@Log (and friends)

Captain's Log, stardate 24435.7: "What was that line again?"

The various `@Log` variants were added in lombok v0.10. *NEW in lombok 0.10:* You can annotate any class with a log annotation to let lombok generate a logger field.
The logger is named `log` and the field's type depends on which logger you have selected.

NEW in lombok v1.16.24: Addition of google's FluentLogger (via `@Flogger`).

NEW in lombok v1.18.10: Addition of `@CustomLog` which lets you add any logger by configuring how to create them with a config key.

Overview

You put the variant of `@Log` on your class (whichever one applies to the logging system you use); you then have a static final `log` field, initialized as is the commonly prescribed way for the logging framework you use, which you can then use to write log statements.

There are several choices available:

`@CommonsLog`
Creates
`private static final org.apache.commons.logging.Log log = org.apache.commons.logging.LogFactory.getLog(LogExample.class);`
`@Flogger`
Creates
`private static final com.google.common.flogger.FluentLogger log = com.google.common.flogger.FluentLogger.forEnclosingClass();`
`@JBossLog`
Creates
`private static final org.jboss.logging.Logger log = org.jboss.logging.Logger.getLogger(LogExample.class);`
`@Log`
Creates
`private static final java.util.logging.Logger log = java.util.logging.Logger.getLogger(LogExample.class.getName());`
`@Log4j`
Creates
`private static final org.apache.log4j.Logger log = org.apache.log4j.Logger.getLogger(LogExample.class);`
`@Log4j2`
Creates
`private static final org.apache.logging.log4j.Logger log = org.apache.logging.log4j.LogManager.getLogger(LogExample.class);`
`@Slf4j`
Creates
`private static final org.slf4j.Logger log = org.slf4j.LoggerFactory.getLogger(LogExample.class);`
`@XSlf4j`
Creates
`private static final org.slf4j.ext.XLogger log = org.slf4j.ext.XLoggerFactory.getXLogger(LogExample.class);`
`@CustomLog`
Creates
`private static final com.foo.your.Logger log = com.foo.your.LoggerFactory.createYourLogger(LogExample.class);`
This option *requires* that you add a configuration to your `lombok.config` file to specify what `@CustomLog` should do.

For example: `lombok.log.custom.declaration = com.foo.your.Logger com.foo.your.LoggerFactory.createYourLog(TYPE)(TOPIC)` which would produce the above statement. First comes a type which is the type of your logger, then a space, then the type of your logger factory, then a dot, then the name of the logger factory method, and then 1 or 2 parameter definitions; at most one definition with `TOPIC` and at most one without `TOPIC`. Each parameter definition is specified as a parenthesised comma-separated list of parameter kinds. The options are: `TYPE` (passes this `@Log` decorated type, as a class), `NAME` (passes this `@Log` decorated type's fully qualified name), `TOPIC` (passes the explicitly chosen topic string set on the `@CustomLog` annotation), and `NULL` (passes `null`).

The logger type is optional; if it is omitted, the logger factory type is used. (So, if your logger class has a static method that creates loggers, you can shorten your logger definition).

Please contact us if there is a public, open source, somewhat commonly used logging framework that we don't yet have an explicit annotation for. The primary purpose of `@CustomLog` is to support your in-house, private logging frameworks.

By default, the topic (or name) of the logger will be the (name of) the class annotated with the `@Log` annotation. This can be customised by specifying the `topic` parameter. For example: `@XSlf4j(topic="reporting")`.

