SLF4J extensions

SLF4J extensions are packaged within *slf4j-ext.jar* which ships with SLF4J.

- Profiler
- Extended logger
- Event Logging
- Logging added with Java agent (requires Java 5)

Profilers

What is a profiler?

According to wikipedia, profiling is the investigation of a program's behavior using information gathered as the program runs, i.e. it is a form of dynamic program analysis, as opposed to static code analysis. The usual goal of performance analysis is to determine which parts of a program to optimize for speed or memory usage.

SLF4J profilers, a.k.a. poor man's profilers, will help the developer gather performance data. Essentially, a profiler consists of one or more stopwatches. Stopwatches are driven (started/stopped) by statements in the *source code*. An example should make the point clearer.

Basic example

Example: Using the profiler: *BasicProfilerDemo*

```
[omitted]
public class BasicProfilerDemo {

  public static void main(String[] args) {
    // create a profiler called "BASIC"
    Profiler profiler = new Profiler("BASIC");
    profiler.start("A");
    doA();

    profiler.start("B");
    doB();

    profiler.start("OTHER");
    doOther();
    profiler.stop().print();
  }
[omitted]
```

Running the above example will output the following output.

```
+ Profiler [BASIC]
|-- elapsed time                      [A]   220.487 milliseconds.
|-- elapsed time                      [B]  2499.866 milliseconds.
|-- elapsed time                  [OTHER]  3300.745 milliseconds.
|-- Total                         [BASIC]  6022.568 milliseconds.
```

Instantiating a profiler starts a global stopwatch. Each call to the start() method starts a new and named stopwatch. In addition to starting a named stopwatch, the start() method also causes the previous stopwatch to stop. Thus, the call to `profiler.start("A")` starts a stopwatch named "A". The subsequent call to `profiler.start("B")` starts stopwatch "B" and simultaneously stops the stopwatch named "A". Invoking the `stop()` on a profiler stops the last stopwatch as well as the global stopwatch which was started when the profiler was instantiated.

Profiler nesting

Profilers can also be nested. By nesting profilers, it is possible to measure a task which itself has subtasks that need to be timed and measured.

Starting a nested profiler will stop any previously started stopwatch or nested profiler associated with the parent profiler.

Often times, the subtask is implemented by a different class as the class hosting the parent profiler. Using the `ProfilerRegistry` is a convenient way of passing a nested profiler to an object outside the current object. Each thread has its own profiler registry which can be retrieved by invoking the `getThreadContextInstance()` method.

Example: *NestedProfilerDemo*

```
public class NestedProfilerDemo {

  public static void main(String[] args) {
    // create a profiler called "DEMO"
    Profiler profiler = new Profiler("DEMO");

    // register this profiler in the thread context's profiler registry
    ProfilerRegistry profilerRegistry = ProfilerRegistry.getThreadContextInstance();
    profiler.registerWith(profilerRegistry);
    // start a stopwatch called "RANDOM"
    profiler.start("RANDOM");
    RandomIntegerArrayGenerator riaGenerator = new RandomIntegerArrayGenerator();
    int n = 1000*1000;
    int[] randomArray = riaGenerator.generate(n);

    // create and start a nested profiler called "SORT_AND_PRUNE"
    // By virtue of its parent-child relationship with the "DEMO"
    // profiler, and the previous registration of the parent profiler,
    // this nested profiler will be automatically registered
    // with the thread context's profiler registry
    profiler.startNested(SortAndPruneComposites.NESTED_PROFILER_NAME);

    SortAndPruneComposites pruner = new SortAndPruneComposites(randomArray);
    pruner.sortAndPruneComposites();

    // stop and print the "DEMO" printer
    profiler.stop().print();
  }
}
```

