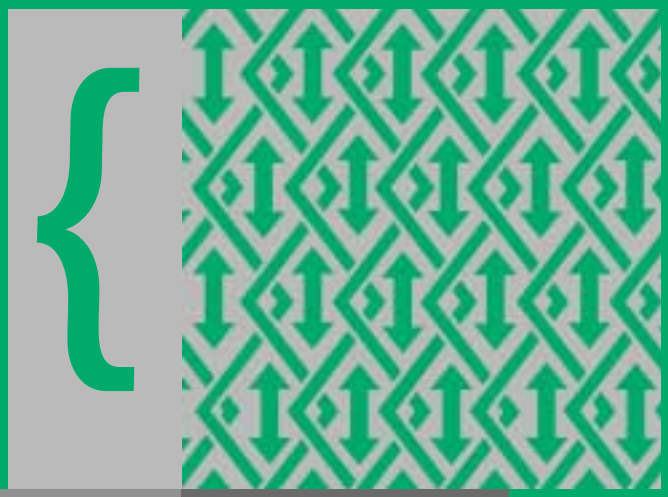
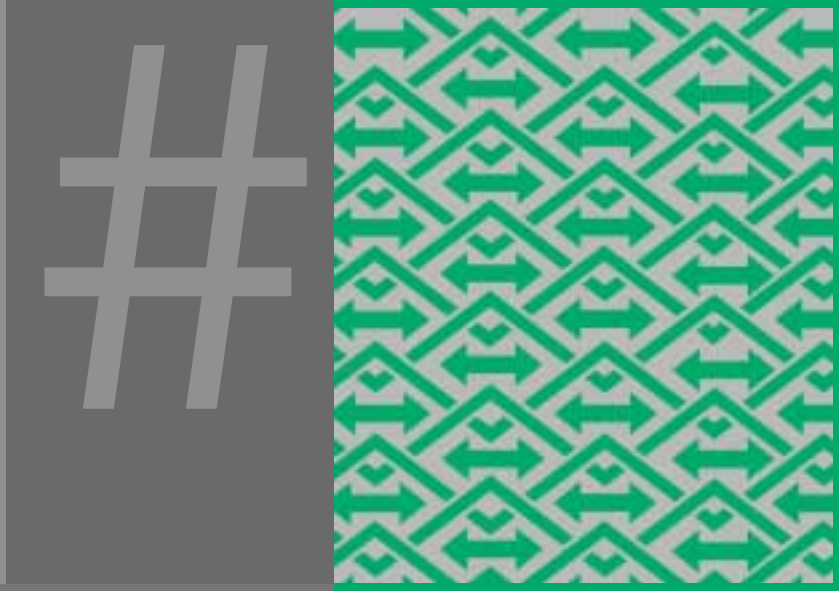
;login:

THE MAGAZINE OF USENIX & SAGE
October 2000 • volume 25 • number 6



inside:

PROGRAMMING
Java Performance



USENIX & SAGE
The Advanced Computing Systems Association &
The System Administrators Guild

java performance

Proxy Classes

by Glen McCluskey

Glen McCluskey is a consultant with 15 years of experience and has focused on programming languages since 1988. He specializes in Java and C++ performance, testing, and technical documentation areas.

<glenm@glenmccl.com>

Proxy classes are a new Java feature, one that allows you to create dynamic classes at runtime. These classes can be used to add a level of indirection to interface method calls, so that calls can be trapped and processed as desired. Using this feature, it's possible to do call tracing for performance purposes, add functionality to existing interfaces, and so on.

We'll start by reviewing some basics of Java interface programming.

Interface Programming

In Java programming, an interface is an abstract type, typically a set of methods that specifies a contract for the type. For example, a List interface might look like this:

```
interface List {
    void add(Object o);
    void del(int i);
    Object get(int i);
    int size();
}
```

In this example, methods are specified for adding, deleting, and retrieving elements from a list, and for obtaining the list size.

An interface itself contains no implementation. That is, you do not say:

```
interface List {
    int size() {...}
}
```

Rather, you implement an interface via a Java class, like this:

```
class LinkedList implements List {
    int size() {...}
}
```

The interface itself does not specify any details of how a list is to be represented (such as with a vector or linked list); this decision is left to an implementing class. Thus the interface specifies a contract, and the class implements the contract.

You can program in terms of interface types, for example:

```
List lst = new LinkedList();
```

and then say things like:

```
int sz = lst.size();
```

without worrying about exactly which class (ArrayList, LinkedList, etc.) is used to implement the List interface, or the details of list representation.

