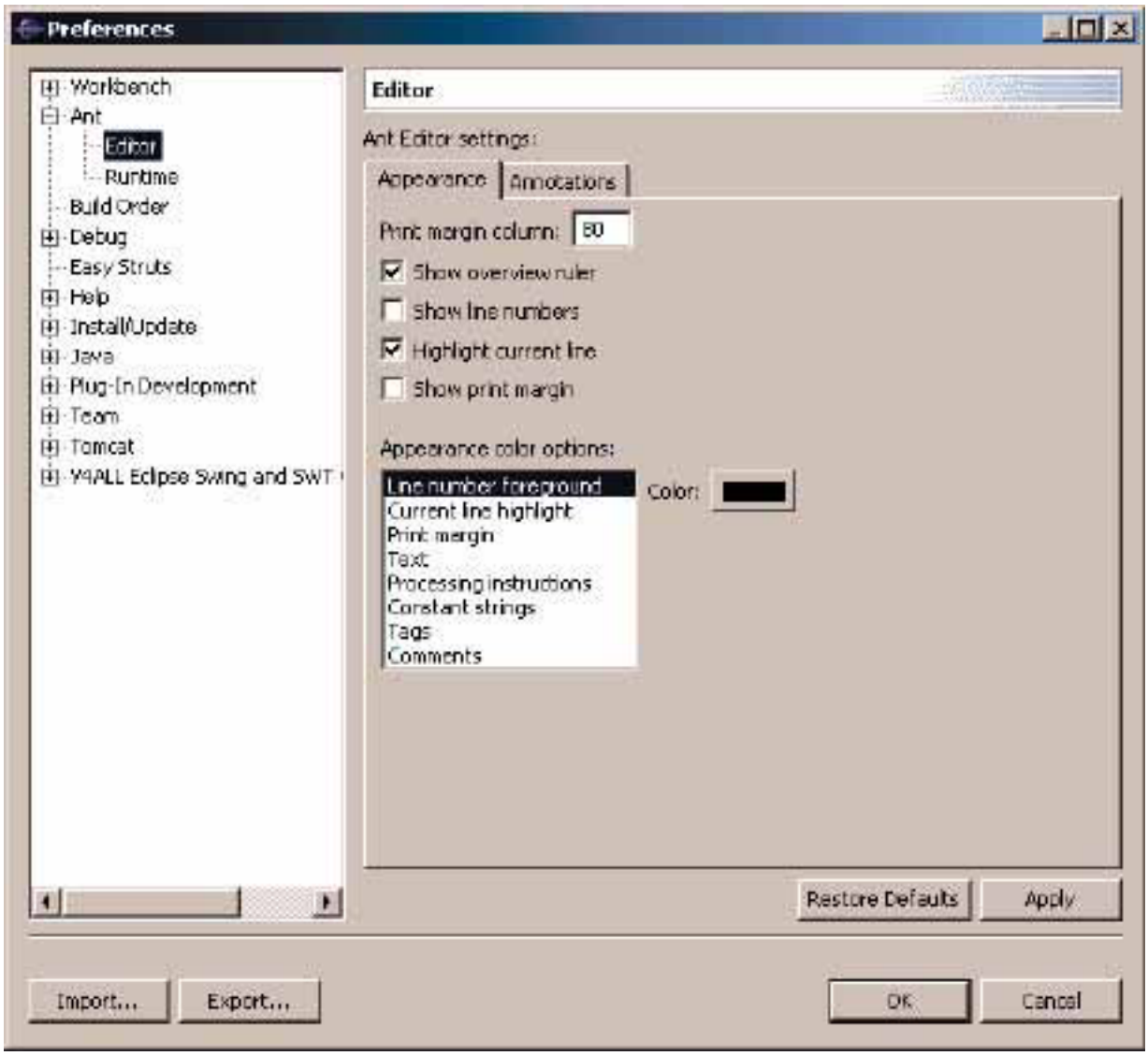

Figure 5-9. Configuring the Ant editor

The Ant editor also offers code assist. For example, if you enter < and then pause in typing, code assist will give you a list of possible Ant build file elements. Entering additional letters narrows down the list—for example, entering <p makes code assist give you the choice of <path>, <patternset>, and <property>. Code assist will also list possible attributes of Ant elements; just click inside the opening tag of an Ant element and press Ctrl+Space. Also, letting the mouse cursor rest on one of the items in the code assist list makes Eclipse display an explanation of what that item does. For example, the explanation for the property element is "Sets a property by name, or a set of properties (from file or resource) in the project."