Here is the relevant excerpt from the SortAndPruneComposites class.

```
[omitted]
public class SortAndPruneComposites {

  static String NESTED_PROFILER_NAME = "SORT_AND_PRUNE";

  final int[] originalArray;
  final int originalArrayLength;

  public SortAndPruneComposites(int[] randomArray) {
    this.originalArray = randomArray;
    this.originalArrayLength = randomArray.length;

  }

  public int[] sortAndPruneComposites() {
    // retrieve previously registered profiler named "SORT_AND_PRUNE"
    ProfilerRegistry profilerRegistry = ProfilerRegistry.getThreadContextInstance();
    Profiler sortProfiler = profilerRegistry.get(NESTED_PROFILER_NAME);

    // start a new stopwatch called SORT
    sortProfiler.start("SORT");
    int[] sortedArray = sort();
    // start a new stopwatch called PRUNE_COMPOSITES
    sortProfiler.start("PRUNE_COMPOSITES");
    int result[] = pruneComposites(sortedArray);

    return result;
  }
[omitted]
```

On a Dual-Core Intel CPU clocked at 3.2 GHz, running the `ProfilerDemo` application yields the following output:

```
+ Profiler [DEMO]
|-- elapsed time                  [RANDOM]    70.524 milliseconds.
|---+ Profiler [SORT_AND_PRUNE]
    |-- elapsed time                    [SORT]   665.281 milliseconds.
    |-- elapsed time        [PRUNE_COMPOSITES]  5695.515 milliseconds.
    |-- Subtotal              [SORT_AND_PRUNE]  6360.866 milliseconds.
|-- elapsed time          [SORT_AND_PRUNE]  6360.866 milliseconds.
|-- Total                           [DEMO]  6433.922 milliseconds.
```

From the above, we learn that generating 1'000'000 random integers takes 70 ms, sorting them 665 ms, and pruning the composite (non-prime) integers 5695 ms, for a grand total of 6433 ms. Given that pruning composites takes most of the CPU effort, any future optimizations efforts would be directed at the pruning part.

With just a few well-placed profiler calls we were able to identify hot-spots in our application. Also note that passing a profiler to a target class could be achieved by registering it in a profiler registry and then retrieving it in the target class.

Printing using a logger

Invoking `profiler.print` will always print the output on the console. If you wish to leave the profiler code in production, then you probably need more control over the output destination. This can be accomplished by associating a logger of your choice with a profiler.

After you have associated a logger with a profiler, you would invoke the `log()` method instead of `print()` previously, as the next example illustrates.

Profiler with a logger: NestedProfilerDemo2

```
[omitted]
public class NestedProfilerDemo2 {

  static Logger logger = LoggerFactory.getLogger(NestedProfilerDemo2.class);

  public static void main(String[] args) {
    Profiler profiler = new Profiler("DEMO");
    // associate a logger with the profiler
    profiler.setLogger(logger);

    ProfilerRegistry profilerRegistry = ProfilerRegistry.getThreadContextInstance();
    profiler.registerWith(profilerRegistry);

    profiler.start("RANDOM");
    RandomIntegerArrayGenerator riaGenerator = new RandomIntegerArrayGenerator();
    int n = 10*1000;
    int[] randomArray = riaGenerator.generate(n);

    profiler.startNested(SortAndPruneComposites.NESTED_PROFILER_NAME);

    SortAndPruneComposites pruner = new SortAndPruneComposites(randomArray);
    pruner.sortAndPruneComposites();

    // stop and log
    profiler.stop().log();
  }
}
```

The output generated by this example will depend on the logging environment, but should be very similar to the output generated by the previous NestedProfilerDemo example.

The log() method logs at level DEBUG using a marker named "PROFILER".

