Easy As Reeling off a Log

James W. Cooper

I work in a complicated building that may have been laid out by the same architect who laid out the streets of downtown Boston. Boston was rumored to have been designed by a drunken surveyor following the path of a cow. Or maybe it was a surveyor following the path of a drunken cow. In any case, most people new to our building find they have to drop bread crumbs in the halls and follow the trail to get back to their office from a distant point.

There are programs like that, too. Sometimes figuring out how a program manages to get the wrong answers so creatively takes more than stepping through it with a debugger. This could happen because there are 50 or 1000 steps before the failure finally occurs, or because the threads in a multi-threaded program interact in an unexpected way. In either case, you really want to watch what happens over time and then use that information to fix the programming problem.

In such cases, debuggers are pretty useless, because there are too many steps before the fault occurs, or because the failure only happens when the threads are running without interruption. You could write a lot of data to the console with good old *System.out.println*, or you could write a little more code and write the stuff to a file so you could study it later. However, there is no reason to write this code when people have already done it for you. There are in fact, two common and somewhat similar Java logging packages available that you can use instead. In this article, we'll discuss the Java 1.4 logging package and the Apache log4j package. Both have analogous features, but there are some significant differences between them as well.

Why Logging?

A logging package has the advantage that it controls what comes out on then console or is written to a file. And, in fact, you can turn the logging behavior on and off using a configuration file without touching your program. This is handy, because you can ship the program with a configuration file having nearly all logging turned off, but if you encounter a bug, you can turn more of the logging on and create a log file to help you understand what is going on. This is particularly helpful when the system in question works perfectly in your test bed, but does not perform correctly on someone else's system.

The Java JDK 1.4 release contains a java.util.logging package that can provide you with pretty sophisticated control of your program's logging behavior. You can specify the level of information to be logged and the place to log it (console, file or memory buffer). You can also create child loggers which inherit the parent's logging properties but with different priorities. For example you could log only SEVERE errors in the entire program, but within one suspect class or package, you might log many more events.

You can begin logging either under program control or using a configuration file to specify what you want to log. We'll start with the programmatic method because it is easier to see.

First we get an instance of the LogManager class:

```
LogManager lMgr = LogManager.getLogManager();
```

Then, we create a named logger and add it to the current list of loggers:

```
String thisName = "Logpkg";
Logger log = Logger.getLogger(thisName);
lMgr.addLogger(log);
```

Note that we did not specify the actual configuration of this logger (where it puts its text and in what format). If we specify nothing, the log manager takes its default behavior from the file logging.properties in the jdk1.4/jre/lib directory. The defaults in this file state that the default handler is the ConsoleHandler, so that the logging information goes to the Console.

Now we can log information throughout our "very complex" program. You can issue log messages by level. The java logging package supports 7 levels of log message priorities:

- SEVERE (highest value)
- WARNING
- INFO
- CONFIG
- FINE
- FINER
- FINEST (lowest value)
- OFF (no logging)

Each of these has a convenience log method, such as

```
log.severe("big problem here");
```

Here are a couple of these messages in a complete program:

```
import java.util.logging.*;
import Logpkg.*;

public class LogProgTest {
      public LogProgTest() {
            //get an instance of the Log Manager
            LogManager lMgr = LogManager.getLogManager();

            //create a name and add a logger with this name
            String thisName = "Logpkg";
            Logger log = Logger.getLogger(thisName);
            lMgr.addLogger(log);
            log.info( "Here we are");
            //start some other class instance
            TestLogger tlog = new TestLogger();
            //and show that we are exiting
            log.warning( "and there we are");
      }
//---------
      public static void main(String[] args) {
            new LogProgTest();
      }
}
```

This produces the output below:

```
Oct 16, 2002 3:34:07 PM LogProgTest <init>
INFO: Here we are
Oct 16, 2002 3:34:07 PM LogProgTest <init>
WARNING: and there we are
```

Now, if we set the level of the logger to suppress INFO messages and only issue warnings and severe errors, we will get less output. For example,

```
      log.setLevel( Level.WARNING);
      log.info( "Here we are");
      //start some other class instance
      TestLogger tlog = new TestLogger();
      //and show that we are exiting
      log.warning( "and there we are");
```

Then the output is limited to the warning line:

```
Oct 16, 2002 3:36:01 PM LogProgTest <init>
WARNING: and there we are
```

Child Loggers

Logger names can be simple text, or they can be dot separated compound names. Any logger whose rootname is the same but has additional name text to the right of a dot is a child of that original logger. You could have a logger named “mylogger” and a child logger named “mylogger.baby”. Clearly, this is intended to be used to name loggers to parallel package names, but they need not be only that. In this example, we named our logger “Logpkg.” Now, if we create loggers with names descended from that, such as “Logpkg.Testlogger,” we can use these child loggers at different priority levels but use the same handlers.

In this example, we wrote another small class called TestLogger and made it part of a package called Logpkg. Then we can name the logger using the actual class name

```
public class TestLogger {
      private Logger log;
      private FileHandler fh;
      public TestLogger() {
            LogManager lMgr = LogManager.getLogManager();
            //get the class name and use it for a logger name
            String thisName = this.getClass().getName();
            log = Logger.getLogger(thisName);
            lMgr.addLogger(log);
      }
```

This logger is now a child of our original logger, and will use the same handlers. Here is the code that logs information to the child.

```
      public void lTester() {
            log.entering( this.getClass().getName(),"ltest" );
            try {
                int k = 6/0;
            }
            catch (ArithmeticException e) {
                log.severe( e.getMessage() );
            }
        }
```

Note that we now finally have a way to avoid the dreaded *System.out.println* in catching exceptions. We just log them to a file for later review by someone knowledgeable. With the info level messages suppressed, we get the following logged messages

```
Oct 16, 2002 4:13:38 PM LogTest <init>
WARNING: Here we are!
Oct 16, 2002 4:13:38 PM Logpkg.TestLogger lTester
SEVERE: / by zero
```

Logging to a File

Now, if we want to log to a file as well as to the Console, we can do that by creating a FileHandler and adding that handler to our logger.

```
        fh = new FileHandler("logtest.log");
        log.addHandler( fh);
```

If we do not specify anything different, the default output format for a file handler is XML. Here is a section of the output:

```
<record>
  <date>2002-10-16T15:57:11</date>
  <millis>1034798231560</millis>
  <sequence>0</sequence>
  <logger>Logpkg</logger>
  <level>WARNING</level>
  <class>LogTest</class>
  <method>&lt;init&gt;</method>
  <thread>10</thread>
  <message>Here we are!</message>
</record>
```

Using a Properties File

A more usual way to define the logger properties you want to use is to store them in a properties file that is read in on startup. The advantage is that you can change these properties, such as the level of data to be logged, and even turn all the logging off by changing only the properties file. You do not need to change or recompile your Java code. Here is a simple method for setting the properties file to be read by the LogManager:

```
private void readLogProperties(LogManager lMgr) {
        try {
        //open the file to be used for configuration
            FileInputStream fl =
                new FileInputStream(new File("logtest.properties"));
            //read the configuration into the LogManager
            lMgr.readConfiguration(fl);
        } catch (IOException e) {
            System.out.println("log property file not found");
        }
}
```
Then, our main program need make no reference to levels or handlers:

```
public LogPropTest() {
      LogManager lMgr = LogManager.getLogManager();
      //read in a properties file
      readLogProperties(lMgr);
      //create a name and add a logger with this name
      String thisName = "Logpkg";
      Logger log = Logger.getLogger(thisName);
      log.log(Level.WARNING, "Here we are!");
      //call another class which has its own logger
      TestLogger tlog = new TestLogger();
      tlog.lTester();
      //log the program exit
      log.info("There we are");
}
```

This configuration file contains the names of the handlers and those to be used, in a standard Java Properties file format. First, we define the handlers we will use throughout, and the default level:

```
handlers=
java.util.logging.FileHandler, java.util.logging.ConsoleHandler

.level= INFO
```

