Obfuscating Classes

3

"Any sufficiently advanced technology is indistinguishable from magic."

Murphy's Technology Laws

IN THIS CHAPTER

Protecting the Ideas Behind Your Code

Reverse engineering and hacking have been around since the early days of software development. As a matter of fact, stealing or replicating someone else's ideas has always been the easiest way of creating competitive products. There is, of course, a perfectly acceptable method of building on previous discoveries by others—and as long as the others don't mind, it works fine. Most inventors and researchers, however, would like to get credit and possibly a financial reward for their work. In simpler terms, they also have mortgages to pay and vacations to take.

A good way of protecting intellectual property is for the author to obtain copyrights and patents on the unique features of the work. This is certainly recommended for inventions and major discoveries that required a lot of investment into research and development. Copyrighting software is a rather easy and cost-effective process, but it protects only the "original" code of the application, not the ideas behind it. Others would not be able to take copyrighted code and use it in their applications without the author's permission, but if they have their own implementation of the same feature, it would not be considered a violation to use that. Patents provide a much better protection because they cover the ideas and algorithms rather than a specific implementation, but they are expensive to file and can take years to obtain.

Is the risk of having your application hacked real? If it has good ideas, then absolutely. Most of the widely publicized reverse engineering cases at the time of this writing did not occur with Java products, but here's an excerpt from a Java vendor (DataDirect Technologies):

> ROCKVILLE, MD., July 1, 2002—DataDirect Technologies, Inc., an industry-leading data connectivity vendor has filed a lawsuit against i-net Software GmbH alleging copyright infringement and breach of contract. DataDirect Technologies is seeking both preliminary and permanent injunctive relief to prevent i-net from engaging in further efforts to market and sell products which DataDirect Technologies believes were illegally reverse-engineered from its products.

DataDirect Technologies claims that a competitor reverse engineered its product, and yet even today its product has only minimal protection from decompiling.

In the real world, copyrighting the code and getting a patent for an approach cannot provide adequate protection if a competitor or hacker can easily learn the implementation from the source code. The issues of legal protection are discussed in a separate chapter, but for now, let's focus on smart ways to protect the intellectual property (IP) of Java applications.

Obfuscation As a Protection of Intellectual Property

Obfuscation is the process of transforming bytecode to a less human-readable form with the purpose of complicating reverse engineering. It typically includes stripping out all the debug information, such as variable tables and line numbers, and renaming packages, classes, and methods to machine-generated names. Advanced obfuscators go further and change the control flow of Java code by restructuring the existing logic and inserting bogus code that will not execute. The premise of the obfuscation is that the transformations do not break the validity of the bytecode and do not alter the exposed functionality.

Obfuscation is possible for the same reasons that decompiling is possible: Java bytecode is standardized and well documented. Obfuscators load Java class files, parse their formats, and then apply transformations based on supported features. When all the transformations are applied, the bytecode is saved as a new class file. The new file has a different internal structure but behaves just like the original file.

Obfuscators are especially necessary for products and technologies in which the implementation logic is delivered to the user. That is the case for HTML pages and JavaScript where the product is distributed in source code form. Java doesn't fare much better because, even though it is typically distributed in binary bytecode, using a decompiler as described in the previous chapter can produce the source code—which is almost as good as the original.

Transformations Performed by Obfuscators

No standards exist for obfuscation, so the level of protection varies based on the quality of the obfuscator. The following sections present some of the features commonly found in obfuscators. We will use `ChatServer`'s `sendMessage` method to illustrate how each transformation affects the decompiled code. The original source code for `sendMessage` is shown in Listing 3.1.

LISTING 3.1 Original Source Code of `sendMessage`

```
public void sendMessage(String host, String message) throws Exception {
    if (host == null || host.trim().length() == 0)
        throw new Exception ("Please specify host name");

    System.out.println("Sending message to host " + host + ": " + message);
    String url = "//" + host + ":" + this.registryPort + "/chatserver";
    ChatServerRemote remoteServer = (ChatServerRemote)Naming.lookup(url);

    MessageInfo messageInfo = new MessageInfo(this.hostName, this.userName);
    remoteServer.receiveMessage(message, messageInfo);
    System.out.println("Message sent to host " + host);
}
```

Stripping Out Debug Information

Java bytecode can contain information inserted by the compiler that helps debug the running code. The information inserted by `javac` can contain some or all of the following: line numbers, variable names, and source filenames. Debug information is not needed to run the class but is used by debuggers to associate the bytecode with the source code. Decompilers use this information to better reconstruct the source code. With full debug information in the class file, the decompiled code is almost identical to the original source code. When the debug information is stripped out, the names that were stored are lost, so decompilers have to generate their own names. In our case, after the stripping, `sendMessage` parameter names would appear as `s1` and `s2` instead of `host` and `message`.

