

Sample Chapter

JAVA Reflection IN ACTION

Ira R. Forman
Nate Forman



 MANNING

Java Reflection in Action

by Ira R. Forman

and

Nate Forman

Sample Chapter 5

Copyright 2004 Manning Publications

contents

- Chapter 1 ■ A few basics
- Chapter 2 ■ Accessing fields reflectively
- Chapter 3 ■ Dynamic loading and reflective construction
- Chapter 4 ■ Using Java’s dynamic proxy
- Chapter 5 ■ Call stack introspection
- Chapter 6 ■ Using the class loader
- Chapter 7 ■ Reflective code generation
- Chapter 8 ■ Design patterns
- Chapter 9 ■ Evaluating performance
- Chapter 10 ■ Reflecting on the future

- Appendix A ■ Reflection and metaobject protocols
- Appendix B ■ Handling compilation errors in the “Hello World!” program
- Appendix C ■ UML

5

Call stack introspection

In this chapter

- Examining the call stack
- Using call stack introspection
- Avoiding infinite recursion during method intercession
- Checking class invariants

Introspection includes more than the structure of an application. Information about the execution state of the running application is also useful for increasing flexibility. Java has metaobjects that represent the execution state of the running program, including metaobjects that represent the call stack.

Each thread of execution has a call stack consisting of stack frames. Each frame in the call stack represents a method call. Stack frames contain information such as an identification of the method, the location of the statement that is currently executing, the arguments to the method, local variables and their values, and so on. Each stack frame represents the method last called by the method in the frame below it. In Java, the frame at the bottom of a call stack represents the main method of an application or the run method of a thread.

Call stack introspection allows a thread to examine its context. This context includes the name of the method that it is currently executing and the series of method calls that led to that method. This information is useful in several ways:

- *Logging*—An application can log more precise messages given this information.
- *Security*—An API can decide whether or not to proceed based upon its caller's package or class.
- *Control flow*—A reflective facility can avoid situations such as infinite recursion.

These and other applications make call stack introspection a useful tool for programmers.

Java supports call stack introspection though an indirect facility. There is no facility to directly modify a call stack or any of its constituent frames. You can consider methods for thread management to be indirect ways to modify the call stack. The rest of this chapter further motivates the use of call stack introspection and details its mechanics in Java.

5.1 *George's logging problem*

When George first came to WCI, he was given the project of designing and implementing their approach to logging.¹ Logging, as an individual feature, was seen as

¹ In JDK 1.4, Java added the logging facility, `java.util.logging`. This facility is useful and flexible. We exclude it here to concentrate on illustrating the details of call stack introspection. In general, the Java logging facility is fairly advanced, providing class name and method name information about the caller. However, it does not provide line numbers or a full stack trace. The scenario in this chapter comes from a real situation faced by a development organization using JDK 1.3, before the release of JDK 1.4.

an opportunity for George to work independently. Because logging is a concern that cuts across all modules, this task was also an opportunity for him to become familiar with the whole code base.

Logging is an important tool for both diagnosing problems and rollback and recovery in an operational application. We concentrate on the mechanics and information flow for a logging API. There are many good resources on the uses for logging, for example, see the *ACM Computing Surveys* article by Elnozahy et al [26].

Tracing, as we presented it in section 4.4, records the entry and exit to a method. As such, tracing is a special kind of logging. Logging is more general because it can record the passing of other control points than method entry and exit, and it records special kinds of events.

Good applications can do quite a bit of logging. For example, figure 5.1 (from [45]) depicts the logging code in Tomcat, which is the servlet container that is the official reference implementation for Java Servlet and Java Server Pages technologies (see jakarta.apache.org/tomcat). The figure is a bar graph where each bar represents the size of a module in the Tomcat implementation. The stripes in each bar represent logging code.

George knows that to be effective, a logging facility must provide metadata in the records that are logged. He decides to log the name of the calling class, the name of the calling method, and the line number where the call was made. The resulting interface to his logging facility is shown in listing 5.1.

