ECSE 321 – Tutorial 10

Logging

Dominic Charley-Roy
https://github.com/dominiccharleyroy
dominic.charley-roy @ mail.mcgill

Let's Talk About System.out

System.out.print(ln)

- Lets you print strings to standard output.
- Great for quick and dirty debugging!

But...

... it's too simple!

Think of these use cases.

- You want all the program output to be saved in a file.
- You aren't running the program from the terminal/from in Eclipse.
- You want to disable logging output without deleting all the System.out code!
- You want to copy everything output in System.out and put it in a textbox on your UI?

Ideally...

- Currently logging code is coupled with what we're actually trying to log.
  - What if you want to change how logging is done?
- We want some **abstract logging classes**!

Enter log4j

http://logging.apache.org/log4j/2.x/

A log4j Primer

- Log4j is a 3rd party library.
- Exposes an API for logging!
- Can easily create your own loggers. Examples:
  - Logging to file
  - Logging to standard output
- Lets you filter logs
- **Separates logging from the text you are logging!**

Adding it to your project...

- **Step 1:** Download the log4j zip files
- **Step 2:** Extract the log4j-core-2.1.jar and log4j-api-2.1.jar files into your project's lib folder
- **Step 3:** Add the libraries to our class path

Adding to Class Path

To work with an external jar file, we need to add it to our class path.

In Eclipse:
In the *Package Explorer*, right-click on the jar file and select Build Path > Add to Build Path

Logging Levels

- Log4j introduces the notion of **levels** associated with a logging message.
- These levels are meant to demonstrate the importance of the log message.
- These follow an order. From lowest priority to highest:
  - **TRACE, INFO, DEBUG, WARN, ERROR, FATAL**

Configuring Log4j

- Put configuration in src/**log4j2.xml** file!
- This default will show all messages in the console, regardless of level.

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
  <Appenders>
    <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
    </Console>
  </Appenders>
  <Loggers>
    <Root level="all">
      <AppenderRef ref="Console"/>
    </Root>
  </Loggers>
</Configuration>
```

Creating a Logger

- Instantiate a logger with a name
  - Usually the class name
- We have to specify how we are configuring the logging system. Let's stick to the defaults for now!

```
public class MyApplication {

        private static final Logger logger =
            LogManager.getLogger("My Application");

}
```

Hello, world!

- Our logger has an **info** method for logging a message at the info level.

```
logger.info("Hello, world!");
```

```
00:51:42.016 [main] INFO  MyApplication - Hello, World!
```

More Logging Methods

The **Logger** has methods for the various levels:

```
logger.warn("This is the warning level.");
logger.error("This is the error level.");
logger.fatal("This is the fatal level.");
logger.debug("This is the debug level.");
logger.trace("This is the trace level.");
```

There is also a generic logging method:

```
logger.log(Level.WARN, "This is a warning.");
```

Time
for log4j
to shine!

Mixing in Parameters

The logging methods let you some basic string interpolation. Instead of...

```
logger.info("The user " + username + " reset their password to " + password);
```

We can do:

```
logger.info("The user {} reset their password to {}", username, password);
```

Formatted Loggers

We can get even fancier! If we use a formatted logger:

```
Logger logger = LogManager.getFormatterLogger("MyApplication");
```

Then we can mix in variables and format them! This is kind of like **printf** in C/C++ or the **StringFormatter**.

```
logger.info("The order of %s had %d item(s) for a total of %10.2f$.",
    username, items, price);
```

```
01:01:03.539 [main] INFO  MyApplication - The order of Dominic had 10 item(s) for a total of    1000.57$.
```

LogEvents

When you call a logging method, a LogEvent is created using your text.

This has a message, a level, and an associated logger.

Appenders

- These classes are responsible for processing LogEvent objects!
- They are called when you log something!
- This is what lets you do all sorts of things like
  - Write message to console
  - Write message to file
  - Write message to a database
  - Write to a socket
  - ... WHATEVER YOU LIKE!

Multiple Appenders

It's very easy to have more than one appender! Each has a unique name. For example, if you have a File appender and a Console appender:

