EXCEPTION HANDLING
&
LOGGING
BEST PRACTICES

Angelin

AGENDA

- Logging using Log4j
- “Logging” Best Practices
- “Exception Handling” Best Practices
- CodePro Errors and Fixes

LOGGING USING LOG4J

Logging using Log4j

- Log4j - logging library for Java
- Logging Levels (in lowest to highest order)
- The standard levels of Log4j are ordered as

ALL < TRACE < DEBUG < INFO < WARN < ERROR < FATAL < OFF

Logging using Log4j

Level	Description
ALL	The lowest possible rank and is intended to turn on all levels of logging including custom levels.
TRACE	Introduced in log4j version 1.2.12, this level gives more detailed information than the DEBUG level.
DEBUG	Designates fine-grained informational messages that are most useful to debug an application.
INFO	Designates informational messages that highlight the progress of the application at coarse-grained level.
WARN	Designates potentially harmful situations. This level can be used to warn usage of deprecated APIs, poor use of API, 'almost' errors and other runtime situations that are undesirable or unexpected, but not necessarily "wrong".

Logging using Log4j

Level	Description
ERROR	Designates error events that might still allow the application to continue running. This level can be used to inform about a serious error which needs to be addressed and may result in unstable state.
FATAL	Designates very severe error events that will presumably lead the application to abort.
OFF	The highest possible rank and is intended to turn off logging.

How Logging Level works?

A logging request of a particular level is said to be *enabled* if that level is higher than or equal to the level of its logger.

Example

```
import org.apache.log4j.*;
public class LogClass {
  private static final org.apache.log4j.Logger LOGGER =
    Logger.getLogger(LogClass.class);

  public static void main(String[] args) {
      LOGGER.setLevel(Level.WARN);
      LOGGER.trace("Trace Message!");
      LOGGER.debug("Debug Message!");
      LOGGER.info("Info Message!");
      LOGGER.warn("Warn Message!");
      LOGGER.error("Error Message!");
      LOGGER.fatal("Fatal Message!");
  }
}
```

Output:

Warn Message!
Error Message!
Fatal Message!

"LOGGING" BEST PRACTICES

Logging - Best Practices

- Declare the logger to be both **static** and **final** to ensure that every instance of a class shares the common logger object.
- Add code to check whether logging has been enabled at the right level.
- Use meaningful log messages that are relevant to the context.

Logging - Best Practices

- Better to use logging only to log the following,
  - ✓ method entry (optionally with the method's input parameter values)
  - ✓ method exit
  - ✓ root cause message of exceptions that are handled at the exception's origin point.

Logging - Best Practices

- Any other intermediate redundant logging statements, which are used just for the purpose of debugging can still be avoided.

  Example

```
try {
    LOGGER.debug("About to enter getSkuDescription method");
    // The above logging statement is not required,
    // if getSkuDescription() method logs its method entry
    String skuDesc = getSkuDescription(skuNumber);
    LOGGER.debug("Exited getSkuDescription method");
    // The above logging statement is not required,
    // if getSkuDescription() method logs its method exit
} catch (ServiceException se) {
    LOGGER.error(se.getErrorMessage());
    throw se;
}
```

Logging - Best Practices

- Avoid logging at ‘every’ place where a custom exception is thrown and instead log the custom exceptions’ message in its ‘catch’ handler.

Example

```
try {
    if (null == skuNumber || skuNumber.isEmpty()) {
            LOGGER.error(“Sku number is invalid”);
            // The above logging statement is not required,
            // since the catch handler logs the message
            throw new ServiceException(“Sku number is invalid”);
    }
```

Logging - Best Practices

```
    try {
            sku = Integer.parseInt(skuNumber);
    } catch (NumberFormatException nfe) {
            LOGGER.error("Sku number is invalid and not a number");
            // The above logging statement is not required,
            // since the catch handler logs the message
            throw new ServiceException("Sku number is invalid and
  not a number", nfe);
    }
    ……
} catch (ServiceException se) {
    LOGGER.error(se.getErrorMessage());
    throw se;
}
```

EXCEPTION HANDLING BEST PRACTICES

Exception Handling - Best Practice #1

- **Handle Exceptions close to its origin**
  - Does NOT mean “catch and swallow” (i.e. suppress or ignore exceptions)

```
try {
    // code that is capable of throwing a XyzException
} catch ( XyzException e) {
    // do nothing or simply log and proceed
}
```

  - It means, “log and throw an exception relevant to that source layer”
    - DAO layer - DataAccessException
    - Business layer - ApplicationException (example - OUSException)

Exception Handling - Best Practice #1

Important Note

In applications using Web Services, the Web Service (a.k.a Resource) layer,

- ✓ should catch ALL exceptions and handle them by creating proper error response and send it back to client.
- ✓ should NOT allow any exception (checked or unchecked) to be "thrown" to client.
- ✓ should handle the Business layer exception and all other unchecked exceptions separately.