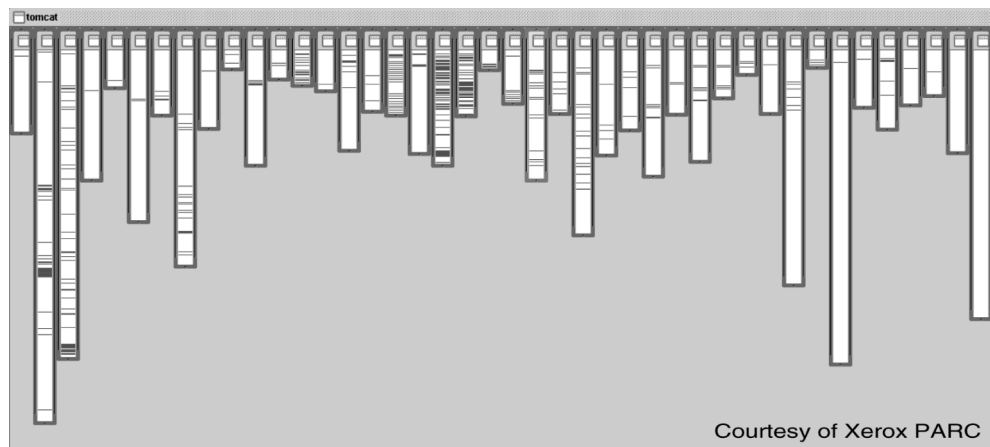


Figure 5.1 The amount of logging in Apache Tomcat. Each vertical bar represents a module. The stripes represent logging code.

Listing 5.1 Interface to an overly simple logging facility

```
public interface Logger {
    // Types for log records
    public static final int ERROR = 0;
    public static final int WARNING = 100;
    public static final int STATUS = 200;
    public static final int DEBUG = 300;
    public static final int TRACE = 400;

    void logRecord( String className,
                   String methodName,
                   int lineNumber,
                   String message,
                   int logRecordType );

    void logProblem( String className,
                    String methodName,
                    int lineNumber,
                    Throwable problem );
}
```

In listing 5.1 the five constants are for classifying the log records. The method `logRecord` writes a log record to whatever medium is used to store such records. The method `logProblem` does the same for a `Throwable`. The line number argument for each method is a nice bit of metadata to have when examining the program with an interactive development environment.

Here is an example of how George's coworkers would use this facility:

```
public class Dog {
    private Logger log = new LoggerImpl();

    public void bark() {
        ...
        this.log.logRecord( "Dog",
                           "bark",
                           23,
                           "Execution point A passed",
                           Logger.STATUS );
        ...
    }
}
```

George's facility fulfills the requirements for storing the desired metadata. However, usage of this facility becomes tedious, at best, and nearly impossible to maintain, at worst. Typing all of that metadata is fragile. Changes to the surrounding code can cause changes in the logging calls. The class name and method name

can easily be incorrect if code is copied and pasted or if a class or method is renamed. In addition, if a subclass inherits code that does logging, the logging call still records the name of the superclass. The line number argument is so unstable that this particular interface is not really practical.

Clearly, this metadata should be available without needing to pass it as parameters. Before JDK 1.4, this information was virtually unavailable programmatically. However, JDK 1.4 provides a set of introspective features to make this metadata available. Next we explore call stack introspection and demonstrate how to simplify George's interface.

5.2 Performing call stack introspection

To achieve call stack introspection, we are going to need a little programming trick, because there is no accessible call stack metaobject in Java. Instead, when an instance of `Throwable` is created, the call stack is saved as an array of `StackTraceElement`. By writing

```
new Throwable().getStackTrace()
```

we have access to a representation of the call stack when the `Throwable` was created. Table 5.1 shows the part of the public interface to the Java `Throwable` class relevant to call stack introspection. `Throwable` has always supported the printing of stack traces. In the past, some ingenious developers would turn this into call stack introspection by capturing the output of `printStackTrace`, parsing it, and making it available programmatically. JDK 1.4 alleviates the need for such solutions by including `StackTraceElement` objects that can be obtained from a `Throwable`.

`Throwable` supports a method named `getStackTrace` that returns an array of `StackTraceElement`. These objects provide access to the same information printed by `printStackTrace`. The array returned by `getStackTrace` represents the call

Table 5.1 Relevant interface to `Throwable`

Method	Description
void printStackTrace ()	Prints this throwable and the call stack to the standard error stream
void printStackTrace (<code>PrintStream s</code>)	Prints this throwable and the call stack to the specified print stream
void printStackTrace (<code>PrintWriter s</code>)	Prints this throwable and the call stack to the specified print writer
<code>StackTraceElement[]</code> getStackTrace ()	Returns the call stack as an array of stack trace elements

stack, with each element representing one stack frame and the first representing the most recent method invocation.