Name Mangling

Developers use meaningful names for packages, classes, and methods. Our sample chat application's server implementation is called `ChatServer` and the method that sends a message to another user is called `sendMessage`. Good names are crucial for development and maintenance, but they mean nothing to the JVM. Java Runtime (JRE) doesn't care whether `sendMessage` is called `goShopping` or `abcdefg`; it still invokes it and executes it. By renaming

the meaningful human-readable names to meaningless machine-generated ones, obfuscators make the task of understanding the decompiled code much harder. What used to be `ChatServer.sendMessage` becomes `d.a;` when many classes and methods exist with the same names, the decompiled code is extremely hard to follow. A good obfuscator takes advantage of polymorphism to make matters worse. Three methods with different names and signatures doing different tasks in the original code can be renamed to the same common name in the obfuscated code. Because their signatures are different, it does not violate the Java language specification but adds confusion to the decompiled code. Listing 3.2 shows an example of a decompiled `sendMessage` after obfuscation that stripped the debugging information and performed name mangling.

LISTING 3.2 Decompiled `sendMessage` After Name Mangling

```
public void a(String s, String s1)
    throws Exception
{
    if(s == null ¦¦ s.trim().length() == 0)
    {
        throw new Exception("Please specify host name");
    } else
    {
        System.out.println(String.valueOf(String.valueOf((
           new StringBuffer("Sending message to host ")
           ).append(s).append(": ").append(s1))));
        String s2 = String.valueOf(String.valueOf((
           new StringBuffer("//")).append(s).append(":")
           .append(b).append("/chatserver")));
        b b1 = (b)Naming.lookup(s2);
        MessageInfo messageinfo = new MessageInfo(e, f);
        b1.receiveMessage(s1, messageinfo);
        System.out.println("Message sent to host ".concat(
              String.valueOf(String.valueOf(s))));
        return;
    }
}
```

Encoding Java Strings

Java strings are stored as plain text inside the bytecode. Most of the well-written applications have traces inside the code that produce execution logs for debugging and audit trace. Even if class and method names are changed, the strings written by methods to a log file or console

can betray the method purpose. In our case, `ChatServer.sendMessage` outputs a trace message using the following:

```
System.out.println("Sending message to host " + host + ": " + message);
```

Even if `ChatServer.sendMessage` is renamed to `d.a`, when you see a trace like this one in the decompiled message body, it is clear what the method does. However, if the string is encoded in bytecode, the decompiled version of the class looks like this:

```
System.out.println(String.valueOf(String.valueOf((new
StringBuffer(a("A\025wV6¦\0279_:a\003xU:2\004v\0227}\003m\022"))
).append(s).append(a("(P")).append(s1))));
```

If you look closely at the encoded string, it is first passed to the `a()` method, which decodes it and returns the readable string to the `System.out.println()` method. String encoding is a powerful feature that should be provided by a commercial-strength obfuscator.

Changing Control Flow

The transformations presented earlier make reverse engineering of the obfuscated code harder, but they do not change the fundamental structure of the Java code. They also do nothing to protect the algorithms and program control flow, which is usually the most important part of the innovation. The decompiled version of `ChatServer.sendMessage` shown earlier is still fairly understandable. You can see that the code first checks for valid input and throws an exception upon error. Then it looks up the remote server object and invokes a method on it.

The best obfuscators are capable of transforming the execution flow of bytecode by inserting bogus conditional and `goto` statements. This can slow down the execution somewhat, but it might be a small price to pay for the increased protection of the IP. Listing 3.3 shows what `sendMessage` has become after all the transformations discussed earlier have been applied.

