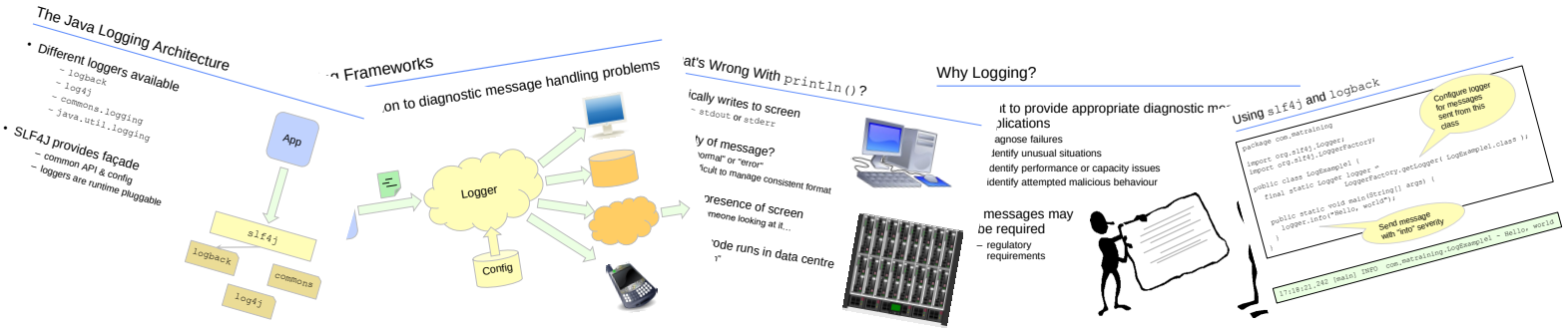
Logging in Java Applications

Java Programming
November 2020



Why Logging?

- Important to provide appropriate diagnostic messages from applications
  - diagnose failures
  - identify unusual situations
  - identify performance or capacity issues
  - identify attempted malicious behaviour
- Audit messages may also be required
  - regulatory requirements

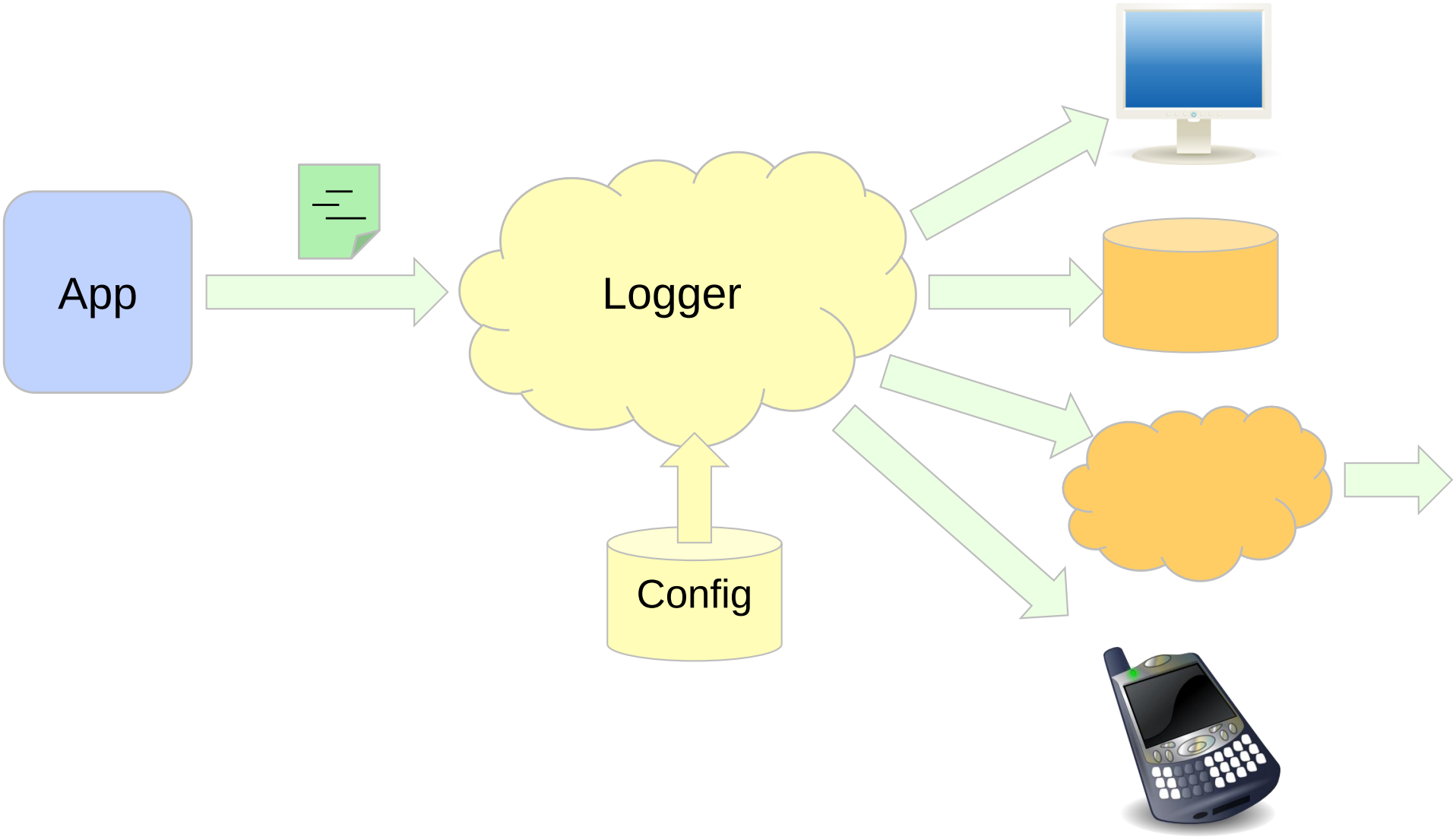
What's Wrong With `println()`?

- Typically writes to screen
  - `stdout` or `stderr`

- Severity of message?
  - "normal" or "error"
  - difficult to manage consistent format
- Assumes presence of screen
  - and someone looking at it…
- Server-side code runs in data centre
  - "dark room"

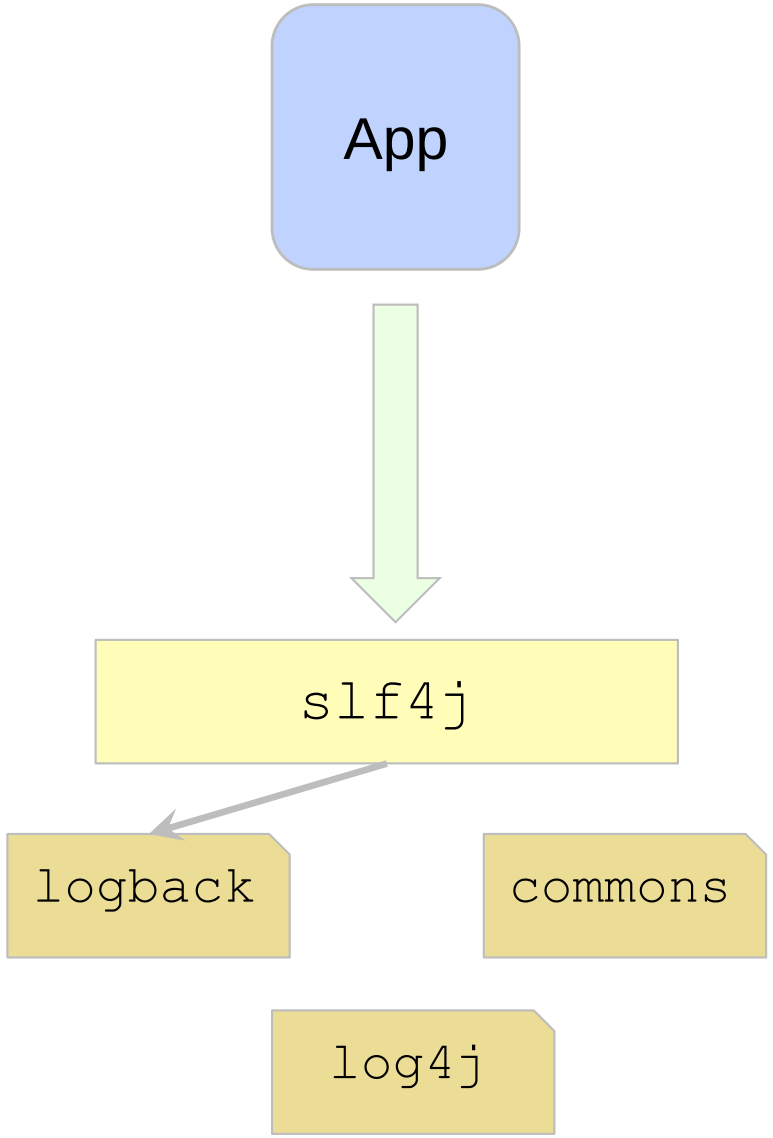
Logging Frameworks

- Provide solution to diagnostic message handling problems



The Java Logging Architecture

- Different loggers available
  - `logback`
  - `log4j`
  - `commons.logging`
  - `java.util.logging`
- SLF4J provides façade
  - common API & config
  - loggers are runtime pluggable