Table 5.2 shows the primary interface to a `StackTraceElement`. Each `StackTraceElement` provides information on the class name, method name, and line number for the execution point that it represents, the name of the file that contains the source code, and an indication as to whether or not the method is native.

Table 5.2 The methods defined by `StackTraceElement`

Method	Description
String <code>getFileName()</code>	The name of the source file containing the execution point represented by this stack trace element is returned.
int <code>getLineNumber()</code>	The line number of the source line containing the execution point represented by this stack trace element is returned.
String <code>getClassName()</code>	The fully qualified name of the class containing the execution point represented by this stack trace element is returned.
String <code>getMethodName()</code>	The name of the method containing the execution point represented by this stack trace element is returned.
boolean <code>isNativeMethod()</code>	If the method containing the execution point represented by this stack trace element is a native method, true is returned.

By creating a new `Throwable`, you can perform call stack introspection. Here is a simple example. In Java, instance methods can easily obtain the name of their class with the following line:

```
this.getClass().getName();
```

However, trying to use this code in a static method or initializer yields a compiler error. This error occurs because `this` cannot be used in a static context. It is necessary to use call stack introspection to obtain the class name from a static context. The following line of code accomplishes that task:

```
(new Throwable()).getStackTrace()[0].getClassName();
```

This is just one problem solved by call stack introspection.

5.3 Logging with call stack introspection

Now let's improve George's logging facility by using call stack introspection. First, we present a better interface in listing 5.2. This interface eliminates those parameters that can be determined reflectively.

Listing 5.2 A better Logger

```
public interface Logger {
    // Types for log records
    public static final int ERROR = 0;
    public static final int WARNING = 100;
    public static final int STATUS = 200;
    public static final int DEBUG = 300;
    public static final int TRACE = 400;

    void logRecord( String message, int logRecordType );

    void logProblem( Throwable problem );
}
```

Listing 5.3 contains an implementation of the `Logger` interface. The second line of `logRecord` constructs a new `Throwable`, making its stack trace information available. Subsequent lines query the stack frame of the caller of `logRecord` to get the necessary metadata.

Listing 5.3 Partial implementation of reflective Logger

```
public class LoggerImpl implements Logger {

    public void logRecord( String message, int logRecordType ) {
        Throwable ex = new Throwable();
        StackTraceElement ste = ex.getStackTrace()[1];

        String callerClassName = ste.getClassName();
        String callerMethodName = ste.getMethodName();
        int callerLineNum = ste.getLineNumber();

        // write of log record goes here
    }

    public void logProblem( Throwable t ) {
        // write of log record goes here
    }
}
```

The implementation is straightforward except for the index into the `StackTraceElement` array. Remember that the top of the stack contains the call to `logRecord`. Consequently, `logRecord` uses the second element in the array. The change to the logger implementation is only a few lines of code. However, the addition of introspection changes its usability dramatically.

5.4 Pitfalls

When an application uses call stack introspection, which stack frame to use becomes an issue. Recall that in the previous example, the `logRecord` method uses the second stack frame. This works correctly. However, imagine if the other method in the interface is implemented as follows:

```
public void logProblem (Throwable problem) {
    this.logRecord( problem.toString(), ERROR );
}
```

Although this looks like an effective implementation, it is defective because `logProblem` adds a stack frame that is unanticipated by `logRecord`. Therefore, the log entry looks like it was entered by the `logProblem` method in the `Logger` class.

This problem may be addressed in several ways according to the circumstances. Enabling the log methods to search the entire call stack for the appropriate frame would be the most general solution. You might think this is simple; however, it presents several difficulties. You could search for the first frame that does not occur in the `LoggerImpl` class. This search might yield another logger that is delegating to the `LoggerImpl`. You could search for the first class that does not exist in the logging package. This search prevents the application from providing logging functionality in another package's facade.

A simpler approach is to ensure that the correct stack element is captured when a call enters the facility. This may be done as follows:

```
public void logRecord( String message, int logRecordType ) {
    logMessage( message,
                logRecordType,
                (new Throwable()).getStackTrace()[1] );
}
public void logProblem( Throwable t ) {
    logMessage( t.toString(),
                ERROR,
                (new Throwable()).getStackTrace()[1] );
}
public void logMessage( String message,
                        int logRecordType,
                        StackTraceElement ste )
{
    String callerClassName = ste.getClassName();
    String callerMethodName = ste.getMethodName();
    int callerLineNumber = ste.getLineNumber();

    // write of log record goes here
}
```

The implementations of `logRecord` and `logProblem` each pass the correct stack trace element to `logMessage`, which does the actual recording of the log entry. The `logMessage` method is public to allow explicit specification of the stack frame in problematic situations. This arrangement ensures that the correct information is entered into the log.

