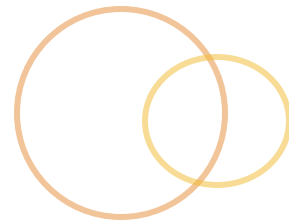
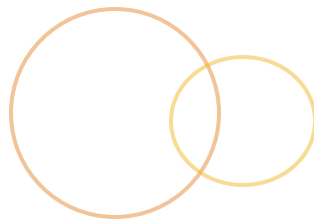


# Java Logging Tools

Yes you did, it says to here!

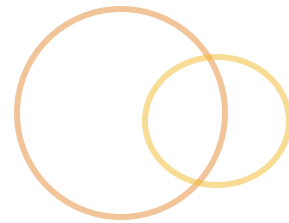
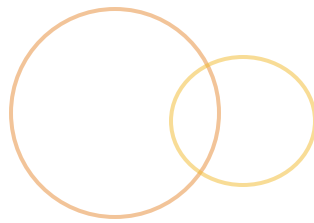


# Objectives



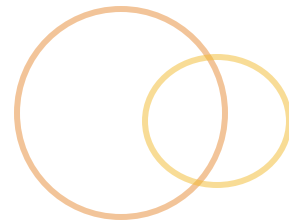
- Write log messages at different severity levels
- Create loggers appropriate to the subsystem they will report for
- Write log messages including exceptions
- Minimize CPU effort involved in preparing messages that are filtered out
- Configure the logging system to write to a file and/or to the console

# Objectives



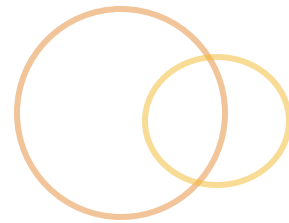
- Configure the filtering of log messages by their severity level, based on the subsystem originating the message and the destination of the message
- Control the format of plain text log messages

# Log Organization



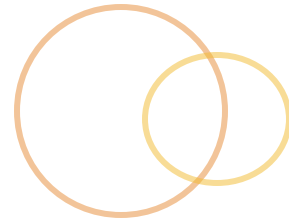
- Logs are typically organized by:
  - Subsystem issuing the message
  - Severity of the issue being reported
- The subsystem is usually identified by the Java package of the class issuing the message
  - The use of package names is not mandatory; a class can use the logger of a different package, but such behavior might be confusing
  - More generally, a dotted hierarchical notation is recommended
- Different subsystems are given their own Logger object

# Log Organization



- The severity of events being logged is categorized by an enumerated Level value
- Log messages sent to a particular logger will be filtered by level by the logger
- Loggers have associated Handler and Formatter objects
- Handlers write to destinations, but also have a filter level

# Obtaining A Logger



- Most classes will grab a Logger during static initialization and keep it for life

```
public class Banana {  
    private static final Logger logger =  
        Logger.getLogger(Banana.class.getName());  
    // ...  
}
```

The package name is often sufficient, unless very fine control is required

# Simple Writing Of A Log Message



- Log messages are simple to write

```
LOG.log(Level.SEVERE, "That broke!");
```

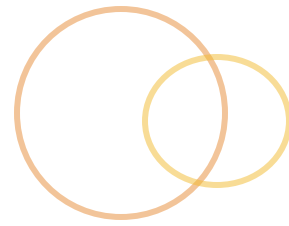
- Can specify class and method:

```
LOG.logp(Level.SEVERE,  
    "com.pkg.MyClass", "myMethod", "That broke!");
```

- And other utility/shortcut methods

Method	Effect	Method	Effect
severe	Log at severe level	finer	Log at finer level
warning	Log at warning level	finest	Log at finest level
info	Log at info level	entering	Log entering method, at finer level
fine	Log at fine level	exiting	Log exiting method, at finer level
		throwing	Log throwing an exception (finer)

# Logging Exceptions



- ⦿ Exceptions can be logged in a number of ways
  - ⦿ Using the throwing method, which takes three arguments, including a Throwable:

```
FileNotFoundException fnfe = new  
    FileNotFoundException(  
        "File data.dat does not exist");  
LOG.throwing("AClass", "main", fnfe);  
throw fnfe;
```

