latex input: mmd-article-header Title: Things to Notes about Programming Author: Ethan C. Petuchowski Base Header Level: 1 latex mode: memoir Keywords: general, databases, testing, unit testing, monads, personal interest, research, fundamentals CSS: http://fletcherpenney.net/css/document.css xhtml header: copyright: 2014 Ethan Petuchowski latex input: mmd-natbib-plain latex input: mmd-article-begin-doc latex footer: mmd-memoir-footer

10 Tips for Proper Application Logging

Attribute	Value
Posted by	Tomasz Nurkiewicz
In	Software Development
On	January 17th, 2011
At	javacodegeeks.com/2011/01/10-tips-proper-application-logging.html

Main things I got out of it

1. Consider the fact that the log levels are not JUST for being able to selectively tune the number of log messages that show up in your console.
    1. They can be grepd for particular levels, post-execution
2. Use SLF4J
3. Look at the table of logging levels and their respective use-cases below
4. Avoid side-effects (and NPEs!) in your logs
5. Log exceptions *only* at the point where you catch them, and use the following code

```
log.error("Error reading configuration file", e);
```

6. Log arguments and return values
7. Log stuff returned from an external system or API

Summary

Use the appropriate tools for the job

1. That means SLF4J because it allows pattern subsitution

```
log.debug("Found {} records matching filter: '{}'", records, f
```

1. Use it with the Logback framework instead of Log4J
    - I'll need to look into *that*...
2. Perf4J is also a great simple tool for making graphs to address performance issues

Don’t forget, logging levels are there for you

Error

An untolerable event occurd that brought the system down and must be investigated immediately. e.g: NPE, database unavailable, mission critical use case cannot be continued.

Warn

"Current data unavailable, using cached values" "Application running in development mode or "Administration console is not secured with a password". The application can tolerate warning messages, but they should always be justified and examined.

Info

Important business process has finished. User should be able to understand INFO messages and quickly find out what the application is doing. E.g. one INFO statement per each ticket saying `"{Who} booked ticket from {Where} to {Where}"`. Also notes each action that changes the state of the application significantly (database update, external system request).

Debug

For developers' eyes.

Trace

Stuff only useful while developing a piece of code. Then should be removed so it doesn't conflict with someone else trying to do the same thing.

Do you know what you are logging?

1. Make sure your log messages are clear
2. Get rid of potential NPEs from code like

```
log.debug("Processing request with id: {}", request.getId());
```

3. Watch out with logging lazily initialized objects, e.g. the collections returned by the Hibernate API
4. Debuggable classes need an appropriate `toString` method

1. Note you may want to use the JDK's `Arrays.deepToString`

Avoid side effects

1. E.g. lazily initialized objects, as noted above
2. Too much logging can slow your app down; e.g. when there's more than say 50MB of logs per hour

Be concise and descriptive

1. Each logging statement should contain both data and description.

```
log.debug("Message with id '{}' processed", message.getJMSMess
```

2. Definitely just a description is not enough. The *context* of the loggable text is important to note.

Tune your pattern

For example, in Logback, the author uses

```
<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender"
    <encoder>
        <pattern>%d{HH:mm:ss.SSS} %-5level [%thread][%logger{0}] %m
    </encoder>
</appender>
```

He doesn't recommend adding filename, class name, and line number--because it *allows* you to feel OK getting sloppy with your debug messages and it is slow--but it sounds like a pretty good idea to me....