LISTING 3.3 Decompiled `sendMessage` After All Transformations

```
    public void a(String s, String s1)
        throws Exception
    {
        boolean flag = MessageInfo.c;
        s;
        if(flag) goto _L2; else goto _L1
_L1:
        JVM INSTR ifnull 29;
           goto _L3 _L4
_L3:
        s.trim();
```

LISTING 3.3 Continued

```
_L2:
        if(flag) goto _L6; else goto _L5
_L5:
        length();
        JVM INSTR ifne 42;
           goto _L4 _L7
_L4:
        throw new Exception(a("\002)qUe7egDs1,rM6:*g@6<$yQ"));
_L7:
        System.out.println(String.valueOf(String.valueOf((
            new StringBuffer(a("\001 zP\177<\"4Ys!6uSsr1{\024~=6`\024"))
            ).append(s).append(a("he")).append(s1))));
        String.valueOf(String.valueOf(
            (new StringBuffer(a("}j"))).append(s).append(":")
            .append(b).append(a("}&¦Ub! fBs "))));
_L6:
        String s2;
        s2;
        covertjava.chat.b b1 = (covertjava.chat.b)Naming.lookup(s2);
        MessageInfo messageinfo = new MessageInfo(e, f);
        b1.receiveMessage(s1, messageinfo);
        System.out.println(a("\037 gGw5 4Gs<14@yr-{Gbr").concat(String.valueOf
        ➥(String.valueOf(s))));
        if(flag)
            b.c = !b.c;
        return;
    }
```

Now that's a total, but powerful, mess! `sendMessage` is a fairly small method with little conditional logic. If control flow obfuscation was applied to a more complex method with `for` loops, `if` statements, and local variables, the obfuscation would be even more effective.

Inserting Corrupt Code

Inserting corrupt code is a somewhat dubious technique used by some obfuscators to prevent obfuscated classes from decompiling. The technique is based on a loose interpretation of the Java bytecode specification by the Java Runtime. JRE does not strictly enforce all the rules of bytecode format verification, and that allows obfuscators to introduce incorrect bytecode into the class files. The introduced code does not prevent the original code from executing, but an attempt to decompile the class file results in a failure—or at best in confusing source code full of `JVM INSTR` keywords. Listing 3.3 shows how a decompiler might handle corrupt code. The risk of using this method is that the corrupted code might not run on a version of JVM that

more closely adheres to the specification. Even if it is not an issue with the majority of JVMs today, it might become a problem later.

Eliminating Unused Code (Shrinking)

As an added benefit, most obfuscators remove unused code, which results in application size reduction. For example, if a class called `A` has a method called `m()` that is never called by any class, the code for `m()` is stripped out of `A`'s bytecode. This feature is especially useful for code that is downloaded via the Internet or installed in unsecured environments.

Optimizing Bytecode

Another added benefit touted by obfuscators is potential code optimization. The vendors claim that declaring nonfinal methods as final where possible and performing minor code improvements can help speed up execution. It is hard to assess the real performance gains, and most vendors do not publish the metrics. What is worth noting here is that, with every new release, JIT compilers are becoming more powerful. Therefore, features such as method finalization and dead code elimination are most likely performed by it anyway.

Knowing the Best Obfuscators

Plenty of obfuscators are available, and most of them contain the same set of core features. Table 3.1 includes just a few of the most popular products, both free and commercial.

TABLE 3.1

Popular Obfuscators

PRODUCT	KLASSMASTER	PROGUARD	RETRO GUARD	DASH-O	JSHRINK
Version	4.1	1.7	1.1.13	2.x	2.0
Price	$199–$399	Free	Free	$895–$2995	$95
Stripping out of debug information	Yes	Yes	Yes	Yes	Yes
Name mangling	Yes	Yes	Yes	Yes	Yes
Encoding of Java strings	Yes	No	No	No	Yes
Changing of control flow	Yes	No	No	No	No
Insertion of corrupt code	Yes	No	No	No	No
Elimination of unused code (shrinking)	Yes	Yes	No	Yes	Yes
Optimizing of bytecode	No	No	No	Yes	Yes
Flexibility of scripting language and obfuscation control	Excellent	Excellent	Good	Not rated	Good
Reconstruction of stack traces	Yes	Yes	No	No	No

For commercial applications that contain intellectual property, I recommend Zelix KlassMaster primarily because of its unique control flow obfuscation. This technique makes the obfuscated code truly hard to crack, so the product is worth every dollar you will pay for it. At the time of writing, it is the only obfuscator known to have this feature. ProGuard is available free from `www.sourceforge.net` and is the best choice for the budget-conscious user with applications that do not require commercial-strength protection.