With Lombok

```
import lombok.extern.java.Log;
import lombok.extern.slf4j.Slf4j;

@Log
public class LogExample {

  public static void main(String... args) {
    log.severe("Something's wrong here");
  }
}

@Slf4j
public class LogExampleOther {

  public static void main(String... args) {
    log.error("Something else is wrong here");
  }
}

@CommonsLog(topic="CounterLog")
public class LogExampleCategory {

  public static void main(String... args) {
    log.error("Calling the 'CounterLog' with a message");
  }
}
```

Vanilla Java

```
public class LogExample {
  private static final java.util.logging.Logger log = java.util.logging.Logger.getLogger(LogExample.class.g

  public static void main(String... args) {
    log.severe("Something's wrong here");
  }
}

public class LogExampleOther {
  private static final org.slf4j.Logger log = org.slf4j.LoggerFactory.getLogger(LogExampleOther.class);

  public static void main(String... args) {
    log.error("Something else is wrong here");
  }
}

public class LogExampleCategory {
  private static final org.apache.commons.logging.Log log = org.apache.commons.logging.LogFactory.getLog("C

  public static void main(String... args) {
    log.error("Calling the 'CounterLog' with a message");
  }
}
```

Supported configuration keys:

`lombok.log.fieldName` = *an identifier* (default: `log`).
The generated logger fieldname is by default '`log`', but you can change it to a different name with this setting.
`lombok.log.fieldIsStatic` = [`true` | `false`] (default: true)
Normally the generated logger is a `static` field. By setting this key to `false`, the generated field will be an instance field instead.
`lombok.log.custom.declaration` = *LoggerType*
LoggerFactoryType.loggerFactoryMethod(loggerFactoryMethodParams)*(loggerFactoryMethodParams)*
Configures what to generate when `@CustomLog` is used. (The italicized parts are optional). loggerFactoryMethodParams is a comma-separated list of zero to any number of parameter kinds to pass. Valid kinds: TYPE, NAME, TOPIC, and NULL. You can include a parameter definition for the case where no explicit topic is set (do not include the TOPIC in the parameter list), and for when an explicit topic is set (do include the TOPIC parameter in the list).
`lombok.log.flagUsage` = [`warning` | `error`] (default: not set)
Lombok will flag any usage of any of the various log annotations as a warning or error if configured.
`lombok.log.custom.flagUsage` = [`warning` | `error`] (default: not set)
Lombok will flag any usage of `@lombok.CustomLog` as a warning or error if configured.
`lombok.log.apacheCommons.flagUsage` = [`warning` | `error`] (default: not set)
Lombok will flag any usage of `@lombok.extern.apachecommons.CommonsLog` as a warning or error if configured.
`lombok.log.flogger.flagUsage` = [`warning` | `error`] (default: not set)
Lombok will flag any usage of `@lombok.extern.flogger.Flogger` as a warning or error if configured.
`lombok.log.jbosslog.flagUsage` = [`warning` | `error`] (default: not set)
Lombok will flag any usage of `@lombok.extern.jbosslog.JBossLog` as a warning or error if configured.
`lombok.log.javaUtilLogging.flagUsage` = [`warning` | `error`] (default: not set)
Lombok will flag any usage of `@lombok.extern.java.Log` as a warning or error if configured.
`lombok.log.log4j.flagUsage` = [`warning` | `error`] (default: not set)
Lombok will flag any usage of `@lombok.extern.log4j.Log4j` as a warning or error if configured.
`lombok.log.log4j2.flagUsage` = [`warning` | `error`] (default: not set)
Lombok will flag any usage of `@lombok.extern.log4j.Log4j2` as a warning or error if configured.
`lombok.log.slf4j.flagUsage` = [`warning` | `error`] (default: not set)
Lombok will flag any usage of `@lombok.extern.slf4j.Slf4j` as a warning or error if configured.
`lombok.log.xslf4j.flagUsage` = [`warning` | `error`] (default: not set)
Lombok will flag any usage of `@lombok.extern.slf4j.XSlf4j` as a warning or error if configured.

Small print

If a field called `log` already exists, a warning will be emitted and no code will be generated.

A future feature of lombok's diverse log annotations is to find calls to the logger field and, if the chosen logging framework supports it and the log level can be compile-time determined from the log call, guard it with an `if` statement. This way if the log statement ends up being ignored, the potentially expensive calculation of the log string is avoided entirely. This does mean that you should *NOT* put any side-effects in the expression that you log.