Log method arguments and return values

1. If you follow the simple rule of logging each method input and output (arguments and return values), you don't even need a debugger any more.
2. Of course, you must be reasonable, but every method that:
    1. accesses external systems (e.g. databases)
    2. blocks
    3. waits, etc. should be considered. Simply follow this pattern:

```
public String printDocument(Document doc, Mode mode) {
    log.debug("Entering printDocument(doc={}, mode={})", doc,
    String id = //Lengthy printing operation
    log.debug("Leaving printDocument(): {}", id);
    return id;
}
```

3. Because you are logging both the beginning and the end of method invocation, you can
    1. manually discover inefficient code
    2. detect possible causes of deadlocks and starvation
        - simply by looking after “entering” without corresponding “leaving”.
    3. If your methods have meaningful names, reading logs would be a pleasure.
    4. Analyzing what went wrong is much simpler
        - since on each step you know exactly what has been processed.
    5. You could build this in using an AOP aspect
        - though that could get messy
4. Consider DEBUG or TRACE levels as best suited for these types of logs.
5. But it is always better to have too much rather than too few logging statements.
6. Treat logging statements with the same respect as unit tests
    1. your code should be covered with logging routines as it is with unit tests.
    2. No part of the system should stay with no logs at all.
    3. Remember, sometimes observing logs rolling by is the only way to tell whether your application is working properly or hangs forever.

Watch out for external systems

1. if you communicate with an external system, consider logging every piece of data that comes out from your application and gets in.
2. Period.
3. What is the point of having a fast, but broken application, that no one can fix?

Log exceptions properly

1. Avoid logging exceptions, let your framework or container (whatever it is) do it for you
2. It is the responsibility of whomever *catches* the exception to log it
3. The *correct* way to log an exception is *exactly* as follows

```
log.error("Error reading configuration file", e);
```

Logs easy to read, easy to parse

1. Both humans and shell scripts should be able to get their own respective benefits from your logs
2. E.g. check this out

```
log.debug("Request TTL set to: {} ({})", new Date(ttl), ttl);
// Request TTL set to: Wed Apr 28 20:14:12 CEST 2010 (12724784

final String duration =
    DurationFormatUtils.formatDurationWords(durationMillis, tr
log.info("Importing took: {}ms ({})", durationMillis, duration
// => Importing took: 123456789ms (1 day 10 hours 17 minutes 3
```

    1. The "millis" are parseable and therefore graphable, and the "duration" is 'understandable'

Other stuff

Programmer Productivity

Fred Brooks (of *Mythical Man-Month* fame) recently (2010) wrote a book (*Design of Designs*) [in part] about programmer productivity, and he mentioned how in the 60s, he was able to follow-along with ALL the major things happening in computer research, but that over time he has had to give up on being able to do that as each thing got WAY more specialized. This is part of why more people are needed to do tasks today: they each require a much larger set of more specific expertise. The other reason people are more often working collaboratively on software than they used to is that by going to market sooner, one can have a higher chance of owning a larger market share with your product. But he notes that we must remain cognizant that even if having more people on a team means we can split up the work as best we can, and each have *less* work to do, as managers, we should realize that in this structure there inevitably ends up being MORE work to do overall, especially considering the communication overhead between project members. Also different tasks lend themselves to being split amongst multiple people than others (think Amdahl's Law).

XML vs JSON

XML is considered to to be a better description format than JSON in certain cases. For example consider how you'd write the following in JSON. The point is not that it would be *impossible* in JSON, just that it would be more difficult to read and write by hand, and potentially more difficult to define transformations for *a la* XSLT/XPath.

```
<foo>
   <x:bar x:prop1="g">
      <quuz />
   </bar>
</foo>
```

Deadlock

6/7/15

How do we ensure that deadlock does not occur?

1. Assign an order to locks
2. Require that they're always acquired in that order

How can we detect a deadlock?

Look for a cycle in the wait-for graph (WFG).

1. Assign each thread a node.
2. If thread *p* wants a lock that thread *q* owns, draw a line $p \rightarrow q$.
3. Having a cycle in the WFG implies the system is in deadlock.

But how can we detect a cycle in a graph?

One recommended option is to use depth-first search (DFS), and keep tracker of vertices on the current recursion stack.