If your logging system supports markers, e.g. logback, you could specifically enable or disable output generated by SLF4J profilers. Here is logback configuration file disabling output for any logging event bearing the "PROFILER" marker, even if the logger used by the profiler is enabled for the debug level.

logback configuration disabling logging from profilers, and only profilers

```
<configuration>

  <turboFilter class="ch.qos.logback.classic.turbo.MarkerFilter">
    <Marker>PROFILER</Marker>
    <OnMatch>DENY</OnMatch>
  </turboFilter>

  <appender name="STDOUT"
    class="ch.qos.logback.core.ConsoleAppender">
    <layout class="ch.qos.logback.classic.PatternLayout">
      <Pattern>%-5level %logger{36} - %msg%n</Pattern>
    </layout>
  </appender>

  <root>
    <level value="DEBUG" />
    <appender-ref ref="STDOUT" />
  </root>
</configuration>
```

MDCStrLookup

StrLookup is a class defined in Apache Commons Lang. It is used in conjunction with the StrSubstitutor class to allow Strings to have tokens in the Strings dynamically replaced at run time. There are many cases where it is desirable to merge the values for keys in the SLF4J MDC into Strings. MDCStrLookup makes this possible.

Apache Commons Configuration provides a ConfigurationInterpolator class. This class allows new StrLookups to be registered and the values can then be used to merge with both the configuration of Commons Configuration as well as the configuration files it manages.

StrLookup obviously has a dependency on Commons Lang. The Maven pom.xml for slf4j-ext lists this dependency as optional so that those wishing to use other extensions are not required to unnecessarily package the commons lang jar. Therefore, when using MDCStrLookup the dependency for commons-lang must be explicitly declared along with slf4j-ext.

Extended Logger

The XLogger class provides a few extra logging methods that are quite useful for following the execution path of applications. These methods generate logging events that can be filtered separately from other debug logging. Liberal use of these methods is encouraged as the output has been found to

- aid in problem diagnosis in development without requiring a debug session
- aid in problem diagnosis in production where no debugging is possible
- help educate new developers in learning the application.

The two most used methods are the entry() and exit() methods. entry() should be placed at the beginning of methods, except perhaps for simple getters and setters. entry() can be called passing from 0 to 4 parameters. Typically these will be parameters passed to the method. The entry() method logs with a level of TRACE and uses a Marker with a name of "ENTER" which is also a "FLOW" Marker.

The exit() method should be placed before any return statement or as the last statement of methods without a return. exit() can be called with or without a parameter. Typically, methods that return void will use exit() while methods that return an Object will use exit(Object obj). The entry() method logs with a level of TRACE and uses a Marker with a name of "EXIT" which is also a "FLOW"

Marker.

The throwing() method can be used by an application when it is throwing an exception that is unlikely to be handled, such as a RuntimeException. This will insure that proper diagnostics are available if needed. The logging event generated will have a level of ERROR and will have an associated Marker with a name of "THROWING" which is also an "EXCEPTION" Marker.

The catching() method can be used by an application when it catches an Exception that it is not going to rethrow, either explicitly or attached to another Exception. The logging event generated will have a level of ERROR and will have an associated Marker with a name of "CATCHING" which is also an "EXCEPTION" Marker.

By using these extended methods applications that standardize on SLF4J can be assured that they will be able to perform diagnostic logging in a standardized manner.

Note that XLogger instances are obtained to through the `XLoggerFactory` utility class.

The following example shows a simple application using these methods in a fairly typical manner. The `throwing()` method is not present since no Exceptions are explicitly thrown and not handled.

```
package com.test;

import org.slf4j.ext.XLogger;
import org.slf4j.ext.XLoggerFactory;

import java.util.Random;

public class TestService {
  private XLogger logger = XLoggerFactory.getXLogger(TestService.class
      .getName());

  private String[] messages = new String[] { "Hello, World",
      "Goodbye Cruel World", "You had me at hello" };

  private Random rand = new Random(1);

  public String retrieveMessage() {
    logger.entry();

    String testMsg = getMessage(getKey());

    logger.exit(testMsg);
    return testMsg;
  }

  public void exampleException() {
    logger.entry();
    try {
      String msg = messages[messages.length];
      logger.error("An exception should have been thrown");
    } catch (Exception ex) {
      logger.catching(ex);
    }
    logger.exit();
  }

  public String getMessage(int key) {
    logger.entry(key);

    String value = messages[key];

    logger.exit(value);
    return value;
  }

  private int getKey() {
    logger.entry();
    int key = rand.nextInt(messages.length);
    logger.exit(key);
    return key;
  }
}
```

