Logging Guidelines

Following the Great Relogging activity that took place recently, a revised format for logging in Java code is established as follows:

1. `Logging.LOGGING` compile-time flag has been removed. Do not use it.

---

2. The logger should always be named **`log`**.

---

3. `org.apache.log4j.Category` is deprecated in favor of `org.apache.log4j.Logger`. Example:
   ```
   private static final Category log = Logging.LOGGING ? Category.getInstance(POD.class) : null;
   ```
   is replaced with
   ```
   private static final Logger log = Logger.getLogger(POD.class);
   ```

---

4. A class should never add “implements Logging” to its declaration statement.

---

5. The following new runtime logging level checks are available:
   ```
   log.isFatalEnabled() // currently unused
   log.isErrorEnabled()
   log.isWarnEnabled()
   log.isInfoEnabled()
   log.isDebugEnabled()
   log.isTraceEnabled() // currently unused
   ```
   Example:
   ```
   if (log.isInfoEnabled())
   {
       log.info("restoreRecordings: Storage not ready - deferring load of recording database");
   }
   ```
   Please note that curly braces are required around the `log.info(...);` call. The following should **not** be used:
   ```
   if (log.isInfoEnabled())
       log.info("restoreRecordings: Storage not ready - deferring load of recording database");
   ```
   or
   ```
   if (log.isInfoEnabled()) log.info("restoreRecordings: Storage not ready - deferring load of recording
   database");
   ```

---

6. Every single log message should have an `if (log.isXXXEnabled())` check, even where there are two consecutive log messages in the source. Example:
   ```
   if (log.isDebugEnabled()
   {
       log.debug("Destroy called. Ok.");
   }
   if (log.isDebugEnabled()
   {
       log.debug("Received Tuning Over failed event");
   }
   ```
   The following should **not** be used:
   ```
   if (log.isDebugEnabled()
   {
       log.debug("Destroy called. Ok.");
       log.debug("Received Tuning Over failed event");
   }
   ```

---

7. Mixing business logic with logging level checks should be avoided. Example:
   ```
   if (low)
   {
       if (log.isWarnEnabled())
       {
           log.warn("VM memory low even after reclamation");
       }
   }
   ```
   The following should **not** be used:
   ```
   if (log.isWarnEnabled() && low)
   {
       log.warn("VM memory low even after reclamation");
   }
   ```

---

8. Also, please note that the business logic checks should be nested outside rather than inside of the logging statements. Using the previous example, the following should **not** be used:
   ```
   if (log.isWarnEnabled())
   {
       if (low)
       {
           log.warn("VM memory low even after reclamation");
       }
   }
   ```

---

9. Do not use inherited loggers. Don't make a logger protected in a parent class and use that logger from a subclass. Every class that needs to log should just have its own logger.
10. Any logging-specific logic that is not part of the actual log call should also be wrapped in a log level check.
11. In the future, we should use 'trace' where we are tempted to add a boolean to control debug level loggin with more granularity.

For reference, here is the link to our coding guidelines. Refer to section 2.4: Logging Guidelines.