Pseudocode

Suppose we are creating a method on `class Graph` with an adjacency-list `adj`. The graph does not have to be connected or anything. It will still work, because we start the search from every node in the graph.

```
# the kicker-off-er
def hasCycle():
    visited = [false] * numNodes
    onStack = [false] * numNodes
    for node in range(numNodes):
        if inCycle(node, visited, onStack):
            return True
    return False

# does the actual DFS
def inCycle(node, visited, onStack):
    if not visited[node]:
        visited[node] = true
        onStack[node] = true
        for adjNode in adj[v]:
            # THIS IS THE KEY PART OF IT ALL
            if !visited[adjNode] and inCycle(adjNode, visited, onSta
                # a cycle was found furth down the search
                # so we must bubble it up
                return True
            elif onStack[adjNode]:
                # a cycle was found here; pass True to parent stack
                return True
    onStack[node] = False
    return False
```

Apache Software Foundation

5/26/15 --- This is basically a summary of the Wikipedia article

- American non-profit corporation [501(c)(3)] supporting Apache software projects, including Apache HTTP Server, formed in 1999
- Apache projects are distrubuted under the Apache License, making them "free and open source software (FOSS)"
- Projects have collaberative, consensus-based development process
- Membership of the foundation is granted only to volunteers who have actively contributed to Apache projects
- The foundation provides legal protection to its volunteers, and prevents its name from being used without permission
- It holds several ApacheCon conferences each year
- History is linked to Apache HTTP Server, development beginning in 1995, where 8 people worked on enhancing NCSA HTTPd
    - NCSA itself was among the earliest webservers developed, following Tim Berners-Lee's CERN httpd and others
- Development occurs in semi-autonomous "top-level projects"
    - Some of which have their own sub-projects

- As part of the process of becoming an Apache project, ASF gains the necessary intellectual property rights for the development and distribution of all its projects

Local databases

After I ran `brew install mysql`, it told me

- to launch mysql, run `mysql.server start`
- to connect, run `mysql -uroot`

The Monad

From Wikipedia

5/27/14

Intro

- A structure used in functional programming to represent computations defined as sequences of steps
- Defining a monad *type* means defining how operations of that type can be chained or nested together
- Facilitates the construction of data processing pipelines
- The monad defines extra code that gets executed *between* statements in the pipeline
- Can be seen as a functional design pattern for building generic types
- Used in "*purely functional*" programs to allow sequenced operation including side effects like I/O, variable assignment, exception handling, concurrency, etc.

What is it

- Consists of a type constructor *M* and two operations *bind* and *return/unit*
- These operations must fulfill several properties to allow proper composition
- *Return* takes puts a value of a plain type into the type constructor, creating a *monadic value*
- *Bind* does the reverse, extracting the original value from the container and passing it to the next function in the pipeline, possibly with additional checks and transformations

Example (Option)

- We want a monadic type such that computations can be chained such that if one computation fails, the rest of the computations simply won't *do* anything.

- So we have a type `Option[T]` s.t. if a computation fails it returns `None`, and if it succeeds it returns `Some[T]`, and when we perform operations on two `Options`, we only actually compute a result if they are both `Somes`, but simply return `None` if at least one operand is a `None`.

The following is my Scala translation of their Haskell example for this. It merits careful study.

```
scala> def maybePlus(a: Option[Int], b: Option[Int]): Option[Int] =
     |   for {
     |     ia <- a
     |     ib <- b
     |   } yield (ia + ib)
     | }
maybePlus: (a: Option[Int], b: Option[Int])Option[Int]

scala> maybePlus(Some(3), Some(4))
res0: Option[Int] = Some(7)

scala> maybePlus(Some(3), None)
res1: Option[Int] = None
```

Example Writer monad

My Scala translation of their Javascript example for this. Also merits study.

```
// the bind() method from the example
case class Writer[A](value: A, log: String) {
  def flatMap[B](f: (A) => Writer[B]): Writer[B] = {
    f(value) match {
      case Writer(newVal, newStr) =>
        Writer(newVal, log + newStr)
    }
  }
}

// the unit() method from the example
object Writer {
  def apply[A](a: A): Writer[A] = Writer(a, "")
}

object Run extends App {
  println(
    Writer(2)
      .flatMap(x => Writer(x*x, "squared."))
      .flatMap(x => Writer(x + 3, "plus threed."))
  )
}
```

Monad laws

There is a small set laws that must be followed in order for a monad to behave correctly, but I don't understand the Haskell used to identify them on the Monad Wikipedia page.

