Free Excerpt

JAVA Reflection IN ACTION

Ira R. Forman
Nate Forman

To order this book, visit
www.manning.com/forman
or your favorite bookseller

MANNING

1 Using Java's dynamic proxy

In this chapter

- How to use `java.lang.reflect.Proxy`
- Using proxy to implement decorators
- Chaining proxies
- Pitfalls of using `Proxy`

The dictionary [68] tells us that a proxy is an "agency, function, or office of a deputy who acts as a substitute for another." When this idea is applied to object-oriented programming, the result is an object, a *proxy*, that supports the interface of another object, its *target*, so that the proxy can substitute for the target for all practical purposes.

The keys to this arrangement are implementation and delegation. The proxy implements the same interface as the target so that it can be used in exactly the same way. The proxy delegates some or all of the calls that it receives to its target and thus acts as either an intermediary or a substitute. In its role as an intermediary, the proxy may add functionality either before or after the method is forwarded to the target. This gives the reflective programmer the capability to add behavior to objects. This chapter discusses this and other uses of proxies.

4.1 Working with proxies

The sequence diagram in figure 4.1 depicts the most common situation where the proxy instance receives a method call and forwards it to the target. Even this arrangement has a use; it hides the location of the target from the client. If you have used remote method invocation, you are familiar with proxies that are local substitutes for remote objects.

The Java reflection API contains a dynamic proxy-creation facility, `java.lang.reflect.Proxy`. This class is part of Java reflection because `Proxy` is Java's only way of approximating method invocation intercession. Let's dissect the previous phrase. **Intercession** is any reflective ability that modifies the behavior of a program by directly taking control of that behavior. Method invocation intercession is the ability to intercept method calls. The intercepting code can determine the behavior that results from the method call.

We say *approximating* because Java does not support reflective facilities for interceding on method calls. Therefore, we must use proxies as an approximation. Referring to figure 4.1, we see that proxies also allow the ability to pre- and post-process method calls. Let's examine the benefits achieved from doing this.

Programmers commonly discuss properties of classes. For example, a class that records its method calls is often referred to as a *tracing* class. A class that ensures that a failed operation does not leave an object in an intermediate state is often referred to as an *atomic* class.

The code that implements such properties is usually spread among the definitions of each of the methods of the class, almost always at the beginning and at the return points. The ability to intercede on method invocation permits the

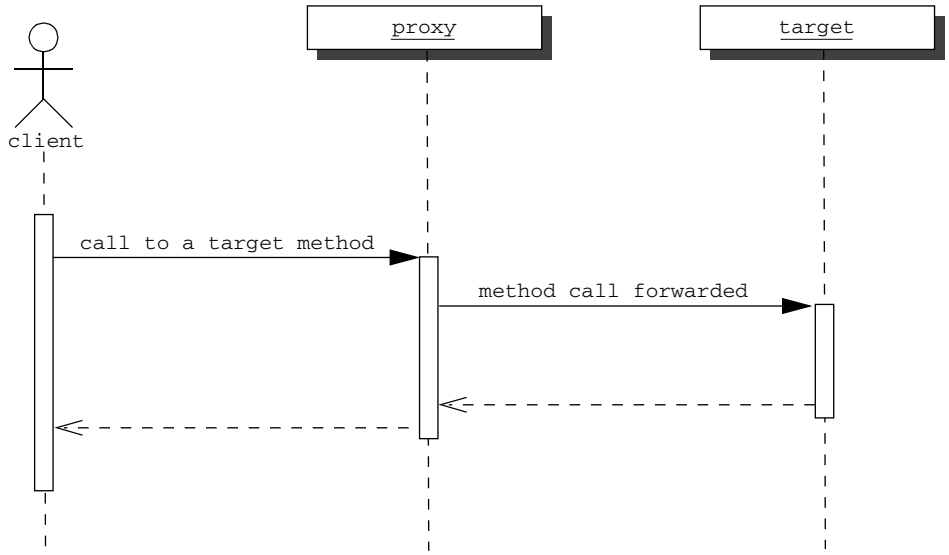

Figure 4.1 Sequence diagram for the typical use of a proxy. The proxy forwards received method calls to its target. The proxy may or may not do some pre- and post-processing.

programmer to gather this property-implementing code together in one place. This property can later combine with classes, yielding the desired effect.

