Logging

- System.out.println
- JSE Logging
- Log4J
- SLF4J
- logback

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Debugging Strategies

- Debugging Java applications is an acquired skill
- There are several useful techniques available to help:
  - Debugging by inspection
  - Using your IDE's GUI debugger
  - Logging

This chapter covers the logging strategy.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Logging

- Logging messages to a console/log file is an age-old programming technique
- Possible logging techniques include:
  - System.out.println
  - The JSE Logging API
  - Apache Log4J
  - Simple Logging Facade for Java (SLF4J)
  - logback

Though logging is rather primitive, it is still quite useful and sometimes is the only way to debug, especially for time-sensitive applications. Luckily, Java gives you several options on how to log diagnostic messages. Note that there are several other logging frameworks available not listed here including LogBack and SLF4J.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using System.out.println for Logging

- This is the most primitive and easiest way to debug your Java code, including server-side programs such as servlets and JSPs -- simply write diagnostic info to the console
- For server-side programs, the *container* may display the output or log it to a file
- You can use a boolean variable to turn the output off and on (requires a re-compile)

```
1    private boolean debugging = true;
2
3    . . .
4
5    if ( debugging )
6        System.out.println ( "
7          Test println logging..." );
```

This technique is familiar to almost every programmer. The only real quirk is that you don't really know where the server-side output will be sent, since the container decides where to send standard output. For example, the Tomcat Web container sends this output to a console window by default.

Here we define a boolean that we can set to false when we go to production to suppress the diagnostic output.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using JSE Logging API

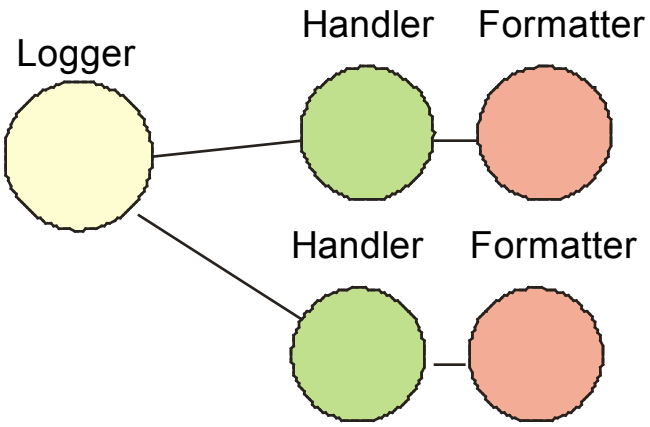
- JSE 1.4 introduced a powerful logging facility that can also be used in JEE applications
- The logging facility gives you compile-time or run-time control over how much information is logged
- Thus you can have verbose logging during development and only log errors in production

Prior to JSE 1.4, many Java programmers wrote their own logging facilities or used the Apache Log4J package. But with the advent of 1.4, now a powerful and flexible logging facility is part of JSE.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

JSE Logging Architecture

- The Logger class acts as the target for logging, but it's the Handler subclasses that actually perform the output, formatting it according to the associated Formatter subclass



All of the classes and interfaces for logging are in the java.util.logging package.

The Logger class defines methods that you can use to write output. A Logger keeps a list of Handler/Formatter pairs that perform the output. There are standard classes (e.g. ConsoleHandler and SimpleFormatter) that you can use, or you can write your own.

It's possible to establish a hierarchy of Logger objects, each with their own list of Handler/Formatters. That lets you control logging on a package level -- for example, you could turn off logging for classes in the "com.mycompany.packageA" package but allow logging for other sub-packages in the "com.mycompany" package. We will not explore that functionality in this chapter.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Standard Logging Classes

- The java.util.logging package contains a few standard handlers and formatter classes
- You can write your own if the standard classes do not meet your needs

MemoryHandler	Implements a memory-based circular buffer for output
ConsoleHandler	Writes output to the system console
FileHandler	Writes output to a file
SimpleFormatter	Formats brief output
XMLFormatter	Formats output as XML content

You can configure which handler/formatter pairs a logger uses either by modifying a system properties file or by writing code. If you do not change the default, the so-called "root" logger uses a ConsoleHandler/SimpleFormatter to perform output to a console.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