The correct type to use for currency values

Don't use `double`, use `java.math.BigDecimal`

Redis

3/23/14

StackOverflow

- ***RE**mote **DI**ctionary **S**erver*
- **NoSQL** key-value data store
- Data structure server
- Similar to **Memcached**, but with built-in *persistence* and more datatypes
- **Persistence** -- *snapshotting*, or *journaling* to disk
- **Datatypes** -- *Dictionary*, *List*, *(Sorted) Set*
- **Pub/Sub** transactions (see glossary at bottom)

- **Optimistic locking** -- (see glossary at bottom)
- The entire data set is stored in-memory (like Memcached)

Use Cases:

- Highly scalable data store shared by multiple processes, applications, or servers
- Caching layer

Philosophy

When something is too slow

5/30/14

Cache it

On the Method of Understanding a Programme

3/13/14

- Run the program in a debugger
- Put breakpoints everywhere
- When it catches the first one, walk up the stack
- Put a breakpoint at the top of that stack
- Now restart from that outermost function and watch the thing unfold

Reactive Applications

2/23/14

According to Typesafe

Reactive applications have one or more of the following **defining traits:**

- **Event driven** -- enables parallel, asynchronous processing of messages or events with ease.
- **Scalable** -- across nodes elastically
- **Resilient** -- recovers and repairs automatically
- **Responsive** -- single-page UIs that provide instant feedback

In particular, the **Typesafe Reactive Platform** consists of the following **stack:**

- **Play!** -- Web Framework
- **Akka** -- Actor Model concurrency library
- **Scala** -- Programming Language
- **Typesafe Console** -- Console

3/23/14

The best description of what it really means, I've found to be this lecture on Vimeo by Sadek Drobi. He describes it as being a pattern in which you do all your computations on `Promises`, and then the whole string of events that will happen once you obtain your `Promise` is computed and then the actual computation takes place whenever you do obtain the `Promise`. But the point is that you're never waiting for the `Promise`, you just "react" by computing it when you *do* receive it. Something like that, anyway.

Regex --- Lookahead & Lookbehind = "Lookaround"

2/17/14

From Regular-Expresions.info

Lookahead

- **Negative lookahead** --- match something *not* followed by something else
  - E.g. q *not* followed by u --- `q(?!u)`
- **Positive lookahead** --- match something *only if* it's followed by something else
  - E.g. q followed by u --- `q(?=u)`
    - Recall that the u is not consumed in this case
- Any valid regular expression can be used inside the lookahead (even capture groups, which do require their own set of parentheses within the lookahead)

Lookbehind

- **Negative lookbehind** --- match something *not* preceded by something else
  - E.g. b *not* preceded by a --- `(?<!a)b`
- **Positive lookbehind** --- match something *only if* it's preceded by something else
  - E.g. `:` preceded by `cite` --- `(?<=cite):`
- Lookbehinds are generally restricted to only allowing some subset of the normal regex vocabulary, but the specifics vary (quite a bit) by language

Notes:

- **They do not consume characters** in the string, they only assert whether a match is possible or not
- Some regexes would be impossible without them

##Newlines

2/16/14

- On **Windows**, they use carriage-return & line-feed (`"\r\n"`)
- On **UNIX/Mac**, they use new-line, which is represented by the same ascii code as line-feed (`'\n'`), but does both CR-LF in one go