The case for this combination of classes and properties is more real for software projects than you would think. A colleague once observed that when an object-oriented database is first brought into a programming shop, the number of classes doubles. The shop has added one property, *persistence*, to their application. Each class now requires a persistent and a nonpersistent version [18].

Developers get many key benefits from separating property-implementing code. One benefit of this separation is low maintenance cost for applications. Each such property can be modified by making a change in only one place in the code base. Another benefit of separating properties is improved reusability. The separated property can be used in many places in many applications.

There is also a compelling argument to present to management for such separation. Consider George's employer, Wildlife Components, which sells a class library of n classes. There are p properties that they wish their classes to have in all combinations. Both the number of classes and the number of properties grow as the company evolves to meet the increasing business demands. WCI faces the possibility of having to support a class library of at least $n2^p$ classes if they must write new classes to implement and combine properties in their original classes.

This additional maintenance is a serious enough concern to win management over. Isolating properties into reusable components and composing them later, as can be done with `Proxy`, yields a much smaller library of size $n+p$. This represents an enormous savings to WCI or any other company. This effect may not be as pronounced in other organizations, but it does exist.

Now that we have discussed the abstract benefits of `Proxy`, let's pay a visit to George and look at a simple example.

4.2 George's tracing problem

George has been assigned the task of creating tracing versions of several of the classes that he maintains. In a tracing class, each method records information about its entry and, after method execution, records information about its return. George's employer, WCI, wants tracing available for their classes because tracing helps with problem determination in deployed software.

Consider the following scenario. A customer calls WCI technical support with a defect report. Tech support asks the customer to turn tracing on in their software and follow the steps to reproduce the defect. Because tracing is turned on, the customer can then send WCI a file containing the path through the WCI source code.

This information solves many problems for the WCI technical team. It tells them a great deal about the state of the program during the failure. It also may prevent them from having to replicate their customer's environment and data.

While tracing is a useful feature, it is also very I/O intensive. Therefore, classes should be able to turn tracing on and off. However, including tracing code and guards to turn it on and off in each class bloats the classes and makes them slower because of the execution of the `if` statements. Due to these constraints, George decides to make tracing and nontracing versions of his classes.

One option George considers is subclassing each nontraced class and overriding each method with traces and `super` calls. He can then set up a process for either instantiating the traced or nontraced version depending upon some command-line argument. George quickly realizes that this option has the following shortcomings:

- *Tedium*—Executing this option is boring and mechanical. In fact, a computer program can be written to do this job.
- *Error-proneness*—George can easily misdeclare an override, misspelling the method name or including the wrong parameter list. He could also forget

or overlook a method. At best, he may have a compile error to warn him that his process broke. Otherwise, the class may not behave as expected.

- *Fragility*—If anyone in George's department adds, deletes, or changes the signature on a method in the superclass, the traced subclass breaks either by not building or by not tracing as expected.

Clearly, George is in need of a better solution. George needs to separate the concern of tracing from the rest of the source code and implement it in a separate module. George reasons that this can be done with a proxy, where the proxy traces the call before and after delegating the method invocation to the target. Although there will be one proxy object for every target, with the use of reflection, all of the proxies can be instances of one proxy class, which addresses the shortcomings raised previously. Before presenting George's solution, let's examine `java.lang.reflect.Proxy`.

4.3 Exploring Proxy

As stated previously, the two important tasks for any proxy are interface implementation and delegation. The Java `Proxy` class accomplishes implementation of interfaces by dynamically creating a class that implements a set of given interfaces. This dynamic class creation is accomplished with the static `getProxyClass` and `newProxyInstance` factory methods, shown in listing 4.1.

Listing 4.1 Partial declaration for `java.lang.reflect.Proxy`

```
public class Proxy implements java.io.Serializable {
   ...
   public static Class getProxyClass( ClassLoader loader,
                                      Class[] interfaces )
                    throws IllegalArgumentException ...

   public static Object newProxyInstance( ClassLoader loader,
                                          Class[] interfaces,
                                          InvocationHandler h )
                    throws IllegalArgumentException ...

   public static boolean isProxyClass( Class cl ) ...

   public static InvocationHandler getInvocationHandler( Object proxy )
                    throws IllegalArgumentException ...
}
```

Each class constructed by these factory methods is a public final subclass of `Proxy`, referred to as a **proxy class**. We refer to an instance of one of these dynamically constructed proxies as a **proxy instance**. We call the interfaces that the proxy class implements in this way **proxied interfaces**. A proxy instance is assignment-compatible with all of its proxied interfaces.