5.5 Class invariant checking

George has been presented with another problem. A major wildlife service wants to track the lifecycles of animals. Life spans, diseases, and pregnancies are just some of the conditions that the service wants to track. George is responsible for implementing the class of time intervals used to track the start and end of these conditions. Listing 5.4 presents George's interface, `TimeInterval`, and his first draft of an implementation.

Listing 5.4 The draft implementation of `TimeInterval`

```
import java.util.Date;

/**
 * Class invariant: start() <= end()
 */
interface TimeInterval {
    Date getStart();
    Date getEnd();
}

-----

import java.util.Date;

public class TimeIntervalImpl1 implements TimeInterval {

    private final Date start;
    private final Date end;

    public TimeIntervalImpl1( Date s, Date e ) {
        start = s;
        end = e;
        assert invariant() : "start>end";
    }

    public Date getStart() { return start; }

    public Date getEnd() { return end; }

    public boolean invariant() { return start.compareTo(end) <= 0; }
}
```

George recognizes that to operate properly, implementations of `TimeInterval` are required to have their start date on or before their end date. He dutifully records this in the comment describing the class. This requirement for implementations of `TimeInterval` is called a class invariant. An **invariant** is a logical condition of the state of a program that is always true, or always true except if control is in some particular piece of code. A **class invariant** is a logical condition that is true for each instance of the class after the instance is constructed and whenever no method of the class is executing. Note that a method is considered to be executing if it is on the call stack, even if that method has passed control by calling another method in the application.

The class invariant for `TimeInterval` is established by the constructor of `TimeIntervalImpl1` and seems to be inviolate. After all, there are only accessor methods and no methods to change the private fields. However, examining the implementation, shows that this invariant can be violated easily from outside the class by any caller that maintains a reference to one of the internal date objects or any caller to one of the accessors.

The problem is that `TimeIntervalImpl1` does not fully encapsulate its components. In this respect, there are two distinct defects. First, in the constructor, `TimeIntervalImpl1` merely assigns the arguments to its private fields rather than making defensive copies. The caller of the constructor may retain access to what becomes the internal parts of a time interval object. Second, the accessors return object references to the internal parts of a time interval. Again, defensive copies should be made.

George quickly fixes the problem with `TimeIntervalImpl2` shown in listing 5.5.² `TimeIntervalImpl2` makes defensive copies in both the constructor and the accessors, which means that no outside object holds a reference to the parts of the time interval. Note that in the constructor, a copy constructor is used rather than `clone`, because the incoming arguments may belong to a subclass of `Date` that overrides `clone` in an undesirable manner. `TimeIntervalImpl2` ensures that its instances are fully encapsulated and that the class invariant is inviolate.

Listing 5.5 A fully encapsulated implementation of `TimeInterval`

```
import java.util.Date;

public class TimeIntervalImpl2 implements TimeInterval {
    private final Date start;
```

² This implementation is based on one we saw in *Effective Java* [7], a book containing many worthwhile lessons for Java programmers.

```
private final Date end;

public TimeIntervalImpl2( Date s, Date e ) {
    start = new Date(s.getTime());
    end = new Date(e.getTime());
    assert invariant() : "start>end";
}

public Date getStart() { return (Date)start.clone(); }

public Date getEnd() { return (Date)end.clone(); }

public boolean invariant() { return start.compareTo(end) <= 0; }
}
```

Writing down the invariants is an important aspect of documenting a class. Maintenance programmers must be informed of the quintessential properties of a class. Classes that are not fully encapsulated need class invariant checking to protect themselves from external code that violates the invariant. A fully encapsulated class can ensure that its class invariants hold based solely on its own code. Nonetheless, checking invariants is useful for fully encapsulated classes to prevent maintenance from inserting code that invalidate invariants.

On recognizing the importance of checking invariants, George decides to provide a facility for his team. First, he specifies an interface, shown in listing 5.6, that all classes using his facility must implement.

Listing 5.6 The `InvariantSupporter` interface

```
public interface InvariantSupporter {
    boolean invariant();
}
```

George envisions writing a class `InvariantChecker` with a static method `checkInvariant` that calls the invariant and provides other services (for example, bypassing invariant checks for customers that require higher performance). With these services in mind, George's facility is a better alternative than establishing a coding standard in which the invariant method is called directly. His teammates would write calls to `InvariantChecker.checkInvariant` at the beginning and end of every method (remember all return statements count as being the end of a method).