Using `slf4j` and `logback`

```
package com.matraining

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class LogExample1 {
  final static Logger logger =
                      LoggerFactory.getLogger( LogExample1.class );

  public static void main(String[] args) {
    logger.info("Hello, world");
  }
}
```

Configure logger for messages sent from this class

Send message with "info" severity

```
17:18:21.242 [main] INFO  com.matraining.LogExample1 - Hello, world
```

Configuration

- May be written in XML
  - `logback.xml`
  - or `logback-test.xml`
- Location can be specified as property

  `-Dlogback.configurationFile=`*`/path/to/config-file`*
- `logback.groovy`
  - located on classpath
- `logback` uses defaults used when no file found

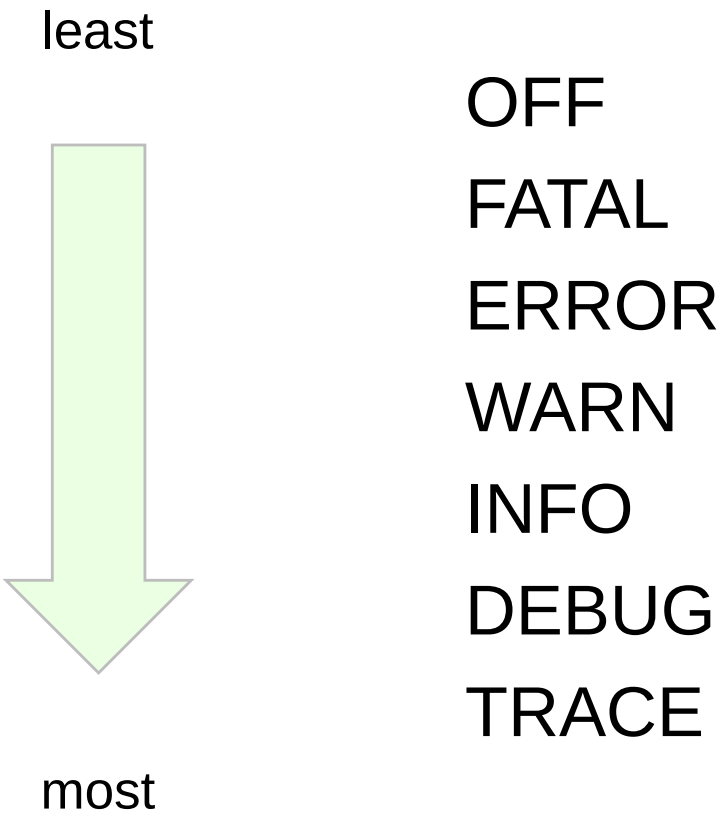
Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration debug="false">

  <appender name="STDERR" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <Pattern>
        %d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n
      </Pattern>
    </encoder>
  </appender>

  <root level="debug">
    <appender-ref ref="STDERR" />
  </root>

</configuration>
```

```
17:18:21.242 [main] INFO  com.matraining.LogExample1 - Hello, world
```

Severity Levels

- Several levels of severity
  - request assigned a level
  - logger has level
  - logger allows messages with level >= logger level



Configuring a Logger

- Specify `name` attribute
  - package or class name
- `level` and `appender` details optional
  - may be inherited

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration debug="false">
  …

  <logger name="com.matraining" level="debug" />

  <root level="off">
    <appender-ref ref="STDERR" />
  </root>

</configuration>
```