```
<Appenders>
  <Console name="Console" ...>
    ...
  </Console>
  <File name="FileAppender" ...>
    ...
  </File>
</Appenders>
```

To use both:

```
<Root level="all">
  <AppenderRef ref="Console"/>
  <AppenderRef ref="FileAppender"/>
</Root>
```

File Appender

The File tag lets you write log events to a file! It can be configured using these parameters:

```
<File name="FileAppender"
          fileName="problems.log"
          append="true"/>
```

- name is a unique name for the appender
- fileName is the name of the file we want to write to
- append means the log file should not be re-created every time!

RollingFileAppender

- This Appender lets us have a bit more control over the way logging to file works!
- Suppose you want to have small log files (no more than 5MB each) and you only want to keep enough logging information for 3 log files. Can't really do this with the standard FileAppender!
- Think of this like a big **queue**.

RollingFileAppender Example

This will keep up to 4 log files (log.txt, log-1.txt, log-2.txt, log-3.txt). The last three files are *archives*, so log.txt has the freshest logs! Each log file is limited to 5MB.

```
<RollingFile name="RollingFile" fileName="log.txt"
      filePattern="log-%i.txt">
 <PatternLayout>
   <Pattern>%d %p %c{1.} [%t] %m%n</Pattern>
 </PatternLayout>
 <Policies>
    <TimeBasedTriggeringPolicy/>
    <SizeBasedTriggeringPolicy size="5 MB"/>
 </Policies>
 <DefaultRolloverStrategy max="3"/>

</RollingFile>
```

You can also do time-based rolling over with this!

STMP Appender

This can send out an email every time you get *x* log events! Suppose every 50 messages you want an email:

```
<SMTP name="Mail" subject="Error Log"
      to="errors@mywebsite.com"
      from="errors@mywebsite.com"
      smtpHost="localhost" smtpPort="25"
      bufferSize="50">
</SMTP>
```

Be careful with this!

Filters

Filters let you filter log events such that only some will get sent to particular log appenders!

For example, you could keep all events of level warn or higher in a file and ignore anything less.

Filters are added as a child node of an Appender.

ThresholdFilter

This lets you specify a particular log level. You can then say what to do with log events that are that level or higher (**match**) as well as with those that are lower (**don't match**).

Example: If you only want to keep events of level WARN or higher:

```
<Console name="Console" target="SYSTEM_OUT">
  <ThresholdFilter level="WARN"
                   onMatch="ACCEPT"
                   onMismatch="DENY"/>
</Console>
```

Burst Filter

- This filter lets you specify a maximum amount of messages to show before silently discarding the rest.
- For this, you specify a **level**, a **rate** and a **max**.
  - The rate is the average amount of events *below the level* per second.
  - The max is the total number of events more than the average that we will keep.
- For example, if our usual amount of events below info is 10 per second, and we want to allow at most 5 times that:

```
<BurstFilter level="INFO" rate="10" maxBurst="50"/>
```

An Example

Suppose you want to write all log messages to a console and only messages with a level of warning or higher to a file "problems.log". We have two appenders:

```
<Appenders>
  <Console name="Console" target="SYSTEM_OUT">
    ...
  </Console>
  <File name="FileAppender"
        fileName="problems.log"
        append="true">
      <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
      <ThresholdFilter level="WARN" onMatch="ACCEPT"/>
  </File>
</Appenders>
```

To use both:

```
<Root level="all">
  <AppenderRef ref="Console"/>
  <AppenderRef ref="FileAppender"/>
</Root>
```

More!

- You can...
  - have many loggers! (Eg. one logger per class)
  - specify specific configurations for each logger!
  - combine filters
  - change the layout of what is printed when logging
  - create your own appenders
  - filter based on regular expressions
  - ... AND MUCH MORE!
- The log4j manual: http://logging.apache.org/log4j/2.0/manual/index.html

Recap

- System.out couples **what** we're logging with **how** we're logging.
- Log4j gives a clean API for logging to separate this!
- Use log4j2.xml to specify **how logging is done**.
- Use Logger objects to log messages without worrying about the how.
  - Log Events go to Appenders which decide what to do with them.
  - Log Events may be filtered out by an Appender's Filters.