Course Application Design

Creating beautiful and reliable applications
Logging

Michiel Noback
Institute for Life Sciences and Technology
Hanze University of Applied Sciences

Part one

Why Logging

Old school coding

- This is probably how you have written Java programs so far

```
private void start() {
    System.out.println("starting the good work");
    try {
        doBusiness();
    } catch (IOException e) {
        e.printStackTrace();
        System.out.println("Something went wrong - exiting!");
    }
    System.out.println("finished");
}
```

The disadvantages of `System.out.println()`

- They clutter the standard out stream (or the standard error stream)
- They have to be
  - Removed (but you have forgotten where the output comes from!)
  - Commented
  - Uncommented
  - Reinserted

…all the time

Advantages of Logging

- Determine run-time which messages are written
- No cluttering of stdout and stderr
- Different output formats can be specified
- Easy bug tracing
- Filtering of log messages
- Highly configurable (too much…)

Disadvantages of Logging

- Highly configurable (too much…)
- Requires your coding time
- Not really SRP
  - but this can be solved using Aspect Oriented Programming (AOP) – a bit like Python decorators

Part one

Getting started

Java.util.logging

- This presentation deals with the Logging API provided by Java itself.
- Other major frameworks are
  - Log4J2
  - Apache commons logging
  - Logback
- The Spring Boot framework provides configurations for all Logging technologies

The Java Logging API

- Here is the same class, but with logging

```
public class LoggingApp {
    public static void main(String[] args) {
        LoggingApp app = new LoggingApp();
        LOGGER.setLevel(Level.ALL);
        app.start();
    }
    private static final Logger LOGGER = Logger.getLogger("nl.bioinf");

    private void start() {
        try {
            LOGGER.log(Level.INFO, "Doing business");
            doBusiness();
        } catch (IOException e) {
            LOGGER.log(Level.SEVERE,
                    "Something went wrong; cause= {0}; message={1}",
                    new Object[]{e.getCause(), e.getMessage()});
        }
    }
}
```

The Level that is set is very important

Logging levels

- Level.***ALL***

```
Jan 18, 2018 10:27:02 AM nl.bioinf.logging.LoggingApp start
INFO: Doing businessJan 18, 2018 10:27:02 AM
nl.bioinf.logging.LoggingApp start
SEVERE: Something went wrong; cause=java.io.FileNotFoundException;
message=exception occurred
```

- Level.***WARNING***

```
Jan 18, 2018 10:28:02 AM nl.bioinf.logging.LoggingApp start
SEVERE: Something went wrong; cause=
java.io.FileNotFoundException; message=exception occurred
```

Logging messages

- You should always be **specific** and **informative** in your logging messages
- Here are a few ways to create them

Logging messages

- Create messages in the simplest form, without message parameters

```
LOGGER.warning("Could not find resource");
//same as
LOGGER.log(Level.WARNING, "Could not find resource");
```

```
Jan 18, 2018 3:08:26 PM nl.bioinf.logging.LoggingApp start
WARNING: Could not find resource
```

Logging messages with parameters

- Create messages with message parameters

```
LOGGER.log(Level.SEVERE,
        "Something went wrong; cause= {0}; message={1}",
        new Object[]{e.getCause(), e.getMessage()});
```

```
Jan 18, 2018 3:08:26 PM nl.bioinf.logging.LoggingApp start
SEVERE: Something went wrong; cause=
java.io.FileNotFoundException; message=exception occurred
```

Logging messages with exceptions

- Create messages with exceptions

```
LOGGER.log(Level.WARNING,
           "An illegal state!",
           new IllegalStateException("No booze"));
```

```
Jan 18, 2018 4:05:47 PM nl.bioinf.logging.LoggingApp start
WARNING: An illegal state!
java.lang.IllegalStateException: No booze
       at nl.bioinf.logging.LoggingApp.start(LoggingApp.java:35)
       at nl.bioinf.logging.LoggingApp.main(LoggingApp.java:15)
```

