Логирование в Java / quick start

* [Java](https://habr.com/ru/hub/java/)

В ходе моей работы в компании DataArt я, в числе прочего, занимаюсь менторской деятельностью. В частности это включает в себя проверку учебных заданий сделанных практикантами. В последнее время в заданиях наметилась тенденция «странного» использования логеров. Мы с коллегами решили включить в текст задания ссылку на статью с описанием java logging best practices, но оказалось, что такой статьи в которой бы просто и без лишних деталей на практике объяснялось бы как надо писать в лог на Java, вот так вот с ходу не находится.   
  
Данная статья не содержит каких-то откровений, в ней не рассматриваются тонкости какого либо из многочисленных java logging frameworks. Здесь рассказываю как записать в лог так, чтобы это не вызвало удивления у Ваших коллег, основная цель написания включить ее в список обязательного чтения для практикантов. Если все еще интересно, читайте дальше

**Что нужно логировать**

Разумеется, логировать все подряд не стоит. Иногда это и не нужно, и даже опасно. Например, если залогировать чьи-то личные данные и это каким-то образом всплывет на поверхность, будут реальные проблемы, особенно на проектах, ориентированных на Запад. Но есть и то, что **логировать обязательно**:

1. **Начало/конец работы приложения.** Нужно знать, что приложение действительно запустилось, как мы и ожидали, и завершилось так же ожидаемо.
2. **Вопросы безопасности.** Здесь хорошо бы логировать попытки подбора пароля, логирование входа важных юзеров и т.д.
3. Некоторые **состояния приложения**. Например, переход из одного состояния в другое в [бизнес процессе](http://www.businessstudio.com.ua/images/cust/proc4.jpg).
4. Некоторая **информация для дебага**, с соответственным уровнем логирования.
5. **Некоторые SQL скрипты.** Есть реальные случаи, когда это нужно. Опять-таки, умелым образом регулируя уровни, можно добиться отличных результатов.
6. **Выполняемые нити(Thread)** могут быть логированы в случаях с проверкой корректной работы.

**Популярные ошибки в логировании**

Нюансов много, но можно выделить несколько частых ошибок:

1. **Избыток логирования.** Не стоит логировать каждый шаг, который чисто теоретически может быть важным. Есть правило: **логи могут нагружать работоспособность не более, чем на 10%.** Иначе будут проблемы с производительностью.
2. **Логирование всех данных в один файл.** Это приведет к тому, что в определенный момент чтение/запись в него будет очень сложной, не говоря о том, что есть ограничения по размеру файлов в определенных системах.
3. **Использование неверных уровней логирования.** У каждого уровня логирования есть четкие границы, и их стоит соблюдать. Если граница расплывчатая, можно договориться какой из уровней использовать.

Несколько разъяснений.

* Весь код примеров использует java.util.logging framework. Вопрос «Какой из фреймворков логирования ниболее кошерен» я оставлю за кадром. Скажу только что до java.util.logging проще всего дотянуться ибо он уже идет вместе с JRE и на самом деле рассказанное в данной статье с минимальными косметическими правками верно для подавляющего большинства систем логирования.
* В целом рецепты приведенные в данной статье не являются единственно верными, есть моменты о которых можно поспорить, но в целом эти рецепты используются многие годы, многими разработчиками, во многих проектах и они достаточно хороши чтобы им следовать если у Вас нет каких-то совсем уже серьезных возражений.
* В статье не рассматриваются такие «продвинутые» топики как:
  + Конфигурирование уровней для отдельных логеров
  + Форматирования логов
  + Асинхронное логирование
  + Создание собственных уровней логирования в Log4J
  + Контекстное логирование
  + И многое другое
* Слово logging я пишу по русски как логирование с одной буквой «г» в основном потом, что такой вариант перевода чаще встречается
* Советы, что, с каким уровнем логировать я пожалуй тоже оставлю за кадром т.к. тут все сильно зависит от приложения, условий эксплуатации, отношений с заказчиком и т.п. тонких вещей.