Logger for package has debug level

Default is not to send any messages

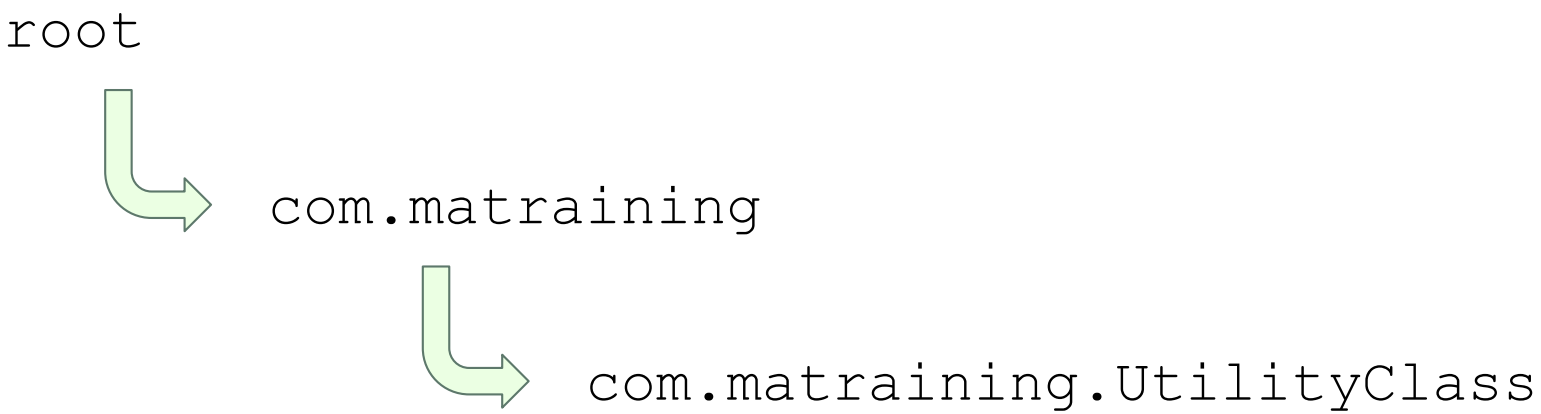
Configuring a Logger

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration debug="false">
  …

 <logger name="com.matraining" level="DEBUG" />

 <logger name="com.matraining.UtilityClass" level="WARN" />

 <root level="off">
    <appender-ref ref="STDERR" />
  </root>

</configuration>
```

```
public class UtilityClass {
  static final Logger logger =
       LoggerFactory.getLogger(UtilityClass.class);
  public void doSomething () {
    logger.debug("I'm doing something important");
  }
}
```

Configuring a Logger

- Logger hierarchy based on name



- Inheritance may be disabled
  - set `additivity` attribute to `"false"`

Appenders

- Define where log messages are sent
  - and how they are formatted

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration debug="false">

  <appender name="STDERR" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <Pattern>
        %d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n
      </Pattern>
    </encoder>
  </appender>

  …

</configuration>
```

Appenders

- Many different Appenders available
  - API allows more to be written

`ConsoleAppender`	logs to `System.out` or `System.err`, depending on configuration
`FileAppender`	logs to specified file
`RollingFileAppender`	logs to file, with roll-over to new file on specified trigger (size or time)
`SocketAppender`	send log messages to remote server (`SSLSocketAppender` also available)

Appenders

- Appender configuration is cumulative
  - appenders from parent loggers added to locally defined ones
  - unless "`additivity=false`" specified

```
…
<appender name="LOGFILE" class="ch.qos.logback.core.FileAppender">
   <file>/tmp/myLog</file>
   <encoder>
     <Pattern>
       %-4relative [%thread] %-5level %logger{36} - %msg%n
     </Pattern>
   </encoder>
 </appender>
 …
 <logger name="com.matraining.UtilityClass" level="DEBUG">
   <appender-ref ref="LOGFILE" />
 </logger>
 <root level="debug">
   <appender-ref ref="STDERR" />
 </root>
 …
```

Performance Considerations

- Avoiding unnecessary work when logging disabled
- Logging overhead is method call plus `int` comparison
- Hidden costs in message parameter construction
  - e.g. `String` concatenation

```
myLogger.debug("Val1: " + val1 + "Val2: " + val2")
```

- Can be avoided by checking before call

```
if ( myLogger.isDebugEnabled() ) …
```

- Alternate solution - use parameterised logging API
  - defers operation until required

```
myLogger.debug("Val1: {} Val2: {}", val1, val2")
```

Log File Best Practices

- Application logs should help diagnose or find a problem
  - must contain enough data to identify what a user was doing
  - must not contain sensitive information
  - separate out security events, use guidelines for log files
  - should send an event to monitoring system if necessary
  - must be stored in a protected location

Secure Logging Guidelines

- What **should** you log:
  - authentication and authorization events and attempts
  - application and network events
  - administrative actions taken within the application
  - any access to sensitive information
- What should you **NOT** log:
  - nature and content of failed login events
  - any sensitive information
- Developers should be aware of Log Injection
  - write to logs in a way that averts these attacks
  - attackers may attempt to cover up their actions

Secure Logging Guidelines

- Where to log:
  - Logging as a Service / Splunk
  - WM has well established high and normal security logging
- How to log (what libraries to use):
  - Java – SLF4J & Logback
  - C/C++, .Net, Perl, Python – MSLog
- Secure systems log files may be used to detect an attack
  - products like Splunk assist with log file analysis