This test application uses the preceding service to generate logging events.

```
package com.test;

public class App {
  public static void main( String[] args )    {
    TestService service = new TestService();
    service.retrieveMessage();
    service.retrieveMessage();
    service.exampleException();
  }
}
```

The configuration below will cause all output to be routed to target/test.log. The pattern for the FileAppender includes the class name, line number and method name. Including these in the pattern are critical for the log to be of value.

```
<?xml version='1.0' encoding='UTF-8'?>
```

```
<configuration>
  <appender name="console" class="ch.qos.logback.core.ConsoleAppender">
    <filter class="ch.qos.logback.classic.filter.LevelFilter">
      <level>ERROR</level>
      <onMatch>ACCEPT</onMatch>
      <onMismatch>DENY</onMismatch>
    </filter>
    <layout class="ch.qos.logback.classic.PatternLayout">
      <Pattern>%d{HH:mm:ss.SSS} %-5level %class{36}:%L %M - %msg%n</Pattern>
    </layout>
  </appender>
  <appender name="log" class="ch.qos.logback.core.FileAppender">
    <File>target/test.log</File>
    <Append>false</Append>
    <layout class="ch.qos.logback.classic.PatternLayout">
      <Pattern>%d{HH:mm:ss.SSS} %-5level %class{36}:%L %M - %msg%n</Pattern>
    </layout>
  </appender>

  <root>
    <level value="trace" />
    <appender-ref ref="log" />
  </root>
</configuration>
```

Here is the output that results from the Java classes and configuration above.

```
00:07:57.725 TRACE com.test.TestService:22 retrieveMessage - entry
00:07:57.738 TRACE com.test.TestService:57 getKey - entry
00:07:57.739 TRACE com.test.TestService:59 getKey - exit with (0)
00:07:57.741 TRACE com.test.TestService:47 getMessage - entry with (0)
00:07:57.741 TRACE com.test.TestService:51 getMessage - exit with (Hello, World)
00:07:57.742 TRACE com.test.TestService:26 retrieveMessage - exit with (Hello, World)
00:07:57.742 TRACE com.test.TestService:22 retrieveMessage - entry
00:07:57.742 TRACE com.test.TestService:57 getKey - entry
00:07:57.743 TRACE com.test.TestService:59 getKey - exit with (1)
00:07:57.745 TRACE com.test.TestService:47 getMessage - entry with (1)
00:07:57.745 TRACE com.test.TestService:51 getMessage - exit with (Goodbye Cruel World)
00:07:57.746 TRACE com.test.TestService:26 retrieveMessage - exit with (Goodbye Cruel World)
00:07:57.746 TRACE com.test.TestService:32 exampleException - entry
00:07:57.750 ERROR com.test.TestService:40 exampleException - catching
java.lang.ArrayIndexOutOfBoundsException: 3
  at com.test.TestService.exampleException(TestService.java:35)
  at com.test.AppTest.testApp(AppTest.java:39)
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
  at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
  at java.lang.reflect.Method.invoke(Method.java:585)
  at junit.framework.TestCase.runTest(TestCase.java:154)
  at junit.framework.TestCase.runBare(TestCase.java:127)
  at junit.framework.TestResult$1.protect(TestResult.java:106)
  at junit.framework.TestResult.runProtected(TestResult.java:124)
  at junit.framework.TestResult.run(TestResult.java:109)
  at junit.framework.TestCase.run(TestCase.java:118)
  at junit.framework.TestSuite.runTest(TestSuite.java:208)
  at junit.framework.TestSuite.run(TestSuite.java:203)
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
  at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
  at java.lang.reflect.Method.invoke(Method.java:585)
  at org.apache.maven.surefire.junit.JUnitTestSet.execute(JUnitTestSet.java:213)
  at org.apache.maven.surefire.suite.AbstractDirectoryTestSuite.executeTestSet(AbstractDirectoryTestSuite.java:140)
  at org.apache.maven.surefire.suite.AbstractDirectoryTestSuite.execute(AbstractDirectoryTestSuite.java:127)
  at org.apache.maven.surefire.Surefire.run(Surefire.java:177)
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
  at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
  at java.lang.reflect.Method.invoke(Method.java:585)
  at org.apache.maven.surefire.booter.SurefireBooter.runSuitesInProcess(SurefireBooter.java:338)
  at org.apache.maven.surefire.booter.SurefireBooter.main(SurefireBooter.java:997)
00:07:57.750 TRACE com.test.TestService:42 exampleException - exit
```