Character Encodings

9/9/15

Java Charset

- In Java, we have the Standard Library's *immutable* (and thread-safe) `public abstract class java.nio.charset.Charset extends Object implements Comparable<Charset>`, which is a "named mapping between sequences of sixteen- bit Unicode *code units* and sequences of *bytes*".
- This class provides methods
    - `ByteBuffer encode(String str)` -- encodes a string into bytes in this charset
    - `CharBuffer decode(ByteBuffer bb)` -- decodes bytes in this charset into Unicode characters
- You might want to use it for example to **declare the encoding of a text file to read**

```
new InputStreamReader(
    new FileInputStream(new File(".")),
    StandardCharsets.UTF_8   // alternative to string constant
);
```

2/4/15

- **ASCII** --- 7-bit character set maps characters to the range [0,127]
    - Sufficient for American English, but that's about it
- **Latin-1** --- 8-bit strict superset of ASCII, adds a few more characters
- **Unicode** --- space for 1m characters, 100K are used so far
    - Since it's only a "character set" not a "character encoding", it specifies what character gets mapped to each number, but it doesn't say how many bytes to use or which endianness; that's why we have the *encodings* UTF-8/16/32
    - **UTF-32** --- each character is a 4-byte int
    - **UTF-16** --- most are 2-bytes, some less common chars are 4-bytes
    - **UTF-8** --- 1-byte for ASCII chars, 2-bytes for many other alphabets, 3-4-bytes for chars from Asian languages
        - This is used by Java's `.class` files to store string literals

- Other --- always use someone *else*'s converter to deal with other encodings

Reference

- Java I/O 2nd ed., Elliotte Rusty Harold, 2010, O'Reilly

FTP vs. HTTP

2/16/14

Mainly from [AlBlue's Blog](http://alblue.bandlem.com/2009/02/why-do-people- still-use-ftp.html)

- FTP was created for transferring files
    - That's why it gives you options for binary or ASCII modes
- HTTP was created for transmitting HTML
    - The file type is guessed based on the extension, though it can be specified in the header
- Originally, FTP was better for file downloads because HTTP/1.0 didn't support resumable downloads (where client disconnects part-way through and needs to restart). However, this was made possible in HTTP/1.1 via headers `Accept-Ranges` and `Content-Range`.
- HTTP also supports automatic data-compression, querying data-type before downloading, proxy support, running over SSL with HTTPS
- **People only use FTP over HTTP out of ignorance**
- In particular, WebDAV -- an extension of HTTP -- allows collaborative editing and management of documents stored on Web servers.

Glossary

- **Reentrant** -- 2/16/14 -- a function that can be interrupted in the middle of its execution and then safely called again ("re-entered") before its previous invocations complete execution. Once the reentered invocation completes, the previous invocations will resume correct execution.
- **Partial Function** -- 3/1/14 -- *normally* a function maps an *entire* domain to some range. In a **Partial Function**, however, not *every* element of the domain must be mapped. Some values in the domain may be *undefined* after passing * through the partial function.
- **Optimistic Locking** -- 3/23/14 -- Before committing, each transaction verifies that no other transaction has modified the data it has read. If the check reveals conflicting modifications, the committing transaction rolls back and can be restarted.
- **Messanging pattern** -- 3/23/14 -- describes how two different parts of a message passing system connect and communicate with each other.

    - E.g. HTTP is a *request-response* pattern, UDP is a *one-way* pattern.
    - **Request-response** -- requester sends request message, replier receives, processes, and responds.
    - **Publish-subscribe** -- *publishers* post to an *intermediary message broker*, and *subscribers* register subscriptions with the broker. The broker might perform a *store and forward* to route messages to their destinations, and may prioritize the orderings.
        - E.g. RSS feeds
        - Provides better scalability than *request-response*
- **[Referential Transparency] (http://en.wikipedia.org/wiki/Referential_transparency_(computer_science))** -- when you can replace an expression with its value without changing the program.
- **Dithering** --- applying noise to randomize quantization error, used when compressing audio and video data