You can also configure Ant when you're about to run it, before selecting an Ant target to build. Right-click build.xml, select Run Ant, and click the Main tab in the dialog that opens, as shown in Figure 5-10. You can set the location of the build file you want to use here, as well as the base directory for the build. You can also set Ant arguments in the Arguments field.

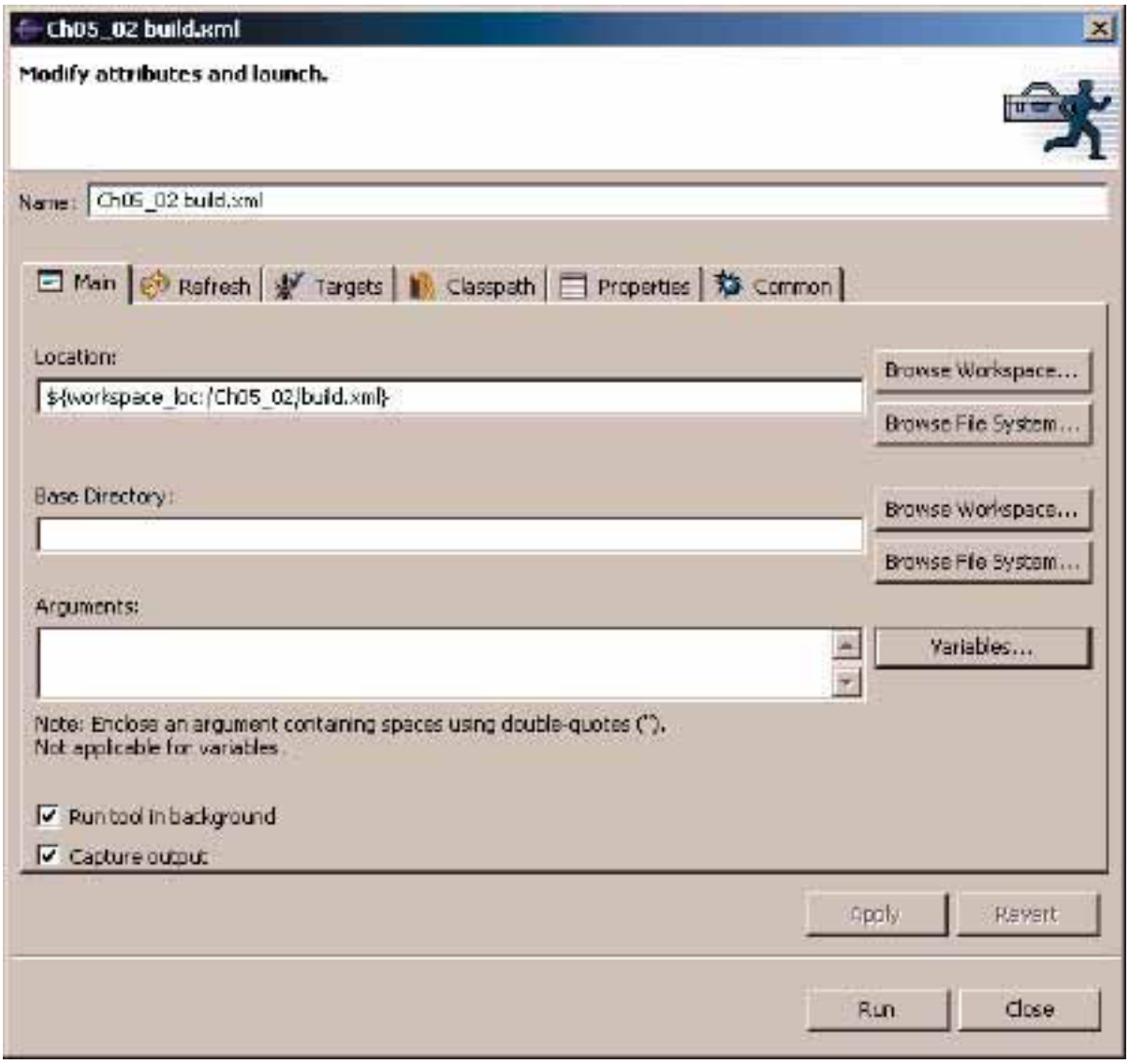

Figure 5-10. Setting the run configuration

You can also run Ant as an external tool: select Run➤ External Tools➤ External Tools, click Program, click New, and enter the name you want to give to the external version of Ant. To fill in the Location field, click Browse File System, find the correct file for your operating system (for example, ant.bat in the Ant bin folder in Windows). In the Working Directory field enter the directory of your buildfile, and click to execute your build file.