Exception Handling - Best Practice #1

Example

```
try {
    // code that is capable of throwing an ApplicationException
} catch (ApplicationException e) {
    // form error response using the exception's
    // data – error code and/or error message
} catch (Exception e) {
    // log the exception related message here, since this block is
    // expected to get only the unchecked exceptions
    // that had not been captured and logged elsewhere in the code.
    // form error response using the exception's
    // data – error code and/or error message
}
```

The catch handler for 'Exception' in the Web Service layer is expected to handle all unchecked exceptions thrown from within 'try' block

Exception Handling - Best Practice #2

❖**Log Exceptions just once and log it close to its origin**

Logging the same exception stack trace more than once can confuse the programmer examining the stack trace about the original source of exception.

```
try {
    // code that is capable of throwing a XyzException
} catch (XyzException e) {
    // log the exception specific information
    // throw exception relevant to that source layer
}
```

Exception Handling - Best Practice #2

#1 - When catching an exception and throwing it through an exception relevant to that source layer, make sure to use the construct that passes the original exception's cause. Otherwise, CodePro will report "No cause specified when creating exception".

```
try {
    // code that is capable of throwing a SQLException
} catch (SQLException e) {
    // log technical SQL Error messages, but do not pass
    // it to the client. Use user-friendly message instead
    LOGGER.error("An error occurred when searching for the SKU details" + e.getMessage());
    throw new DataAccessException("An error occurred when searching for the SKU details", e);
}
```

Exception Handling - Best Practice #2

#2 - There is an exception to this rule, in case of existing code that may not have logged the exception details at its origin. In such cases, it would be required to log the exception details in the first method up the call stack that handles that exception. But care should be taken not to COMPLETELY overwrite the original exception's message with some other message when logging.

Example

DAO Layer:

```
try {
    // code that is capable of throwing a SQLException
} catch (SQLException e) {
    // LOGGING missed here
    throw new DataAccessException("An error occurred
    when processing the query.", e);
}
```

Exception Handling - Best Practice #2

Processor Layer:

```
try {
    // code that is capable of throwing a DataAccessException
} catch (DataAccessException e) {
    // logging is mandated here as it was not logged
    // at its source (DAO layer method)
    LOGGER.error(e.getMessage());
    throw new OUSException(e.getMessage(), e);
}
```

Exception Handling - Best Practice #3

❖ **Do not catch "Exception"**

Accidentally swallowing RuntimeException

```
try {
   doSomething();
} catch (Exception e) {
    LOGGER.error(e.getMessage());
}
```

This code

1. also captures any RuntimeExceptions that might have been thrown by doSomething,
2. ignores unchecked exceptions and
3. prevents them from being propagated.

Exception Handling - Best Practice #3

Important Note (about some common RuntimeExceptions)

- NullPointerException – It is the developer's responsibility to ensure that no code can throw it. Run CodePro and add null reference checks wherever it has been missed.

- NumberFormatException, ParseException – Catch these and create new exceptions specific to the layer from which it is thrown (usually from business layer) using user-friendly and non technical messages.

Exception Handling - Best Practice #3

Important Note (about some common RuntimeExceptions)

- Example

```
try {
    int sku = Integer.parseInt(skuNumber);
} catch (NumberFormatException nfe) {
    LOGGER.error("SKU number is invalid and not a number");
    throw new OUSException("SKU number is invalid and not a
    number", nfe);
}
```

- All other unchecked exceptions (RuntimeExceptions) will be caught and handled by the Web Service layer (as explained in Best Practice #1).

CODEPRO ERRORS & FIXES

Fix to common CodePro errors

- **"Invalid exception parameter name"**

  Solution

  Rename the parameter to “e”

  Example

```
try {
    // code that is capable of throwing a DataAccessException
} catch (DataAccessException e) {
    throw new OUSException(e.getMessage(), e);
}
```

Fix to common CodePro errors

- **"No cause specified when creating exception"** when wrapping an exception into another exception.

  Solution

  Use the construct that passes the original exception's cause

  Example

```
try {
    // code that is capable of throwing a SQLException
} catch (SQLException e) {
    LOGGER.error(e.getMessage());
    throw new DataAccessException("An error occurred
    when searching for the SKU details", e);
}
```

THANK YOU

(Best Practices)

High BP is good for ~~health~~ code