SimpleLogging Package

Version 2.1
February 2023

Gray Watson

Table of Contents

SimpleLogging

Version 2.1 – February 2023

The goal of this library is to be a small logging facade that backends to a number of standard logging packages and that can be copied into another project. This allows you to write your code and include log messages without having a fixed dependency on any one logging package. I include this code into my libraries and so they can stay agnostic. This logging code allows you to write messages with the slf4j-style `{}` argument support, handles arrays appropriately and supports up to 4 arguments before forcing the caller to pass in an object array or calling a different `logArgs(...)` method for variable arguments.

I understand that this facade is similar to other logging systems which separate their API from the implementation. I think SimpleLogging is better than the others because it doesn't use the classpath order to satisfy the connection between the API and the backend. SimpleLogging includes calls directly to specific backend APIs which can be chosen through code or configuration. This direct calling allows for more control over the backend selection and usually fewer dependencies.

SimpleLogging is also designed to be copied into your open source project so you don't have to add a maven dependency. Just copy the java files from `src/main/java` into your source tree and rename the packages as necessary. Please also copy the `SIMPLELOGGING_LICENSE.txt` file which is the very permissive ISC license.

To get started quickly using SimpleLogging, see Chapter 1 [Quick Start], page 2. There is also a HTML version of this documentation.

Gray Watson `http://256stuff.com/gray/`

1 Start Using Quickly

To use SimpleLogging in your code is similar to other logging libraries that you are used to.

```
// usually a logger will be per-class
// getLogger() also can take a String label
private static final Logger logger =
   LoggerFactory.getLogger(MyClass.class);
...

// log trace message with arguments
// toString() on the args only called if trace messages enabled
logger.trace("some trace information: {} and {}", arg1, arg2);
...

// NOTE: exception argument comes _before_ the
// message format to not confuse the arguments
logger.error(exception, "http client threw getting URL: {}", url);
```

For more extensive instructions, see Chapter 2 [Using], page 3.

2 Using SimpleLogging

2.1 Downloading Jar

To get started with SimpleLogging, you will need to download the jar file. The SimpleLogging release page is the default repository but the jars are also available from the central maven repository.

The code works with Java 6 or later.

2.2 Depending on SimpleLogging in Your Library

To use the SimpleLogging code in your library, you can either depend on the jar or just copy the classes into a logging package in your code and rename the package as necessary. You may want to tune the order of discovery in the `LogBackendType` enumerated class or remove support for certain loggers that your library doesn't need to support.

SimpleLogging is designed to be copied into your open source project so you don't have to add a dependency. Just copy the java files from `src/main/java` into your source tree and rename the packages as necessary. Please also copy the `SIMPLELOGGING_LICENSE.txt` file which is the very permissive ISC license. You may want to change the constants in `LoggerConstants.java`.

2.3 Logging Messages with {} Arguments

SimpleLogging supports "slf4j" type of arguments. Whenever the `{}` characters appear in your logging message, the library will look for arguments to substitute into the `{}`. For example:

```
String host = "server1";
...
logger.info("connected to host {}", host);
```

If info messages are being displayed then the following message will be logged:

```
connection to host server1 failed
```

This can be accomplished with a simple `StringBuilder` but the big difference is that no unnecessary objects will be created *unless* the log message is to be written. If the logger is limited to warning messages and above, no `StringBuilder` will be created and the `toString()` method for the arguments will never be called if the above info log call is made.

