

[Home](#)[Newsletter](#)[Java Specialist Club](#)[Java Training](#)[Conference Venue Hire](#)[Java Resources](#)[Contact](#)

The Java Specialists' Newsletter

➤ Issue 168 ➤ 2009-01-15 ➤ Category: [Language](#) ➤ Java version: Java 5+

0

[Subscribe Free](#) [RSS Feed](#) [Print](#)

The Delegator

by Dr. Heinz M. Kabutz

Abstract:

In this newsletter we show the reflection plumbing needed for writing a socket monitor that sniffs all the bytes being sent or received over all the Java sockets. The Delegator is used to invoke corresponding methods through some elegant guesswork.

Welcome to the 168th issue of **The Java(tm) Specialists' Newsletter** and welcome to 2009! On Christmas Day (25th December), I was playing tennis with Helene and with a hard whack broke a string, the first in 30 years of playing tennis! The tennis lessons are obviously having some effect on my game. It took only 15 days to have the string repaired in Crete. I need to either save up for a stringing machine or buy a spare raquet for next time. According to Coach Kathy, I can expect this to happen every few months.

Upcoming Java Master Courses:

Duesseldorf, Germany, Aug 22
Chania, Crete, Sep 6
Cape Town, South Africa, Sep 12

In-house courses if these dates or locations do not suit you. Note that the course in Crete may also be attended remotely via webinar.

The Delegator

A few years ago, I showed how to count bytes sent over RMI sockets. This mechanism helped me tune the performance for an ERP system in 2002, though I'm not sure it still works with modern versions of Java.

This time, however, I wanted to build a general mechanism for listening to bytes flowing over sockets in Java, not just RMI socket. This should be done with minimal code impact. Subclassing Socket would thus probably not be an option.

It turns out that the implementation to do this is possible, but non-trivial. I have broken the problem up into two digestible chunks. This first part shows the reflection plumbing that we will need to get this magic working. The second part will come in our next newsletter and will show how to use that magic to sniff sockets and to then integrate the resulting measurements with MXBeans so we can view the statistics in JConsole.

Socket uses the strategy pattern for the actual communication and we are able to specify our own implementation. Thus all we would need to do is write our own strategy that counts the bytes flowing backwards and forwards. Unfortunately the standard strategy implementations are package access in the java.net.* package, so we are not able to use them directly. We certainly cannot subclass them, but we could call the methods with reflection. However, because the classes themselves are package access, we need to find the declared constructor, set that to be accessible and then instantiate it. [Note: In the original version of this newsletter, I offered a more complicated way to instantiate the object.]

```
// then we load the class, e.g. "java.net.SocketImpl"
Class implCl = Class.forName(delegateClass);
// we find the constructor
Constructor delegateConstructor =
    implCl.getDeclaredConstructor();
delegateConstructor.setAccessible(true);
// we have constructed the package access class
this.delegate = delegateConstructor.newInstance();
```

However, even if we have constructed the object, all of the methods inside the superclass, in our case "java.net.SocketImpl", are protected, so we cannot call them directly, even if our class is a subclass of SocketImpl.

So, I ventured out to write a Delegator class, which allows me to automatically delegate the method call to the correct matching method. So all we have to do in our own SocketImpl is to write:

```
public void close() throws IOException {
    delegator.invoke();
}
```

Newsletter Links

- [Book Review](#)
- [Concurrency](#)
- [Exceptions](#)
- [GUI](#)
- [Inspirational](#)
- [Language](#)
- [Performance](#)
- [Software Engineering](#)
- [Tips and Tricks](#)

Java Courses

- [Java Master](#)
- [Java Foundation](#)
- [Java 5 Tiger](#)
- [Design Patterns](#)

[find out more](#)

What is the Java Specialists Club?

Venue for Hire

Looking for a super conference room for your next company event?

Crete is your perfect destination.

[Click here for more details](#)

This even works when the method has parameters, such as:

```
public void listen(int backlog) throws IOException {
    delegator.invoke(backlog);
}
```

The invoke() method discovers what to invoke by finding the method name using the stack trace and then searching for a matching method based on the parameters of the arguments.

