Advanced Exception Handling

Introduction

In article I will be showing you how to handle Exceptions in Java properly. Most programming books don't thoroughly explain exception handling and give the wrong perception that exception handling is just some nasty clean up code. You will usually see auto-generated exception handling code that calls `printStackTrace()`. While that works, most of the time it is not the best option. Proper exception handling is a vital skill, which few developers apply properly.

Exception Handling

In Example 1, we have a public process method that calls two private methods, `process1()`, and `process2()`. In `process1()` we create a `Reader`, which we try to read from. Using `reader.read()` will throw a `java.io.IOException` which we have to handle. An `IOException` is a `CheckedException`.

```
package com.marcusbiel.java8course;

import java.io.IOException;
import java.io.InputStreamReader;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Training {

     private static final Logger LOG =
LoggerFactory.getLogger(Training.class);

     public void process() throws IOException, InterruptedException {
          process1();
          process2();
     }

     private void process1() throws IOException {
          InputStreamReader reader = new InputStreamReader(System.in);
          reader.read();
     }

     private void process2() throws InterruptedException {
```

```
            Thread.sleep(100);
      }
}
```

Example 1

Java is the only programming language that supports Checked Exceptions. This means that the compiler insists that you handle the Exception, or at least declare it. What most people do in this case, is they have their IDE “Add Exception to the Method Signature”, and that gets rid of the problem by forwarding the problem to the method’s caller. The problem then reappears in the caller function `process()`, so most people will just do the same thing again.

Additionally, `process2()` might throw an InterruptedException, so you have to also declare that exception in `process()`. That will eventually lead to cluttering the code with many exceptions in the throws clause as you can see in Example 2:

```
public void process() throws IOException, InterruptedException {
    process1();
    process2();
 }
```

Example 2

This is just a small example, but in real world cases, when designing an application with hundreds or thousands of classes in a hierarchy, where they call one another, declaring exceptions this way can cause **a lot of clutter**. Declaring Checked Exceptions instead of directly handling them is really epidemic, and it will soon spread like cholera over your entire code base. Therefore, this approach is definitely not recommended.

Using a try/catch

A solution to the previous problem would be to surround the potentially problematic code with a try / catch as shown in Example 3. We could try to use the reader, and if the exception is thrown, call `printStackTrace()`.

```
private void process1() {
      InputStreamReader reader = new InputStreamReader(System.in);
      try {
            reader.read();
      } catch (IOException e) {
            e.printStackTrace();
      }
}
```

Example 3

However, calling printStackTrace() can cause problems. This method collects the whole stack, which may use a lot of memory, in addition to printing, which also uses a lot of resources. It is therefore better to avoid using it.

The best way of handling an exception is to do something to help the client. For example, to read from an alternative source. Instead of doing what we did in Example 3, we could use `readFromDatabase()` to read from an alternative database.

```
private void process1() {
     InputStreamReader reader = new InputStreamReader(System.in);
     try {
          reader.read();
     } catch (IOException e) {
          readFromDatabase();
     }
}

public void readFromDatabase() {
    // read from a database as an alternate source
}
```

Example 4

In conclusion, the best option when dealing with exceptions is to handle them as early as possible. If for any reason you can't handle an exception, you can log it, using a logging framework.

Logging an Error

Another option is to log the error. Most of the time I see developers using LOG.error () to do this, like below:

```
private void process1() {
     InputStreamReader reader = new InputStreamReader(System.in);
     try {
          reader.read();
     } catch (IOException e) {
          LOG.error(e.getMessage(), e);
     }
}
```

Example 5

Developers tend to use `error()` *whenever* they log an exception, however in many situations `warn()` might be enough. Using error () is useful if some human intervention is needed to fix the exception, like if a database needs to be manually restarted. However, if the system can recover on its own, consider using `warn()` instead.

Besides a `CheckedException`, a method call could also produce an unchecked exception, such as a `RuntimeException`, which I will deal with in the next example. As for the logging level, all the same rules apply, but there is one small difference for runtime exceptions. It might be better to log using the `toString()` method, because there are some exceptions, such as the `NullPointerException`, where the error would not contain a message. In this case logging the message would just return an empty string. So if you use `toString()`, you would at least see the name of the class that was thrown, such as `NullPointerException`, which is more helpful than an empty string.