The `getProxyClass` method retrieves the proxy class specified by a class loader and an array of interfaces. If such a proxy class does not exist, it is dynamically constructed. Because each Java class object is associated with a class loader, in order to dynamically create a proxy class, `getProxyClass` must have a class loader parameter (the reason for this requirement is explained in chapter 6). The name of each proxy class begins with `$Proxy` followed by a number, which is the value of an index that is increased each time a proxy class is created.

All proxy classes have a constructor that takes an `InvocationHandler` parameter. `InvocationHandler` is an interface for objects that handle methods received by proxy instances through their proxied interfaces. We discuss invocation handlers further after we finish with the methods of `Proxy`. A combination of `getConstructor` and `newInstance` may be used to construct proxy instances, as in the following lines

```
Proxy cl = getProxyClass( SomeInterface.getClassLoader(),
                          Class[]{SomeInterface.class} );
Constructor cons = cl.getConstructor( new Class[]{InvocationHandler.class} );
Object proxy = cons.newInstance( new Object[] { new SomeIH( obj ) } );
```

where `SomeIH` is a class that implements `InvocationHandler`. Alternatively, this sequence can be accomplished with a single call to `newProxyInstance`:

```
Object proxy = Proxy.newProxyInstance( SomeInterface.getClassLoader(),
                                       Class[]{SomeInterface.class},
                                       new SomeIH( obj ) );
```

This call implicitly creates the proxy class, which can be retrieved with `getProxyClass`.

The static method `isProxyClass` is used to determine if a class object represents a proxy class. The line

```
Proxy.isProxyClass(obj.getClass())
```

may be use to determine if `obj` refers to a proxy instance. If `p` refers to a proxy instance,

```
Proxy.getInvocationHandler(p)
```

returns the `InvocationHandler` that was used to construct `p`.

4.3.1 Understanding invocation handlers

`Proxy` allows programmers to accomplish the delegation task by providing the `InvocationHandler` interface. Instances of `InvocationHandler`, also referred to as **invocation handlers**, are objects that handle each method call for a proxy instance. Invocation handlers are also responsible for holding any references to targets of the proxy instance. Listing 4.2 shows the `InvocationHandler` interface.

Listing 4.2 The `InvocationHandler` interface

```
public interface InvocationHandler {

      public Object invoke( Object proxy, Method method, Object[] args )
                                        throws Throwable;

}
```

A proxy instance forwards method calls to its invocation handler by calling `invoke`. The original arguments for the method call are passed to `invoke` as an object array. In addition, the proxy instance provides a reference to itself and to a `Method` object representing the invoked method.

Notice that the parameters passed to `invoke` are exactly the objects needed to forward a method call to another object reflectively. If `target` refers to the object being proxied, the lines

```
public Object invoke( Object proxy, Method method, Object[] args)
   throws Throwable
{
   return method.invoke(target, args);
}
```

implement an `invoke` method that passes every call transparently. More complex invoke methods may perform pre- and post-processing on the arguments. Note that invocation handlers may also forward to many targets or none at all.

Figure 4.1 depicts an abstraction of forwarding a method through a proxy. Figure 4.2 depicts that actual sequence of calls when the invocation handler is implemented as shown previously. For clarity, UML is often used to present the minimal relevant detail to convey understanding. With this idea in mind, our subsequent diagrams for proxy present the abstraction rather than the implementation detail.

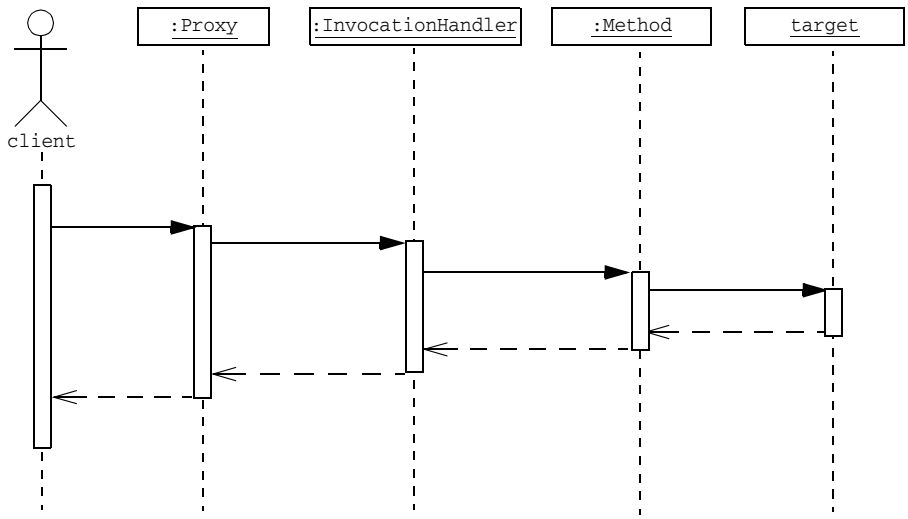

Figure 4.2 Sequence diagram illustrating the actual objects involved in forwarding a method when the invocation handler of the proxy uses the `invoke` method of `Method`.

4.3.2 Handling the methods of Object

A proxy instance is an object, and so it responds to the methods declared by `java.lang.Object`. This raises the issue of whether or not these methods should be handled by `invoke`. The issue is resolved as follows:

- `hashCode`, `equals`, and `toString` are dispatched to the invoke method in the same manner as any other proxied method.
- If a proxied interface extends `Cloneable`, then the invocation handler does intercede on the invocations to `clone`. However, unless the proxied interface makes `clone` public, it remains a protected method.
- If any proxied interface declares an override to `finalize`, then invocation handlers do intercede on calls to `finalize`.
- Method intercession does not take place for the other methods declared by `java.lang.Object`. Consequently, these methods behave as expected for any instance of `java.lang.Object`. In other words, a call to `wait` on a proxy instance waits on the proxy instance's lock, rather than being forwarded to an invocation handler.

The information in the last bullet is welcome because it means that an invocation handler cannot make a proxy instance lie about its class or interfere with multi-

threaded locking. Now that you understand the basics of `Proxy`, let's return to George's tracing problem.

4.4 Implementing a tracing proxy

George solves his tracing problem using `Proxy`. From his exploration of `Proxy`, George readily understands that his solution must have an invocation handler in which the `invoke` method forwards all method calls to the target. This forwarding is readily accomplished with the `invoke` method of `Method`. The next design decision involves the creation of the proxy and the invocation handler. George decides that all of his creation code can be located in the class written for the invocation handler. This is accomplished with a static method, `createProxy`. This static method is passed the target, which is examined introspectively to create an appropriate proxy and invocation handler. Listing 4.3 shows the invocation handler that George created. With this invocation handler, George can add tracing of any interface to an individual object. Let's examine the solution in detail.

Listing 4.3 An invocation handler for a proxy that traces calls

```
import java.lang.reflect.*;
import java.io.PrintWriter;

public class TracingIH implements InvocationHandler {

    public static Object createProxy( Object obj, PrintWriter out ) {
        return Proxy.newProxyInstance( obj.getClass().getClassLoader(),
                                       obj.getClass().getInterfaces(),
                                       new TracingIH( obj, out ) );
    }

    private Object target;
    private PrintWriter out;

    private TracingIH( Object obj, PrintWriter out ) {
        target = obj;
        this.out = out;
    }

    public Object invoke( Object proxy, Method method, Object[] args )
        throws Throwable
    {
        Object result = null;
        try {
            out.println( method.getName() + "(...) called" );
            result = method.invoke( target, args );
        } catch (InvocationTargetException e) {
            out.println( method.getName() + " throws " + e.getCause() );
            throw e.getCause();
```

```
            }
            out.println( method.getName() + " returns" );
            return result;
        }
    }
```

The implementation part of George's solution happens in the `createProxy` method. The static factory method `createProxy` wraps its argument in a proxy that performs tracing. First, `createProxy` examines its argument object for the direct interfaces that its class implements. It sends that array of interfaces to `Proxy.newProxyInstance`, which constructs a proxy class for those interfaces.[1] Next, a `TracingIH` is constructed with the argument as its target. Finally, `createProxy` constructs and returns a new proxy that forwards its calls to the `TracingIH`. This proxy implements all of the interfaces of the target object and is assignment-compatible with those types.

The delegation part of George's solution happens in the `invoke` method. The `invoke` method in listing 4.3 first records the method name to a `java.io.PrintWriter`. A more complete facility would also include the arguments, but we omit them for brevity. Then the `invoke` method forwards the call to the target and, subsequently, stores the return value. If an exception is thrown, the exception is recorded with the print writer; otherwise, the return value is recorded. Finally, the result of the call is returned.

