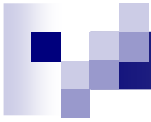




Basics of programming 3

Logging



Log4J



Logging motivation

■ Logging

□ Runtime activity

- Observing application operation

□ Persistent

- Available even after application stops
 - c.f. runtime debugging

□ Helps debugging

- During errors causes might be found
- The only help in many cases
 - Multithreaded debugging is hard



Manual logging

- `System.out.println()`
 - Simple
 - Regular and logging messages mixed
 - No filtering
 - Immovable solution
 - Switching on-off?
 - Hard to change output channel
 - Embedded systems don't have a console



System logging

- Solution: logging support
 - Log messages sent to the logging system
 - Built in support for printout
 - Requirements
 - Easy reconfiguration
 - Filtering
 - Modifiable output format
 - console, file, OS, DB
 - Efficiency
 - Logging takes as much as 4% of time on average
 - No overhead when switched off



Logging framework

■ Apache Log4J

- Born in 1999, current version is 2.0
 - 1.x and 2.x has usage differences
 - basic architecture and concepts are same
- Java class library, open source
 - single .jar file must be added to classpath

■ java.util.logging

- Since 2002 in JDK 1.4
- Very similar to log4j
 - fewer built in functionality



Log4J Logger

- The application uses a Logger object for logging

```
import org.apache.logging.log4j.*;

class MyClass {
    private static final Logger logger =
        LogManager.getLogger("MyLogger");

    public void foo() {
        logger.info("Log this");
    }
}
```



Logger methods

- Methods for sending log messages
 - `void trace(Object message [,Throwable t])`
 - `void debug(Object message [,Throwable t])`
 - `void info(Object message [,Throwable t])`
 - `void warning(Object message [,Throwable t])`
 - `void fatal(Object message [,Throwable t])`
 - For each logging level
 - Message is usually String, but can be any object
 - Even exceptions can be logged with an optional parameter



LogManager methods

- Static methods for accessing Logger objects
 - static `Logger getLogger()`
 - root instance (loggers form a hierarchy)
 - static `Logger getLogger(String name)`
 - static `Logger getLogger(Class clazz)`
 - static `Logger getLogger(Object o)`
 - instance with the given name, class, object



Logger hierarchy

- Logger objects organized into a hierarchy
 - Based on their names, automatically
 - Hierarchy levels are separated by dots (“.”)
 - Similar to package names and members
 - E.g. *hu.bme* logger is parent of *hu.bme.iit*
 - Top of hierarchy is a noname root
 - Log messages reach parent loggers as well
 - This can be explicitly overridden
 - `void setAdditivity(boolean flag)`



Logging levels

- Best practice
 - Logger name = full name of class
- Each logger instance has a logging level
 - Messages below level are discarded
 - Parents get discarded messages as well
 - If no level is set, inherits from parent