- ⦿ Using the Logger method

```
log(Level l, String msg, Throwable t)
```



# Efficient Logging



- Complex log messages can be expensive to prepare, involving method calls to find data, extraction of stack traces, and concatenation of strings. If done extensively, this can waste CPU power—unfortunate if no output will result
  - Test if this Logger will do anything with the message:  
`if (LOG.isLoggable(Level.FINE)) //...`  
But Note this test is not fully accurate as messages can be filtered later
  - Since Java 8, can use a `Supplier<String>` instead of a `String`  
`LOG.log(Level.SEVERE, () -> "That broke!");`

# Configuration Of Loggers



- Loggers use Handlers to write to destinations, such as disk files
- Loggers and Handlers both have severity levels, and Filters.
- Handlers can also have formatters
- These options can be controlled programmatically, but are almost always controlled by a configuration file

# Configuration Of Loggers



- Default config is in `JRE_HOME/lib/logging.properties`
- Override the default logging config file with the property `java.util.logging.config.file`  
**-Djava.util.logging.config.file=LogConfig.properties**
- General layout of configuration file is:
  - Handlers
  - Handler level configuration
  - Per-handler configuration, including format configuration
  - Per-subsystem level config

# Sample Configuration File Part 1



```
handlers = java.util.logging.FileHandler,  
           java.util.logging.ConsoleHandler
```

List of Handler classes to be installed.  
ConsoleHandler writes to the console,  
FileHandler writes to files

Note, these are  
single lines,  
wrapped to fit

```
java.util.logging.ConsoleHandler.level = INFO  
java.util.logging.ConsoleHandler.formatter =  
    java.util.logging.SimpleFormatter
```

Output format  
control, XML optional

Level limit for this  
handler

# Sample Configuration File Part 2



```
java.util.logging.FileHandler.pattern =  
    %h/NetBeansProjects/LogScratch/java%u.log  
java.util.logging.FileHandler.limit = 50000  
java.util.logging.FileHandler.count = 4  
java.util.logging.FileHandler.formatter =  
    java.util.logging.SimpleFormatter
```

Output  
formatting  
control

Number of files  
permitted by this  
logger.

Output file for  
file-based logger  
%h = home dir  
(single line)

# Sample Configuration File Part 3



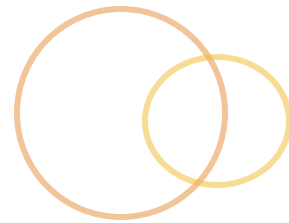
```
.level = WARNING  
com.di.sample.level = WARNING
```

Sets log filtering level for  
com.di.sample logger to  
**WARNING**

Note, if filter level for  
handler is higher, that  
overrides this  
configuration

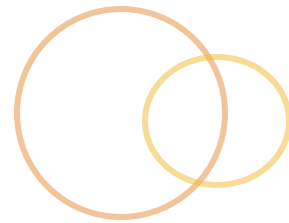
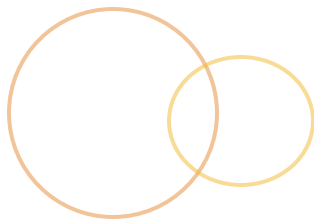
Sets log filtering level for  
the root name, this is  
typically inherited into all  
other loggers, unless  
explicitly reconfigured,  
as at left

# Logging Handlers



- ◎ Four handlers are built into the logging API
  - ◎ ConsoleHandler—Sends log messages to the system console (System.err)
  - ◎ FileHandler—Sends log messages to a configurable file
  - ◎ SocketHandler—Sends log messages over a network socket
  - ◎ MemoryHandler—Stores log messages in a circular buffer in memory. Allows the current buffer to be pushed out to a separate handler class on-demand

# Formatters



- Handlers use formatters to prepare messages for publication. There are two built-in formatters
  - SimpleFormatter
  - XMLFormatter
- The SimpleFormatter allows control of the format of log messages, by contrast the XML format is standardized



# Configuring SimpleFormatter



- The property `java.util.logging.SimpleFormatter.format` is a format string (similar to `printf` formatting) that is used to control the output format.
- Positional arguments to the format operation are
  - `%1$tc` — The date
  - `%2$s` — Source (fully qualified class name)
  - `%3$s` — Logger name
  - `%4$s` — Severity level
  - `%5$s` — Message
  - `%6$s` — Exception stack trace (if applicable)