By default the various log methods (`logger.trace(...)`, `logger.debug(...)`, ...) take 0 to 4 arguments with no objects being created (aside from some possible auto-boxing of numbers). If you need more arguments you can use the argument-array with a explicit `new Object[] { ... }` pattern or use variable arguments with a call to `logger.traceArgs(...)`, `logger.debugArgs(...)`, etc.. The variable arguments also creates an `Object[]` regardless of whether or not the log message will be logged so SimpleLogging wants you to be explicit about creating it. It is recommended that you test

for the log level with an `if` statement when you are logging with the `Object[]` or variable arguments.

```
if (logger.isLevelEnabled(Level.TRACE)) {
   // here's with an explicit object array
   logger.trace("Method args: {}, {}, {}, {}, {}",
       new Object[] { schema, host, port, path, query });
   // here's with a variable arguments with traceArgs()
   logger.traceArgs("Method args: {}, {}, {}, {}, {}",
       schema, host, port, path, query);
}
```

Here's how the arguments are turned into strings in the log message:

1. if you pass no arguments to the logger call, then any `{}` patterns will be displayed with no substitution
2. if you have more `{}` patterns in the message than arguments, then any extras will output the empty string
3. if you have more arguments than you have `{}` patterns in the message then they will be ignored
4. `null` values will output the string `null`
5. arrays will be displayed as `[ele1, ele2, ...]` with each element going through the same logic here
6. if the argument implements `LogArgumentCreator` then the results of `arg.createLogArg()` will be displayed
7. by default `arg.toString()` will be displayed

Here are examples of these rules.

```
// no arguments but a {} pattern
logger.info("connected to host '{}'");
// outputs: connected to host '{}'

// missing argument to second {} displays as empty string
String host = "host1";
logger.info("connected to host '{}' port '{}'", host);
// outputs: connected to host 'host1' port ''

// extra argument (port) is ignored
String host = "host1";
int port = 443;
logger.info("connected to host '{}'", host, port);
// outputs: connected to host 'host1'

// null argument
String host = null;
logger.info("connected to host '{}' failed", host);
// outputs: connected to host 'null'
```

```
// arguments that are arrays
String[] hosts = new String[] { "srv1", "srv2" };
logger.info("connected to hosts {} failed", hosts);
// outputs: connected to hosts [srv1, srv2]

// logging of Host which implements LogArgumentCreator where
// host.createLogArg() method returns the string: host4
Host host = new Host();
logger.info("connected to host '{}'", host);
// outputs: connected to host 'host4'

// logging of Server where server.toString() returns: srv3
Server server = new Server();
logger.info("connected to host '{}'", server);
// outputs: connected to host 'srv3'
```

2.4 Setting Log Level or Disabling Logging

By default the choice of whether or not to log, for example, a trace log message is up to the specific log backend. For example. if you are using LOG4J2, your LOG4J2 config file determines whether or not trace logging is enabled for a particular package. That said, there are some ways that you can impact in code which messages will get logged with calls to the SimpleLogging methods regardless of the log backend.

```
// have all INFO or above messages to be logged
// regardless of the backend configurations
Logger.setGlobalLogLevel(Level.INFO);
```

You can also disable all log messages by using the `OFF` level:

```
// force all messages to be disabled
Logger.setGlobalLogLevel(Level.OFF);
```

Another way to not have any log messages show would be use use the `NULL` log backend.

```
// set the backend type to be the null logger
LoggerFactory.setLogBackendType(LogBackendType.NULL);
```

2.5 Using "Fluent" Logging Method Chaining

Another pattern supported by SimpleLogging is the "fluent" logging which uses method chaining to build up a log message. The benefit of this method are that no objects are created by the log call unless the log level is enabled. Typically, primitives are auto-boxed into their object equivalents in a standard SimpleLogging logger before it can check to see if the log level is enabled.

For example, even with port being an `int` primitive below, there are no objects generated by thus call unless `TRACE` log level is enabled.

```
// create a fluent logger instead of a logger
private static final FluentLogger fluentLogger =
   LoggerFactory.getFluentLogger(MyClass.class);
...
// this generates no additional objects even due to auto-boxing
// unless trace logging is enabled
fluentLogger.atTrace()
    .msg("connected to host '{}' port '{}'")
    .arg(host)
    .arg(port)
    .log();
```

Logging of arrays in fluent logging can be done one of two ways. You can use the `flientLogger.args(...)` method to add an array of object arguments to the log message, each element of which will match a `{}` from the message. If you want an array to be associated with a single `{}` and displayed as `[arg1, arg2, ...]` then you need to use the method `fluentLogger.arg(Object)` which will interpret the array as a single object.