In addition, there's also an Ant view in Eclipse, which you open with Window➤ Show View➤ Other➤ Ant. This view provides an Ant-based overview of build files; to add a build file to this view, right-click the view, select Add Buildfile, and navigate to the build file you want to display. The view will display a breakdown of the build file, as you see in Figure 5-11, and you can run various Ant targets by right-clicking them and selecting Run.

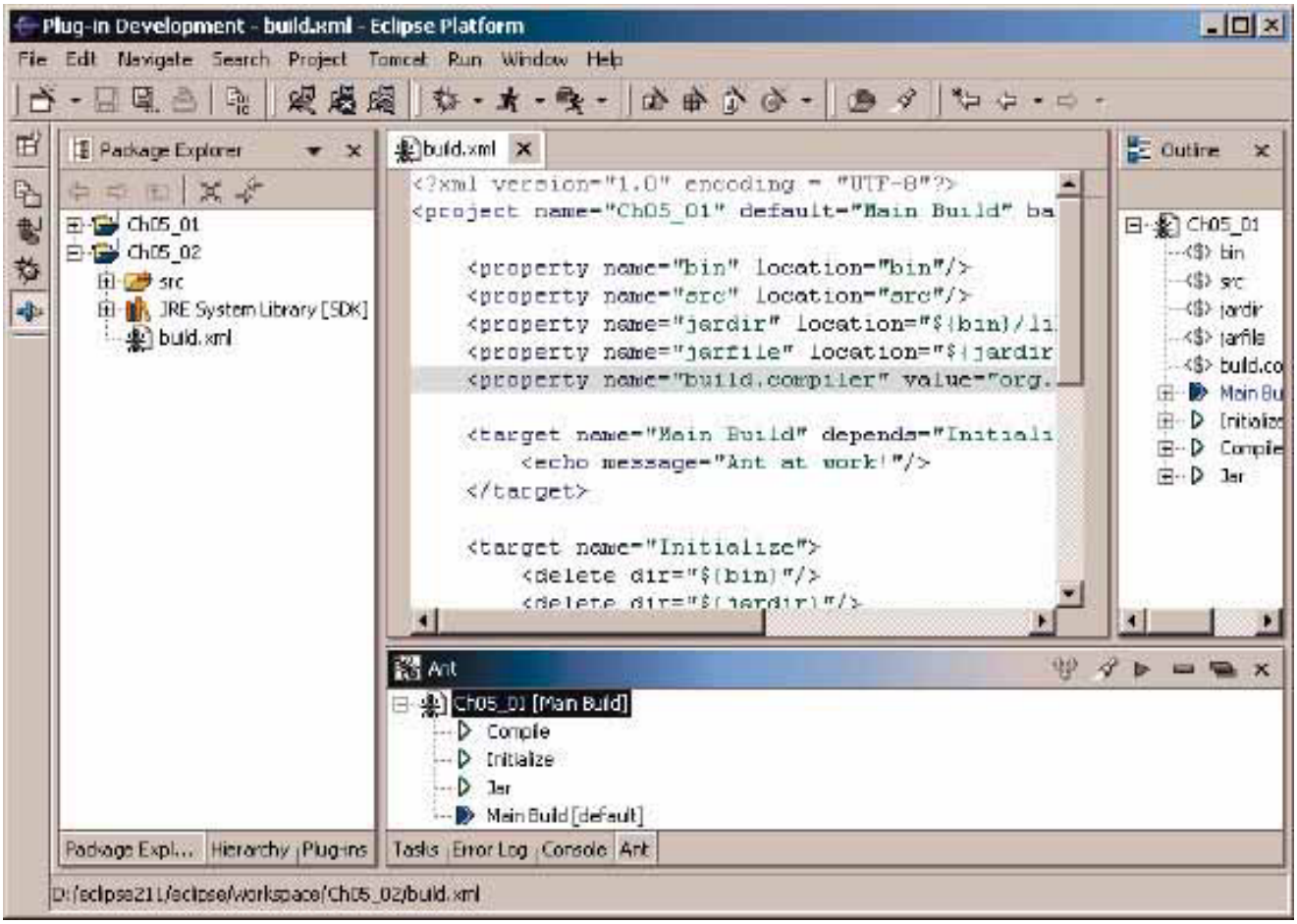

Figure 5-11. The Ant view

Catching Errors in Build Files

Eclipse gives you some support for catching Ant errors before you run Ant, but not much. The Eclipse Ant editor doesn't handle syntax errors as the JDT editor does for Java—for example, if you misspell the project element's default attribute as deefault, the Ant editor won't have a problem. However, if the XML in your build file has syntax errors, such as missing a closing tag or improper nesting(in XML terms: if your XML is not well-formed), you'll see the same wavy red line and hollow red box you see in the JDT editor in the Ant editor, indicating a syntax error, as shown in Figure 5-12. You can determine what error occurred by looking at the wavy line's tooltip, as shown in the figure, where we haven't closed the mkdir element.

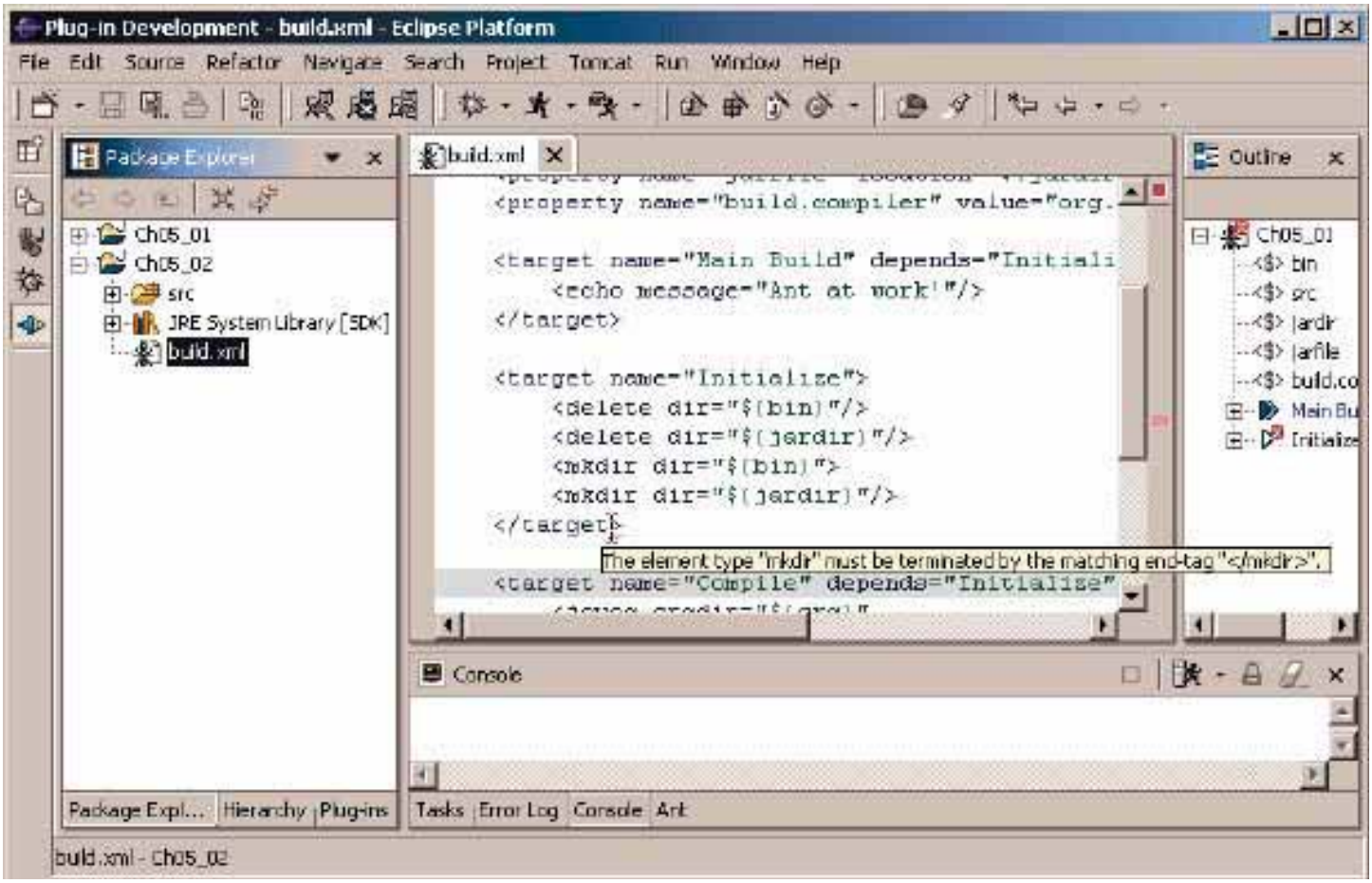

Figure 5-12. Handling a syntax error

If you miss an XML error like this in the editor, you'll see a message reminding you of it as soon as you try to run Ant to build the project. The Ant editor doesn't display non-XML syntax errors (like spelling the project attribute default to deefault), but you'll automatically see any syntax errors listed when you try to run Ant, as shown at the top of Figure 5-13.

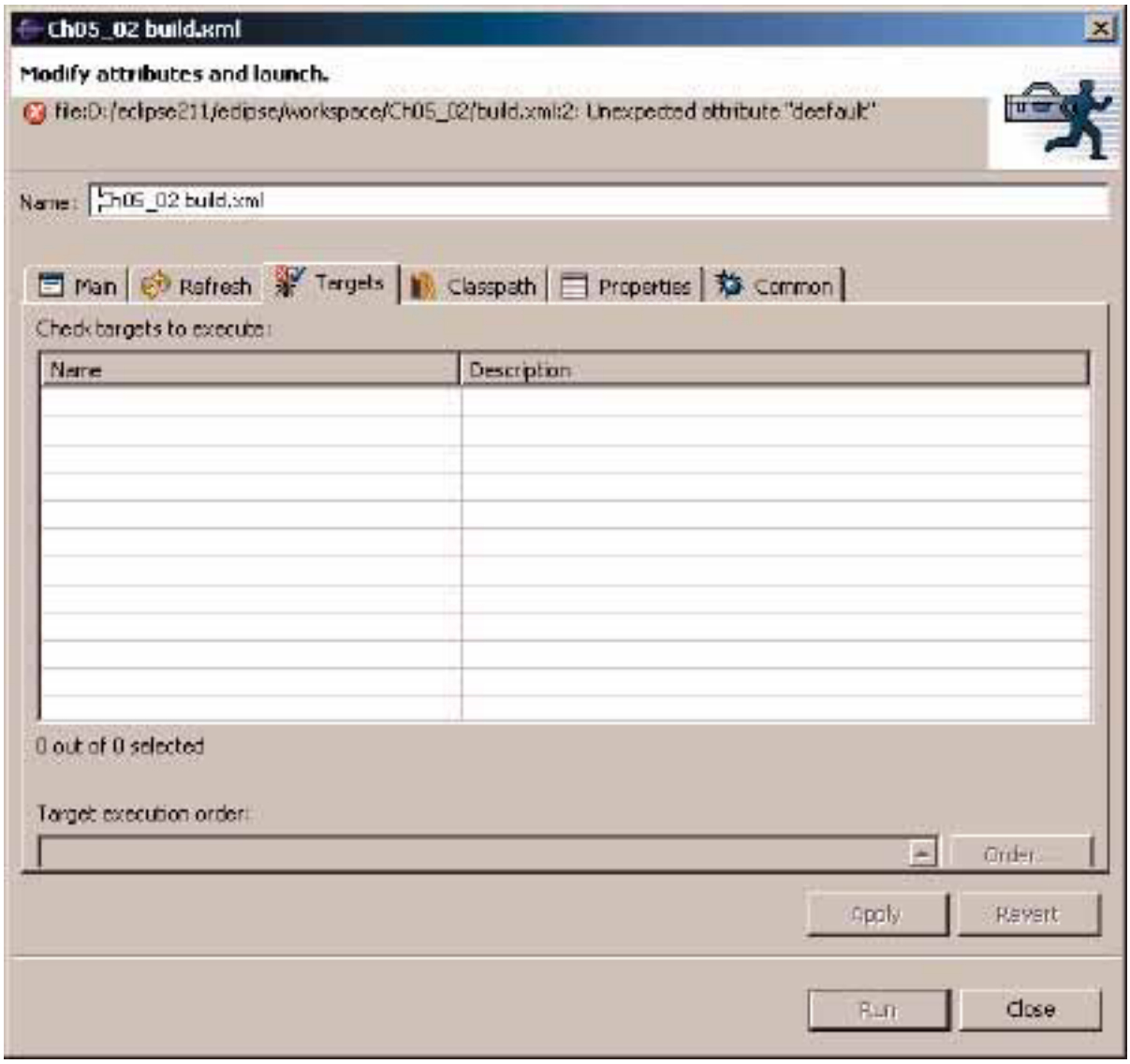

Figure 5-13. An error description

Unfortunately, that's as much support as Eclipse gives you for handling Ant errors—when Ant runs, it'll display its own errors in the Console view, and it's up to you to take it from there (although Eclipse does hyperlink errors to the associated lines in the build or Java file). Eclipse doesn't support interactive debugging of Ant scripts yet, which is a pity, given how complex those scripts can become. Perhaps we'll see that one day in Eclipse; in fact, given the popularity of Ant, it's possible that future versions of Eclipse may include an Ant wizard, invoked with File γ New, that will let you set up build targets, directories, and Ant tasks as easily as creating a new Eclipse project.