JSE Logging Levels

- The Logger class provides methods with names identical to the logging levels that let you log info at that level
- You can control the "threshold" level for logging either at run-time or by editing a standard properties file (the default is INFO)

SEVERE	For messages that must be logged in all cases (e.g. fatal errors)
WARNING	For warning messages
INFO	For informational message
CONFIG	For messages about configuration issues
FINE	Finely detailed messages for debugging and diagnosing problems
FINER	More detail than FINE
FINEST	The most detail possible

The logger maintains a threshold level for logging -- for example, if you set the level to Warning, only Warning and Severe messages will go through.

Having all of these levels configurable at runtime lets you log more detail during development than you do during production. That increases runtime performance, since fewer messages are being logged.

For example, as you code a servlet, you could use myLogger.fine() to output detailed messages about a JDBC connection and used myLogger.severe() to output a message if the connection fails entirely. That way, during development, you obtain detailed diagnostic information, but don't take a huge performance hit once you go to production.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

JSE Logging Example

```
1    Logger theLogger =
2      Logger.getLogger ( MyClass.class.getName() );
3
4    theLogger.info ( "Test JSE logging..." );
5
6    theLogger.setUseParentHandlers ( false );
7    FileHandler fileHandler =
8         new FileHandler ( "%t/logs/mylog", 10000, 5 );
9    XMLFormatter xmlFormatter =
10        new XMLFormatter();
11   fileHandler.setFormatter ( xmlFormatter );
12   theLogger.addHandler ( fileHandler );
13
14   for ( int j = 0; j < 1000; j++ )
15   {
16     theLogger.info (
17         "Test JSE logging, msg #: " + j );
18   }
```

This code fragment first creates a logger and then in line 4, logs a string using the default setup (we did not modify any properties file), which uses a ConsoleHandler/SimpleFormatter to write to the console. The output looks something like:

Oct 28, 2002 2:08:16 PM MyClass myMethod INFO: Test J2SE 1.4 logging...

Then in lines 6 to 18, we disable the default output and create a FileHandler that writes to files in the system "temp" directory. It will create a maximum of five files using "mylog" as a pattern and will store at most 10,000 bytes in each. If the program creates more output than that, the Logger will cycle through the five files. We also specify that the files should be formatted as XML content.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

JSE Logging Example Output

```
1    <?xml version="1.0"
2          encoding="windows-1252" standalone="no"?>
3    <!DOCTYPE log SYSTEM "logger.dtd">
4    <log>
5    <record>
6      <date>2012-10-28T12:02:30</date>
7      <millis>1035828150883</millis>
8      <sequence>932</sequence>
9      <logger>ServletLogging</logger>
10     <level>INFO</level>
11     <class>MyClass</class>
12     <method>myMethod</method>
13     <thread>11</thread>
14     <message>
15       Test JSE logging, msg #: 924
16     </message>
17   </record>
```

This listing shows a portion of the file c:\temp\logs\mylog.2 that was created by the logger based on the FileHandler we just created. Note that the information is formatted as XML and contains a "record" element for each log output. The record includes the time and date when the message was generated, along with context information and finally the message itself.

The beauty of using XML is that you can then use tools such as XSLT or DOM/SAX to process the message logs, perhaps creating HTML output or performing data processing on the logs.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Introduction to Log4J

- Log4J is an open-source logging framework hosted by the Apache group
- Log4J is similar to the JSE logging framework and in fact influenced its design



To use log4j, you must put its JAR file in the program's CLASSPATH.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Steps For Using Log4J

1. Configure your program's CLASSPATH to include *log4j.jar*
2. Write a Log4J configuration file either as a *properties* file or as an XML file
3. In your program:
    - Instantiate a logger
    - Add logging statements at appropriate points

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Sample Log4J Properties File

- This file is named **log4j.properties** and should be in the program's CLASSPATH
- This file configures Log4J to output to the console

```
1    log4j.appender.stdout=org.apache.log4j.ConsoleAppender
2    log4j.appender.stdout.Target=System.out
3    log4j.appender.stdout.layout=\
4      org.apache.log4j.PatternLayout
5    log4j.appender.stdout.layout.ConversionPattern=\
6      %d{ABSOLUTE} %5p %c{1}:%L - %m%n
7    log4j.rootLogger=debug, stdout
```


Note that Log4J allows for hierarchical loggers - that lets you configure logging at a higher level, then override it on sub-loggers. The root logger we configure here is at the top of the hierarchy.