When a proxied method is called on a proxy instance, control first passes to the `invoke` method with the following arguments:

- *`proxy`*—The proxy instance on which the method was invoked. `TracingIH` happens to make no use of this parameter.
- *`method`*—A `Method` object for the invoked method.
- *`args`*—An array of objects containing the values of the arguments passed in the method invocation on the proxy instance. `args` is `null` if the `method`

[1] The `getInterfaces` method returns only the direct interfaces of a class. As George has written the invocation handler, only methods declared by direct interfaces are traced. In chapter 8, we present a method, `Mopex.getAllInterfaces`, that finds all of the interfaces implemented by a class. What about methods that are not implemented in an interface? George might be asked to supply a tool that finds those methods and puts them in an interface. Reflection can help here, too, but you will have to wait until chapter 7 to read how.

takes no arguments. Arguments of primitive types are wrapped in instances of the appropriate primitive wrapper class; for example, `java.lang.Integer` wraps an `int`.

The declared return type of `invoke` is `Object`. The value returned by `invoke` is subject to the following rules:

- If the called method has declared the return type void, the value returned by `invoke` does not matter. Returning `null` is the simplest option.
- If the declared return type of the interface method is a primitive type, the value returned by `invoke` must be an instance of the corresponding primitive wrapper class. Returning `null` in this case causes a `NullPointerException` to be thrown.
- If the value returned by `invoke` is not compatible with the interface method's declared return type, a `ClassCastException` is thrown by the method invocation on the proxy instance.

The exception `UndeclaredThrowableException` may be thrown by the execution of the `invoke` method. `UndeclaredThrowableException` wraps non-runtime exceptions that are not declared by the interface for the method being called. The cause of the wrapped exception may be accessed with `getCause`. This wrapping of an exception may seem odd, but it is necessary when you consider the difficulty of programming invocation handlers that are limited to throwing just those exceptions known at the origin of the call.

To fully understand the class `TracingIH` in listing 4.3, it is best to understand how a using application is changed by the execution of the statement

```
Dog proxyForRover = (Dog) TracingIH.createProxy( rover );
```

where `Dog` is a Java interface and `rover` contains an instance of a class `DogImpl` that implements that interface. Note that the proxy facility ensures that the proxy instance returned by `createProxy` can be cast to `Dog`. Figure 4.3 presents a diagram that shows all of the objects and classes that are relevant to the previous line of code. The objects created by that line of code are in the gray area.

This invocation handler in listing 4.3 provides the module that George wants. Instead of having to change source code, he can wrap objects with proxies and have the users of the objects reference the proxies. This technique avoids all of the shortcomings of the process George would have to follow without `Proxy`.

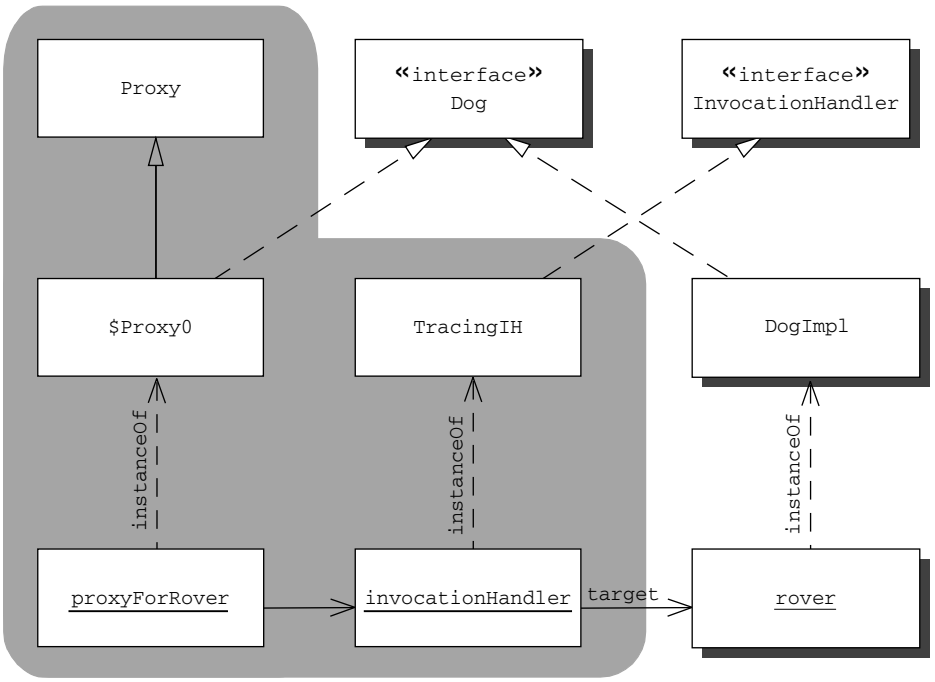

Figure 4.3 A class diagram illustrating the execution of the `createProxy` factory method from listing 4.3.