Logging levels 2

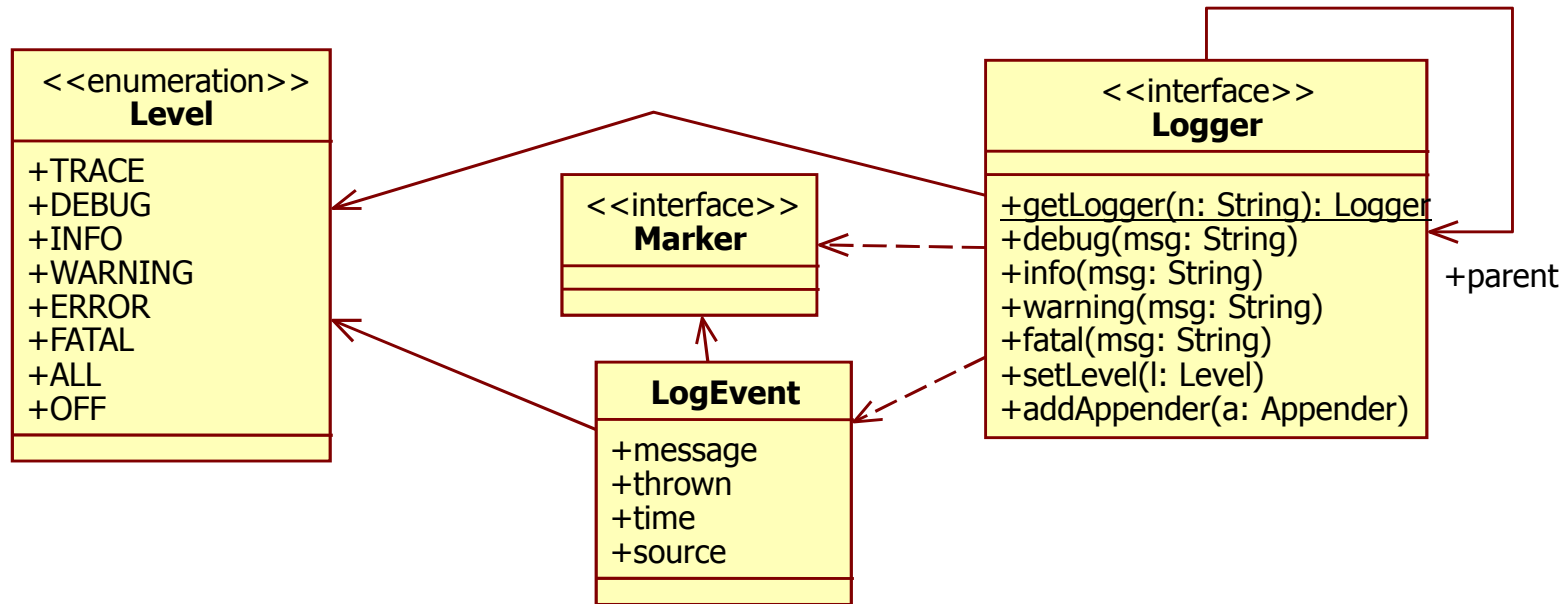
- `org.apache.log4j.Level`:
 - TRACE: Low level, relating to method calls
 - DEBUG: Log messages for debugging
 - INFO: Messages during normal operations
 - WARN: Warnings
 - ERROR: Error, an operation failed
 - FATAL: The application can not continue
- Each log message carries its level



Logging levels 3

- Special values
 - Level.ALL: All messages
 - Level.OFF: No messages
- Only used for logger configuration

Log4J class hierarchy





Appender

- Prints out the log messages
 - Each logger object can have any number of appenders
 - `void addAppender(Appender appender)`
 - Enables multiple target for log messages
 - In addition to parents' appenders
- Good range of Appenders available



Appender classes

- ConsoleAppender
 - Logging for standard output or std error
 - setTarget(String)
- FileAppender
 - Logging into a file
 - setFile(String), setAppend(boolean)
 - RollingFileAppender
 - If size limit is reached, new file is opened
 - DailyRolledFileAppender
 - After a predefined time new file is opened