A logger is considered a child based on its package name. So for example, given classes in two packages, com.goliath and com.goliath.util, the hierarchy would be:

```
root
  |
com.goliath.ClassA
  |
com.goliath.util.ClassB
```

So we could set global configuration on the root, and then override it on the "second" level, also configuring the "third" level:

```
log4j.rootLogger=info
. . .
log4j.logger.com.goliath=debug
```

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Sample Log4J Program

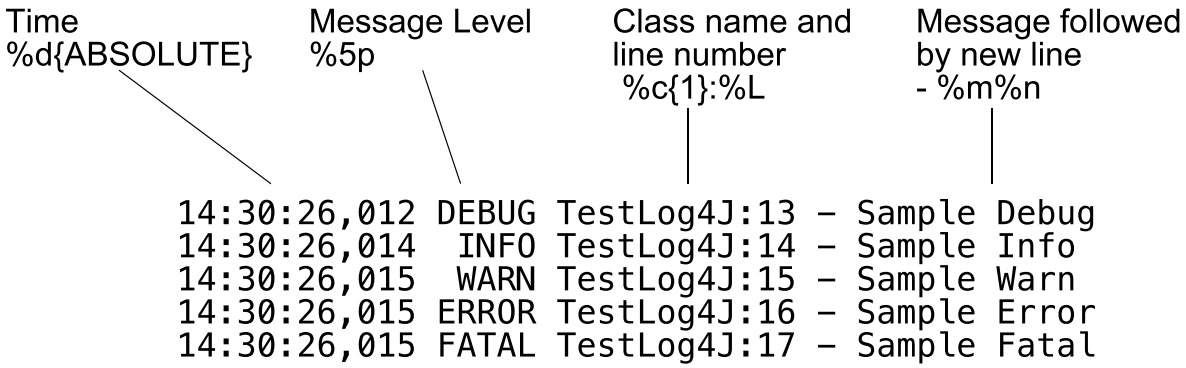
```
1    package log4j;
2
3    import org.apache.log4j.Logger;
4
5    public class TestLog4J
6    {
7        private static org.apache.log4j.Logger log = Logger
8           .getLogger(TestLog4J.class);
9
10       public static void main(String[] args)
11       {
12           log.trace("Sample Trace");  // lowest level
13           log.debug("Sample Debug");
14           log.info("Sample Info");
15           log.warn("Sample Warn");
16           log.error("Sample Error");
17           log.fatal("Sample Fatal");  // highest level
18       }
19   }
```


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Sample Log4J Program's Output

- Using a *layout* pattern of:

```
%d{ABSOLUTE} %5p %c{1}:%L - %m%n
```



Note that on the last page, the "trace" message would not appear -- that's because the properties file configured the level to be "debug" and trace is a lower-priority level than debug.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Log4J Logging Levels

ALL	The ALL has the lowest possible rank and is intended to turn on all logging
TRACE	The TRACE Level designates finer-grained informational events than the DEBUG
DEBUG	The DEBUG Level designates fine-grained informational events that are most useful to debug an application
INFO	The INFO level designates informational messages that highlight the progress of the application at coarse-grained level
WARN	The WARN level designates potentially harmful situations
ERROR	The ERROR level designates error events that might still allow the application to continue running
FATAL	The FATAL level designates very severe error events that will presumably lead the application to abort
OFF	The OFF has the highest possible rank and is intended to turn off logging

These levels are defined in the org.apache.log4j.Level class. Note that OFF and ALL can only be configured as a threshold: there are no methods for these levels.