Simply changing the root logger level to DEBUG in the example above will reduce the output considerably.

```
00:28:06.004 ERROR com.test.TestService:40 exampleException - catching
java.lang.ArrayIndexOutOfBoundsException: 3
  at com.test.TestService.exampleException(TestService.java:35)
  at com.test.AppTest.testApp(AppTest.java:39)
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
  at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
  at java.lang.reflect.Method.invoke(Method.java:585)
  at junit.framework.TestCase.runTest(TestCase.java:154)
  at junit.framework.TestCase.runBare(TestCase.java:127)
  at junit.framework.TestResult$1.protect(TestResult.java:106)
  at junit.framework.TestResult.runProtected(TestResult.java:124)
  at junit.framework.TestResult.run(TestResult.java:109)
  at junit.framework.TestCase.run(TestCase.java:118)
  at junit.framework.TestSuite.runTest(TestSuite.java:208)
  at junit.framework.TestSuite.run(TestSuite.java:203)
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
  at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
  at java.lang.reflect.Method.invoke(Method.java:585)
  at org.apache.maven.surefire.junit.JUnitTestSet.execute(JUnitTestSet.java:213)
  at org.apache.maven.surefire.suite.AbstractDirectoryTestSuite.executeTestSet(AbstractDirectoryTestSuite.java:140)
  at org.apache.maven.surefire.suite.AbstractDirectoryTestSuite.execute(AbstractDirectoryTestSuite.java:127)
  at org.apache.maven.surefire.Surefire.run(Surefire.java:177)
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
  at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
  at java.lang.reflect.Method.invoke(Method.java:585)
  at org.apache.maven.surefire.booter.SurefireBooter.runSuitesInProcess(SurefireBooter.java:338)
  at org.apache.maven.surefire.booter.SurefireBooter.main(SurefireBooter.java:997)
```

Event Logging

The EventLogger class provides a simple mechanism for logging events that occur in an application. While the EventLogger is useful as a way of initiating events that should be processed by an audit Logging system, it does not implement any of the features an audit logging system would require such as guaranteed delivery.