Appender classes 2

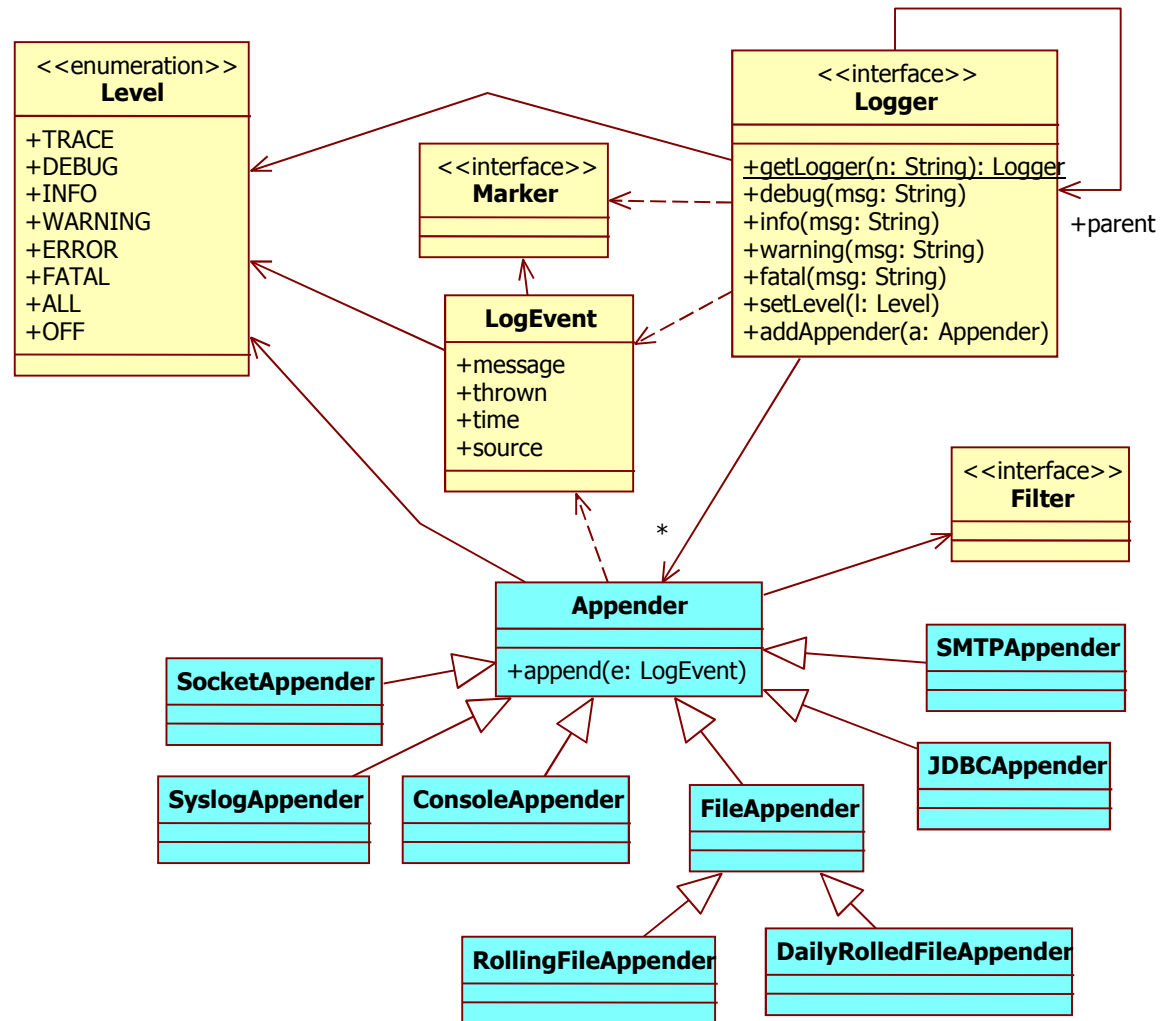
- SocketAppender
 - Sending log messages over the network
- SMTPAppender
 - Last N log messages are sent in an email
- JDBCAppender
 - Log messages stored into a DB
- SyslogAppender, NTEventAppender
 - OS specific logging is used



Appender usage

- Appenders have a wide range of configurability
 - getter-setter function provided
- Accept LogEvents
 - `accept(LogEvent e)`: abstract method
- Can have Filters
 - Filters filter events, etc.
 - decision: accept, deny, neutral

Log4J class hierarchy





LoggingEvent

- LoggingEvent object
 - Created when a logger method is called
 - Holds the *message*
 - The level of the message
 - Place and time of calling the logger method
 - Name of logger, etc.
 - From this object readable output is generated
 - Done by *Layout* objects



Layout

- Each *Appender* has a *Layout* object
 - void setLayout(Layout layout)
- Simplest *Layout* classes
 - SimpleLayout
 - Prints the level and text of the message
 - HTMLLayout
 - Creates an HTML table from the messages
 - XMLLayout
 - Creates an XML document from the messages