Пример №1

Хорошо

public class SomeClass {

private static Logger log = Logger.getLogger(SomeClass.class.getName());

public void someMethod()

{

log.info("Some message");

}

...

1. Логер это статическое поле класса инициализируемое при загрузке класса, имеет простое, короткое имя, важно чтобы во всех Ваших классах переменная логера называлась одинаково (это диктуется общим правилом, одинаковые вещи в программе должны делаться одинаковым образом).
2. В качестве имени логера я использую имя класса, на самом деле это не единственный способ, можно пытаться организовать какую-то свою иерархию логирования (например transport layer/app layer для подсистем имеющих дело с обменом данными), но как показывает практика выдумывать и главное потом неукоснительно следовать такой иерархии крайне сложно, а вариант с именами логеров совпадающими с именами классов весьма хорош и используется в 99% проектов
3. Здесь для записи в лог я использую короткий метод .info, а не более общий метод .log, так много лаконичнее
4. Имя логера берется как SomeClass.class.getName(), а не как «com.dataart.demo.java.logging.SomeClass», оба способа по идее одинаковы, но первый защищает Вас от сюрпризов при рефакторинге имени/пакета класса

Плохо

public class SomeClass {

public void someMethod()

{

Logger.getLogger("com.dataart.demo.java.logging.SomeClass").log(Level.INFO,"Some message");

}

...

По сути тоже самое но букв больше и читается не так легко.

Замечание между примерами

Вы наверное обратили внимание, что все сообщения в примерах на английском языке. Это не случайно. Дело в том, что даже если все-все кто работает и будет работать с Вашим кодом говорят по русски, есть вероятность, что Вам придется просматривать лог сообщения на удаленном компьютере например через ssh при этом в большом количестве случаев Вы увидите примерно такое сообщение "????, ???? ?????!!!!" (я безусловно знаю что через ssh можно протащить русские буквы, но вот почему-то далеко не всегда все оказывается настроенным должным образом).  
Или даже на локальной машине в cmd вы можете увидеть что вот такое:  
INFO: ╨Ъ╨░╨║╨╛╨╡-╤В╨╛ ╤Б╨╛╨╛╨▒╤Й╨╡╨╜╨╕╨╡ ╨▓ ╨╗╨╛╨│  
  
С этим безусловно тоже можно бороться. Но не всегда легко объяснить заказчику на том конце телефонной трубки, как сделать так чтобы вместо крякозябр были видны русские буквы.   
Совет: Пишите лог сообщения на английском языке, ну или в крайнем случае латинскими буквами. 

Методы логгера могут принимать в качестве аргументов строковые сообщения, шаблоны сообщений, исключения, ресурсы локализованных текстовок сообщений, а также, начиная с Java 8, поставщиков строковых сообщений:

*Строковое сообщение* String stringMessage = "Сообщение";

*Строковое сообщение с параметрами* String stringMessageFormat ="Сообщение {0}";

*Исключение* Throwable throwable = new Throwable();

*ResourceBundle хранящий сообщения* ResourceBundle resourceBundle = ResourceBundle.getBundle("logging.jul.bundle");

*Поставщик сообщений* Supplier<String> stringMessageSupplier = ()->"Сообщение";

Выделяется две группы методов: 1. название которых соответствует уровню логгирования и 2. методы log, loggp, logrb, принимающие уровень логгирования в качестве параметра с типом Level. Первая группа содержит методы двух типов: принимающих строковое сообщение или поставщика строковых сообщений:

log.info(stringMessage);

log.info(stringMessageSupplier);

Вторая группа методов имеет следующие вариации:

*// Вывести сообщение с указанием уровня логгирования*

log.log(new LogRecord(Level.INFO, stringMessage));

log.log(Level.INFO, stringMessage);

log.log(Level.INFO, stringMessageSupplier);

log.log(Level.INFO, stringMessageFormat, args);

log.log(Level.INFO, stringMessage, throwable );

log.log(Level.INFO, throwable, stringMessageSupplier);