The recommended way of using the EventLogger in a typical web application is to populate the SLF4J MDC with data that is related to the entire lifespan of the request such as the user's id, the user's ip address, the product name, etc. This can easily be done in a servlet filter where the MDC can also be cleared at the end of the request. When an event that needs to be recorded occurs an EventData object should be created and populated. Then call EventLogger.logEvent(data) where data is a reference to the EventData object.

```java
import org.slf4j.MDC;
import org.apache.commons.lang.time.DateUtils;

import javax.servlet.Filter;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.FilterChain;
import javax.servlet.http.HttpSession;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.TimeZone;

public class RequestFilter implements Filter
{
  private FilterConfig filterConfig;
  private static String TZ_NAME = "timezoneOffset";

  public void init(FilterConfig filterConfig) throws ServletException {
    this.filterConfig = filterConfig;
  }

  /**
   * Sample filter that populates the MDC on every request.
   */
  public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse,
                       FilterChain filterChain) throws IOException, ServletException {
    HttpServletRequest request = (HttpServletRequest)servletRequest;
    HttpServletResponse response = (HttpServletResponse)servletResponse;
    MDC.put("ipAddress", request.getRemoteAddr());
    HttpSession session = request.getSession(false);
    TimeZone timeZone = null;
    if (session != null) {
      // Something should set this after authentication completes
      String loginId = (String)session.getAttribute("LoginId");
      if (loginId != null) {
        MDC.put("loginId", loginId);
      }
      // This assumes there is some javascript on the user's page to create the cookie.
      if (session.getAttribute(TZ_NAME) == null) {
        if (request.getCookies() != null) {
          for (Cookie cookie : request.getCookies()) {
            if (TZ_NAME.equals(cookie.getName())) {
              int tzOffsetMinutes = Integer.parseInt(cookie.getValue());
              timeZone = TimeZone.getTimeZone("GMT");
              timeZone.setRawOffset((int)(tzOffsetMinutes * DateUtils.MILLIS_PER_MINUTE));
              request.getSession().setAttribute(TZ_NAME, tzOffsetMinutes);
              cookie.setMaxAge(0);
              response.addCookie(cookie);
            }
          }
        }
      }
    }
    MDC.put("hostname", servletRequest.getServerName());
    MDC.put("productName", filterConfig.getInitParameter("ProductName"));
    MDC.put("locale", servletRequest.getLocale().getDisplayName());
    if (timeZone == null) {
      timeZone = TimeZone.getDefault();
    }
    MDC.put("timezone", timeZone.getDisplayName());
    filterChain.doFilter(servletRequest, servletResponse);
    MDC.clear();
  }

  public void destroy() {
  }
}
```

Sample class that uses EventLogger.

```
import org.slf4j.ext.EventData;
import org.slf4j.ext.EventLogger;

import java.util.Date;
import java.util.UUID;

public class MyApp {

  public String doFundsTransfer(Account toAccount, Account fromAccount, long amount) {
    toAccount.deposit(amount);
    fromAccount.withdraw(amount);
    EventData data = new EventData();
    data.setEventDateTime(new Date());
    data.setEventType("transfer");
    String confirm = UUID.randomUUID().toString();
    data.setEventId(confirm);
    data.put("toAccount", toAccount);
    data.put("fromAccount", fromAccount);
    data.put("amount", amount);
    EventLogger.logEvent(data);
    return confirm;
  }
}
```

The EventLogger class uses a Logger named "EventLogger". EventLogger uses a logging level of INFO. The following shows a configuration using Logback.

```
<configuration>
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <layout class="ch.qos.logback.classic.PatternLayout">
      <Pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</Pattern>
    </layout>
  </appender>

  <appender name="events" class="ch.qos.logback.core.ConsoleAppender">
    <layout class="ch.qos.logback.classic.PatternLayout">
      <Pattern>%d{HH:mm:ss.SSS} %X - %msg%n</Pattern>
    </layout>
  </appender>

  <logger name="EventLogger" additivity="false">
    <level value="INFO"/>
    <appender appender-ref="events"/>
  </logger>

  <root level="DEBUG">
    <appender-ref ref="STDOUT" />
  </root>

</configuration>
```

Adding logging with Java agent

NOTE: BETA RELEASE, NOT PRODUCTION QUALITY

Quickstart for the impatient:

1. Use Java 5 or later.
2. Download slf4j-ext-1.7.12.jar and javassist.jar, and put them both in the same directory.
3. Ensure your application is properly configured with slf4j-api-1.7.12.jar and a suitable backend.
4. Instead of "java ..." use "java --javaagent:PATH/slf4j-ext-1.7.12.jar=time,verbose,level=info ..."
   (replace PATH with the path to the jar)
5. That's it!

