Logging

Contents

Overview

Logging is the process of writing messages during the execution of a program to a central place. This allows us to report and persist error, warning, and info messages, which can be retrieved and analysed later.

Java contains its own logger in `java.util.logging.Logger`, though it is generally accepted to use a third-party logger instead. The most well-used third party logger framework is `Log4j`.

We can also use an abstraction layer, such as `SLF4J`, which decouples our code from the underlying logging framework so that we can switch between different logging frameworks.

Tutorial

For the purposes of this tutorial, we shall use `Log4j`. The latest version at the time of writing is `2.13`.

(*note: do not use any `1.x` version of `Log4j`, as it is **extremely** deprecated.*)

`Log4j` has extended functionality over Java's boilerplate logger:

- it allows for greater definition of logging, known as *log levels*
- it allows output of log messages to different destinations than the standard console output (files, databases, etc.)
- it contains an interface so that we can create our own implementation
- each logger we create is managed by a single `LogManager`, which allows us to use multiple loggers within the same program

Previously we might have been using `System.out` or `System.err` to print messages back to the console, which is considered bad practice due to its performance issues.

Log Levels

`Log4j` provides more flexibility with what we print in which environments through the use of **log levels**.

This allows for a greater degree of configuration, which can be useful when debugging code. For instance, we could turn on (or off!) every type of log statement, which reduces the time taken to debug code by not reading through a gigantic stack trace, and thus improves the maintainability of the code which we create.

`Log4j`'s log levels cascade from highest to lowest according to the below list - activating a higher log level will print all levels beneath it too:

- `all` : Used to turn on all logging levels, including custom levels
- `trace` : Provides more detailed information than the debug level
- `debug` : Focused on providing code debugging support to the developer
- `info` : Provides information on the progress and chosen state - generally useful for the end user
- `warn` : Warns a user of unexpected events within the application which do *not* cause the application to break
- `error` : Provides information on serious errors which need addressing, and may cause the application to break
- `fatal` : Informs the user of complete application failure
- `off` : Used to turn off all logging levels, including custom levels

Maven Dependency

The dependency for log4j is as follows, as retrieved from maven repository:

```
<!-- https://mvnrepository.com/artifact/org.apache.logging.log4j/log4j-core -->
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.13.3</version>
</dependency>
```

Configuration File

For configuring which log messages to display, how they look, and where to log those messages to, you must use a separate `log4j2.xml`, which should be stored in your `src/main/resources` folder.

Luckily, the `xml` is not too difficult to create:

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration>
  <Appenders>
    <File name="FILEOUT" fileName="errors.log" append="false">
      <PatternLayout>
        <Pattern>%d{yyyyMMdd HH:mm:ss} %-5p [%t] %C{2} %F%L - %m%n</Pattern>
      </PatternLayout>
    </File>
    <Console name="STDOUT" target="SYSTEM_OUT">
      <PatternLayout pattern="%m%n"/>
    </Console>
  </Appenders>
  <Loggers>
    <Root level="debug">
      <AppenderRef ref="FILEOUT" level="debug"/>
      <AppenderRef ref="STDOUT" level="info"/>
    </Root>
  </Loggers>
</Configuration>
```

There are two sections to note here - `Appenders` and `Loggers`:

- `Appenders` lets us set what our messages look like (with `PatternLayout` and `Pattern`), and where they are sent - here, we're outputting both to a `File` (`errors.log`) and to the `Console` (in a standard `System.out`).
- `Loggers` lets us choose the initial `Root` log level, and gives us the choice of which `Appender` to use with which log level - here, we're passing our `debug` messages to the `errors.log` file and the `info` messages to the console.

Import

Since `Log4j` is a separate library, we need to import it:

```
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
```

The `LogManager` manages all the `Logger`s we might have in our application, so we need to import both.

Creating a Logger

To create a logger we need to do the following inside a class.

```
public static Logger LOGGER = LogManager.getLogger();
```

The logger is instantiated as `static` so that there is only one logger instance per class.

In previous versions, you would have had to pass the class name to the `getLogger()` method, which is no longer necessary in version `2.x`.

Using a Logger

Now that we have created our logger we can use it from anywhere in the class.

We can change the `.info` to whatever log level we want to print at:

```
public void logMessage(String message) {
    LOGGER.info(message);
}
```

We could also hard-code the log message:

```
LOGGER.fatal("The program seems to have exploded.");
```

Similarly, we could pass in exception variables, for instance, in a `catch`-block:

```
catch (SQLException e) {
    LOGGER.debug(e.getStackTrace());
    LOGGER.error(e.getMessage());
}
```

Setting a testing configuration

By default, any JUnit tests you write in an application will use the `log4j2.xml` file.

Luckily, it is possible to use a secondary `xml` for testing purposes. This can be useful for splitting development debugging from test debugging.

Two criteria need to be fulfilled to get this working:

- placing a `log4j2-test.xml` inside the `src/test/resources` folder
- reference this configuration file by name inside your `@BeforeClass`-annotated test setup method:

```
@BeforeClass
public static void setup() {
    System.setProperty("log4j.configurationFile", "log4j2-test.xml");
}
```

You can now use your logger in the same way as any of your other classes alongside any of the test configurations you set in your testing `xml`.

Exercises

There are no exercises for this module.