DEBUG - Message 1



PatternLayout

- One of the most commonly used Layout class
- Format is set by a format string
 - via *ctr* or *setConversionPattern(String pattern)*
 - example format string:
 - `"%-5p [%t]: %m%n"`

```
DEBUG [main]: Message 1
WARN  [main]: Message 2
```

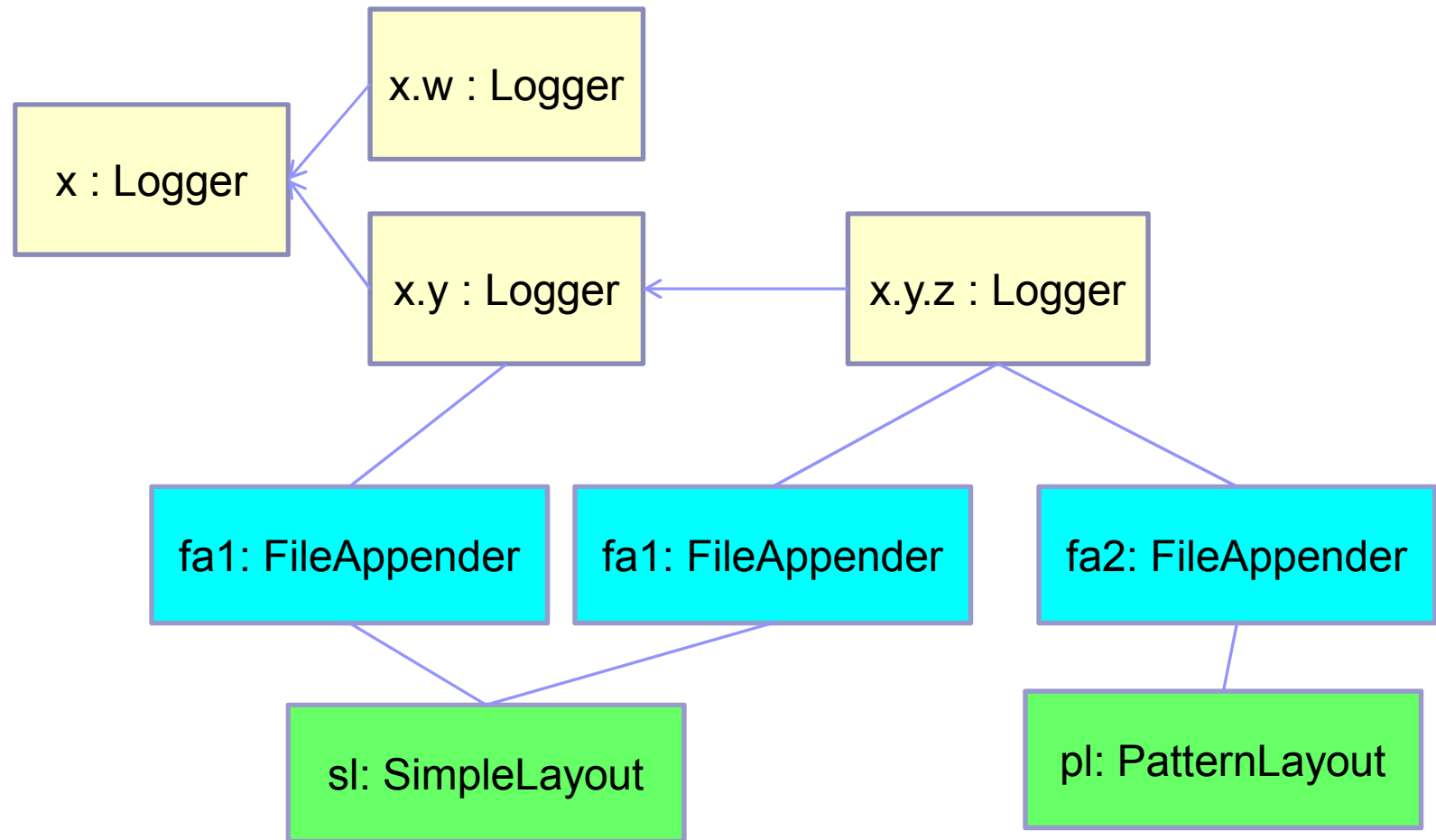


PatternLayout format

- Format string

- Any character sequence is allowed
 - Printed without modification
- Percent sign ("%") for placeholders (c.f. *printf*)
 - %c: name of logger
 - %C: name of caller class
 - %d{yyyy-MM-dd HH:mm:ss}: timestamp
 - %m: message text
 - %p: level
 - %n: newline
 - etc.

Log4J example architecture





Configuring Log4J

- Appenders have to be assigned to loggers
 - From within our application (*forbidden since 2.0*)
 - e.g. at the beginning of the main() method
 - this configuration can not be modified later

```
Logger.getRootLogger().addAppender(  
    new ConsoleAppender(  
        new PatternLayout("%p - %m")  
    ));  
Logger.getRootLogger().setLevel(Level.DEBUG);
```



Configuring Log4J /2

- Using a configuration file
 - Two formats: property file, XML file (v1.x)
 - Additional format: JSON (v2.x)
 - No need for recompilation
 - Programmatic setting for v1.0:

```
public static void main(String[] args) {  
    // either  
    PropertyConfigurator.configure("log4j.properties");  
    // or  
    DOMConfigurator.configure("log4j.xml");  
}
```



Configuring Log4J /3

- Configuration files are automatically found
 - At application startup searching in CLASSPATH
 - *log4j.xml, log4j.properties*
 - *log4j2.xml, log4j2.properties, log4j2.json*
 - They are usually put among *.class* files
 - bin directory
 - root of jar file



Configuring Log4J /4

- Log4J can be configured via system properties
 - log4j.debug: debug on/off
 - mostly for configuration errors
 - log4j.configuration: name of default config file
 - Absolute URL for exact specification
 - Relative URL for search in CLASSPATH

```
java -Dlog4j.configuration=log.conf myapp.Application
```



Log4J properties file

- Stores key=value pairs
 - different in 1.x and 2.x
 - Keys form a hierarchy
 - Mapping Logger hierarchy
 - JavaBean attributes of the classes
 - Classic solution, widespread

```
log4j.rootLogger=DEBUG, R
log4j.appender.R=org.apache.log4j.ConsoleAppender
log4j.appender.R.layout=org.apache.log4j.SimpleLayout
log4j.appender.R.threshold=INFO
```



Log4J properties in XML

- XML file (1.x differs from 2.x)
 - Sui generis hierarchical, easier to read (?)
 - Newer approach
 - Newer config options only in XML

```
<log4j:configuration xmlns:log4j="...">
  <appender name="R" class="org.apache.log4j.ConsoleAppender">
    <param name="threshold" value="info" />
    <layout class="org.apache.log4j.SimpleLayout" />
  </appender>
  <root>
    <priority value="debug" />
    <appender-ref ref="R" />
  </root>
</log4j:configuration>
```




Log4J properties in XML (v2)

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration package="logging" status="WARN">
  <Appenders>
    <Console name="Screen" target="SYSTEM_OUT">
      <PatternLayout pattern="%msg%n"/>
    </Console>
    <File name="LogFile" fileName="log4j.txt">
      <PatternLayout pattern="File: %d{HH:mm:ss} [%t] %-5level
        %logger{36} - %msg%n"/>
    </File>
  </Appenders>
  <Loggers>
    <Logger name="logging.Main" level="trace">
      <AppenderRef ref="Screen"/>
      <AppenderRef ref="LogFile"/>
    </Logger>
    <Root level="trace">
      <AppenderRef ref="Screen"/>
    </Root>
  </Loggers>
</Configuration>
```



Log4J properties in JSON

- JSON file (2.x only)
 - Sui generis hierarchical, easier to read (?)
 - Even newer approach
 - Direct mapping between XML and JSON

```
{"configuration":  
  { "package": "logging", "status": "warn",  
    "appenders": { "appender": [  
      { "type": "Console", "name": "Screen", "target": "SYSTEM_OUT",  
        "PatternLayout": { "pattern": "%msg%n" } },  
      ...  
    ] },  
    "loggers": { "logger": [  
      { "name": "logging.Main", "level": "trace",  
        "AppenderRef": { "ref": "Screen" } },  
      "root": { "level": "trace", "AppenderRef": { "ref": "Screen" } }  
    ] }  
  }  
}
```



Property vs XML – an example

```
log4j.debug=true
```

```
log4j.rootLogger=INFO, logfile
```

```
log4j.logger.hu.bme=DEBUG, console
```

```
...
```

```
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/"  
  debug="true">
```

```
<root>  
  <priority value="info" />  
  <appender-ref ref="logfile" />  
</root>
```

```
<logger name="hu.bme">  
  <level value="debug" />  
  <appender-ref ref="console" />  
</logger>
```

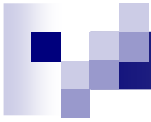
```
...
```

Property vs XML – an example

```
log4j.appender.console= \
    org.apache.log4j.ConsoleAppender
log4j.appender.console.target= \
    System.out
log4j.appender.console.layout= \
    org.apache.log4j.SimpleLayout

log4j.appender.logfile= \
    org.apache.log4j.FileAppender
log4j.appender.logfile.fileName= \
    app.log
log4j.appender.logfile.layout= \
    org.apache.log4j.PatternLayout
log4j.appender.logfile.layout. \
    ConversionPattern=%d %c - %m%n
```

