
MAHARISHI UNIVERSITY of MANAGEMENT
Engaging the Managing Intelligence of Nature
Computer Science Department
CS401 Modern Programming Practices (MPP)
Professor Paul Corazza

Lecture 10:Best Programming Practices with Java 8

Living Life in Accord with Natural Law

REVIEW: Exception-Handling and Logging

Exception-Handling in Java
(Optional Module #2)

Executive Summary

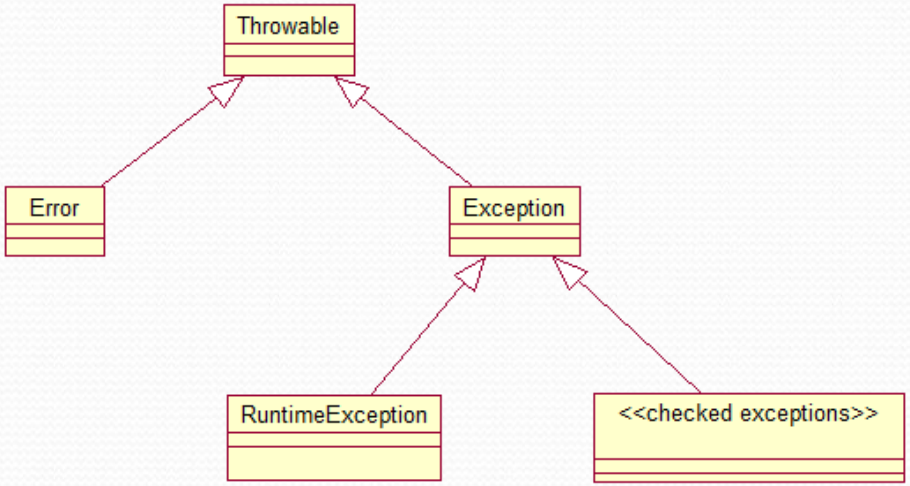
Best practices for exception-handling:

1. An exception should be *thrown* by the first object in a call stack that encounters an error condition
2. An object should *handle* a thrown exception only if it is qualified to handle it; otherwise, it should propagate the thrown exception up the call stack
3. Information about the state of the application should be *logged* by objects in the call stack that have useful information for tracking down the error

Review of Exception-Handling in Java (Optional Module #2)

- In Java, error conditions are represented by Java classes, all of which inherit from Throwable.

The Hierarchy of Exception Classes



Classification of Error-Condition Classes (Optional Module #2)

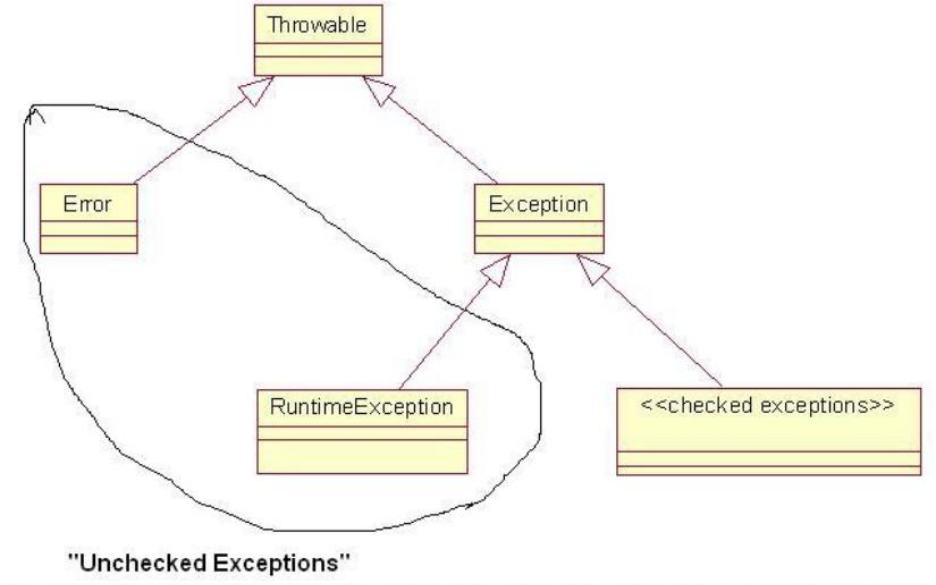
In Java, error-condition classes belong to one of three categories:

- *Error* – Objects in this category belong to the inheritance hierarchy headed by the Error class. Examples include OutOfMemoryError, StackOverflowError, VirtualMachineError. If these are thrown, it indicates an unrecoverable error and the application should terminate.

- *Other Unchecked Exceptions* – Besides Error objects, unchecked exceptions include all objects that belong to the inheritance hierarchy headed by the class RuntimeException. Examples include NullPointerException, ArrayIndexOutOfBoundsException, NumberFormatException. When these are thrown, it indicates that a programming error has occurred and needs to be fixed. Typically, these types of exceptions are not caught and handled – they simply indicate that some logic error needs to be corrected

Classification of Exception Classes (Optional Module #2) (cont.)

- *Checked Exceptions* – Exceptions in this category are subclasses of Exception but not subclasses of RuntimeException. Examples include FileNotFoundException, IOException, SQLException. The idea behind checked exceptions is that it should be possible to handle them in such a way that the application can continue; for example, if a database is unavailable, a SQLException would be thrown, and the user could be given the option to continue on to other features of the application even if the database is down for awhile.