We find the method name like this:

```
private String extractMethodName() {
    Throwable t = new Throwable();
    String methodName = t.getStackTrace()[2].getMethodName();
    return methodName;
}
```

We match the parameters to the method as follows:

```
private Method findMethod(String methodName, Object[] args)
    throws NoSuchMethodException {
    Class<?> clazz = superclass;
    if (args.length == 0) {
        return clazz.getDeclaredMethod(methodName);
    }
    Method match = null;
next:
    for (Method method : clazz.getDeclaredMethods()) {
        if (method.getName().equals(methodName)) {
            Class<?>[] classes = method.getParameterTypes();
            if (classes.length == args.length) {
                for (int i = 0; i < classes.length; i++) {
                    Class<?> argType = classes[i];
                    argType = convertPrimitiveClass(argType);
                    if (!argType.isInstance(args[i])) continue next;
                }
                if (match == null) {
                    match = method;
                } else {
                    throw new DelegationException(
                        "Duplicate matches");
                }
            }
        }
    }
    if (match != null) {
        return match;
    }
    throw new DelegationException(
        "Could not find method: " + methodName);
}
```

The convertPrimitiveClass() method is needed because of autoboxing. When we pass a primitive to an Object.. varargs list, it gets converted to its wrapper class. The convertPrimitiveClass() method converts the primitive class of the delegate object parameter to its matching wrapper class, so we can check that the parameter is an instance. (Maybe read that sentence again until it makes sense.)

There are some limitations to my approach. It cannot handle null parameters. It cannot handle methods where the parameters are subclasses of each other. However, in these cases it will throw an exception, rather than do the wrong thing.

If a method call is not delegated correctly, you can specify the method name and parameter types explicitly, like this:

```
public void connect(InetAddress address, int port)
    throws IOException {
    delegator
        .delegateTo("connect", InetAddress.class, int.class)
        .invoke(address, port);
}
```

As you will see in the code, it does this by creating an instance of an inner class that then stores the method name and parameter types.

Return Types

The invoke() method always returns the correct type, using generics. Thus we do not need to type cast when we write:

```
public FileDescriptor getFileDescriptor() {
    return delegator.invoke();
}
```

There are two exceptions where we need to do some more work. The first is with primitives and the second is when we use the result of the invoke() method as a method parameter directly. We solve the primitives using autoboxing:

```
public int getPort() {
    Integer result = delegator.invoke();
    return result;
}
```

The second issue is also easy to solve, by writing the result of invoke() first to a local variable:

```

public InputStream getInputStream() throws IOException {
    InputStream real = delegator.invoke();
    return new DebuggingInputStream(real, monitor);
}

```

Fixing Broken Encapsulation

The programmers who wrote the java.net package took some short-cuts by modifying fields directly from other classes, rather than calling methods. It is thus not enough to simply delegate the method calls. We have to go one step further. Before we call any method, we need to copy all the fields in our shared superclass to our delegated object. After the method is called, we need to copy the fields back to our object. Here is how we do that:

```

writeFields(superclass, source, delegate);
method.setAccessible(true);
Object result = method.invoke(delegate, args);
writeFields(superclass, delegate, source);
return result;

```

The writeFields method is quite simple:

```

private void writeFields(Class clazz, Object from, Object to)
    throws Exception {
    for (Field field : clazz.getDeclaredFields()) {
        field.setAccessible(true);
        field.set(to, field.get(from));
    }
}

```

Putting it all together

When we put all these elements together, we get the Delegator class, which we can use for delegating method calls easily to another class. Here is the full monty;

```

package util;

public class DelegationException extends RuntimeException {
    public DelegationException(String message) {
        super(message);
    }

    public DelegationException(String message, Throwable cause) {
        super(message, cause);
    }

    public DelegationException(Throwable cause) {
        super(cause);
    }
}

package util;

import java.lang.reflect.*;

public class Delegator {
    private final Object source;
    private final Object delegate;
    private final Class superclass;

    public Delegator(Object source, Class superclass,
                    Object delegate) {
        this.source = source;
        this.superclass = superclass;
        this.delegate = delegate;
    }

    public Delegator(Object source, Class superclass,
                    String delegateClassName) {
        try {
            this.source = source;
            this.superclass = superclass;
            Class implCl = Class.forName(delegateClassName);
            Constructor delegateConstructor =
                implCl.getDeclaredConstructor();
            delegateConstructor.setAccessible(true);
            this.delegate = delegateConstructor.newInstance();
        } catch (RuntimeException e) {
            throw e;
        } catch (Exception e) {
            throw new DelegationException(
                "Could not make delegate object", e);
        }
    }

    public final <T> T invoke(Object... args) {
        try {
            String methodName = extractMethodName();
            Method method = findMethod(methodName, args);
            @SuppressWarnings("unchecked")
            T t = (T) invoke0(method, args);
            return t;
        } catch (NoSuchMethodException e) {
            throw new DelegationException(e);
        }
    }

    private Object invoke0(Method method, Object[] args) {
        try {
            writeFields(superclass, source, delegate);
            method.setAccessible(true);