4.5 A note on factories

As mentioned earlier, the tracing invocation handler of listing 4.3 is missing a test to turn tracing on and off dynamically. Instead, the application uses either traced or nontraced versions of its classes. This is accomplished by applying the Abstract Factory pattern for construction of the potentially traced objects. That is, a class is declared that contains a method for creating new instances of `Dog`. This method chooses whether to create instances of the `Dog` class that traces or instances of the one that does not trace. An example factory for implementations of the `Dog` interface is shown in listing 4.4.

Listing 4.4 A factory that chooses between traced and untraced versions of a class

```
import java.lang.reflect.*;
import java.io.PrintWriter;

public class DogFactory {
```

```
    private Class dogClass;
    private boolean traceIsOn = false;

    public DogFactory( String className, boolean trace ) {
        try {
            dogClass = Class.forName( className );
        } catch (ClassNotFoundException e){
            throw new RuntimeException(e); // or whatever is appropriate
        }
        traceIsOn = trace;
    }

    public Dog newInstance( String name, int size ) {
        try {
            Dog d = (Dog)dogClass.newInstance();
            d.initialize(name,size);
            if ( traceIsOn ) {
                d = (Dog)TracingIH.createProxy( d,
                                          new PrintWriter(System.out) );
            }
            return d;
        } catch(InstantiationException e){
            throw new RuntimeException(e); // or whatever is appropriate
        } catch(IllegalAccessException e){
            throw new RuntimeException(e); // or whatever is appropriate
        }
    }
}
```

Notice that the factory method `newInstance` is enhanced reflectively by using the class object to create a new instance the same way as the factory method in the previous chapter. The lines

```
if (traceIsOn) {
   d = (Dog) TracingIH.createProxy(d, new PrintWriter(System.out));
}
```

assure that each `Dog` is wrapped in a tracing proxy when required. This puts the tests for tracing at construction time rather than during execution of the methods of `Dog`.

The factory method also conforms to design recommendations presented in section 3.4.2. The `newInstance` method constructs instances using the `newInstance` method of `Class`. After construction, the new `Dog` is made ready for use with a call to `initialize`.

4.6 Chaining proxies

One of the strengths of using proxies is that they can be arranged in a chain, with each proxy but the last having another proxy as its target. The last target in the chain is the real target object. When done properly, this chaining has the effect of composing the properties implemented by each proxy.

4.6.1 Structuring invocation handlers for chaining

Ensuring that proxies can be chained requires careful design. For example, the invocation handler for tracing is programmed with the assumption that its target is the real target and not another proxy. If the target is another proxy, the invocation handler may not perform the correct operation. To remedy this problem, we present `InvocationHandlerBase`, an abstract class for deriving invocation handlers for chainable proxies. The source code for `InvocationHandlerBase` is shown in listing 4.5.

Listing 4.5 InvocationHandlerBase

```
import java.lang.reflect.*;
import mopex.*;

public abstract class InvocationHandlerBase implements InvocationHandler {

    protected Object nextTarget;
    protected Object realTarget = null;

    InvocationHandlerBase( Object target ) {
        nextTarget = target;
        if ( nextTarget != null ) {
            realTarget = findRealTarget(nextTarget);
            if (realTarget == null)
                throw new RuntimeException("findRealTarget failure");
        }
    }

    protected final Object getRealTarget() { return realTarget; }

    protected static final Object findRealTarget( Object t ) {
        if ( !Proxy.isProxyClass(t.getClass()) )
            return t;
        InvocationHandler ih = Proxy.getInvocationHandler(t);
        if ( InvocationHandlerBase.class.isInstance( ih ) ) {
            return ((InvocationHandlerBase)ih).getRealTarget();
        } else {
            try {
                Field f = Mopex.findField( ih.getClass(), "target" );
                if ( Object.class.isAssignableFrom(f.getType()) &&
                     !f.getType().isArray() ) {
```

```
                    f.setAccessible(true); // suppress access checks
                    Object innerTarget = f.get(ih);
                    return findRealTarget(innerTarget);
                }
                return null;
            } catch (NoSuchFieldException e){
                return null;
            } catch (SecurityException e){
                return null;
            } catch (IllegalAccessException e){
                return null;
            } // IllegalArgumentException cannot be raised
        }
    }
}
```

The service provided by `InvocationHandlerBase` is the recursive search `findRealTarget` that traverses the chain of proxy instances and invocation handlers to find the real target at the end of the chain. If each invocation handler in the chain extends `InvocationHandlerBase`, the traversal is simply accomplished with calls to `getRealTarget`, because `findRealTarget` is used in the constructor to initially set `realTarget`.