Potential Problems and Common Solutions

Obfuscation is a reasonably safe process that should preserve application functionality. However, in certain cases the transformations performed by obfuscators can inadvertently break code that used to work. The following sections look at the common problems and recommended solutions.

Dynamic Class Loading

The renaming of packages, classes, methods, and variables works fine as long as the name is changed consistently throughout the system. Obfuscators ensure that any static references within the bytecode are updated to reflect the new name. However, if the code performs dynamic class loading using `Class.forName()` or `ClassLoader.loadClass()` passing an original class name, a `ClassNotFound` exception can result. Modern obfuscators are pretty good with handling such cases, and they attempt to change the strings to reflect the new names. If the string is created at runtime or read from a properties file, though, the obfuscator is incapable of handling it. Good obfuscators produce a log file with warnings pointing out the code that has potential for runtime problems.

STORIES FROM THE TRENCHES

The most innovative product from CreamTec is WebCream, which is available for a free download from the Web. The free edition is limited to five concurrent users; to get more users, you must buy a commercial license. Having grown up in the Ukraine, I knew many people who would prefer to crack the licensing to turn the free edition into an unlimited edition that would normally be worth thousands of dollars. At CreamTec, we used a simple, free obfuscator that didn't do much more than name mangling. We thought it was good enough until a friend of mine, who views limited-functionality commercial software as a personal insult, cracked our licensing code in less than 15 minutes. The message was clear enough, and we decided to purchase Zelix KlassMaster to protect the product as well as we could. After we used the aggressive control flow obfuscation with a few extra tricks, our friend has not been able to get to the licensing code with the same ease as before—and because he didn't want to spend days figuring it out, he has given up.

The simplest solution is to configure the obfuscator to preserve the names of dynamically loaded classes. The content of the class, such as the methods, variables, and code, can still be transformed.

Reflection

Reflection requires compile-time knowledge of method and field names, so it is also affected by obfuscation. Be sure to use a good obfuscator and to review the log file for warnings. Just as with the dynamic class loading, if runtime errors are caused by obfuscation, you must exclude from obfuscation the method or field names that are referenced in `Class.getMethod` or `Class.getField`.

Serialization

Serialized Java objects include instance data and information about the class. If the version of the class or its structure changes, a deserialization exception can result. Obfuscated classes can be serialized and deserialized, but an attempt to deserialize an instance of a nonobfuscated class by an obfuscated class will fail. This is not a very common problem, and it can usually be solved by excluding the serializable classes from obfuscation or avoiding the mixing of serialized classes.

Naming Conventions Violation

The renaming of methods can violate design patterns such as Enterprise JavaBeans (EJB), where the bean developer is required to provide methods with certain names and signatures. EJB callback methods such as `ejbCreate` and `ejbRemove` are not defined by a super class or an interface. Providing these methods with a specific signature is a mere convention prescribed by EJB specification and enforced by the container. Changing callback method names violates the naming convention and makes the bean unusable. You should always be sure to exclude the names of such methods from obfuscation.

Maintenance Difficulties

Last, but not least, obfuscation makes maintaining and troubleshooting applications more difficult. Java exception handling is an effective way of isolating the faulty code, and looking at the stack trace can generally give you a good idea of what went wrong and where. Keeping the debugging information for source filenames and line numbers enables the runtime to report the exact location in code where the error occurred. If done carelessly, obfuscation can inhibit this feature and make debugging harder because the developer sees only the obfuscated class names instead of the real class names and line numbers.

You should preserve at least the line number information in the obfuscated code. Good obfuscators produce a log of the transformations, including the mapping between the original class names and methods and the obfuscated counterparts. The following is an excerpt from the log file generated by Zelix KlassMaster for the `ChatServer` class:

```
Class: public covertjava.chat.ChatServer    =>    covertjava.chat.d
    Source: "ChatServer.java"
    FieldsOf: covertjava.chat.ChatServer
        hostName    =>    e
        protected static instance    =>    a
        messageListener    =>    d
        protected registry    =>    c
        protected registryPort    =>    b
        userName    =>    f
    MethodsOf: covertjava.chat.ChatServer
        public static getInstance()    =>    a
        public getRegistry(int)    =>    a
        public init()    =>    b
        public receiveMessage(java.lang.String, covertjava.chat.MessageInfo)
        ➥NameNotChanged
        public sendMessage(java.lang.String, java.lang.String)    =>    a
        public setMessageListener(covertjava.chat.MessageListener)    =>    a
```

So, if an exception stack trace shows the `covertjava.chat.d.b` method, you can use the log and find out that it was originally called `"init"` in a class that was originally called `covertjava.chat.ChatServer`. If the exception occurred in `covertjava.chat.d.a`, you would not know the original method name for sure because multiple mappings exist (witness the power of overloading). That's why line numbers are so important. By using the log file and the line number in the original source file, you can quickly locate the problem area in the application code.

Some obfuscators provide a utility that reconstructs the stack traces. This is a convenient way of getting the real stack trace for the obfuscated stack trace. The utility typically uses the same method as we used earlier, but it automates the job—so why not save ourselves some time? It also allows scrambling the line numbers for extra protection.

Using Zelix KlassMaster to Obfuscate a Chat Application

Even though each obfuscator has its own format of configuring the transformations, they all support a common set of features. The Chat application does not contain state-of-the-art algorithms or patent-pending inventions, but it is dear to our hearts so we are going to use Zelix KlassMaster to protect it from the prying eyes of hackers and thieves.

First, we obtain a copy of Zelix KlassMaster and install it on a local machine. Remember that we refer to the Chat application's home directory as `CovertJava`. Next, we copy `ZKM.jar` from KlassMaster's installation directory to our project `lib` directory so we can script against it. The easiest way to create the obfuscation script is with KlassMaster's GUI. Using the command

```
java -jar ZKM.jar
```

from the `lib` directory, we run the GUI. Then, in the initial helper dialog box that appears, we select the Set Classpath option. We now select the runtime libraries of the JDK we're using and, in the Open Classes dialog box that appears next, we select `CovertJava/lib/chat.jar`. After that, KlassMaster should load all the classes of the Chat application and we should be able to view the internal structure of the bytecode. The screen should look similar to Figure 3.1.

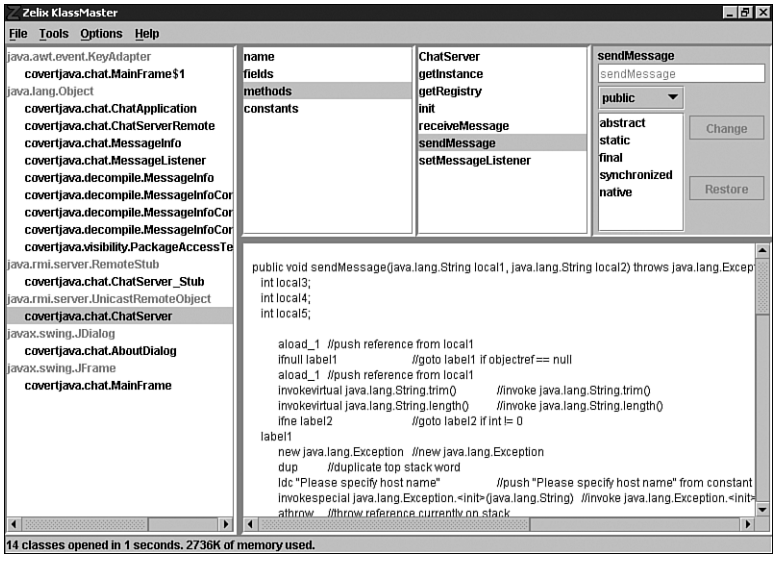

FIGURE 3.1 Chat classes loaded into the KlassMaster GUI.

While working with the GUI, you can easily see just how flexible KlassMaster is. You can manually change the names of classes, methods, and fields; modify the visibility of classes or methods; make methods final; change text strings; and do other cool stuff. KlassMaster attempts to propagate the changes throughout the loaded code, so if other classes refer to a method and you change its name, the referring classes are updated to reflect the change. After making all your changes, you can save the classes as is or trim and obfuscate them first. Classes loaded into the GUI environment can be further modified after the obfuscation, even though I can't think of a reason why someone would need to do so. For details of KlassMaster's features and how to use it, please refer to its user manual.