```

```

        Object result = method.invoke(delegate, args);
        writeFields(superclass, delegate, source);
        return result;
    } catch (RuntimeException e) {
        throw e;
    } catch (InvocationTargetException e) {
        throw new DelegationException(e.getCause());
    } catch (Exception e) {
        throw new DelegationException(e);
    }
}

private void writeFields(Class clazz, Object from, Object to)
    throws Exception {
    for (Field field : clazz.getDeclaredFields()) {
        field.setAccessible(true);
        field.set(to, field.get(from));
    }
}

private String extractMethodName() {
    Throwable t = new Throwable();
    String methodName = t.getStackTrace()[2].getMethodName();
    return methodName;
}

private Method findMethod(String methodName, Object[] args)
    throws NoSuchMethodException {
    Class<?> clazz = superclass;
    if (args.length == 0) {
        return clazz.getDeclaredMethod(methodName);
    }
    Method match = null;
next:
    for (Method method : clazz.getDeclaredMethods()) {
        if (method.getName().equals(methodName)) {
            Class<?>[] classes = method.getParameterTypes();
            if (classes.length == args.length) {
                for (int i = 0; i < classes.length; i++) {
                    Class<?> argType = classes[i];
                    argType = convertPrimitiveClass(argType);
                    if (!argType.isInstance(args[i])) continue next;
                }
                if (match == null) {
                    match = method;
                } else {
                    throw new DelegationException(
                        "Duplicate matches");
                }
            }
        }
    }
    if (match != null) {
        return match;
    }
    throw new DelegationException(
        "Could not find method: " + methodName);
}

private Class<?> convertPrimitiveClass(Class<?> primitive) {
    if (primitive.isPrimitive()) {
        if (primitive == int.class) {
            return Integer.class;
        }
        if (primitive == boolean.class) {
            return Boolean.class;
        }
        if (primitive == float.class) {
            return Float.class;
        }
        if (primitive == long.class) {
            return Long.class;
        }
        if (primitive == double.class) {
            return Double.class;
        }
        if (primitive == short.class) {
            return Short.class;
        }
        if (primitive == byte.class) {
            return Byte.class;
        }
        if (primitive == char.class) {
            return Character.class;
        }
    }
    return primitive;
}

public DelegatorMethodFinder delegateTo(String methodName,
    Class<?>... parameters) {
    return new DelegatorMethodFinder(methodName, parameters);
}

public class DelegatorMethodFinder {
    private final Method method;

    public DelegatorMethodFinder(String methodName,
        Class<?>... parameterTypes) {
        try {
            method = superclass.getDeclaredMethod(
                methodName, parameterTypes
            );
        } catch (RuntimeException e) {
            throw e;
        } catch (Exception e) {
            throw new DelegationException(e);
        }
    }
}

```

```
public <T> T invoke(Object... parameters) {  
    @SuppressWarnings("unchecked")  
    T t = (T) Delegator.this.invoke0(method, parameters);  
    return t;  
}  
}
```

In our next newsletter we will combine this with Sockets to count how many bytes are transferred and then write an MBean to show this information in JConsole.

Kind regards

Heinz

▶ Language Articles	▶ Related Java Course	▶ Discuss at The Java Specialist Club
-------------------------------------	---------------------------------------	---

© 2010 Heinz Kabutz - All Rights Reserved

[Sitemap](#)

[seo web design](#) Catch22 Marketing

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners. JavaSpecialists.eu is not connected to Oracle, Inc. and is not sponsored by Oracle, Inc.