```
<appender name="console" class=
"org.apache.log4j.ConsoleAppender">
  <param name="target"
    value="System.out" />
  <layout class=
"org.apache.log4j.SimpleLayout" />
</appender>
<appender name="logfile" class=
"org.apache.log4j.FileAppender">
  <param name="fileName"
    value="app.log" />
  <layout class=
"org.apache.log4j.PatternLayout">
    <param name=
      "ConversionPattern"
      value="%d %c - %m%n"/>
  </layout>
</appender>
</log4j:configuration>
```



java.util.logging



java.util.logging

- Similar architecture to Log4J

- *Logger*

- used to log messages for a specific component.
 - methods with level names (`logger.info("message")`)
 - hierarchical namespace

- *LogRecord*:

- passes logging requests between the logging framework and individual log handlers.
 - level, source name, message, millis, sequenceNumber, etc.



java.util.logging

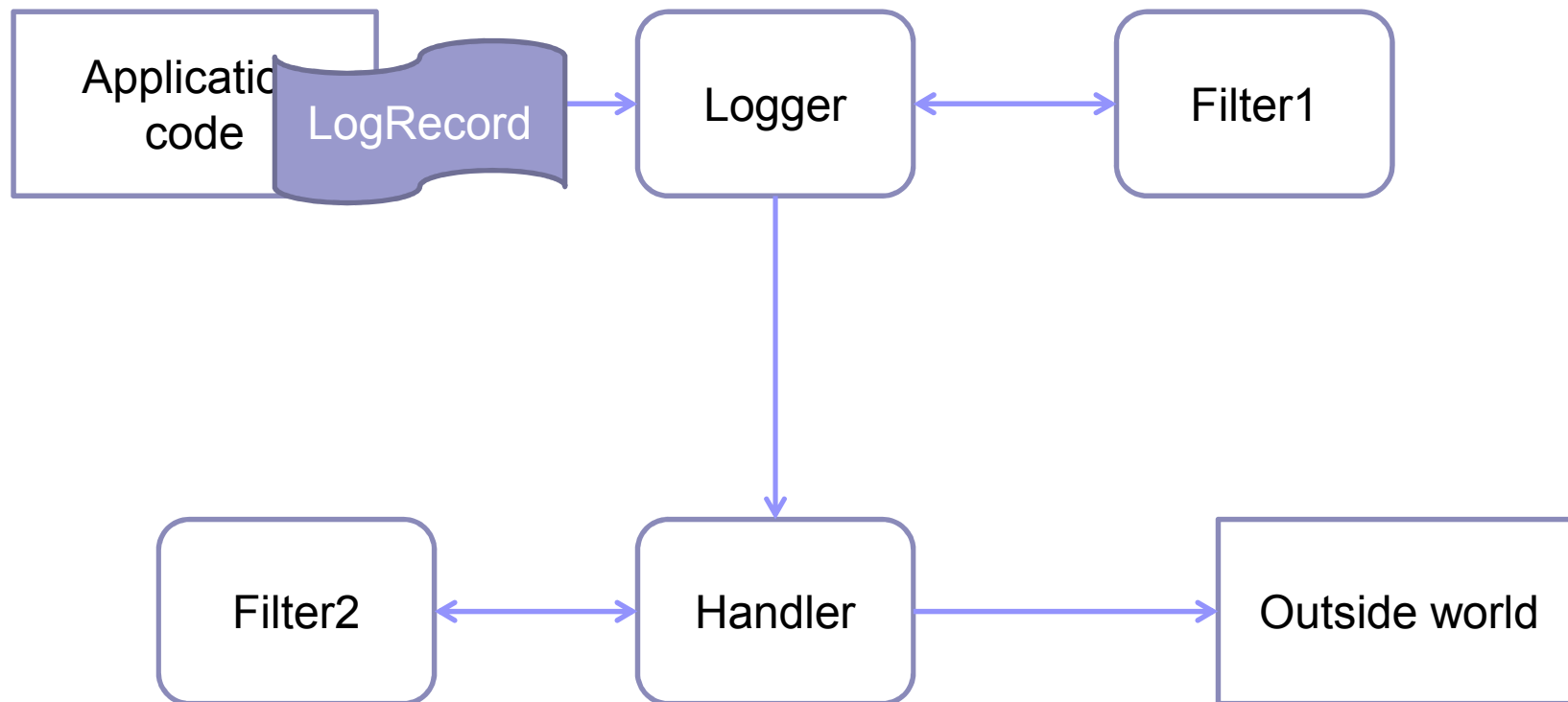
- *Level*
 - a set of standard logging levels
 - OFF, SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST, ALL
- *Handler (like Appender in log4j):*
 - Exports *LogRecord* objects to a variety of destinations
 - e.g. ConsoleHandler, FileHandler, SocketHandler, StreamHandler
- *Formatter (like Layout in log4j)*
 - supports formatting *LogRecord* objects
 - e.g. SimpleFormatter, XMLFormatter



java.util.logging

- *Filter (interface)*
 - fine-grained control over what gets logged
 - `boolean isLoggable(LogRecord record)`
- *LogManager*
 - keeps track of global logging information
 - logging control properties read from the configuration file


Logging in action





Example use (def. config.)

```
public class Nose {
    // Obtain a suitable logger.
    private static Logger logger =
        Logger.getLogger("com.wombat.nose");
    public static void main(String argv[]) {
        // Log a FINE tracing message
        logger.fine("doing stuff");
        try {
            wombat.sneeze();
        } catch (Exception ex) {
            // Log the exception
            logger.log(Level.WARNING, "trouble sneezing", ex);
        }
        logger.fine("done");
    }
}
```



Example use (inline config)

```
public class Nose {
    // Obtain a suitable logger.
    private static Logger logger =
        Logger.getLogger("com.wombat.nose");
    private static FileHandler fh =
        new FileHandler("mylog.txt");
    public static void main(String argv[]) {
        // Send logger output to our FileHandler.
        logger.addHandler(fh);
        // Request that every detail gets logged.
        logger.setLevel(Level.ALL);

        //...
    }
}
```