Then, we define the properties for each handler:

```
# define the file handler output
java.util.logging.FileHandler.pattern = LogTestu.log
java.util.logging.FileHandler.limit = 50000
java.util.logging.FileHandler.count = 2
java.util.logging.FileHandler.formatter =
            java.util.logging.XMLFormatter

# Limit the console messages to WARNING and above.
java.util.logging.ConsoleHandler.level = WARNING
java.util.logging.ConsoleHandler.formatter =
            java.util.logging.SimpleFormatter
```

Note that by just changing one line:

```
.level= OFF
```
we can total disable logging.

Log4J

The Log4J package has the same basic capabilities but also allows more sophisticated control of output and more types of handlers. If you want your program to run using Java JVMs before 1.4, you can specify this package instead. Log4J is available for free download from the Apache.org website. It was originally written by Ceki Gulcu, but has been an open source project with a number of contributors for some time now.

Log4J differs in that it has no default output handlers, which in its terminology are called *appenders*. You need to specify appenders either in code or in a configuration file.

Further, you can specify the format of the output using a set of format specifiers reminiscent of those use for output formatting in the C and C++ languages.

The same logging test program looks like this when written to create loggers programma tically using Log4J.

```
import org.apache.log4j.*;
import java.util.*;
import java.io.*;
import Logpkg.*;

public class Log4JTest {
      public Log4JTest() {
            String thisName = "Logpkg";
            Logger log = Logger.getLogger(thisName);
            BasicConfigurator.configure();
            //create another class with its own child loggers
            TestLogger tlog = new TestLogger();
            log.log(Priority.WARN, "We are really here now!");
            //log information in the child class and logger
            tlog.lTester();
            log.info("That was it");
      }
      //------
      public static void main(String[] args) {
            new Log4JTest();
      }
}
```

Note that you can set up a basic configuration that defines a Console logger with formatted output by simply calling the static *configure* method of the BasicConfigurator class. This call can be used to set up the standard Console output, but if the program finds a file called “log4j.properties” it will take its configuration from it. The output from this simple logging program, including that from the child class is

```
0 [main] WARN Logpkg  - We are really here now!
0 [main] INFO Logpkg.TestLogger  - entering ltest
0 [main] FATAL Logpkg.TestLogger  - / by zero
0 [main] INFO Logpkg  - That was it
```

This program generates a slightly formatted output of one line per log message. You can reduce the forma tting by creating your own Si mpleFormat Console appender, by replacing the call to BasicConfigurator.configure with

```
log.addAppender( new ConsoleAppender(new SimpleLayout()));
```

Then, the output is

```
WARN - We are really here now!
INFO - entering ltest
FATAL - / by zero
INFO - That was it
```

Finally, you can specify your own pattern format by writing:

```
log.addAppender(new ConsoleAppender(
      new PatternLayout("%d %r [%t] %-5p %c %x - %m\n")));
```

which produces the formatted output:

```
2002-10-17 09:05:46,659 0 [main] WARN  Logpkg  - We are really here now!
2002-10-17 09:05:46,659 0 [main] INFO  Logpkg.TestLogger  - entering ltest
2002-10-17 09:05:46,669 10 [main] FATAL Logpkg.TestLogger  - / by zero
2002-10-17 09:05:46,669 10 [main] INFO  Logpkg  - That was it
```

Child Loggers

The child class naming conventions are exactly the same in Log4J. We named our root logger as “Logpkg” and our child logger will be named after the package and class it occurs in as before.

```
public class TestLogger {
      private Logger log;
      public TestLogger() {
            String thisName = this.getClass().getName();
            log = Logger.getLogger(thisName);
      }
      //-----
      public void lTester() {
            log.info ("entering ltest" );
            try {
                  int k = 6/0;
            }
            catch (ArithmeticException e) {
                  log.fatal( e.getMessage() );
            }
      }
}
```

Here, the new logger is called “Logpkg.TestLogger”.