It's also possible to programmatically define custom logging levels.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Log4J Layouts

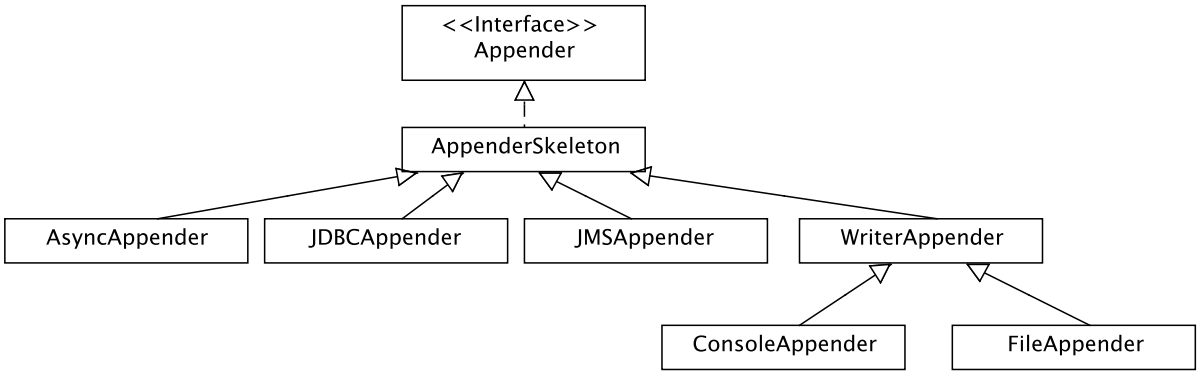
- Log4J provides several standard **layout** classes that format output in various ways
- You can write custom layouts by writing a subclass of *org.apache.log4j.Layout*

HTMLLayout	This layout formats events in a HTML table
PatternLayout	This layout formats events in a String using conversion patterns
SimpleLayout	This layout formats events by displaying only the log level and message
XMLLayout	This layout formats events as XML elements. The output is NOT well-formed XML, but is intended to by included into a separate XML file

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Log4J Appenders

- Log4J provides several standard **appender** classes that control where the output is sent
- You can write custom appenders by writing a class that implements *org.apache.log4j.Appender*



There are a few standard appender classes not shown in the figure. Here is the complete list:

AsyncAppender, JDBCAppender, JMSAppender, LF5Appender, NTEventLogAppender, NullAppender, SMTPAppender, SocketAppender, SocketHubAppender, SyslogAppender, TelnetAppender, WriterAppender

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Introduction to SLF4J

- The **Simple Logging Facade for Java** (SLF4J) defines a logging-framework independent client API for logging
- If you log using this facade API, then you can switch between JSE logging, Log4J or logback at deployment, without needing to change your source code



Many open source Java project support SLF4J, including Hibernate and the Spring Framework.

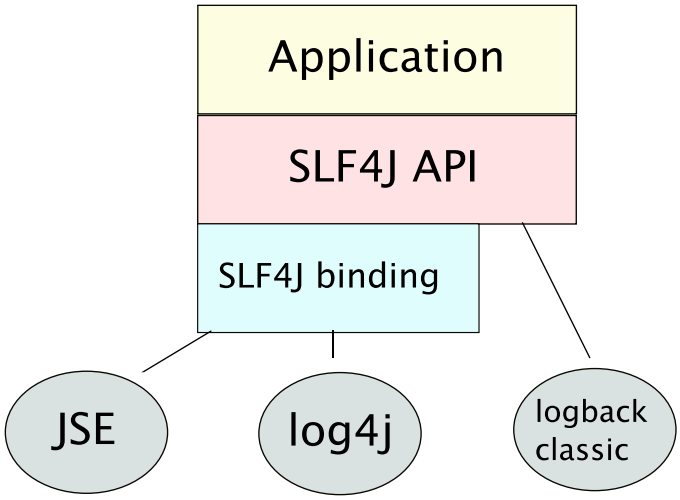
The homepage for SLF4J is at:

`http://www.slf4j.org`

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using SLF4J

- To use SLF4J, you need the SLF4J API JAR as well as a SLF4J **binding** JAR and the JAR(s) for the logging framework you wish to use
- The SLF4J client API is very similar to log4J



Logback is considered to be the "native" back end for SLF4J, so you don't need a binding JAR.

