Logging

- What is Logging?
- Anatomy of a Logging Statement
- Why not just use *println*()?
  - Filtering: Log Hierarchy, Log Levels and How to Choose Them
  - Building Log Messages: Concatenation vs Args vs Lambdas
  - Formatting: Layouts (Pattern, JSON, whatever…)
  - Output: Appenders (Console, [Rolling] File, syslog, socket…)
  - Context: Time, Exception, Thread, MDC (RID, User, Method...)
- Structured Logging: Machine Readability FTW!
- Impact of Logging. Typical Pitfalls
- Popular Java Logging Frameworks
- Alternatives to Logging

What is Logging?

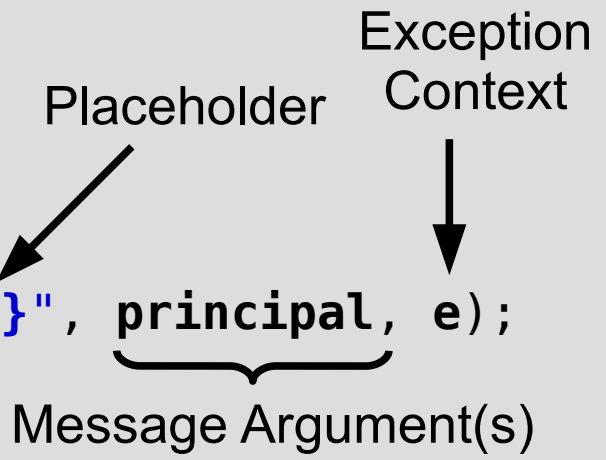
- One of the Pillars of **Classic Observability** (Logging, Metrics, Tracing)
- Emitting **Events**:
  - About **Known Unknowns** in your code (e.g. network errors, anticipated unhappy code paths etc.)
  - With **Low Cardinality** (tens of attributes, typically less)
  - **Schemaless** / **Low-Schema**

Anatomy of a Logging Statement

```
private static final Logger log = LoggerFactory.getLogger(My.class);

// <...>

var principal = ...;
try (var creditClient = ...) {
    return creditClient.score(principal);
} catch (ClientException e) {
    log.error("Could not calculate score for {}", principal, e);
    return DEFAULT_CREDIT_SCORE;
}
```



Anatomy of a Logging Statement

```
2022-04-08 08:42:57,458 ERROR [credit-client-pool-68]
{da7b76be-7a98-4a2f-9758-4b3df818d903}
Could not calculate score for PremiumBankingAccount[id=2001000, ...]
ru.hse.java.client.ClientException: java.util.concurrent.ExecutionException:
io.grpc.StatusRuntimeException: {FAILED_PRECONDITION} Blahblahblah...
        at ClientBase.a(ClientBase.java:13)
        at CreditClient.score(CreditClient.java:438)
Caused by: java.util.concurrent.ExecutionException: io.grpc.StatusRuntimeException:
{FAILED_PRECONDITION} Blahblahblah...
        at ClientBase.c(ClientBase.java:23)
        at ClientBase.b(ClientBase.java:17)
        at ClientBase.a(ClientBase.java:11)
        ... 1 more
Caused by: io.grpc.StatusRuntimeException: {FAILED_PRECONDITION} Blahblahblah...
        at ClientBase.e(ClientBase.java:30)
        at ClientBase.d(ClientBase.java:27)
        at ClientBase.c(ClientBase.java:21)
        ... 3 more
```

Why not just use a *println*()?

```
var principal = ...;
try (var creditClient = ...) {
    return creditClient.score(principal);
} catch (ClientException e) {
    System.err.println(java.time.Instant.now() + " "
        + "ERROR "
        + "[" + Thread.currentThread().getName() + "] "
        + "{" + MyMagicContext.getRequestId() + "} "
        + "Could not calculate score for " + principal);
    e.printStackTrace(); // Defaults to System.err output
    return DEFAULT_CREDIT_SCORE;
}
```

Copy-Paste: Code Duplication, Error Prone (*Shotgun* changes)
Make logging an utility method: Reinventing the wheel

Log Message Filtering

- Log Levels
  - **TRACE**: Super-detailed messages. *E.g.* generated SQL queries
  - **DEBUG**: Detailed and/or frequent messages with information useful for debugging.
    E.g. details about repository operations
  - **INFO**: Normal messages about happy path / progress of your processes.
    E.g. processed 150/1500 data migration tasks
  - **WARN**: Potentially unexpected or harmful situations warranting a closer look once in a while
    E.g. optional service dependency is temporarily unavailable
  - **ERROR**: Severe errors warranting a look by the oncall and/or code developer
    E.g. datastore down, data corruption etc.
- Hierarchy of Loggers: `ru` ← `ru.hse` ← `ru.hse.java` ← `ru.hse.java.MyClass` ← ...

  E.g. show only warnings and errors from `org.apache.http.*` classes
  => `org.apache.http` logger level = **WARN**

Building Log Messages

- Eager (**Not recommended**):

```
log.error("x = " + x + ", y = " + y);
```

- Semi-Lazy (Classic Way):

```
log.error("x = {}, y = {}", x, y);
```

- Lazy (Newfangled Way):

```
log.error(() → "x = " + x.get() + ", y = " + y);
```

Formatting: Layouts

- Formatting is Decoupled from Message Building: A major advantage of Logging Frameworks
- Most Popular: Pattern Layout, *printf*() on steroids

```
%d %-5level [%t] %c{1.}: %X{rid} %msg%n%throwable
```

Timestamp	Log Level (left padded)	Thread	Logger (shortened)	Value from Context	Message	Exception (if any)

- JSON Layout, XML Layout, …

Output: Appenders

- Log Message Output is Easily Customizable:
  A major advantage of Logging Frameworks
- Console Appender (reasonable default)
- [Rolling] File Appender: Write to file(s), optionally compressing and/or deleting old log files
- Also: syslog, socket, …

Log Message Context

- Date and Time
- Thread
- Exceptions and their causes (and also suppressed exceptions, see `try`-with-resources)
- Custom Attributes (MDC)
  - Request ID / Correlation ID
  - Trace ID, Span ID
  - User / Principal ID
  - API Method
  - …

Structured Logging

- Text is so 1980's and text parsers are **SLOW** and **UNRELIABLE**
- Let's output to **some structured format instead**
  - Preferrably, with schema
- Old Try: syslog (RFC 5424)
- New Tries: journalctl (systemd), JSON

Structured Logging

```
{ "timestamp": "2022-04-08 08:42:57.458",
  "level": "ERROR",
  "thread": "credit-client-pool-68"
  "rid": "da7b76be-7a98-4a2f-9758-4b3df818d903",
  "message": "Could not calculate score",
  "account": {
    "@type": "PremiumBankingAccount",
    "id": 201000,
    ...
  },
  "exception": {
    "@type": "ru.hse.java.client.ClientException",
    "message": ...,
    "cause": { ... }
  }
}
```

Impact of Logging. Typical Pitfalls

- **Improperly used, logging is a DoS waiting to happen, or worse**
- CPU: Log level checking, Message Building, Message Formatting
- Logging changes timings → Uncovers or masks **data races**
- Memory: Large messages and/or args cause **GC pressure** → **OutOfMemoryError**
- Disk: Absence of / poorly configured log rotation → **No space left on device**
- Synchronous Logging will cause **hangs/hiccups** under heavy load, especially if you are logging to a network disk (incl. in the cloud)
- Async logging fixes hangs/hiccups but will **drop your log messages**!
- Logging Frameworks definitely have undetected 0-day vulnerabilities
  e.g. log4shell

Popular Java Logging Frameworks

- For Standalone Services/Apps:
  - **Log4j 2.x**
  - **Logback**
- For Libraries: **SLF4J** (Simple Logger Facade for Java)
  - Allows the user to pick **any** compatible logging framework
  - Both Log4j 2.x and Logback are compatible!
- **Log4j 1.x: Legacy Version, Do Not Use in New Code!**

Alternatives to Logging

- "New Observability": Events+Traces+Logs all-in-one
  Firehose of **high cardinality** events (1000s of attributes and values)
  → Sample **interesting events**
  → Index → Search and Visualise
  → **Discover complex system behavior** (manual or AI-assisted)
  E.g. Honeycomb, VividCortex, Dynatrace, Instana…
- Open Source: Jaeger tracing is better than nothin'
  - Simple sampling strategies ([adaptive] probabilistic, leaky bucket)
  - High cardinality is difficult