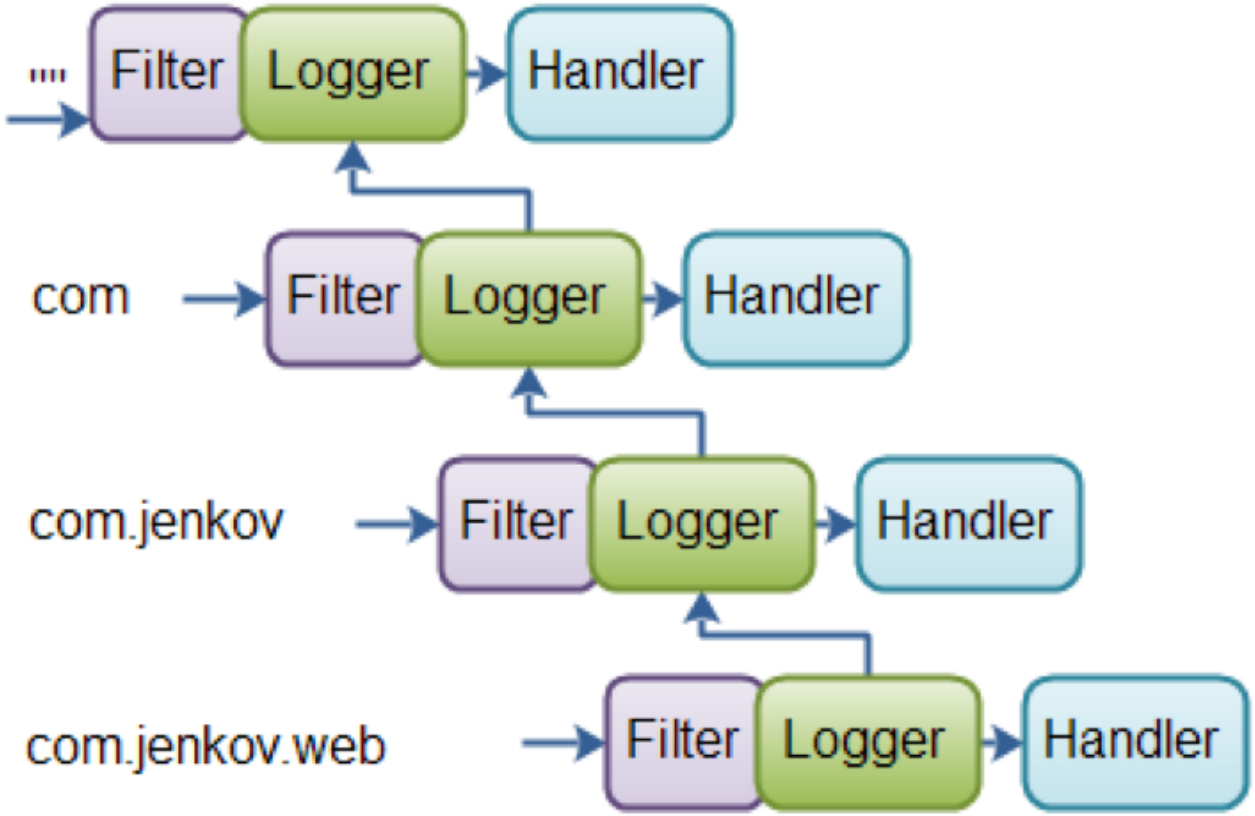
Logging levels

- The `Level` class defines a set of logging levels that can be used to control logging output
- Enabling logging at a given level also enables logging at all higher levels.
- The levels in descending order are:
  - SEVERE (highest value)
  - WARNING
  - INFO
  - CONFIG
  - FINE
  - FINER
  - FINEST (lowest value)
- In addition there is a level OFF that can be used to turn off logging, and a level ALL that can be used to enable logging of all messages.

Logger hierarchy

- When a message is passed to a Logger, the message is passed through the Logger's Filter, if it is set
- The Filter can either accept or reject the message. If accepted, it is forwarded to the Handlers set on the Logger.
- If a message is accepted by the Filter, the message is also forwarded to the Handler's of the parent Loggers
- However, when a message is passed up the hierarchy, the message is not passed through the Filters of the parent Loggers

Logger hierarchy



Handlers and Formatters

- Besides the ConsoleHandler, you can use FileHandler (or others)

```
//remove default handlers
Handler[] handlers = LOGGER.getHandlers();
for(Handler handler : handlers) {
    LOGGER.removeHandler(handler);
}
//define and set handler with formatter
FileHandler fileHandler = new FileHandler("logfile.html", false);
fileHandler.setFormatter(new MyHtmlLogFormatter());
LOGGER.addHandler(fileHandler);
```

Handlers and Formatters

- Handlers have default Formatters, but you can define your own – here is an example generating html

```
public class MyHtmlLogFormatter extends Formatter {
    public String format(LogRecord record) {
        return ("<tr><td>"
                    + (new Date(record.getMillis())).toString()
                    + "</td><td>”
                    + record.getMessage()
                    + "</td></tr>\n");
    }
    public String getHead(Handler h) {
        return ("<html>\n  <body>\n"
                    + "<Table border>\n<tr>"
                    + "<td>Time</td><td>Log Message</td></tr>\n");
    }
    public String getTail(Handler h) {
        return ("</table>\n</body>\n</html>");
    }
}
```

Logging in Spring Boot applications

- See https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/html/boot-features-logging.html for the official doc
- and https://www.concretepage.com/spring-boot/spring-boot-logging-example for a tutorial
- But if you really want to kick ass, you should do it with Aspect Oriented Programming (AOP) – see https://egkatzioura.com/2016/05/29/aspect-oriented-programming-with-spring-boot/ for a starter