The Logger As a Singleton

The Singleton pattern provides one and only one instance of a class, and a global point of access to it. If we create and name a logger as “Logpkg” in our main program, we can access that same logger anywhere else in the program by that name, and the same instance of the logger will be returned from the Logger.getLogger method call. This means that you don’t have to pass references to the logger around your code. You can just fetch the logger by name wherever you need it.

The Configuration File

Now, if you want to configure Log4J using a configuration file, you write even less code, since the properties file does all the work:

```
public class Log4JConfig {
      public Log4JConfig() {
            String thisName = "Logpkg";
            Logger log = Logger.getLogger( thisName);
            PropertyConfigurator.configure("log4j.prop");
            TestLogger tlog = new TestLogger();
            log.log(Priority.WARN, "We are really here now!");
            //-------------
            tlog.lTester() ;
            log.info( "That was it");
      }
      //------
      public static void main(String[] args) {
            new Log4JConfig();
      }
}
```

In a typical configuration file, you define the root level appender’s characteristics, and then derive all the actual appenders from it:

```
# set root level appender
log4j.rootLogger=DEBUG, appender1, f1

log4j.appender.appender1=org.apache.log4j.ConsoleAppender
log4j.appender.appender1.layout=org.apache.log4j.PatternLayout
log4j.appender.appender1.layout
      .ConversionPattern=%r [%t] %p %c %x - %m%n

log4j.appender.f1=org.apache.log4j.FileAppender
log4j.appender.f1.File=config4j.log
log4j.appender.f1.layout=org.apache.log4j.PatternLayout
log4j.appender.f1.layout
      .ConversionPattern=%r [%t] %p %c %x - %m%n
```

The priorities you specify in Log4J are part of the Priority class, and are named INFO, DEBUG, WARN, ERROR and FATAL.

Cutting Up with a Chainsaw

The Log4J package also provides a stand-alone GUI display called *Chainsaw*, that allows you to monitor log messages visually. You need to start Chainsaw before you start your program, either by

```
java org.apache.log4j.chainsaw.Main
```

or by creating a desktop shortcut (on Windows) to a “javaw” invocation of the same program.

