Приложение 2

Log4J

В процессе функционирования сложных приложений необходимо вести журнал сообщений и ошибок, чтобы была возможность отследить время входа и выхода пользователя из системы, возникновение исключительных ситуаций, сбоев и т. д. Существуют различные API регистрации сообщений и ошибок, среди которых на данный момент можно выделить следующие.

1. **Log4J.** Apache Log4J был первым регистратором. Изначально обладает качественной архитектурой, в следствие чего быстро занял доминирующие позиции и применяется в большинстве промышленных приложений. Разработан в рамках проекта Jakarta Apache.
2. **java.util.logging.** Пакет появился в JavaSE в версии 1.4 в 2001 году после появления Log4J. Возможностей фреймворк предоставляет меньше, чем Log4J, тем не менее, у **java.util.logging** есть некоторое преимущество — он является частью JavaSE.
3. **Apache Commons Logging.** Предназначен для абстрагирования разработчика от конкретной библиотеки логгирования. Он предоставляет некоторый унифицированный интерфейс, транслируя его вызовы в использование конкретных возможностей фреймворков. Commons Logging абстрагирует следующие фреймворки: Log4J, java.util.logging, Avalon LogKit, Lumberjack.
4. **SLF4J.** Simple Logging Facade for Java абстрагирует еще больше технологий логгирования, чем Commons Logging.
5. **Logback.** Расширение Log4J с добавлением к нему новых возможностей.

Любой регистратор событий состоит из трех элементов:

— собственно регистрирующего — logger;
— направляющего вывод — appender;
— форматирующего вывод — layout.

В итоге logger регистрирует и направляет вывод события в пункт назначения, определяемый направляющим элементом, в формате, заданном форматирующим элементом.

Log4j

В современном практическом программировании представляет основной инструмент журналирования событий. Формирует журнал сообщений (отладочных, информационных, системных, security, сообщений об ошибках). Log4j

можно загрузить по адресу: **http://logging.apache.org/log4j/**. Перед использованием необходимо зарегистрировать библиотеку **log4j-[***версия***].jar** в приложении.

Logger

Основным элементом API регистрации событий и ошибок является регистратор **org.apache.log4j.Logger**, который управляет регистрацией сообщений. Вывод регистратора может быть направлен на консоль, в файл, базу данных, GUI-компонент или сокет. Это компонент приложения, принимающий и выполняющий запросы на запись в регистрационный журнал. Apache Log4J поддерживает несколько способов конфигурации. Позволяет управлять своим поведением во время исполнения.

Каждый класс приложения может иметь свой собственный logger или быть прикреплен к общему для всего приложения. Регистраторы образуют иерархию, как и пакеты Java. Каждый логгер имеет имя, описывающее иерархию, к которой он принадлежит. Разделителем является точка. Принцип полностью аналогичен формированию имени пакета в Java.

Регистратор может быть создан или получен с помощью статического метода **getLogger(String name)** или **getLogger(Class name)**, где **name** — имя пакета или класса. На вершине иерархии находится корневой регистратор. Он всегда существует и у него нет имени. Ссылку на корневой регистратор можно получить статическим методом **getRootLogger()**.

У каждого регистратора есть уровень сообщения по возрастанию (**TRACE**, **DEBUG**, **INFO**, **WARN**, **ERROR**, **FATAL, OFF**), который управляет выводом сообщений. Для вывода сообщений конкретного уровня используются методы **trace()**, **debug()**, **info()**, **warn()**, **error()**, **fatal()**. Чтобы вывести информацию о возникшем исключении в качестве второго параметра, в перечисленные методы нужно передать объект класса, производного от **Throwable**. Для вывода сообщения необходимо, чтобы уровень выводимого сообщения был не ниже, чем уровень регистратора (**TRACE** < **DEBUG** < **INFO** < **WARN** < **ERROR** < **FATAL** < **OFF**), т. е. если уровень регистратора **INFO**, то вызов **logger.debug("message")** не даст никакого эффекта, т. к. **DEBUG** < **INFO**. Уровень регистратора можно указать с помощью метода **setLevel(Level level)**, который принимает объект класса **Level**, содержащий одноименные константы для каждого уровня. Если уровень регистратора не указывается, то применяется уровень, унаследованный от его родителя. Уровень корневого регистратора **DEBUG**. Таким образом, сообщения, выводимые с уровнем ниже установленного, в лог не попадут. И в этом заключается основное преимущество — можно вставлять в программный код вывод информации на различных уровнях (об ошибках — на уровне **ERROR**, о нормальном ходе выполнения — на уровне **INFO**, отладочную — на уровне **DEBUG**), а потом гибко регулировать, что именно будет выводиться.

Некоторые общие методы для вывода сообщений:

log(Priority priority, Object message, Throwable t) — выводит сообщения указанного уровня с информацией об исключительной ситуации **t**;

log(Priority priority, Object message) — выводит сообщения указанного уровня.

Appender

Вывод регистратора может быть направлен в различные места назначения: файл, консоль и т. д. Каждому из них соответствует класс, реализующий интерфейс **org.apache.log4j.Appender**. Кроме того, вывод в базу данных можно произвести с помощью класса **JDBCAppender**, в журнал событий ОС — **NTEventLogAppender**, на SMTP-сервер — **SMTPAppender**.

Если логгер — это та точка, откуда уходят сообщения в коде, то аппендер — это та точка, куда они приходят в конечном итоге. Например, файл или консоль. Список таких точек, поддерживаемых Log4J:

— консоль;
— файлы (несколько различных типов);
— JDBC;
— темы (topics) JMS;
— NT Event Log;
— SMTP;
— Сокет;
— Syslog;
— Telnet;
— любой **java.io.Writer** или **java.io.OutputStream**.

Существует возможность написать собственный класс аппендер и использовать его.

Основными аппендерами, использующимися наиболее широко, являются файловые аппендеры. Их есть несколько типов:

— **org.apache.log4j.FileAppender**
— **org.apache.log4j.RollingFileAppender**
— **org.apache.log4j.DailyRollingFileAppender**

Логгеры связываются с аппендерами в соотношении «многие ко многим» — у одного логгера может быть несколько аппендеров, а к одному аппендеру может быть привязано несколько логгеров. Важно понимать, что аппендеры наследуются от родительских логгеров.

Уровень логирования наследуется (или устанавливается) независимо от аппендера. Иначе, если на логгере root сконфигурирован вывод в вывод в консоль с уровнем **ERROR**, а на дочернем логгере — в файл с уровнем **INFO**, то вывод в дочерний логгер с уровнем **INFO** попадет и в файл, и в консоль.

Существует возможность отказаться от наследования аппендеров. Для этого логгеру надо выставить свойство **additivity** в **false**, по умолчанию оно выставлено

в **true**. Конкретные детали описания и использования наиболее употребимых аппендеров будут приведены ниже.

Layout

Вывод регистратора может иметь различный формат. Каждый формат представлен классом, производным от **Layout**. Все методы класса **Layout** предназначены только для создания подклассов.

Для конфигурирования формата вывода используются наследники класса **org.apache.log4j.Layout**:

— **org.apache.log4j.SimpleLayout**
— **org.apache.log4j.HTMLLayout**
— **org.apache.log4j.xml.XMLLayout**
— **org.apache.log4j.PatternLayout**

Установить **Layout** для **FileAppender** или **ConsoleAppender** можно с помощью метода **setLayout(Layout layout)** или передать его в конструкторы перечисленных классов.

org.apache.log4j.SimpleLayout — наиболее простой вариант. На выходе читается уровень вывода и сообщение.

org.apache.log4j.HTMLLayout — данный компоновщик форматирует сообщения в виде HTML-страницы.

org.apache.log4j.xml.XMLLayout — формирует сообщения в виде XML.

org.apache.log4j.PatternLayout и **org.apache.log4j.EnhancedPatternLayout —** используют шаблонную строку для форматирования выводимого сообщения.

Например:

```
<layout class="org.apache.log4j.PatternLayout" >
          <param name="ConversionPattern"
                    value="%d{dd.MM.yyyy HH:mm:ss} [%t] %-5p %c - %m%n"/>
</layout>
```

Данный форматтер принимает параметром **ConversionPattern** — шаблон вывода лога, где

%d{DATE} — выводит дату-время. В скобках можно указать собственный формат вывода. Также применимы именованые шаблоны, а именно **DATE**, **ISO8601** и **ABSOLUTE**. Последний содержит формат **HH:mm:ss,SSS**, подходящий для логов с кратким сроком хранения. По умолчанию дата будет выведена в формате **ISO8601**;

%t — выводит имя потока, выводящего сообщение, для однопоточного приложения будет выводить **main**;

%5p — выводит уровень лога (**ERROR**, **DEBUG**, **INFO** и пр.), где цифра указывает число выводимых символов, если символов меньше, то сообщение будет дополнено пробелами;

%c{6} — категория с числом выдаваемых уровней. Категорией в общем случае будет имя класса с пакетом. Обычно это строка, где уровни разделены точками. По умолчанию без {} будет выводить полный путь к корню проекта. Верхний уровень при значении **1** будет выводить только имя класса;

%M — имя метода, в котором произошел вызов записи в лог;

%L — номер строки, в которой произошел вызов записи в лог;

%m — собственно сообщение, передаваемое в лог;

%n — перевод строки.

Конфигурация

Перед использованием Log4j его необходимо сконфигурировать. Конфигурирование осуществляется способами — через файл свойств, через xml-файл, через программное конфигурирование и по умлчанию. Преимущество следут отдать способам, использующим конфигурационные файлы, как динамичным и легко изменяемым. Более удобным для понимания считается xml-конфигурирование.

Простейший конфигурационный файл **log4j.xml** может находиться в корне проекта в виде:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
        <appender name="TxtAppender" class="org.apache.log4j.FileAppender">
                <param name="Encoding" value="UTF-8" />
                <param name="File" value="logs/log.txt" />
                <layout class="org.apache.log4j.SimpleLayout" />
        </appender>
        <logger name="by.bsu">
                <level value="debug" />
        </logger>
        <root>
                <appender-ref ref="TxtAppender" />
        </root>
</log4j:configuration>
```

В итоге создан файловый аппендер с именем **TxtAppender** для записи в файл **logs/log.txt** с поддержкой кодировки UTF-8 и упрощенным компоновщиком. Сконфигурирован корневой логгер уровня **debug**. Аппендер **FileAppender** добавляет данные в файл до бесконечности, поэтому файл может быть очень большого размера, что значительно усложняет его чтение при превышении разумных размеров. Запись в большой текстовый файл также может замедлять работу системы. В чистом виде практически не используется. Он является основой для других, предлагая ключевые способы взаимодействия с файлами. Поддерживает следующие свойства: **append** — дописывать

существующий файл или каждый раз создавать новый, **bufferedIO** — буферизовать ли вывод в файл (по умолчанию **false**), **file** — имя файла, **encoding** — кодировка вывода, **bufferSize** — размер буфера (по умолчанию 8192).

Подключается данная конфигурация вызовом метода **doConfigure()** класса **org.apache.log4j.xml.DomConfigurator**:

```
new DOMConfigurator().doConfigure("log4j.xml", LogManager.getLoggerRepository());
```

В приведенном ниже примере производятся регистрация и вывод как обычных информационных сообщений о выполненных действиях, так и сообщений о возникающих ошибках (попытке вычисления факториала отрицательного числа).

/ # 1 # регистратор ошибок # DemoLog.java */*

```
package by.bsu.log4j.base;
import org.apache.log4j.LogManager;
import org.apache.log4j.Logger;
import org.apache.log4j.xml.DOMConfigurator;
public class DemoLog {
          static {
               new DOMConfigurator().doConfigure("log4j.xml", LogManager.getLoggerRepository());
          }
          static Logger logger = Logger.getLogger(DemoLog.class);
          public static void main(String[ ] args) {
                    try {
                              factorial(9);
                              factorial(-3);
                    } catch (IllegalArgumentException e) {
                              // вывод сообщения уровня ERROR
                              logger.error("negative argument: ", e);
                    }
          }
          public static int factorial(int n) {
                    if (n < 0) {
                              throw new IllegalArgumentException(
                                                  "argument " + n +" less than zero");
                    }
                    // вывод сообщения уровня DEBUG
                    logger.debug("Argument n is " + n);
                    int result = 1;
                    for (int i = n; i >= 1; i--) {
                              result *= i;
                    }
                    // вывод сообщения уровня INFO
                    logger.info("Result is " + result);
                    return result;
          }
}
```

Вывод регистратора **"by.bsu.log4j.base.DemoLog"**, в файл **log.txt** будет следующим:

DEBUG - Argument n is 9
INFO - Result is 362880
ERROR - negative argument:
java.lang.IllegalArgumentException: argument -3 less than zero
at by.bsu.log4j.base.DemoLog.factorial(DemoLog.java:22)
at by.bsu.log4j.base.DemoLog.main(DemoLog.java:14)

Порции выводимых данных в Log4J называются сообщениями.

Для вывода одновременно в текстовый файл и в файл в виде XML необходимо добавить следующую информацию о новом аппендере с простым XML компоновщиком:

```
<appender name="XMLAppender" class="org.apache.log4j.FileAppender">
          <param name="File" value="logs/log.xml" />
          <layout class="org.apache.log4j.XMLLayout"/>
</appender>
```

и добавить строку о появлении еще одного корневого логгера:

```
<root>
          <appender-ref ref="TxtAppender" />
          <appender-ref ref="XMLAppender" />
</root>
```

Таким образом, логгеров может быть несколько и использовать их можно не только для дублирования информации в разные точки сохранения, но и независимо друг от друга.

Файл **log.xml** будет содержать следующую информацию:

```
<log4j:event logger="by.bsu.log4j.base.DemoLog" timestamp="1355923596119" level="DEBUG"
thread="main">
<log4j:message><![CDATA[Argument n is 9]]></log4j:message>
</log4j:event>
<log4j:event logger="by.bsu.log4j.base.DemoLog" timestamp="1355923596121" level="INFO"
thread="main">
<log4j:message><![CDATA[Result is 362880]]></log4j:message>
</log4j:event>
<log4j:event logger="by.bsu.log4j.base.DemoLog" timestamp="1355923596122" level="ERROR"
thread="main">
<log4j:message><![CDATA[negative argument: ]]></log4j:message>
<log4j:throwable><![CDATA[java.lang.IllegalArgumentException: argument -3 less than zero
          at by.bsu.log4j.base.DemoLog.factorial(DemoLog.java:27)
          at by.bsu.log4j.base.DemoLog.main(DemoLog.java:19)
]]></log4j:throwable>
</log4j:event>
```

Если использовать вместо компоновки **SimpleLayout**

```
<layout class="org.apache.log4j.xml.SimpleLayout"/>
```

компоновку **PatternLayout**, приведенную выше в виде

```
<layout class="org.apache.log4j.PatternLayout" >
         <param name="ConversionPattern"
                  value="%d{dd.MM.yyyy HH:mm:ss} [%t] %-5p %c - %m%n"/>
</layout>
```

то в файл **log.txt** будет выведено

```
19.12.2012 17:21:59 [main] DEBUG by.bsu.log4j.base.DemoLog - Argument n is 9
19.12.2012 17:21:59 [main] INFO  by.bsu.log4j.base.DemoLog - Result is 362880
19.12.2012 17:21:59 [main] ERROR by.bsu.log4j.base.DemoLog - negative argument:
java.lang.IllegalArgumentException: argument -3 less than zero
        at by.bsu.log4j.base.DemoLog.factorial(DemoLog.java:27)
        at by.bsu.log4j.base.DemoLog.main(DemoLog.java:19)
```

Следует привести еще один востребованный способ конфигурирования с помощью файлов **properties**:

```
PropertyConfigurator.configure("log4j.properties");
```

Пример простейшего файла **log4j.properties** для вывода ссобщений на консоль:

```
log4j.rootLogger=debug, stdout
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.target=System.out
log4j.appender.stdout.layout=org.apache.log4j.SimpleLayout
```

У компоновщика **HTMLLayout** есть два свойства — **Title** и **LocationInfo**, задающие заголовок HTML-документа и режим вывода информации о точке, где сгенерировано сообщение: имя файла и номер строки в нем. По умолчанию **LocationInfo** имеет значение **false**.

Файл **log4j.properties** для вывода сообщений в html-файл:

```
log4j.rootLogger = DEBUG, html
log4j.appender.html=org.apache.log4j.FileAppender
log4j.appender.html.File=logs/log.html
log4j.appender.html.layout=org.apache.log4j.HTMLLayout
log4j.appender.html.layout.Title=HTML Layout Example
log4j.appender.html.layout.LocationInfo=true
```

У данного компоновщика есть существенный недостаток: формат HTML требует корректного закрытия документа. А при генерации вывода непрерывно добавляются сообщения, т. е. строки в таблицу, и автоматического закрытия документа не происходит.

Log4j поддерживает конфигурирование по умолчанию без всяких файлов, только при этом все сообщения будут выводиться на консоль:

```
BasicConfigurator.configure();
```

Те же действия можно выполнить, не прибегая к конфигурационным файлам непосредственно в коде приложения. С помощью метода **addAppender(Appender newAppender)** класса **Logger** можно добавить **Appender** к регистратору. Один

регистратор может иметь несколько элементов **Appender**. Вывод на консоль осуществляется с помощью класса **ConsoleAppender**. Класс **FileAppender** используется для вывода сообщений в файл. Для установки файла, в который будет выполняться вывод, нужно передать имя файла в конструктор **FileAppender(Layout layout, String filename)** или метод **setFile(String filename)**. По умолчанию любые сообщения, записываемые в файл, будут добавляться к уже имеющимся. Но с помощью метода **setAppend(boolean append)** это можно отменить, сбросив флаг **append**.

```
ConsoleAppender consAppender = new ConsoleAppender(new SimpleLayout());
FileAppender xmlAppender = new FileAppender(new XMLLayout(), "logs/log.xml");
          logger.addAppender(consAppender);
          logger.addAppender(xmlAppender);
FileAppender fileAppender = new FileAppender(new SimpleLayout(), "logs.log.txt");
          logger.addAppender(fileAppender);
          logger.setLevel(Level.DEBUG);
```

Любой вывод, сделанный в регистраторе, будет направлен всем его предкам. Чтобы этого избежать, в регистраторе следует установить флаг аддитивности с помощью метода **setAdditivity(boolean additive)**. В этом случае вывод будет направлен всем его предкам вплоть до регистратора с установленным флагом аддитивности.

Программная конфигурация логгеров используется редко из-за своей громоздкости и отсутствии гибкост при изменении настроек.

При конфигурировании Log4j для веб-проекта следует поместить код процесса конфигурации в метод **init()** сервлета:

```
String prefix = getServletContext().getRealPath("/");
String filename = getInitParameter("init_log4j");
if (filename != null) {
          PropertyConfigurator.configure(prefix + filename);
}
```

где путь вместе с именем файла вида **WEB-INF/classes/log4j.properties** объявляется в параметрах инициализации сервлета.

Аппендеры RollingFileAppender и DailyRollingFileAppender

Аппендер **RollingFileAppender** позволяет создавать новый файл по достижении определенного размера. «Создавать» — означает изменить имя текущего файла путем добавления ему расширения «.0» и открыть следующий. По достижении им максимального размера — первому вместо расширения «.0» выставляется «.1», текущему — «.0», открывается следующий. А именно:

logs.txt.2
logs.txt.1
logs.txt.0
logs.txt

Максимальный размер файла и максимальный индекс, устанавливаемый сохраняемым предыдущим файлам, задаются свойствами **maximumFileSize** и **maxBackupIndex** соответственно. Если индекс должен быть превышен — файл не переименовывается, а удаляется. Таким образом, всегда будет в наличии больше определенного количества файлов, каждый из которых не больше определенного объема.

Пример применения **RollingFileAppender** в **log4j.properties**:

```
log4j.rootLogger=info, fileout
log4j.appender.fileout =org.apache.log4j.RollingFileAppender
log4j.appender.fileout.file=logs/log.txt
log4j.appender.fileout.file.maxBackupIndex=10
log4j.appender.fileout.maximumFileSize=15KB
log4j.appender.fileout.layout=org.apache.log4j.PatternLayout
log4j.appender.fileout.layout.ConversionPattern=%p %d %t %c - %m%n
```

При этом каждый раз при превышении файлом размера, указанного в свойстве **maximumFileSize**, будет создаваться новый файл.

Аппендер **DailyRollingFileAppender** в отличие от **RollingFileAppender**, создающего новый файл по достижении определенного размера, **DailyRollingFileAppender** создает файл с определенной частотой, которая зависит от шаблона, указанного в конфигурации:

'.'yyyy-MM — раз в месяц,
'.'yyyy-ww — раз в неделю,
'.'yyyy-MM-dd — раз в день,
'.'yyyy-MM-dd-a — раз в полдня,
'.'yyyy-MM-dd-HH — раз в час,
'.'yyyy-MM-dd-HH-mm — раз в минуту.

В кавычках в начале шаблона указан символ, который будет использоваться как разделитель между значением даты-времени и именем файла. При создании к имени файла в конце приписываются текущие дата и время, отформатированные согласно указанному шаблону (с помощью класса **java.text.SimpleDateFormat**).

log_time.txt
log_time.txt.2013-01-30-14-29
log_time.txt.2013-01-30-14-30

```
log4j.rootCategory=INFO, fileout
log4j.appender.fileout = org.apache.log4j.DailyRollingFileAppender
log4j.appender.fileout.File=logs/log_time.txt
log4j.appender.fileout.Append = true
log4j.appender.fileout.DatePattern ='.'yyyy-MM-dd-HH-mm
log4j.appender.fileout.layout = org.apache.log4j.PatternLayout
log4j.appender.fileout.layout.ConversionPattern = %d{yyyy-MM-dd HH:mm:ss} %c{3} [%p] %m%n
```

Этот аппендер может быть весьма удобен в случае, если организована автоматическая архивация лога. Наличие в имени файла метки времени делает его по определению уникальным.

Фильтры

Необходимость записывать не все сообщения, а только удовлетворяющее некоторым условиям, реализуется с использованием фильтров. Стандартные фильтры бывают двух видов: **LevelMatchFilter** — пишет сообщения заданного уровня и **LevelRangeFilter** — пишет сообщения в диапазоне уровней.

Например:

```
LevelMatchFilter filter = new LevelMatchFilter();
filter.setLevelToMatch("INFO");
```

в конфигурационном файле в теле тега **<appender>** вставляется тег **<filter>** в виде:

```
<filter class="org.apache.log4j.varia.LevelRangeFilter">
          <param name="LevelMax" value="WARN"/>
          <param name="LevelMin" value="INFO"/>
</filter>
```

Для создания собственного фильтра требуется создать подкласс класса **org.apache.log4j.spi.Filter**. Каждому фильтру соответсвует **Appender**. Отношение между ними может быть установлено как многие ко многим. При вызове логгирующего метода генерируется объект события **LoggingEvent**, которое и передается в метод **int decide()** фильтра. Метод, в свою очередь, должен вернуть результат своей работы в виде одного из значений: **Filter.ACCEPT** — разрешение записать сообщение в лог, **Filter.NEUTRAL** — передать дальше по цепочке фильтров или записать, если фильтр последний, **Filter.DENY** — запрет записи сообщения и обрыв цепочки.

Пусть существует журнал событий, который фиксирует сообщения, связанные с коллекцией монет. Но логгировать события следует только для монет, чьи идентификационные номера больше некоторого значения, попадают в определенный диапазон или удовлетворяют другим условиям.

Чтобы решить поставленную задачу, следует в реализации метода **decide()** указать условия разрешения/игнорирования фиксации события.

/ # 2 # собственный фильтр # CoinFilter.java */*

```
package by.bsu.log.filter;
import org.apache.log4j.spi.Filter;
import org.apache.log4j.spi.LoggingEvent;
import by.bsu.log.entity.Coin;
public class CoinFilter extends Filter {
public int decide(LoggingEvent event) {
                    int result = Filter.NEUTRAL;
                    Object object = event.getMessage();
                    if (object instanceof Coin) {
                              Coin coin = (Coin) object;
                              int id = coin.getId();
```

```
                    // игнорировать, если id меньше 1000, записывать – если больше
                              result = id < 1_000 ? Filter.DENY : Filter.ACCEPT;
                    }
                    return result;
          }
}
```

Из экземпляра события извлекается объект-сообщение, на основе содержимого которго и осуществляется решение.

/ # 3 # бизнес-класс # Coin.java */*

```
package by.bsu.log.entity;
public class Coin {
          private int id;
          private int value;
          private String currencyName;
          public Coin() {
          }
          public Coin(int id, int value, String currencyName) {
                    this.id = id;
                    this.value = value;
                    this.currencyName = currencyName;
          }
          public String getCurrencyName() {
                    return this.currencyName;
          }
          public int getId() {
                    return this.id;
          }
          public int getValue() {
                    return this.value;
          }
}
```

Как экземпляр класса **Coin** попадает в событие? Методы класса **Logger**, отвечающие за запись, принимают сообщение в виде экземпляра класса **Object**. В общем случае туда передаются объекты типа **String**, информация из которых выводится в место назначения в виде логов. Но при этом генерируется событие **LoggingEvent**, в которое и помещается экземпляр-сообщение.

```
Coin coin = // init object
logger.info(coin);
```

Программисту только и остается с помощью фильтра определить реакцию на контент сообщения, какого бы типа он не был.

Конфигурационный файл **log4j.xml** будет выглядеть:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
```

```
        <renderer renderedClass="by.bsu.log.entity.Coin"
                  renderingClass="by.bsu.log.renderer.CoinRenderer"/>
        <appender name="ConsAppender" class="org.apache.log4j.ConsoleAppender">
                  <layout class="org.apache.log4j.PatternLayout">
                            <param name="ConversionPattern" value="%t %-5p %c{5} - %m%n" />
                  </layout>
                  <filter class="by.bsu.log.filter.CoinFilter" />
        </appender>
        <logger name="by.bsu">
                  <level value="debug" />
                  <appender-ref ref="ConsAppender" />
        </logger>
</log4j:configuration>
```

Способ вывода сообщения и его внешний вывод при использовании фильтра также должен быть реализован разработчиком. После возвращения методом **decide()** значения **Filter.ACCEPT** управление передается переопределенному методу **doRender()** интерфейса **ObjectRenderer**, который определяет формат и содержание выводимого сообщения.

/ # 4 # построение сообщения для регистратора # CoinRenderer.java */*

```
package by.bsu.log.renderer;
import org.apache.log4j.or.ObjectRenderer;
import by.bsu.log.entity.Coin;
public class CoinRenderer implements ObjectRenderer {
          public String doRender(Object obj) {
                    StringBuilder builder = new StringBuilder(32);
                    if (obj instanceof Coin) {
                              Coin coin = (Coin) obj;
                              String currency = coin.getCurrencyName();
                              int id = coin.getId();
                              int value = coin.getValue();
                              builder.append( id + ": " + value + "(" + currency + ")");
                    }
                    return builder.toString();
          }
}
```

/ # 5 # регистратор ошибок # FilterDemoLog.java */*

```
package by.bsu.log.base;
import java.util.ArrayList;
import org.apache.log4j.LogManager;
import org.apache.log4j.Logger;
import org.apache.log4j.xml.DOMConfigurator;
import by.bsu.log.entity.Coin;
public class FilterDemoLog {
          static {
              new DOMConfigurator().doConfigure("log4j.xml", LogManager.getLoggerRepository());
          }
```

```
        private static Logger logger = Logger.getLogger(FilterDemoLog.class);
        public static void main(String args[]) {
                ArrayList<Coin> list = new ArrayList<Coin>() {
                        {
                                this.add(new Coin(956, 1, "$"));
                                this.add(new Coin(3462, 10, "руб"));
                                this.add(new Coin(758, 2, "тенге"));
                                this.add(new Coin(2101, 5, "zl"));
                        }
                };
                for (Coin coin: list) {
                        logger.info(coin);
                }
        }
}
```

В консоль будут выведены сообщения только о двух монетах из четырех.

main INFO by.bsu.log.base.FilterDemoLog - 3462: 10(руб)
main INFO by.bsu.log.base.FilterDemoLog - 2101: 5(zl)

Из фильтров можно построить цепочку фильтров.