*// Вывести сообщение с указанием уровня логгирования, класса и метода*

log.logp(Level.INFO, "ClassName", "MethodName", stringMessage);

log.logp(Level.INFO, "ClassName", "MethodName", stringMessageSupplier);

log.logp(Level.INFO, "ClassName", "MethodName", stringMessageFormat, args);

log.logp(Level.INFO, "ClassName", "MethodName", stringMessage, throwable);

log.logp(Level.INFO, "ClassName", "MethodName", throwable, stringMessageSupplier);

*// Вывести сообщение с указанием уровня логгирования, класса,*

*// метода и resourceBundle, хранящего сообщения*

log.logrb(Level.INFO, "ClassName", "MethodName", resourceBundle, "messageId");

log.logrb(Level.INFO, "ClassName", "MethodName", resourceBundle, "messageId", throwable);

*// Вывести сообщение об ошибке*

log.throwing("ClassName","MethodName", throwable);

Пример №2

Хорошо

try {

throw new Exception("Some exception");

} catch (Exception ex) {

log.log(Level.SEVERE, "Exception: ", ex);

}

//В стандартной лог конфигурации вы это сообщение не увидите

log.fine("some minor, debug message");

*/\**

*Иногда вывод лог сообщений требует достаточно больших ресурсов (например дамп какого-то пакета данных и т.п.). В таких случаях стоит проверить выведется ли в лог сообщение для этого уровня логирования*

*\*/*

if (log.isLoggable(Level.FINE)) {

log.fine("Some CPU consuming message: " + prepareCPUConsumingLogMessage());

}

1. Если Вам необходимо залогировать исключение, для этого служит метод .log(level,message,exception)
2. Если вы специально не настроили конфигурацию лог системы, сообщения с уровнем ниже info, например fine выводиться не будут. Но писать их по крайней мере для важных частей системы стоит. Когда что-то пойдет не так, Вы настроите более подробный уровень логирования и увидите много интересного.
3. Слишком много лог сообщений, даже если они физически не пишутся в лог файл из-за своего слишком маленького уровня, могут существенно замедлить выполнение программы. Особенно если для подготовки самого сообщения надо потратить много ресурсов. Для этого есть метод .isLoggable(level) — он позволяет узнать пропустит ли текущая конфигурация логера данное сообщение

Плохо

try { throw new Exception("Some exception");}

catch (Exception ex) {log.severe("Exception: " + ex.toString()); }

log.fine("Some CPU consuming message: " + itTakes500MillisecondsToPrepageThisMessage());

Если логировать только ex.toString(), то потом Вы не сможете понять в какой строке изначально сработало исключение. 

Пример №3

Логер надо конфигурировать. Есть конфигурация по умолчанию она выводит в консоль все сообщения с уровнем INFO и выше. Она достаточно хороша, для разработки из IDE, но для реального приложения ее обычно неплохо бы подправить.   
  
Какие тут есть варианты

По умолчанию: Файл logging.properties для уровня INFO, вывод в консоль

#Console handler  
handlers= java.util.logging.ConsoleHandler  
.level=INFO

Делаем логирование более подробным выводим еще и сообщения уровня FINE

#Console handler  
handlers= java.util.logging.ConsoleHandler  
.level=FINE  
java.util.logging.ConsoleHandler.level = FINE  
  
  
Что мы тут сделали

* Установили уровень FINE для корневого логера, просто чтобы сообщения пролезали внутрь лог системы.
* И сказали что все что пролезет через лог систему надо выводить на консоль от уровня FINE и выше.

Выводим лог сообщения куда-то еще

Чем плох вывод на консоль? Консоль это по сути дела старый добрый [stderr](http://en.wikipedia.org/wiki/Standard_streams). Что это значит:

* Если приложение запускается с помощью javaw Вы вообще ничего не увидите.
* Если вывод идет в консоль и нужное вам сообщение промелькнуло 4 часа назад буфер консоли его уже съел, информация пропала.
* Если вывод консоли направлен в файл *java com.yourcompanyname.EntryClass 2>>application\_log.txt* и приложение работает не останавливаясь несколько недель — файл будет весьма и весьма большим, рискуя занять весь диск.