However, it is rather inflexible to assume that all of the invocation handlers encountered will extend `InvocationHandlerBase`. For invocation handlers that do not extend `InvocationHandlerBase`, we attempt to find a target using reflection. The `findRealTarget` method searches the target proxy instance's invocation handler for an `Object` field named `target`. The search for the `target` field is accomplished using `Mopex.findField`, defined in listing 4.6. If that field exists and has a non-array type assignable to `Object`, it is assumed that the field contains the next link in the chain of proxies.

Listing 4.6 The `findField` method in `Mopex`

```
public static Field findField( Class cls, String name )
                                throws NoSuchFieldException {
    if ( cls != null ) {
        try {
            return cls.getDeclaredField( name );
        } catch(NoSuchFieldException e){
            return findField( cls.getSuperclass(), name );
        }
    } else {
        throw new NoSuchFieldException();
    }
}
```

The interface in the Java Reflection API for querying a class object for its members is not always ideal. For example,

```
X.class.getDeclaredField("foo")
```

throws a `NoSuchFieldException` if the sought field `foo` is declared by a superclass of the target `X`. `Mopex` contains `findField` to make queries for fields more convenient. It recursively searches up the inheritance hierarchy and returns the first field with the specified name. This search furthers our goal of chaining invocation handlers that do not extend `InvocationHandlerBase` with those that do. Let's use it.

4.6.2 Implementing a synchronized proxy

To illustrate the concept of proxy chaining, we need another kind of proxy to chain with the tracing proxy. In this section, we present a proxy for making an object synchronized. This proxy has the effect of using the `synchronized` modifier on a class declaration if Java allowed such a combination. Listing 4.7 presents an invocation handler for synchronized access to its target object. All method forwarding occurs inside a `synchronized` statement.

Listing 4.7 An invocation handler for synchronized access

```
import java.lang.reflect.*;

public class SynchronizedIH extends InvocationHandlerBase {

    public static Object createProxy( Object obj ) {
        return Proxy.newProxyInstance( obj.getClass().getClassLoader(),
                                       obj.getClass().getInterfaces(),
                                       new SynchronizedIH( obj ) );
    }

    private SynchronizedIH( Object obj ) { super( obj ); }

    public Object invoke( Object proxy, Method method, Object[] args )
        throws Throwable
    {
        Object result = null;
        synchronized ( this.getRealTarget() ) {
            result = method.invoke( nextTarget, args );

        }
        return result;
    }
}
```

The lock acquired by the synchronized statement in listing 4.7 is the one belonging to the real target, which is the better design decision. The alternative decision is acquiring the lock associated with the proxy instance. This alternative is likely the wrong design decision. For example, if there were multiple proxy instances for a single target, each proxy instance would be acquiring a different lock. For this reason, it is vital to discover the real target.

4.6.3 Chaining the two proxies

As mentioned previously, chaining is one of the more elegant properties of `Proxy`. That is, by using a synchronizing proxy in front of a tracing proxy, we achieve the effect of an object that both synchronizes and traces. As we did earlier, suppose `Dog` is a Java interface and `DogImpl` is an implementation of that interface. The statement

```
Dog rover = (Dog)SynchronizedIH.createProxy(
                  TracingIH.createProxy( new DogImpl(),
                                         new PrintWriter(System.out) ) );
```

constructs a synchronized proxy instance for a tracing proxy instance for a `Dog` object. For all practical purposes, this is a `Dog` object that synchronizes and traces. This is illustrated in figure 4.4, which shows that a call is passed from one proxy to the next until the call reaches the target.

When you chain proxies, the order usually makes a difference. That is, there is a difference between a synchronized tracing object and tracing synchronized object. The difference is whether or not the synchronization applies to the printing of the trace. In any multithreaded application, this is an important nuance because if the tracing is not conducted inside the synchronization, the trace output of two threads might be mixed so as to appear that the synchronization were not working. That is, the trace would not reflect the true behavior of the application, which would be a poor outcome of the chaining of the proxies.

