HBO Graduaat Informatica
Optie Programmeren

Logging

cvo leerstad

volwassenenonderwijs

Chapter Overview

- Recording the progress of a program
- Collecting more or less information depending on the severity, in different possible formats without recompiling the code

Introduction

- Objectives
  - To add logging to your programs
  - To introduce the Log4J library
- Chapter content
  - Why is logging so important?
  - The limitations of System.out or System.err
  - Oracle's recommendation JSR 47 and Apache's Log4J implementation
- Practical content
  - Familiarization with logging
- Summary

Logging

- The need for logging
  - Info about events and exceptions (broken DB connection, memory exhausted, accounting purpose, tracing process in test environment, ...)
  - Using System.out.println has a lot of limitations

Logging

- Limitations of `System.out.println()` (as well as `System.err.println()`)
  - Can’t turn on and off at run-time without recompiling
  - Can’t turn on and off at run-time for specific classes or packages
  - No concept of severity in messages
  - Not persistent, only one destination
  - Only one format (text)

Logging

- Java Specification Request 47
  - Enable/disable logging at runtime
  - Control logging at a fairly fine granularity (Disable logging for specific functionality)
  - Bridge services that connect the logging APIs to existing logging services (Operating System Logs, Third party logs)

Logging

- What about debugging?
  - Only possible in development environment
  - Logging instead is possible in the field

Logging

- Apache Log4J architecture
  - Log4j has three main components: *loggers*, *appenders* and *layouts*.

component	description
Loggers	Loggers are the things you interract with in code to send messages to. You can have one logger per class, per package or inherit a more generic logger.
Appenders	Appenders are output destinations: console window, SMTP destination, Windows Event Log, …
Layouts	Layouts are the formats that appenders use to write their output

Logging

- Loggers
  - A logger is what your code interacts with to log a message.
  - It takes all message requests, regardless of level, severity, eventual destination, etc.
  - Every logger has a java-class-like name
    - Dots between words work like package and sub-package separators
    - example: logger for classes under package `be.leerstad` and its subpackages:

```
log4j.logger.be.leerstad=info
```

log4j.properties

Logging

- Loggers are organized hierarchically by the dot-delimited segments
  - Logger `be.leerstad.util` is under logger `be.leerstad` and implicitly inherits its configuration unless overridden
  - There is a root logger from which all loggers inherit their default configuration; example:
    - root logger configured for at least `error` severity, logging messages to the `stdout` appender
    - logger configured for at least `info` severity for classes under `be.leerstad` and its subpackages; inherits the `stdout` appender from the root logger

```
log4j.rootLogger=error, stdout
log4j.logger.be.leerstad=info
```

log4j.properties

Logging

- Invoking logging from your code
  - You declare loggers in your code then start logging to them
  - You do this using the `Logger.getLogger(Class c)` or `Logger.getLogger(String s)` methods
    - If the logger specified does not exist it is automatically created
    - You generally pass the class (or name of the class) that is declaring the logger

Logging

- Invoking logging from your code: example:

```
package be.leerstad.utils;

import org.apache.log4j.Logger;

public class Helper {
  public int process(int[] numbers, int length) {
    final Logger LOGGER = Logger.getLogger(Helper.class.getName());
    ...
  }
}
```

Logging

- Using a logger
  - To use a logger, log a message at the appropriate level of severity:

```
log.debug("The value of x=" + x);
log.info("System started at " + new Date());
log.warn("SQL Server down – returning");
log.error("Could not access server", ex);
log.fatal("Cannot proceed”);
```

  - Note that at any severity, you can optionally pass an exception as a param

Severity configured for the logger will determine which of these methods will result in a message in the log appender:

- if `log4j.logger.be.leerstad=info` is set, debug-level messages will not be sent to the appender; info-level, warning-level, error level and fatal-level messages will be sent to the appender
- a subclass inherits its immediate parent level severity

Logging

- Loggers logic:
  - The logger will always accept your message
  - However, the logger might not always route it anywhere
  - Where it goes is a function of
    - the message severity
    - the logger's name
    - what appenders are attached to it

Logging

- Appenders
  - Appenders are actual output destinations for the messages.
  - There are lots of different kinds of appenders, of which `System.out` (stdout) and `System.err` (stderr) are just two
  - There are also appenders for files, Swing components, remote socket servers, JMS, NT Event loggers, and remote Unix Syslog daemons

Logging

- Appenders: some examples

```
###  file log appender config
log4j.appender.file=org.apache.log4j.FileAppender
log4j.appender.file.File=myLogFile.log
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=...

###  NT event log appender config
log4j.appender.NTEvent=org.apache.log4j.nt.NTEventLogAppender
log4j.appender.NTEvent.layout=org.apache.log4j.PatternLayout
log4j.appender.NTEvent.layout.ConversionPattern=%p: %c - %m

###  console log appender config
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=...
```

log4j.properties

Logging

- Appenders
  - Log4j allows attaching multiple appenders to any logger – example:

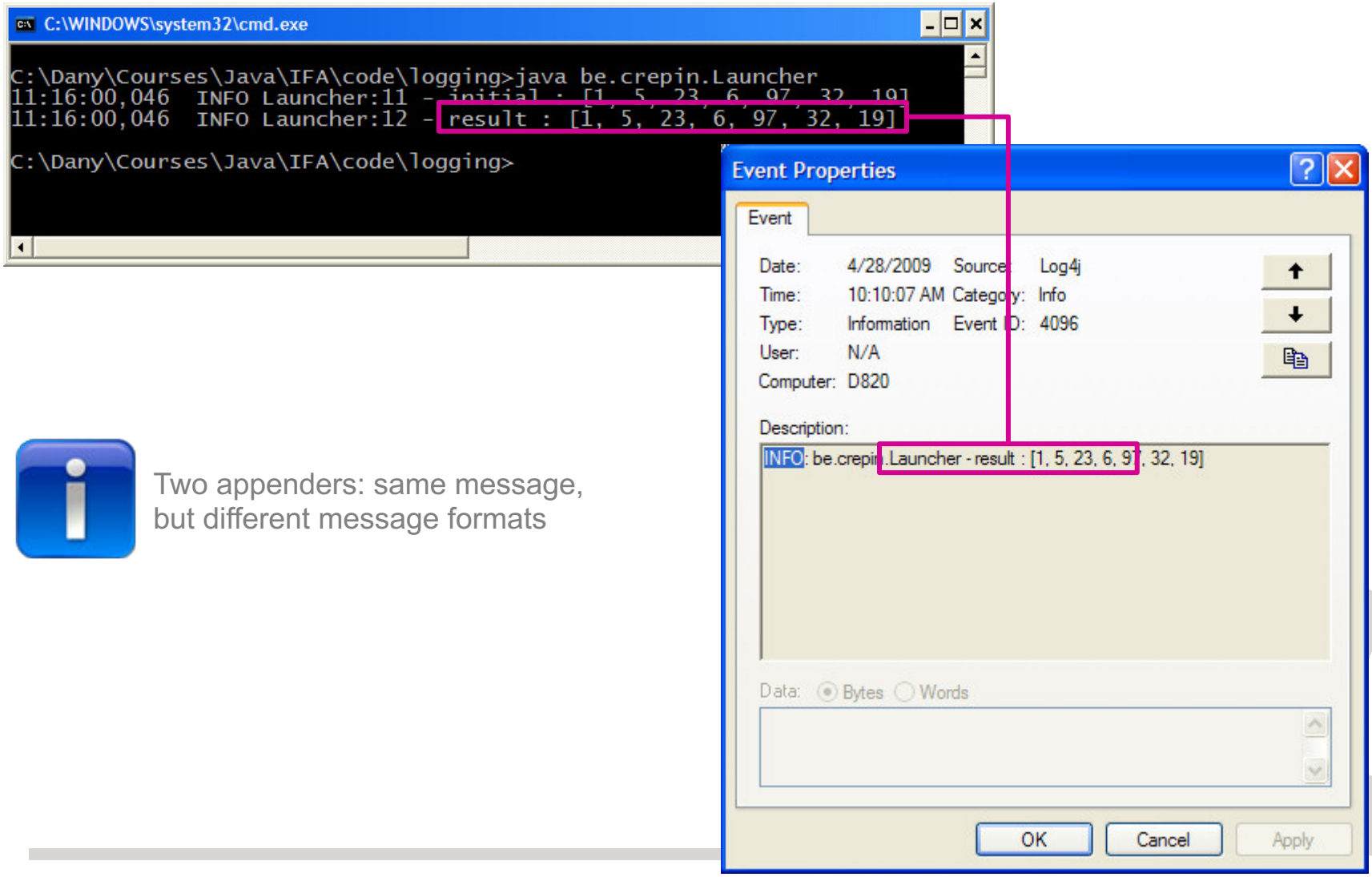
```
log4j.rootLogger=error, stdout, NTEvent
```

log4j.properties

  - You can write your own
  - You can attach or remove appenders dynamically

Logging

- Appenders: demo



Two appenders: same message,
but different message formats

Logging

- Layouts
  - Layouts are how appenders format the log messages sent to them by loggers
  - There are three layouts provided by log4j:
    - *XMLLayout* – format the logging event as an xml line
    - *HTMLLayout* – format the logging event as an HTML table
    - *PatternLayout* – format the logging event as a printf-like string

Logging

switch	description
`c`	Used to output the category of the logging event.
`C`	Used to output the fully qualified class name of the caller issuing the logging request.
`d`	Used to output the date of the logging event. The date conversion specifier may be followed by a date format specifier enclosed between braces. For example, `%d{HH:mm:ss,SSS}` or `%d{dd MMM yyyy HH:mm:ss,SSS}`. If no date format specifier is given then ISO8601 format is assumed
`F`	Used to output the file name where the logging request was issued.
`l`	Used to output location information of the caller. (C+M+L)
`L`	Used to output the line number from where the logging request was issued.
`n`	Outputs the platform dependent line separator character or characters.
`M`	Used to output the method name where the logging request was issued.
`p`	Used to output the priority (severity) of the logging event.
`t`	Used to output the name of the thread that generated the logging event.
`x`	Used to output the NDC (nested diagnostic context) associated with the thread that generated the logging event.

Exercise

Summary

- Logging with a logging framework like Log4J is more flexible than the System.out.println() method
- Logging provides information when necessary and in different possible formats
- Multiple destinations are possible, even simultaneaously
- Recompiling is not necessary