George prototypes the facility with a `checkInvariant` that merely calls the invariant method and throws `IllegalStateException` if the class invariant does not hold. His first case is shown in listing 5.7. It contains a problem that demonstrates the wisdom of the decision to write an invariant-checking facility.

Listing 5.7 The Monkey class

```
public class Monkey implements InvariantSupporter {  
    public void hang() {  
        InvariantChecker.checkInvariant( this );  
        // ...  
        // implementation of hang  
        // ...  
        InvariantChecker.checkInvariant( this );  
    }  
  
    public boolean invariant(){  
        screech();  
        return true;  
    }  
  
    public void screech() {  
        InvariantChecker.checkInvariant( this );  
        // ...  
        // implementation of screech  
        // ...  
        InvariantChecker.checkInvariant( this );  
    }  
}
```

Monkey is an invariant supporter that exhibits one of the potential pitfalls involved in invariant checking. Its `invariant` method uses another instance method of `Monkey`. This causes an infinite recursion, because the invocation of `screech` immediately calls `InvariantChecker.checkInvariant`, which calls `screech`, and so on. Clearly, this is unacceptable.

We could adopt a programming convention that invariants may not call methods on the target object. But such programming conventions are easily forgotten or misunderstood. It is better to avoid programming conventions in favor of more flexible programs. This is accomplished by using call stack introspection to check for the infinite recursion and break it.

Listing 5.8 show the actual implementation of `checkInvariant`. This implementation looks back in the call stack to see if `InvariantChecker.checkInvariant` is present. If so, there is an infinite recursion that must be broken by immediately returning. If not, `invariant` may be called safely.

Listing 5.8 The InvariantChecker class

```
public class InvariantChecker {  
    public static void checkInvariant( InvariantSupporter obj ) {  
        StackTraceElement[] ste = (new Throwable()).getStackTrace();  
        for ( int i = 1; i < ste.length; i++ )  
            if ( ste[i].getClassName().equals("InvariantChecker")  
                && ste[i].getMethodName().equals("checkInvariant") )  
                return;  
        if ( !obj.invariant() )  
            throw new IllegalStateException("invariant failure");  
    }  
}
```

The expense of call stack introspection can be avoided with a simpler check. Listing 5.9 shows a different invariant checker that avoids the call stack introspection with a test of a static boolean field. This is accomplished at the expense of funneling all of the class invariant checking in the application into one synchronized static method.

Listing 5.9 The SynchronizedInvariantChecker class

```
public class SynchronizedInvariantChecker {  
    private static boolean invariantCheckInProgress = false;  
    synchronized public static void checkInvariant( InvariantSupporter obj )  
    {  
        if ( invariantCheckInProgress )  
            return;  
        invariantCheckInProgress = true;  
        if ( !obj.invariant() )  
            throw new IllegalStateException("invariant failure");  
        invariantCheckInProgress = false;  
    }  
}
```

If the application is not multithreaded, the `synchronized` modifier may be removed to get a better performing solution. For multithreaded applications, it is not clear which choice for invariant checking (listing 5.8 or listing 5.9) is better. To make the design choice even more complex, the near future will bring us personal computers with multiple processors. Consequently, a faster test at the expense of greater synchronization may not be a good trade-off.

For Java reflective programming, the complexity of multithreading usually has no impact because of the design of the Java Reflection API. This is not true for reflection in general. In languages that have the capability to make dynamic changes to the running program, multithreading can be more problematic. Java reflective programs may only introspect (in particular, a class object may not be changed dynamically and an object may not change the class to which it belongs). Certainly, call stack introspection is one area where the multithreading issues must be addressed. We will see another area in the next chapter (section 6.4).

5.6 Summary

Call stack introspection allows a program to obtain information about its static context including class name, method name, and program line number. It also makes dynamic context available such as the sequence of method calls leading to the current one. This information is accessed by examining metaobjects that represent the program's call stack.

Java's call stack introspection facility is somewhat improvised in JDK 1.4. `Throwable` objects are populated with programmatic representation of the call stack when constructed. This representation can be introspected over, but it cannot be changed.

Though improvised, this facility is still useful. Logging components and similar applications can use call stack introspection to obtain context information for recording. Without this ability, context information must be provided by hand, which becomes difficult to maintain.

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



Authors respond to reader questions



Ebook edition available