Proxies

java.lang.reflect.Proxy is a class that allows you to create proxy classes at runtime, classes that implement specified interfaces. Once you have an instance of a proxy class, you can use it to provide a level of indirection when you make interface method calls. Calls to methods of interfaces implemented by the proxy class are dispatched to a single invoke() method in an invocation handler. At this point, you can intercept the calls, for example to provide method logging for performance analysis purposes.

In other words, you create a proxy class and instance by specifying a set of interfaces, and an invocation handler to call when any of the methods in the interfaces are invoked. You then use this instance as a proxy for some other object; the proxy sits on top of the other object and allows you to trap method calls that you would normally make on the underlying object.

When a method call is trapped, you are supplied with the signature of the method and a list of all the method arguments. You can then invoke the method on the real underlying object, or take various other actions as desired.

An Example

Here's an example of setting up a proxy:

```
import java.lang.reflect.*;
import java.util.*;
public class ProxyDemo implements InvocationHandler {

    // the underlying object that the proxy is based on private Object proxyobj;
    // create a ProxyDemo object (an invocation handler)
    private ProxyDemo(Object obj) {
        proxyobj = obj;
    }

    // create a proxy object
    public static Object makeProxy(Object obj) {
        Class cls = obj.getClass();
            return Proxy.newProxyInstance(cls.getClassLoader(),
            cls.getInterfaces(),
        new ProxyDemo(obj));
    }

    // handle interface method calls
    public Object invoke(Object proxy, Method meth, Object args[])
        throws Throwable {

        // print the method name and arguments
        System.out.println(meth);
    if (args != null) {
        System.out.print(" args: ");
        for (int i = 0; i < args.length; i++)
            System.out.print(args[i] + " ");
        System.out.println();
        }

        // invoke the method on the original object
        return meth.invoke(proxyobj, args);
    }
    public static void main(String args[]) {

        // create a proxy
        List lref = (List)makeProxy(new ArrayList());

        // create a list of Integer objects
        lref.add(new Integer(57));
        lref.add(new Integer(37));
        lref.add(new Integer(47));

        // sort the list in ascending order
        int size = lref.size();
        for (int i = 0; i < size - 1; i++) {
            for (int j = i + 1; j < size; j++) {
```

```
                Comparable obj1 =
                    (Comparable)lref.get(i);
                Comparable obj2 =
                    (Comparable)lref.get(j);
                if (obj1.compareTo(obj2) > 0) {
                    Object t = lref.get(i);
                        lref.set(i, lref.get(j));
                    lref.set(j, t);
                }
            }
        }
    }
}
```

In this example, ProxyDemo is a class that implements the InvocationHandler interface, that is, defines a method invoke() that is called when any of the methods in the interfaces implemented by the proxy are called. The ProxyDemo constructor records the underlying object that the proxy is created for.

ProxyDemo.makeProxy() is a static method that is used to create a proxy class instance, based on the set of interfaces implemented by a passed-in object.

The proxy in this example is used to track operations on an ArrayList object, an object of a class that implements the java.util.List interface. A proxy is created for an ArrayList object, and then List interface methods are called through the proxy to add elements to the list and sort the list.

When methods are called through the proxy, what actually happens is that the invoke() method in the specified invocation handler is called. It displays the method name and arguments, and then calls the actual method for the underlying object. So in this example, the proxy is used to log method calls, and then the "real" method is called. The first few lines of output from running the program are these:

```
public abstract boolean java.util.List.add(java.lang.Object)
    args: 57
public abstract boolean java.util.List.add(java.lang.Object)
    args: 37
public abstract boolean java.util.List.add(java.lang.Object)
    args: 47
public abstract int java.util.List.size()
public abstract java.lang.Object java.util.List.get(int)
    args: 0
```

The example uses java.lang.Class and java.lang.reflect.Method. These classes are part of what is called "reflection," the ability to examine and manipulate types at runtime. For example, a Method object represents a particular method of a class, and you can invoke the method by specifying an object and a set of arguments, so that:

```
meth.invoke(obj, args);
```

is equivalent to:

```
obj.meth(arg1, arg2, ...);
```

Other Uses For Proxies

We illustrated method logging above, using proxies to display each interface method just before it's invoked. You can also use proxies in other ways, for example, to add functionality to an existing interface, or to modify method arguments before calling a method.

You can also use proxies in other ways, for example, to add functionality to an existing interface, or to modify method arguments before calling a method.