The chaining of proxies is one way to address the problem of exponential growth in the size of the class hierarchy when you need to mix properties and classes. More concretely, the above proxy constructions are much more convenient than maintaining a synchronized version, a tracing version, and a synchronized tracing version of each class that requires these properties. Chapter 7 discusses another way to address this problem.

Irrespective of the approach taken, the importance of the problem and the fundamental reliance of the various solutions on reflection cannot be stressed

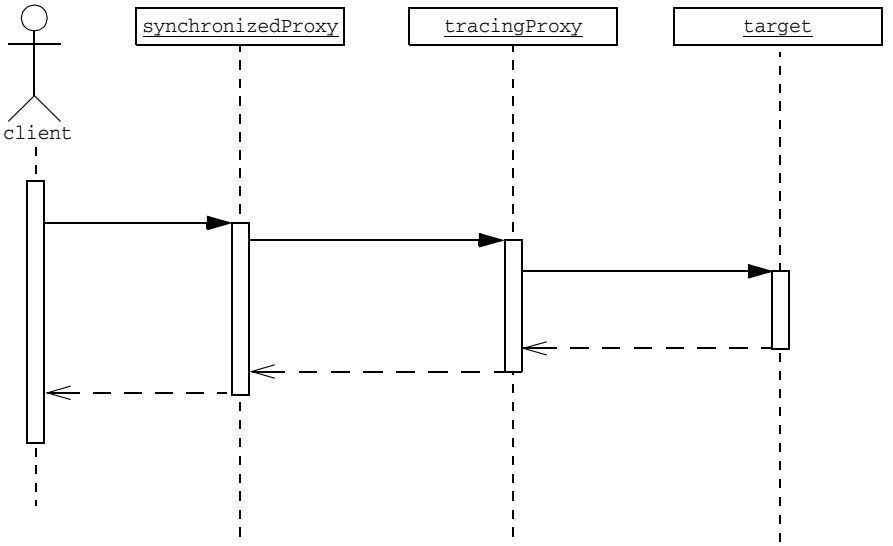

Figure 4.4 Sequence diagram illustrating the chaining of the synchronized proxy and the tracing proxy

enough. In order to be the most flexible, both adaptive and reusable, software must be able to examine itself and its environment (introspection), change itself (intercession), and combine with that environment.

JAVA

JAVA Reflection IN ACTION

Ira R. Forman and Nate Forman

Imagine programs that are able to adapt—with no intervention by you—to changes in their environment. With Java reflection you can create just such programs. Reflection is the ability of a running program to look at itself and its environment, and to change what it does depending on what it finds. This inbuilt feature of the Java language lets you sidestep a significant source of your maintenance woes: the "hard-coding" between your core application and its various components.

Java Reflection in Action shows you that reflection isn't hard to do. It starts from the basics and carefully builds a complete understanding of the subject. It introduces you to the reflective way of thinking. And it tackles useful and common development tasks, in each case showing you the best-practice reflective solutions that replace the usual "hard-coded" ones. You will learn the right way to use reflection to build flexible applications so you can nimbly respond to your customers' future needs. Master reflection and you'll add a versatile and powerful tool to your developer's toolbox.

What's Inside

- Practical introduction to reflective programming
- Examples from diverse areas of software engineering
- How to design flexible applications
- When to use reflection—and when not to
- Performance analysis

Dr. Ira Forman is a computer scientist at IBM. He has worked on reflection since the early 1990s when he developed IBM's SOM Metaclass Framework. **Nate Forman** works for Ticom Geomatics where he uses reflection to solve day-to-day problems. Ira and Nate are father and son. They both live in Austin, Texas.

"Even occasional users [of reflection] will immediately adopt the book's patterns and idioms to solve common problems."

—DOUG LEA
SUNY Oswego, author of
CONCURRENT PROGRAMMING IN JAVA

"... guide[s] you through one compelling example after another, each one illustrating reflection's power while avoiding its pitfalls."

—JOHN VLISSIDES
IBM, coauthor of
DESIGN PATTERNS

www.manning.com/forman

AUTHOR ONLINE Authors respond to reader questions

Ebook edition available

To order this book, visit
www.manning.com/forman
or your favorite bookseller

MANNING $44.95 US/$67.95 Canada

9 781932 394184 54495
ISBN 1-932394-18-4