In this case I'll use `warn()` instead of `error()` to demonstrate its usage.

```
private void process1() {
     InputStreamReader reader = new InputStreamReader(System.in);
     try {
          reader.read();
     } catch (IOException e) {
          LOG.warn(e.getMessage(), e);
     } catch (RuntimeException e) {
          LOG.warn(e.toString(), e);
     }
}
```

Example 6

Wrapping Checked Exceptions

A `CheckedException` forces you to deal with it. This means you must have a way of handling it which you may not always have! In this case, it doesn't really make sense to have a `CheckedException`. You might as well wrap the `CheckedException` with a RuntimeException. In the following example, I have used the generic `RuntimeException`. You might also declare your own Exception extending `RuntimeException`, but in this example I just wanted to show you how to wrap a `CheckedException`:

```
private void process1() {
     InputStreamReader reader = new InputStreamReader(System.in);
     try {
          reader.read();
     } catch (IOException e) {
          throw new RuntimeException(e);
     } catch (RuntimeException e) {
          LOG.warn(e.toString(), e);
     }
}
```

Example 7

Now we have converted the checked `IOExeption` to an unchecked `RuntimeException`, which doesn't have to be handled anymore. You can propagate this all the way up, then you would have just one final catch-block that would log all `RuntimeExceptions` or handle all `RuntimeExceptions` in some generic way.

Log and Rethrow

One common bad practice is to “log and rethrow” an exception, which as the name implies, logs the exception, but then rethrows it. The rethrown exception will either be handled or logged again. This results in a redundant log message in both cases, as handled exceptions usually don’t have to be logged. To monitor your program, you should also have exact statistics of how many times an exception happens. Logging certain exceptions twice or more will tamper with your error statistics immensely. Irrelevant exceptions and / or exceptions happening rarely could get precedence over frequent and / or severe exceptions. Further, it may bloat up your log. I have seen cases where on production one exception was logged and rethrown ten times, which resulted in **ten log messages instead of one for a single error**. If you had this happen for every exception, you’d end up with a log file **ten times** larger than it needs to be; that could be the difference between 10GB and 100GB!

```
private void process1() {
     InputStreamReader reader = new InputStreamReader(System.in);
     try {
          reader.read();
     } catch (IOException e) {
          LOG.error(e.getMessage(), e);
          throw new RuntimeException(e);
     } catch (RuntimeException e) {
          LOG.warn(e.toString(), e);
     }
}
```

Example 8

Another important thing that I haven't mentioned about logging is that you should avoid logging generic messages as much as you can. Logging the values of the parameters that brought you to the exception is very helpful when you try to find the reason for an exception. You can also log the full name of the class and method where the error occurred and the exact line of the error to better help you locate it.

Including all relevant method parameters in the log message is very helpful for identifying the reason that led to the exception. To facilitate a grouping or prefiltering of error messages, put the variable parameters at the end of the log message, if possible.

```
private void process1(String parameter1, Object parameter2) {
     InputStreamReader reader = new InputStreamReader(System.in);
     try {
          reader.read();
     } catch (IOException e) {
          LOG.error(e.getMessage() + ", parameter: " + parameter1 +
"parameter2" + parameter2, e);
     } catch (RuntimeException e) {
          LOG.warn(e.toString(), e);
     }
}
```

Example 9

Logging with Debugging

Another bad habit I see is developers logging intensively on DEBUG level. With the existence of debuggers in IDEs, there is no need for extensive use of debugging in loggers.

Extensive debugging can easily slow down performance. Logging on a debug level *might* be tolerable in cases where there is an external QA team that wants to check if certain methods are called. In case it is a necessity in your team, then you can use:

```
if(LOG.isDebugEnabled()) {
     LOG.debug("");
}
```

Example 10

In the above example, we have a simple `String`, but in real time cases, there would be many parameters, and when calling `toString()` method, it might call many other objects contained in the first object. This could eventually lead to poor performance. By checking first if `DEBUG` is enabled, we avoid redundant operations, such as accessing parameters and concatenating them to the print `String`.

In production environments I generally recommend disabling debug mode on production. This is because debug mode is incredibly taxing on memory and resources, something that might be a major issue in a project with a lot of users, such as a 24/7 server application.

Logging with Info Level

Finally, you can log some information with a log level of `INFO`. Enabling `INFO` also depends on your production settings.

```
if(LOG.isInfoEnabled()) {
     LOG.info("");
}
```

Example 11

On production, you might have `INFO` log level as default. If it is the default, then you can skip checking for it. However, I would really only do this for big starting up processes and not in small methods that are called constantly. For example, your operations team would probably like to see some messages “popping up” if an important server or system is starting.

Thanks for reading!