In some applications logging is used to trace the actual execution of the application as opposed to log an occasional event. One approach is using the extended logger to add statements as appropriately, but another is to use a tool which modifies compiled bytecode to add these statements! Many exist, and the one included in slf4j-ext is not intended to compete with these, but merely provide a quick way to get very basic trace information from a given application.

Java 5 added the Java Instrumentation mechanism, which allows you to provide "Java agents" that can inspect and modify the byte code of the classes as they are loaded. This allows the original class files to remain unchanged, and the transformations done on the byte codes depend on the needs at launch time.

Given the well-known "Hello World" example:

```
public class HelloWorld {
  public static void main(String args[]) {
    System.out.println("Hello World");
  }
}
```

a typical transformation would be similar to: (imports omitted)

```
public class LoggingHelloWorld {
  final static Logger _log = LoggerFactory.getLogger(LoggingHelloWorld.class.getName());

  public static void main(String args[]) {
    if (_log.isInfoEnabled()) {
      _log.info("> main(args=" + Arrays.asList(args) + ")");
    }
    System.out.println("Hello World");
    if (_log.isInfoEnabled()) {
      _log.info("< main()");
    }
  }
}
```

which in turn produces the following result when run similar to "java LoggingHelloWorld 1 2 3 4":

```
1 [main] INFO LoggingHelloWorld - > main(args=[1, 2, 3, 4])
Hello World
1 [main] INFO LoggingHelloWorld - < main()
```

The same effect could have been had by using this command (with the relative path to javassist.jar and slf4j-ext-1.7.12.jar being ../jars):

```
java -javaagent:../jars/slf4j-ext-1.7.12.jar HelloWorld 1 2 3 4
```

How to use

The javaagent may take one or more options separated by comma. The following options are currently supported:

level=X

The log level to use for the generated log statements. X is one of "info", "debug" or "trace". Default is "info".

time

Print out the current date at program start, and again when the program ends plus the execution time in milliseconds.

verbose

Print out when a class is processed as part of being loaded

ignore=X:Y:...

(Advanced) Provide full list of colon separated prefixes of class names NOT to add logging to. The default list is "org/slf4j/:ch/qos/logback/:org/apache/log4j/". This does not override the fact that a class must be able to access the slf4j-api classes in order to do logging, so if these classes are not visible to a given class it is not instrumented.

Some classes may misbehave when being rendered with "object.toString()" so they may be explicitly disabled in the logback configuration file permanently. For instance the ToStringBuilder in the Apache Jakarta commons lang package is a prime candidate for this. For logback add this snippet to logback.xml:

```
<logger name="org.apache.commons.lang.builder" level="OFF" />
```

Note: These are not finalized yet, and may change.

Locations of jar files

The javassist library is used for the actual byte code manipulation and must be available to be able to add any logging statements. slf4j-ext-1.7.12 has been configured to look for the following:

- "javassist-3.4.GA.jar" relatively to slf4j-ext-1.7.12.jar as would be if Maven had downloaded both from the repository and slf4j-ext-1.7.12.jar was referenced directly in the Maven repository in the "-javaagent"-argument.
- "javassist-3.4.GA.jar" in the same directory as slf4j-ext
- "javassist.jar" in the same directory as slf4j-ext

A warning message is printed if the javassist library was not found by the agent, and options requiring byte code transformations will not work.

Misc notes

- A java agent is not invoked on any classes already loaded by the class loader.
- Exceptions in the java agent that would normally have been printed, may be silently swallowed by the JVM.
- The javaagent only logs to System.err.
- The name of the logger variable is fixed (to a value unlikely to be used) so if that name is already used, a failure occurs. This should be changed to determine an unused name and use that instead.
- Empty methods are not instrumented (an incorrect check for an interface). They should be

(The agent is an adaption of the java.util.logging version described in http://today.java.net/pub/a/today/2008/04/24/add-logging-at-class-load-time-with-instrumentation.html)