A well-written Java application provides scripts to build it, so let's integrate obfuscation into our build script. We start by using KlassMaster's GUI to create the obfuscation script. Then, we update it manually to make it more flexible. It is entirely possible to write the script manually or copy and modify a sample script. We run the GUI and select ZKM Script Helper from the Tools menu. Then, we do the following:

1. Read the instructions on the Introductory Page and click Next.
2. On the Classpath Statement page, select `rt.jar` and click Next.
3. On the Open Statement page, navigate to `CovertJava/distrib/chat.jar` and click > to select it for opening. We only need one file because all our application classes are packaged in it. Click Next.
4. On the TrimExclude Statement page, the default exclusions are preset to exclude the cases where obfuscation is likely to result in an error. For example, renaming methods of an EJB implementation class makes it unusable, so EJBs are excluded by default.
5. On the Trim Statement page, select the Delete Source File Attributes check box and the Delete Deprecated Attributes check box to get rid of the debug information; then click Next.
6. In the Don't Change Main Class Name combo box on the Exclude Statement page, select `covertjava.chat.ChatApplication` to preserve its name. This keeps JAR manifest entries valid and enables users to continue invoking the chat using a human-readable name.
7. On the Obfuscate Statement page, select Aggressive in the Obfuscate Control Flow combo box. Then select Aggressive in the Encrypt String Literals combo box, and select Scramble in the Line Number Tables combo box. This ensures adequate protection for the code but enables us to translate stack traces later. Make sure that Produce a Change Log File is checked and click Next.
8. On the SaveAll Statement page, navigate to `CovertJava/distrib` and create a subdirectory called `obfuscated`. Select the newly created directory for output and click Next.
9. The next page should show the script text and allow us to save it to a directory. Save it as `obfuscate_script.txt` in the `CovertJava/build` directory and exit the GUI.

The resulting script should look similar to Listing 3.4.

LISTING 3.4 Obfuscation Script Generated by the GUI

```
/*****************************************************************************/
/* Generated by Zelix KlassMaster 4.1.1 ZKM Script Helper 2003.08.13 17:03:43 */
/*****************************************************************************/
```

```
classpath   "c:\java\jdk1.4\jre\lib\rt.jar"
            "c:\java\jdk1.4\jre\lib\sunrsasign.jar"
            "c:\java\jdk1.4\jre\lib\jsse.jar"
            "c:\java\jdk1.4\jre\lib\jce.jar"
            "c:\java\jdk1.4\jre\lib\charsets.jar";

open        "C:\Projects\CovertJava\distrib\chat.jar";

trim        deleteSourceFileAttributes=true
            deleteDeprecatedAttributes=true
            deleteUnknownAttributes=false;

exclude     covertjava.chat.^ChatApplication^ public static main(java.lang.String[]);

obfuscate   changeLogFileIn=""
            changeLogFileOut="ChangeLog.txt"
            obfuscateFlow=aggressive
            encryptStringLiterals=aggressive
            lineNumbers=scramble;

saveAll     archiveCompression=all "C:\Projects\CovertJava\distrib\obfuscated";
```

A good idea would be to replace the absolute file paths with the relative ones, so that instead of opening `C:\Projects\CovertJava\distrib\chat.jar`, the script opens `distrib\chat.jar`. Finally, we will integrate obfuscation into the build process by declaring a custom task and adding a target that calls it. KlassMaster is written in Java and can be called from any build script. Conveniently, it provides a wrapper class for Ant integration, so all we have to do is add the following to Chat's `build.xml`:

```
<!-- Define a task that will execute Zelix KlassMaster to obfuscate classes -->
<taskdef name="obfuscate" classname="ZKMTask" classpath="${basedir}/lib/ZKM.jar"/>
...
<!-- Define a target that produces obfuscated version of Chat -->
<target name="obfuscate" depends="release">
    <obfuscate scriptFileName="${basedir}/build/obfuscate_script.txt"
        logFileName="${basedir}/build/obfuscate_log.txt"
        trimLogFileName="${basedir}/build/obfuscate_trim_log.txt"
        defaultExcludeFileName="${basedir}/build/obfuscate_defaultExclude.txt"
        defaultTrimExcludeFileName="${basedir}/build/obfuscate_defaultTrimExclude.txt"
        defaultDirectoryName="${basedir}"
    />
</target>
```

We can now run Ant on obfuscate target. If the build is successful, a new file (`chat.jar`) is created in `CovertJava/distrib/obfuscated`. This file contains the obfuscated version of Chat that can still be invoked using the `java -jar chat.jar` command. Take a few moments to look inside that JAR and try decompiling some of the classes.

Before we close the subject of using KlassMaster, I'd like to give a few more examples of script file syntax for excluding classes and class members from obfuscation. The format shown in Table 3.2 can be used for statements of obfuscation script that take in names as parameters. ZKM script language supports wildcards, such as `*` (any sequence of characters) and `?` (any one character), and boolean operations, such as `||` (or) and `!` (not). For a detailed explanation and full syntax, please refer to KlassMaster documentation.

TABLE 3.2

Commonly Used Name Patterns for KlassMaster

SYNTAX	WHAT IT MATCHES
`package1.package2.`	Package names `package1` and `package2`. Other package names and children of `package2` are not matched.
`*.`	All package names in the application.
`Class1`	The name of the class `Class1`.
`package1.Class1`	The name of `Class1` in package `package1` but *not* `package1`'s name.
`package1.^Class1`	The names of `Class1` *and* `package1`.
`package1.^Class1^ method1()`	The names of `package1`, `Class1`, and `method1` with no parameters.
`package1.^Class1^ method1(*)`	The names of `package1`, `Class1`, and all overloaded versions of `method1`.

Cracking Obfuscated Code

Now that we have spent so much time talking about how to protect intellectual property through obfuscation, a few words are due on the strength of the protection. Does a good obfuscator make it hard to hack an application? Absolutely. Does it guarantee that the application will not be hacked? Not at all!

Unless flow control obfuscation is used, reading and working with the obfuscated code is not that difficult. The key point is finding a good starting point for decompiling. Chapter 2, "Decompiling Classes," presented several techniques for reverse engineering of applications, but obfuscation can defeat many of them. For example, the most effective way of locating a starting point is text searching through the class files. With string encoding, the search will yield no results because the strings are not stored as plain text. Package names and class names can no longer be used to learn about the application structure and to select a good starting point. It is still technically possible to decompile the application entry point and work your way through the control flow for a decent-size application, but it is not feasible.

For flow-obfuscated code, the most sensible method of learning the application implementation is using a good old debugger. Most IDEs come with debugging capabilities, but our case will require a heavyweight debugger capable of working without the source code. To find a good starting point for decompiling, the application needs to be run in debug mode. Java has a standard API for debuggers called Debugger API (duh!) that is capable of local as well as remote debugging. Remote debugging enables the debugger to attach itself to an application running in debug mode and is a preferred way of cracking the application. Good debuggers display in-depth information about running threads, call stacks for each thread, loaded classes, and objects in memory. They enable you to set a breakpoint and trace the method executions. A key for working with obfuscated applications is to use the regular interface (UI or programming API) to navigate to a feature of interest and then to rely on the debugger to learn about the class or classes that implement the feature. After the classes are identified, they can be decompiled and studied as described in Chapter 2. Working with debuggers is covered in detail in Chapter 9, "Cracking Code with Unorthodox Debuggers."

Quick Quiz

1. What are the means of protecting intellectual property in Java applications?
2. Which transformations provided by obfuscators offer the strongest protection?
3. For each of the potential problems listed in this chapter, which transformation(s) can cause it?
4. What is the most efficient way to study the obfuscated code?

In Brief

- Obfuscation can be the best way to protect the intellectual property in Java bytecode.
- Obfuscators perform some or all of the following transformations: stripping out debug information, name mangling, encoding strings, changing control flow, inserting corrupt code, eliminating unused code, and optimizing bytecode.
- Obfuscation introduces maintenance difficulties that can be minimized by configuring the obfuscator.
- Obfuscated code is still readable unless control flow obfuscation and string encoding is used.