Чтобы решить эти проблемы был придуман java.util.logging.FileHandler — хэндлер который выводит лог сообщения в файл. При этом он умеет ротировать файлы, т.е. после достижения максимально допустимого размера, он дописывает в файл текщуее лог сообщение и открывает новый файл с инкрементальным префиксом. И так по кругу. Например 

handlers= java.util.logging.FileHandler

java.util.logging.FileHandler.pattern = application\_log.txt

java.util.logging.FileHandler.limit = 50

java.util.logging.FileHandler.count = 7

java.util.logging.FileHandler.formatter = java.util.logging.SimpleFormatter

создаст вот такие файлы (последняя колонка — размер в байтах)

application\_log.txt.0 │ 0

application\_log.txt.1 │ 79

application\_log.txt.2 │ 79

application\_log.txt.3 │ 676

application\_log.txt.4 │ 87

application\_log.txt.5 │ 114

Мы указали максимальный размер 50 байтов, в реальной жизни надо скорее указывать не меньше мегабайта, например вот так (я знаю, что 1000000 это чуть меньше мегабайта, но кому охота по памяти писать 1048576, если суть дела это фактически не меняет)

java.util.logging.FileHandler.limit = 1000000

В примере, как мы видим, файлы получились больше 50 байт потому что размер по сути округляется вверх до последнего целого лог сообщения. Т.е. если Вы укажете размер 1 байт и запишете лог сообщение размером в 1000 байт то размер файла станет 1000 байт и после этого лог сообщения файл закроется и откроется следующий.   
  
Ниже записан copy & paste конфиг для реальной жизни, его вполне хватает для большинства service, console и desktop приложений.

handlers= java.util.logging.FileHandler

java.util.logging.FileHandler.pattern = application\_log.txt

java.util.logging.FileHandler.limit = 1000000

java.util.logging.FileHandler.count = 5

java.util.logging.FileHandler.formatter = java.util.logging.SimpleFormatter

Последняя часть магии

Ну и последнее о чем осталось рассказать — как собственно сконфигурировать логер из файла свойств. Есть два способа:

1. Из командной строки запуска приложения
2. В первых строчках кода Вашего приложения

Первый чуть более правильный ибо он декларативный и работает сразу, до того как начал работать код Вашего приложения. 

Вот так

java Djava.util.logging.config.file=logging.properties com.dataart.application.ClassName  
  
Но к сожалению менять строку запуска не всегда можно или не всегда удобно. Второй способ тоже неплохо работает.

public static void main(String[] args) {

try {

LogManager.getLogManager().readConfiguration(

MainApplicationEntryClass.class.getResourceAsStream("/logging.properties"));

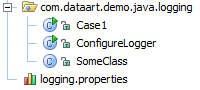
} catch (IOException e) {

System.err.println("Could not setup logger configuration: " + e.toString());

}

.....

* Здесь MainApplicationEntryClass — это класс — точка входа в Ваше приложение, видимо имя класса у Вас будет другое
* Сам файл logging.properties как правило в таких случаях кладется в корень иерархии классов и выглядит это например вот так



Что осталось за кадром

В реальной жизни как минимум половина всех Java приложений это web приложения. Сама техничка логирования в них совершенно не отличается от изложенного выше. Ну может быть за тем исключением что разные сервера приложений могут использовать разные библиотеки логирования такие например как:

* Log4J
* JULI logger (строго говоря это не вполне самостоятельный фреймворк, а своего рода надстройка над java.util.logging)
* SLF4J
* Commons Logging

Соответственно несколько отличается настройка и имена методов. Но сам принцип меняется мало. Конкретные особенности как правило хорошо описаны в документации на сам сервер приложений, например [Tomcat](http://tomcat.apache.org/tomcat-7.0-doc/logging.html) [JBoss](http://docs.jboss.org/process-guide/en/html/logging.html) [Resin](http://www.caucho.com/resin-3.0/config/log.xtp)