For example, the following code which calls this `args(...)` method will output: "1 + 2 = 3"

```
fluentLogger.msg("{} + {} = {}").args(new Object[] { 1, 2, 3 }).log();
```

While the following code which calls `arg(...)` will output: "port numbers: [1, 2, 3]" which interprets the array as an `Object` and will match a single `{}` from the message.

```
fluentLogger.msg("port numbers: {}").arg(new Object[] { 1, 2, 3 }).log();
```

2.6 How SimpleLogging Discovers the Logging Backend

Built into the `LoggerFactory` class is a `LogBackendType` enumerated type which SimpleLogging uses to try to locate what logging packages are on the classpath and therefore should be used.

The following logging implementations will be discovered on the classpath in this order.

1. SLF4J – (often paired with logback)
2. ANDROID – Android native Log
3. LOGBACK – Logback direct without SLF4J
4. COMMONS_LOGGING – Apache Commons Logging
5. LOG4J2 – version 2+
6. LOG4J – older, only used via reflection if available on the classpath
7. LAMBDA – AWS Lambda systems logging
8. LOCAL – log implementation that can write to a simple file
9. CONSOLE – log writing to `System.out` or `System.err`
10. JAVA_UTIL – Java util logging which is usually available in the JRE but never chosen directly
11. NULL – null logger to log no messages

If you need to force the logging type, you can use the following static methods.

```
// used to set a specific LoggerFactory.LogBackendType
LoggerFactory.setLogBackendType(LogBackendType.LOGBACK);
```

Or to specify a specifically factory class including custom ones:

```
// used to set a specific LoggerFactory or custom
LoggerFactory.setLogBackendFactory(new LogbackLogBackendFactory());
```

You can also set the `com.j256.simplelogger.backend` Java system property to one of the `LogBackendType` values in the above list.

```
java -Dcom.j256.simplelogger.backend=LOGBACK ...
```

You can also set the property to be a class name that implements `LogBackendFactory`.

```
java -Dcom.j256.simplelogger.backend=com.j256.simplelogging.backend.LogbackLogBackend$
```

2.7 More Usage Examples

Usually the logger is defined per-class as a `static` field.

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

The logger handles trace, debug, info, warn, error, and fatal messages. Not all of these message types are supported by all of the logger backends so SimpleLogging tries to map where appropriate.

```
logger.trace("read '{}' bytes", readCount);
logger.debug("host '{}', port-number {}", host, port);
logger.info("connecting to: {}:{}", host, port);
logger.warn("retry connect to host '{}', port {}", host, port);
logger.error("connect failed to host '{}', port {}", host, port);
logger.fatal(exception, "host '{}' threw exception", host);
```

If you want to use more than 4 arguments, you will either need to pass in an `Object[]` or call the methods with the `Args` suffix. This is a specific design decision because in either case, an `Object[]` is created even if the message is never logged.

```
logger.debug("schema '{}', host '{}', port {}, path {}, query {}",
     new Object[] { schema, host, port, path, query });
```

Or with variable arguments with methods that have an `Args` suffix such as `debugArgs(...)`.

```
logger.debugArgs("scheme '{}', host '{}', port {}, path: {}",
     schema, host, port, path);
```

2.8 Using With Maven

To use SimpleLogging with maven, include the following dependency in your '`pom.xml`' file:

```
<dependency>
   <groupId>com.j256.simplelogging</groupId>
   <artifactId>simplelogging</artifactId>
   <version>2.1</version>
</dependency>
```

3 Open Source License

Index of Concepts

U

V

W