# Configuring SimpleFormatter



## Example format string:

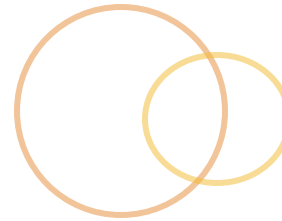
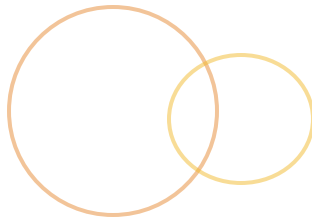
```
java.util.logging.SimpleFormatter.format =  
%4$s--%1$tc, %2$s: %5$s %6$s%n
```

All one line

Severity, date, originating  
class, message, exception

%n = Append end  
of line character!

- Note: any error in the format string result in the default format being used



- Log4J is conceptually similar to the Java Logger
  - But is generally considered to have greater flexibility in configuration
- Log4J predates Java Logging, and has a significant foothold
  - Many third party libraries expect to use it, making it sometimes hard to avoid
  - Other frameworks exist, including some that attempt to integrate other frameworks behind a single facade

# Log4J Usage Overview



- Logging with Log4J is similar to using Java logging
  - Obtain a logger for the current subsystem
  - Send a message with a severity level to that logger
- As with Java logging, it's preferable to defer creation of messages until it's certain they'll be needed
  - Log4J does not currently support the use of `Supplier<String>` in messages, but does allow the provision of a format string and details so that final assembly of the message can be avoided when unnecessary

# Log4J Usage Overview



- Usually built with maven, dependency is

```
<groupId>org.apache.logging.log4j</groupId>  
<artifactId>log4j-core</artifactId>
```

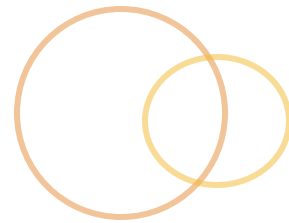
- Obtain a “formatting Logger”

```
private static Logger logger =  
    LogManager.getLogger(MyClass.class);
```

- Sending messages

```
logger.error("That broke!");  
logger.info("There was a disturbance in the %s",  
    "Force");
```

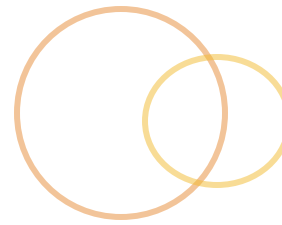
# Configuring Log4J



- Log4J reads configuration from a file
  - Many formats, including XML, JSON, YAML, and many locations are supported
- Most rudimentary form is to place a file `log4j2.xml` on the classpath
- Another simple approach is to define the property `log4j.configurationFile` to specify the location of the xml configuration file

Notice that the default file has a “2” in the name, the config property does not.

# Configuring Log4J



## XML config file has this basic structure:

```
<Configuration ...>
```

```
  <Appenders>
```

```
    <Appender name="xxx"...> ... </Appender>
```

```
  </Appenders>
```

**Appenders write output, e.g.  
to files, or console**

```
  <Loggers>
```

```
    <Logger name="a.b.c" level="error">
```

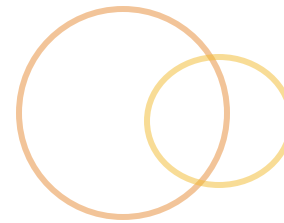
```
      <AppenderRef ref="xxx"/>
```

```
    </Loggers>
```

```
</Configuration>
```

**Loggers accept messages from  
subsystems, filter by severity, then  
send to one or more Appenders**

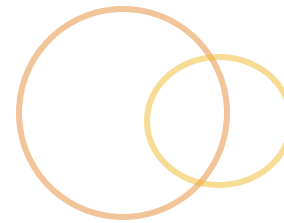
# Lab Exercise



- Create a configuration file for Java Logging that sends all messages at all levels to both the console, and to a file in your home directory
- Create a main method in a class `com.mystuff.TestLogging` that sends three messages, at levels severe, warning, and finer.
  - Verify that these show up on the console and in the expected file



# Lab Exercise



- ⦿ Arrange that messages from the subsystem `com.mystuff` are filtered at the warning level, then re-run the program
  - ⦿ Verify that only two messages are logged this time
- ⦿ Modify the configuration so that the console logger only logs messages at a severe level
  - ⦿ Verify that two messages are logged in the files, but only one shows up on the console