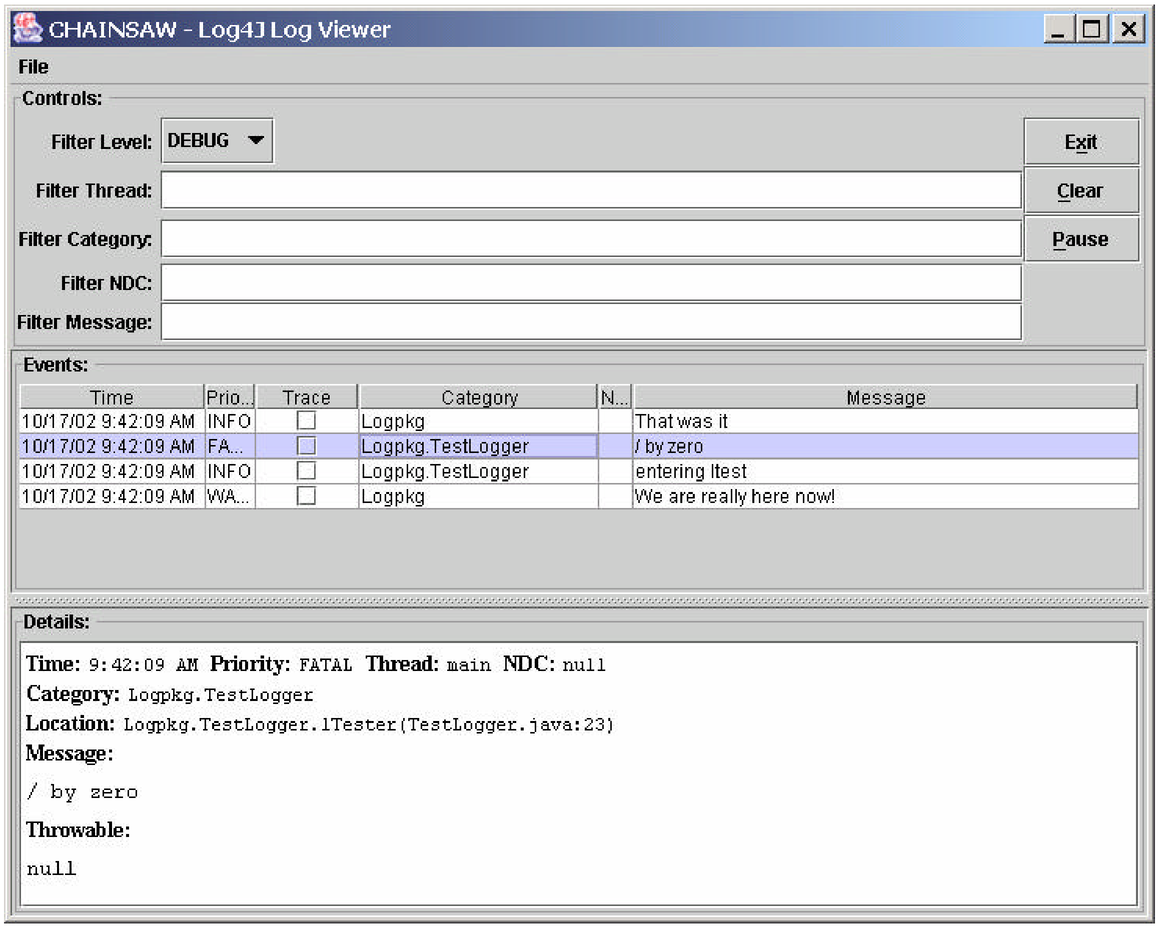
Then you need to add Chainsaw to your configuration file:

```
# set root level appenders
log4j.rootLogger=DEBUG, CHAINSAW, appender1, f1
```

The CHAINSAW config statements are much the same as for the other appenders:

```
log4j.appender.CHAINSAW=org.apache.log4j.net.SocketAppender
log4j.appender.CHAINSAW.RemoteHost=localhost
log4j.appender.CHAINSAW.Port=4445
log4j.appender.CHAINSAW.LocationInfo=true
```

Note that Chainsaw communicates using socket 4445 (by default) and that it can send the logging to any TCP/IP host. Here we use the same machine. The Chainsaw display looks like this:



Filter Level: DEBUG
Filter Thread:
Filter Category:
Filter NDC:
Filter Message:

Time	Prio...	Trace	Category	N...	Message
10/17/02 9:42:09 AM	INFO	☐	Logpkg		That was it
10/17/02 9:42:09 AM	FA...	☐	Logpkg.TestLogger		/ by zero
10/17/02 9:42:09 AM	INFO	☐	Logpkg.TestLogger		entering ltest
10/17/02 9:42:09 AM	WA...	☐	Logpkg		We are really here now!

Time: `9:42:09 AM` **Priority:** `FATAL` **Thread:** `main` **NDC:** `null`
Category: `Logpkg.TestLogger`
Location: `Logpkg.TestLogger.lTester(TestLogger.java:23)`
Message:

```
/ by zero
```

Throwable:

```
null
```

Timing Issues

But do you really want all that logging garp cluttering up your code? Even turned off, there must be some performance penalty. There probably is a small one, primarily based on the cost of constructing the string you send to the logger, if that string contains a number of pieces to be converted to a single string value. You can avoid even that cost by wrapping each call with a test as to whether the logging is turned on"

```
if (log.isDebugEnabled() ) {
            log.info("That was it" + test.timer());
}
```

Summary

Both of these logging packages are worth a look. They provide a really useful alternative to step-by-step debugging whenever the problem you are trying to solve is rather complex and involves some interactions you can't find easily with a debugger. They drop the breadcrumbs for you all through the maze of twisty passages in your code, and you can sweep away the breadcrumbs with the flick of the OFF switch.