Note that you configure logging (e.g. the current level) using the underlying framework. For example, if you are using Log4j, you can write a log4j.properties file.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Sample SLF4J Program

```
1    package logging;
2
3    import org.slf4j.Logger;
4    import org.slf4j.LoggerFactory;
5
6    public class TestSLF4J
7    {
8        private static Logger log =
9           LoggerFactory.getLogger(TestSLF4J.class);
10
11       public static void main(String[] args)
12       {
13           log.trace("Sample Trace"); // lowest level
14           log.debug("Sample Debug");
15           log.info("Sample Info");
16           log.warn("Sample Warn");
17           log.error("Sample Error");
18       }
19   }
```

This code is nearly identical to the Log4J example shown earlier - main difference is in the imports (and that SLF4J doesn't support the fatal() level).

Since SLF4J is just a facade, you need to configure an actual logging back end by putting the appropriate JARs in your CLASSPATH. For example, to use JSE 1.4 logging, use:

```
slf4j-api-xxxx.jar
slf4j-jdk14-xxxx.jar
```

Or to use Log4J:

```
slf4j-api-xxxx.jar
slf4j-log4j12-xxxxx.jar
log4j-1.2.x.jar
```

It's also important to note that you should put at most ONE binding JAR on the CLASSPATH. In other words, to switch backends, you should remove the binding JAR for the old backend and replace it with the new JAR.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Introduction to Logback

- Logback is a successor to Log4J that builds upon Log4j but promises better performance and easier configuration
- Logback natively implements the SLF4J API, so you don't need a binding JAR - should be fast
- You can configure logback using either XML or Groovy



Logback was founded by the same developer who originally created Log4J.

The homepage for logback is:

`http://logback.qos.ch`

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using Logback

- To use logback, your code uses the SLF4J API and the CLASSPATH contains the SLF4J API JAR as well as the logback core and classic JARs
- You can configure the logging by creating logback.xml or logback.groovy - if neither is found, logback uses a default configuration

The default configuration uses a ConsoleAppender with a PatternLayout at DEBUG level.

Note that logback first looks for a Groovy config file, then looks for XML. If logback finds neither, it uses a default configuration.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Configuring Logback with XML

```
1    <configuration>
2
3      <appender name="STDOUT"
4         class="ch.qos.logback.core.ConsoleAppender">
5        <encoder>
6          <pattern>
7            %d{HH:mm:ss.SSS} %-5level %logger{36} - %msg%n
8          </pattern>
9        </encoder>
10     </appender>
11
12     <root level="info">
13       <appender-ref ref="STDOUT" />
14     </root>
15   </configuration>
```


This from a file named logback.xml - it is similar to the default configuration but has a slightly shorter pattern and specifies the root level of "info" rather than the default "debug".

You put this file on the program's CLASSPATH.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Configuring Logback with Groovy

```
1    import ch.qos.logback.classic.encoder.PatternLayoutEncoder
2    import ch.qos.logback.core.ConsoleAppender
3
4    import static ch.qos.logback.classic.Level.DEBUG
5
6    appender("STDOUT", ConsoleAppender) {
7      encoder(PatternLayoutEncoder) {
8        pattern =
9           "%d{HH:mm:ss.SSS} %-5level %logger{36} - %msg%n"
10     }
11   }
12   root(DEBUG, ["STDOUT"])
```


Groovy syntax is essentially a simplified Java syntax.

For this to work, create this content in logback.groovy and then also put the Groovy JARs in the program's CLASSPATH.

Logback defines a Groovy DSL (domain specific language) that mirrors the XML configuration. For example, here we use the "appender" DSL object, which lets us configure in a similar fashion to the "appender" XML element. For details on the Groovy logback DSL, see:

`http://logback.qos.ch/manual/groovy.html`

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Chapter Summary

In this chapter, you learned:

- How to use the JSE Logging Framework
- How to use the Apache Log4J Framework
- How to use the SLF4J API

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum