Лабораторная работа №2

Code Convention. Логирование.

Цель работы: изучение правил документа code convention, изучение основ логирования в Java при помощи библиотеки log4j.

Основные понятия Code Convention

Code Convention (стандарт кодирования) – набор правил и соглашений, используемых при написании исходного кода на некотором языке программирования. Наличие общего стиля программирования облегчает понимание и поддержание исходного кода, написанного более чем одним программистом, а также упрощает взаимодействие нескольких человек при разработке программного обеспечения.

Стандарт оформления кода обычно принимается и используется некоторой группой разработчиков программного обеспечения для единообразного оформления совместно используемого кода.

Образцом для стандарта кодирования может стать набор соглашений, принятых в какой-либо распространённой печатной работе по языку, широко применяемая библиотека или API.

Реже разработчик языка выпускает подробные рекомендации по кодированию. Например, выпущены стандарты кодирования на C# от Microsoft и на Java от Sun. Предложенная разработчиком или принятая в общеизвестных источниках манера кодирования в большей или меньшей степени дополняется и уточняется в корпоративных стандартах.

Обычно, стандарт оформления кода описывает:

- способы выбора названий и используемый регистр символов для имён переменных и других идентификаторов:
  - запись типа переменной в её идентификаторе (венгерская нотация) и
  - регистр символов (нижний, верхний, «верблюжий», «верблюжий» с малой буквы), использование знаков подчёркивания для разделения слов;
- стиль отступов при оформлении логических блоков — используются ли символы табуляции, ширина отступа;
- способ расстановки скобок, ограничивающих логические блоки;
- использование пробелов при оформлении логических и арифметических выражений;

- стиль комментариев и использование документирующих комментариев.
- ограничение размера кода по горизонтали (чтобы помещался на экране) и вертикали (чтобы весь код файла держался в памяти), а также функции или метода в размер одного экрана.

Причины принятия конвенций

1. В процессе разработки на чтение кода тратится больше времени, чем на его написание.
2. 80% времени жизни ПО находится в поддержке.
3. Почти никогда ПО не поддерживается автором.
4. Совместная работа над кодом, имеющим единый стиль написания, значительно легче.
5. Правильное форматирование улучшает понимание работы кода и облегчает поиск ошибок.

Цели принятия конвенций

Следование конвенциям форматирования позволяет:

1. Точно представлять логическую структуру кода.
2. Улучшить читаемость кода.
3. Выдерживать изменения кода.

Основные понятия логирования

Логирование – ведение протокола, или протоколирование, т.е хронологическая запись с различной (настраиваемой) степенью детализации сведений о происходящих в системе событиях (ошибки, предупреждения, сообщения), обычно в файл.

Назначение

Логи предназначены, как правило, для разработчиков чтобы:

- определить, что же делает система прямо сейчас, не прибегая к помощи отладчика, т.к. это иногда не оправдано;
- провести «расследование» обстоятельств, которые привели к определённому состоянию системы (например, падению или багу);
- проанализировать, на что тратится больше времени/ресурсов, т.е. профилирование.

Code Convention for Java

Имена файлов

Имена файлов Java должны совпадать с именем класса, описанного в нем. Единственно допустимое расширение файла, содержащего Java код - .java.

Скомпилированный байт-код - .class

ReadMe файл в репозитории должен иметь имя README.md.

Файл лицензии должен иметь имя LICENSE.md.

Файл .gitignore должен иметь имя .gitignore.

Содержимое файла с исходным кодом Java

Файл с исходным кодом Java должен содержать следующее:

1. Комментарий в начале – Имя класса, версия, дата, копирайт.
2. Указание имени пакета и импорты.
3. Javadoc комментарии к классу.
4. Единственный `public` класс или интерфейс. Связанные с ним `private` классы могут быть помещены в этот же файл **после** `public` класса/интерфейса.

Длина строк

Следует избегать строк длиннее чем 80 символов и разбивать их на несколько. Отступ 4 пробела, а не табуляция.

Разбиение строки

Перенос на следующую строку может быть:

- после запятой;
- до оператора;
- предпочтительно более высокоуровневый перенос;
- согласовать отступ с началом выражения;
- если предыдущее правило дает плохой результат, сделать отступ 8 пробелов;

При переносе, для остатка строки используется двойной отступ от начала предыдущей строки.

Примеры допустимых переносов строк:

```
function(longExpression1, longExpression2, longExpression3,
```

```
        longExpression4, longExpression5);

longName1 = longName2 * (longName3 + longName4 - longName5)
        + 4 * longname6;
```

Комментарии

В большинстве случаев код должен быть самодокументированным, комментарии следует добавлять только при необходимости.

Отдельные комментарии предшествуют коду и имеют такой же отступ. Начинаются с заглавной буквы.

Комментарии в конце строки должны быть однострочными и имеют отступ в 1 пробел от кода. Начинаются с прописной буквы.

Примеры:

```
if (condition) {
    // Комментарий
    foobar(); // комментарий
}
```

Javadoc комментарии

Javadoc комментарии стоит указывать для `public` элементов и для нетривиальных `protected`, `package`, `private` элементов.

Следует разбивать текст на параграфы с помощью тега `<p>...</p>`. Параграфы отделяются друг от друга пустой строкой.

Порядок следования элементов представлен ниже:

```
/**
 * <p>Описание метода.</p>
 *
 * @param value описание параметра метода.
 * @param value2 очень очень очень очень очень очень очень очень
 *               очень очень длинное описание параметра.
 * @return описание значения.
 * @throws КлассИсключения1 описание условий.
 * @throws КлассИсключения2 описание условий.
 * @see полное имя класса с учетом пакетов
 */
@Override
void methodWithAnnotations(int value, int value2) {}
```

Объявления переменных

Предпочтительнее писать одно объявление в строке.

После указания типа данных следует 1 пробел и имя переменной.

Обязательно один тип данных в строке.

Для указания массива квадратные скобки помещаются после имени.

Примеры:

```
int foo;
int foobar[] = new int[5];
```

Объявление блоков кода

При объявлении блоков кода следует использовать фигурные скобки. Открывающая скобка остается на той же строке что и определение, а закрывающая - на отдельной строке и имеет такой же отступ как и объявление блока. Перед открывающей скобкой ставится пробел.

```
void func() {
    // Тело метода
}
```

При пустом теле допускается оставлять обе скобки на одной строке:

```
void doNothing() {}
```

Инициализация

Инициализация по возможности должна происходить при объявлении переменных.

Определение классов и интерфейсов

Классы определяются следующим образом:

```
class Sample extends Object {
    int ivar1;
    int ivar2;

    Sample(int i, int j) {
        ivar1 = i;
        ivar2 = j;
    }

    int emptyMethod() {}

}
```

Таким образом:

- Там где необходимы пробелы по синтаксису Java указывается **один** пробел.
- Нет пробела между именем метода и " (".
- Открывающая "{" в конце строки определения.
- Закрывающая "}" на отдельной строке с отступом.
- Определения методов выделяются пустыми строками..

Выражения

Каждая строка должна содержать одно выражение.

Избегайте двойных присваиваний.

Значение `return` и `throw`, метки для `break` и `continue` не заключается в скобки.

Примеры:

```
a++;
if (a == 1) {
    return a;
} else if (b == 2) {
    throw new Exception();
}
```

Конструкции языка

```
if (condition) {
    expr1;
} else if (condition2) {
    expr2;
} else {
    expr3;
}

// Допустимо писать в одну строку выражения, которые осуществляют выход
// из текущей области видимости
if (cond1) return;
if (cond2) break;
if (cond3) continue;
if (cond4) throw new Exception();

for (initialization; condition; update) {
    expr;
}

// Для цикла с пустым телом
for (initialization; condition; update) {}

while (condition) {
    expression;
}

while (condition) {}

do {
    expr1;
} while (condition);

switch (c) {
    case value1:
        expr1;
        break;

    case value2:
        expr2;
        /* При необходимости "проваливания" необходимо явно указать это
         * следующим комментарием
         */
        // no break!

    default:
        expr;
        // Не стоит опускать break даже для default ветки
```

```
        break;
}

try {
    expr;
    return 0; // не вернет 0, переходит к блоку finally
} catch (Exception e) {
    showError();
} finally {
    doSomethingElse();
    /* При необходимости указания return в блоке finally нужно учитывать,
     * каким образом он работает и указать в комментарии его особенность.
     */
    return 5; // метод всегда возвращает 5
}
```

Пустые строки

Следует ставить пустые строки между логически связанными блоками кода:

- Между секциями файла
- Между описаниями классов и интерфейсов
- Между методами
- Между локальными переменными и операторами
- Перед комментарием
- Между логическими секциями в методе

Пробелы

Следует ставить пробелы:

- вокруг операторов
- после ключевых слов
- после `,`
- после `;` внутри `for (init; cond; upd)`
- после приведений типов, например: `int a = (int) 1.5f;`

Не следует ставить больше одного пробела (везде).

Именование

Пакеты именуются прописными буквами, слитно.

Классы, интерфейсы, перечисления именуются по типу "с заглавной буквы каждое слово": `ApplicationModule`.

Методы именуются с прописной буквы, но каждое последующее слово с заглавной: `getApplicationModule`.

Переменные именуются таким же образом, как и методы.

При использовании общепринятых аббревиатур, заглавной делается только первая буква: `Html`, `Url`...

Константы именуются заглавными буквами, слова через "_": `MAX_ITEMS_COUNT`.

Выбор имен

Для интерфейсов выбирается имя без префиксов или суффиксов.

Реализации интерфейса имеют суффикс - имя интерфейса. Например:

```
interface Repository {
    //...
}
class DatabaseRepository implements Repository {
    //...
}
```

Для переменных также не используются префиксы/суффиксы. Для `boolean` переменных не используются в том числе префиксы `is`, `not`.

При именовании коллекций множественное число только у последнего слова: `userProfiles`...

Callback методы именуются, начиная с префикса `on`. Для обозначения успешного/неудачного завершения используются суффиксы `Success`, `Failure`: `onLoadingSuccess`, `onLoadingFailure`.

Тестовые классы следует именовать также, как и классы которые они тестируют с суффиксом `Test`. Например для класса `MyClass` - `MyClassTest`.

Тестовые методы именуются следующим образом:

```
public void testИмяМетода_условие_ожидаемыйРезультат();
```

Например:

```
public void testOnExecute_whenUserProfileIsNull_returnNull();
```

Вызов методов

При вызове метода после имени **отсутствует** пробел.

Статические члены

При обращении к статическим членам следует использовать имя класса, а не объекта:

`MyClass.staticMethod();` вместо `obj.staticMethod();`

Круглые скобки

Для сложных выражений следует использовать скобки для указания приоритета операций, даже если это не влияет на очередность выполнения: `(5 * 8) + (9 / 2.0) / 8`

Условие для тернарного оператора "if" следует брать в скобки: `(x >= 0) ? x : -x;`

Полная документация находится по адресу http://www.oracle.com/technetwork/java/codeconvtoc-136057.html.

Библиотека log4j

Log4j – фреймворк для упрощения реализации рутинных операций по логированию некоторых событий, которые происходят во время работы Вашего приложения, написанного на Java. Что значит рутинных – при отслеживании работы приложения, т.е. трассировке, выполнения конструкторов, методов, блоков обработки критических ситуаций, разработчики сталкиваются с одними и теми же операциями, а именно:

- Выбор хранилища – консоль, файл, СУБД;
- Конфигурация хранилища – именование хранилища, объем хранилища, путь к хранилищу, сохранение конфигурации;
- Форматирование записей журнала – дата/время, класс/метод и т.д., гибкое форматирование.

Все эти операции фреймворк позволяет автоматизировать, а конфигурацию параметров хранить в одном файле конфигураций отдельно.

Таким образом, от пользователей фреймворка необходимо лишь:

- загрузить библиотеку фреймворка и подключить её к проекту;
- создать конфигурационный файл с параметрами логирования;
- создать Logger – объект журнала (лога) в своем приложении;
- воспользоваться методами для записи в журнал. Он представляет собой набор API, с помощью которых разработчики могут вставлять в свой код выражения, выводящие некоторую информацию (отладочную, информационную, сообщения об ошибках и т.д.), и конфигурировать этот вывод с помощью внешних конфигурационных файлов.

Полная документация приведена по адресу https://logging.apache.org/log4j/2.x/

Загрузка и подключение log4j в IDE Eclipse

Загрузить библиотек можно на странице http://logging.apache.org/log4j/2.x/download.html. Последняя версия на данный момент 2.8.1.

После разархивирования необходимо подключить файлы log4j-api-2.8.1.jar и log4j-core-2.8.1.jar к проекту. Как правило, все используемые сторонние библиотеки располагаются в отдельной папке. Поэтому в коневой папке проекта необходимо создать каталог lib и скопировать туда вышеперечисленные файлы. Затем в меню «Project -> Properties» выбрать пункт «Java Build Path» и перейти на вкладку «Libraries».

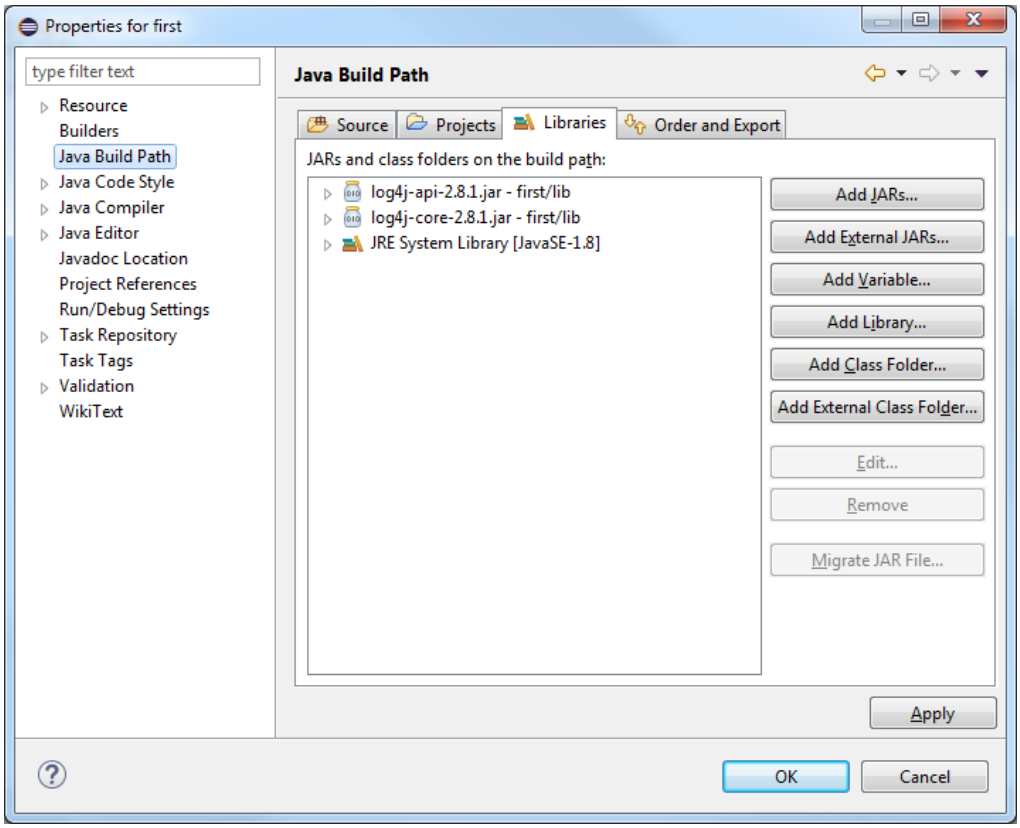

Рисунок 1. Подключение библиотеки log4j.

Выбрать файлы можно нажав кнопку «Add JARs».

Также необходимо перейти на вкладку «Order and Export» и активировать флажки напротив указанных файлов.

Далее нужно импортировать пакет командой import org.apache.logging.log4j.*;

Создание конфигурационного файла

Существует множество способов конфигурации библиотеки, но наиболее распространенными являются использование файлов `*.properties` или `*.xml`. Файл обязательно должен называться `log4j2.properties` (`log4j2.xml`) и располагаться в

директории classpath проекта, т.е. в корневой директории файлов *.class вашего проекта (*не исходных кодов *.java*). В IDE Eclipse это папка /имя_проекта/bin.

Пример файла конфигурации log4j.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
  <!-- Секция аппендеров -->
  <Appenders>
      <!-- Консольный аппендер -->
    <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout pattern="%d %-5p [%t] %C{2} (%F:%L) - %m%n"/>
    </Console>
    <!-- Файловый аппендер -->
    <File name="file" fileName="log.log">
      <PatternLayout>
        <Pattern>%d %p %c{1.} [%t] %m %ex%n</Pattern>
      </PatternLayout>
    </File>
  </Appenders>
  <!-- Секция логгеров -->
  <Loggers>
    <Logger name="ru.bstu.kit.Veretennikov.Two" level="debug"/>
    <Root level="all">
      <AppenderRef ref="Console"/>
      <AppenderRef ref="file"/>
    </Root>
  </Loggers>
</Configuration>
```

Объект Logger

Любой регистратор событий состоит из трех элементов:

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

Регистрирующий элемент представляет собой создаваемый программистом объект класса Logger или LogManager, позволяющий инициализировать запись в лог тех или иных сообщений:

```
Logger log = LogManager.getLogger(Two.class);
```

Уровни логирования в log4j:

- FATAL - произошла фатальная ошибка - у этого сообщения наивысший приоритет
- ERROR - в программе произошла ошибка
- WARN - предупреждение в программе что-то не так
- INFO - информация.

- DEBUG - детальная информация для отладки
- TRACE– трассировка всех сообщений в указанный аппендер

OFF< TRACE< DEBUG< INFO< WARN< ERROR< FATAL< ALL

Такая иерархия означает что если установлен уровень логирования DEBUG, то лог будет содержать и все вышестоящие уровни (INFO, WARN, ERROR, FATAL).

Данный объект для поддержки указанных уровней имеет соответствующие методы .trace(), .debug(), .info(), .warn(), .error(), .fatal() которые передают в лог сообщения с соответствующем уровнем.

В случае, если сообщение для лога строится из нескольких строк, или вообще делается какая то операция, то лучше перед этим сделать проверку на то что данный уровень логирования в данный момент включён, методом isУровеньEnabled (например isDebugEnabled ()). Это стоит проверять для того чтобы не выполнять лишние операции когда результат нам просто далее не понадобится. Если пишется просто строка то нет смысла проверять уровень, внутри log4j и так все проверится и не пройдёт в лог если он выключен. Если в качестве параметра записывающего метода передать exception, то log4j выведет stacktrace в лог.

Appender'ы указывают возможные места назначения выводимого в лог сообщения: 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, консоль, база данных, SMTP, Telnet, SysLog, Сокет и др.

Наиболее распространенными классами appender являются:

- org.apache.log4j.ConsoleAppender,
- org.apache.log4j.FileAppender – добавляет данные в один файл. До бесконечности, т.е. без каких-либо ограничений по размеру. Потому этот аппендер сам по себе практически не используется. Он является базой для остальных, предоставляя общие средства работы с файлами.

- org.apache.log4j.RollingFileAppender – позволяет ротировать файл по достижении определенного размера. "Ротировать" означает, что текущему файлу приписывается расширение ".0" и открывается следующий. По достижении им максимального размера – первому вместо расширения ".0" выставляется ".1", текущему – ".0", открывается следующий. И так далее. 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 имеется не больше определенного количества файлов, каждый из которых не больше определенного объема. Этот тип аппендеров очень часто используемый.
- org.apache.log4j.DailyRollingFileAppender – ротирует файл с определенной частотой, зависящей от формата используемой даты. При ротации к имени файла в конце приписываются текущие дата и время, отформатированные согласно указанному шаблону (с помощью класса java.text.SimpleDateFormat). В кавычках в начале шаблона указан символ, который будет использоваться как разделитель между значением даты/времени и именем файла. Наличие в имени файла временной метки делает его по определению уникальным – лог не потеряется, как это может произойти с обычным ротирующим аппендером.

Для одного логгера может быть указано множество аппендеров. Их параметры задаются в файле конфигурации, но могут быть изменены и программно при обращении к методам вышеперечисленных классов-аппендеров.

Формат вывода записи лога определяется параметрами классов, производных от родительского Layout. Все методы класса Layout предназначены только для создания подклассов.

Наиболее распространенные виды форматов:

- org.apache.log4j.SimpleLayout - наиболее простой вариант. На выходе отображается 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. Формат чем-то напоминает `printf` – тот же знак `'%'`, после которого (возможно) идет модификатор формата и дальше символ, обозначающий тип выводимых данных. Кроме таких служебных комбинаций в строку шаблона можно вставлять любые символы, что позволяет еще более гибко конфигурировать лог.

Параметры форматирования лога:

Опция	Значение, выводимое в лог
`c`	**Категория сообщения**. После символа категории в фигурных скобках может следовать указание – сколько частей имени категории выводить. Они отсчитываются с конца, что логично – это позволяет отсечь длинное имя пакета. Т.е., например, при имени категории *ru.skipy.logging.tests.Log4JTest* комбинация `%c{3}` приведет к выводу в лог *logging.tests.Log4JTest* (три части имени с конца). Если такого указания нет – имя выводится целиком. Еще один вариант сокращения – запись вида `%c{1.2.3.}`. Означает она, что от первой части остается одна буква, от второй – две, от третьей – три. На оставшиеся части распространяется последнее значение. Последняя часть имени выводится целиком. Т.е. из имени *ru.skipy.logging.tests.Log4JTest* форматом `%c{1.2.1.}` останется *r.sk.l.t.Log4JTest* – одна буква, две, далее опять одна. Можно задать еще и символ, которым будут замещаться убранные символы: `%c{1*.2#.1$}` даст результат *r*.sk#.l$.t$.Log4JTest*. При длинных именах категорий такой формат может оказаться удобным.
`C`	**Полное имя класса, в котором сгенерировано сообщение** После имени класса также может идти указание на то, сколько частей имени выводить – полностью аналогично опции `'%c'`.
`d`	**Дата и/или время** Выводит в лог текущие дату и/или время. В фигурных скобках после данной опции указывается формат даты – либо шаблон `java.text.SimpleDateFormat`, либо один из предустановленных – `DATE`, `ABSOLUTE` или `ISO8601`. Документация рекомендует использовать предопределенные форматы вместо собственных шаблонов – под них разработаны специальные классы для более оптимального форматирования, чем это делает `java.text.SimpleDateFormat`.
`F`	**Имя файла, в котором было сгенерировано сообщение**. Не путайте с именем класса. В данном случае в лог выведется именно имя файла, в общем случае не совпадающее с именем класса, например, для любых внутренних классов.
`l`	**Полная информация о точке генерации сообщения**. Содержит имя класса, имя метода, имя файла и строку, в которой было сгенерировано сообщение. **Важно!** Фактически эта опция является аналогом следующей конструкции: `%C.%M(%F:%L)`. Генерация **каждой** из частей в этом наборе – **крайне** медленная процедура. Ну и вся комбинация, естественно, быстрой не будет. Поэтому опции `%l` необходимо категорически избегать в режиме промышленной эксплуатации. В то же время в процессе отладки она может оказать

	неоценимую помощь.
`L`	**Номер строки, в которой было сгенерировано сообщение**. Имеется в виду номер строки в файле.
`m`	**Сообщение** То самое сообщение, которое передается в метод логгера.
`M`	**Имя метода, в котором было сгенерировано сообщение**.
`n`	**Перевод строки** Переводит в логе строку. Это необходимо, иначе все сообщения будут писаться в одну строку.
`p`	**Приоритет сообщения**. Выводит уровень логирования для сообщения.
`r`	**Количество миллисекунд с момента инициализации системы логирования**. Аналог формата даты `RELATIVE` компоновщика *TTCCLayout*. Может использоваться вместо даты, если есть такая необходимость.
`t`	**Имя потока**. Выводит имя потока, в котором сгенерировано сообщение.
`x`	**Вложенный диагностический контекст (NDC)** Выводит связанный с текущим потоком вложенный диагностический контекст.
`X`	**Ассоциативный диагностический контекст (MDC)**. Выводит связанный с текущим потоком ассоциативный диагностический контекст. После опции в фигурных скобках **должно** идти имя ключа, по которому выбирается значение из контекста: `%X{username}` – вывод из контекста имени пользователя, если оно там есть.
`%`	**Знак процента**. Поскольку знак '%' является частью формата, а необходимость в его выводе периодичеси присутстует, конструкция `'%%'` выводит в лог знак '%'.

С учетом баланса между требованиями производительности и объемом информации, которого достаточно для анализа логов, в промышленном режиме рекомендовано использование следующих опций: **%c, %d, %m, %n, %p, %t, %x, %X, %%**. Остальные – **%C, %F, %l, %L, %M** – способны вызвать сильное падение производительности**.**

Данный компоновщик поддерживает, кроме всего прочего, позиционное форматирование. Означает оно, что под каждую опцию можно выделить некоторое место – задать минимальный и максимальный размер значения, а также выравнивание, если значение меньше минимальной выделенной области. Модификаторы форматирования задаются между символом '%' и опцией. На примере опции `%c` рассмотрим действие модификаторов:

Модификатор	Выравнивание	Минимальная ширина	Максимальная ширина	Действие
`%10c`	вправо	10	нет	Отводит минимум 10 символов под имя категории, если длина значения меньше – выравнивает его по правому краю поля
`%-10c`	влево	10	нет	Отводит минимум 10 символов

				под имя категории, если длина значения меньше – выравнивает его по левому краю поля
`%.20c`	нет	нет	20	Отводит максимум 20 символов под имя категории, если длина значения больше – обрезает с начала, оставляя указанное количество символов. Поскольку длина значения не может быть меньше предопределенной, о выравнивании говорить не приходится.
`%10.20c`	вправо	10	20	Отводит минимум 10 и максимум 20 символов под имя категории, если длина значения меньше – выравнивает его по правому краю поля, если больше – обрезает с начала, оставляя 20 символов.
`%-10.20c`	влево	10	20	Отводит минимум 10 и максимум 20 символов под имя категории, если длина значения меньше – выравнивает его по левому краю поля, если больше – обрезает с начала, оставляя 20 символов.

Смысл ограничения минимальной ширины становится ясным из следующих примеров логов:

```
11:31:32,342 Thread-1 ERROR ru.skipy.tests.audit.LoadTest - Check in 344ms: GlobalID=2
11:31:32,358 Thread-17 WARN ru.skipy.tests.ServiceLoadTest - Check in 156ms: GlobalID=8
11:31:32,378 Thread-2 INFO ru.skipy.tests.trace.ServiceLoadTrace - Check in 328ms: GlobalID=3
11:31:35,358 Thread-44 DEBUG ru.skipy.tests.parallel.ext.ServiceParallelLoadTest - Check in 250ms: GlobalID=5
11:31:36,637 Thread-503 INFO ru.skipy.tests.ServiceLoadTest - Check in 219ms: GlobalID=6
11:31:37,846 Thread-59 INFO ru.skipy.tests.extract.Extractor - Check in 94ms: GlobalID=10
11:31:39,072 Thread-86 DEBUG ru.skipy.tests.ServiceLoadTest - Check in 188ms: GlobalID=7
11:31:41,309 Thread-10 INFO ru.skipy.tests.back.BackLoaderInfo - Check in 47ms: GlobalID=11
```

```
11:31:32,342 Thread-1   ERROR    ru.skipy.tests.audit.LoadTest - Check in 344ms: GlobalID=2
11:31:32,358 Thread-17  WARN    ru.skipy.tests.ServiceLoadTest - Check in 156ms: GlobalID=8
11:31:32,378 Thread-2   INFO  ipy.tests.trace.ServiceLoadTrace - Check in 328ms: GlobalID=3
11:31:35,358 Thread-44  DEBUG llel.ext.ServiceParallelLoadTest - Check in 250ms: GlobalID=5
11:31:36,637 Thread-503 INFO    ru.skipy.tests.ServiceLoadTest - Check in 219ms: GlobalID=6
11:31:37,846 Thread-59  INFO  ru.skipy.tests.extract.Extractor - Check in 94ms: GlobalID=10
11:31:39,072 Thread-86  DEBUG   ru.skipy.tests.ServiceLoadTest - Check in 188ms: GlobalID=7
11:31:41,309 Thread-10  INFO  .skipy.tests.back.BackLoaderInfo - Check in 47ms: GlobalID=11
```

Рисунок 2. Примеры логов.

Второй вариант намного более удобен для чтения.

API библиотеки log4j

В классическом для логгеров стиле методы делятся на два типа: совпадающие с названием уровня логирования и методы log, принимающие уровень логирования в качестве параметра. Первые имеют вид:

```
log.info(mapMessage);
log.info(object);
log.info(stringMessage);
log.info(marker, mapMessage);
log.info(marker, object);
log.info(marker, stringMessage);
log.info(object, throwable);
log.info(stringMessage, throwable);
log.info(stringMessageFormat, args);
log.info(marker, mapMessage, throwable);
log.info(marker, object, throwable);
log.info(marker, stringMessageFormat, args);
log.info(marker, stringMessage, throwable);
log.throwing(throwable);
```

Аналогичные методы существуют и для остальных уровней логирования.

Методы log в log4j2 выглядят так:

```
log.log(Level.INFO, mapMessage);
log.log(Level.INFO, object);
log.log(Level.INFO, stringMessage);
log.log(Level.INFO, marker, mapMessage);
log.log(Level.INFO, marker, object);
log.log(Level.INFO, marker, stringMessage);
log.log(Level.INFO, object, throwable);
log.log(Level.INFO, stringMessageFormat, args);
log.log(Level.INFO, stringMessage, throwable);
log.log(Level.INFO, marker, mapMessage, throwable);
log.log(Level.INFO, marker, object, throwable);
log.log(Level.INFO, marker, stringMessageFormat, args);
log.log(Level.INFO, marker, stringMessage, throwable);
log.throwing(Level.INFO, throwable);
```

Пример

```
package ru.bstu.kit.Veretennikov;
import org.apache.logging.log4j.*;

public class Two {

	static final Logger Logger = LogManager.getLogger(Two.class);

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		Logger.debug("GO");
		System.out.println("Hello!!!");
	}

}
```

Задание к лабораторной работе:

Привести код из 1-й лабораторной работы к требованиям конвенции.
Так же необходимо в 1-ю лабораторную добавить логирование основных действий приложения с использованием библиотеки log4j. Логи должны выводится в файл и на консоль. Логи, выводимые в файл нужно разделять по уровням INFO, DEBUG, ERROR:
в один файл записывать логи INFO, DEBUG, WARNING, ERROR и FATAL
во второй - только WARNING, ERROR и FATAL
Маска лога должна содержать следующую информацию:
<уровень> <дата> <время> (короткое имя класса) - <текст лога>
<error.printStackTrace> (для ERROR и FATAL)

например:

```
ERROR 2010-09-01 10:01:02,525  ProcThread1 (RMIServiceServer    ) - Unable to look up client
under the RMI name "//localhost/MyService_and_MPRMIServiceClient"
java.rmi.NotBoundException: MyService_and_MPRMIServiceClient
        at sun.rmi.registry.RegistryImpl.lookup(RegistryImpl.java:106)
        at sun.rmi.registry.RegistryImpl_Skel.dispatch(Unknown Source)
        at sun.rmi.server.UnicastServerRef.oldDispatch(UnicastServerRef.java:375)
        at sun.rmi.server.UnicastServerRef.dispatch(UnicastServerRef.java:240)
        at sun.rmi.transport.Transport$1.run(Transport.java:153)
        at java.security.AccessController.doPrivileged(Native Method)
        at sun.rmi.transport.Transport.serviceCall(Transport.java:149)
        at sun.rmi.transport.tcp.TCPTransport.handleMessages(TCPTransport.java:466)
        at sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.run(TCPTransport.java:707)
        at java.lang.Thread.run(Thread.java:595)
```

Каждый отчет должен содержать:

1. Заголовок лабораторной работы (название и цель работы).
2. Фамилия, инициалы и группа студента.
3. Задание к лабораторной работе.
4. Краткие теоретические сведения.
5. Описание алгоритмов, функций, примененных решений.
6. Результаты выполнения программ.
7. Исходный код программ.
8. Выводы о проделанной работе.