Four Ways to Deal with Checked Exceptions (Optional Module #2)

1. Declare that your method throws the same kind of exception (and do not handle the exception)
2. Put the exception-creating code in a try block, and write a catch block to handle the exception in case it is thrown – in other words, use try/catch blocks.
3. Use try/catch blocks – catch block can log information about the current state – and then re-throw the exception. In this case, as in (1), you must declare that the method throws this type of exception
4. Use try/catch blocks as in (3), but, when an exception is caught, wrap it in a new instance of another type of exception class and re-throw

See Demo lesson10.lecture.checkedexceptions

Summary of Best Practices
(Optional Module #2)

1. *Log when exception first arises*. When an exception is first caught, information about the state of the object should be logged – logging can be done in a catch block. However, if the try/catch are inside a loop that has many iterations or that could even possibly fail to terminate, logging should be done outside the loop.

```
public class MyClass {
    public final static Logger LOG
        = Logger.getLogger(MyClass.class.getName());
```

Obtain instance of jdk Logger

```
public void handleFile(File f) {
    FileReader fileReader = null;
    BufferedReader buf = null;
    try{
        fileReader = new FileReader(f);
        buf = new BufferedReader(fileReader);
        String line = buf.readLine();
        System.out.println(line);
    } catch(IOException e) {
        LOG.warning("1st IOException: "
            + e.getMessage());
```

Write to the LOG. Use LOG.severe, LOG.warning, LOG.info, LOG.fine

Summary of Best Practices
(Optional Module #2) (cont.)

2. *Handler of exception should be chosen carefully*. An exception should be handled by an object that "knows" what to do with the exception – typically, it should have the responsibility of communicating a message to the user. Therefore, when an exception is thrown, it should propagate up the call stack until it reaches an appropriate handler.

Summary of Best Practices
(Optional Module #2) (cont.)

3. *Never create an "empty" catch block*. Exceptions should never be ignored, as in

```
try { . . .
} catch {}
```

← This is so bad

If nothing needs to be done, there should at least be a comment stating this fact – and probably some message to the log – rather than dead silence.

```
try { . . .
} catch {
    //nothing needed here
    LOG.info("Exception thrown by . . .");
}
```

← This is acceptable

Summary of Best Practices (Optional Module #2) (cont.)

4. *Never catch Exception or Throwable*. Your code should (almost) never catch Exception or Throwable. One reason is that doing so means that you will be handling any RuntimeExceptions that are thrown (like NullPointerException), and these should not be caught. [One exception to this rule arises sometimes when communicating with external APIs – it may not always be possible to anticipate which types of Exceptions will be thrown by API methods, and you may want to make sure your application does not shut down because of an uncaught exceptions coming from the outside.

Summary of Best Practices (Optional Module #2) (cont.)

5. *Always validate input arguments.* Important methods that take input arguments should validate input values and throw an IllegalArgumentException in case of invalid inputs.

```
void myMethod(String arg) {
  if(arg == null || arg.length() == 0)
  throw new IllegalArgumentException("Input must
  be nonempty");
  //more
}
```

Note that throwing any type of RuntimeException never requires a throws declaration.

Summary of Best Practices (Optional Module #2) (cont.)

6. *Don't throw instances of RuntimeException*. If you need to throw some kind of runtime exception, either use one of the specific subclasses of RuntimeException available in the Java libraries (as in the previous example: IllegalArgumentException, or others: IllegalStateException, NumberFormatException) or, if nothing fits, create your own subclass of RuntimeException. Never simply throw a RuntimeException – it is too general.

```
public class MyClass {
    void myMethod() {
        //problem arises . . .
        throw new RuntimeException("A problem...");
    }
}
```

← This is bad

```
public class MyClass {
    void myMethod() {
        //problem arises . . .
        throw new MyRuntimeException("A problem...");
    }
}

class MyRuntimeException
    extends RuntimeException {
    public MyRuntimeException() {
        super();
    }
    public MyRuntimeException(String msg) {
        super(msg);
    }
}
```

← This is good

Summary of Best Practices (Optional Module #2) (cont.)

7. *Using a finally block*.
   A. *Always executes*. When finally is used, the code in the finally block is executed even if the try block succeeds and returns (finally block executes before performing the return) or an exception is thrown. When an exception is thrown and caught, before control is passed up the stack, finally clause executes; when it is not caught, before a stack trace is displayed, finally clause is executed.
   B. *Used for cleanup*. Traditionally, finally is used to clean up resources before exiting the application. Files are closed, database connections closed, etc. Java 7/8 provides a new approach (*try with resources*) to handle this pattern – discussed below
   C. *No return statement in a finally block.* A return statement should not occur in a finally block – if the try block also has a return statement, then the finally block's return statement will be the one that executes.
   D. *Do not throw an exception within a finally block*. An exception should not be thrown from within a finally block – if an exception is thrown during execution of the try block, and then in the finally block another exception is thrown, the exception from the finally block is the one that is actually thrown.

Setting up the JDK Logger (Optional Module #2)

1. The JDK Logger can always be accessed like this:

```
private final static Logger LOG =
Logger.getLogger(<any string>)
```

The string argument should be the current package name. Typical way of obtaining this string:

```
MyClass.class.getPackage().getName();
```

The top-level logger is indicated by the empty string "":

```
Logger LOG = Logger.getLogger("");
```

There is also a global logger that can be obtained like this:

```
Logger LOG = Logger.getGlobalLogger();
```

For smaller applications, this one is fine to use.

Setting up the JDK Logger (Optional Module #2)

2. Configuring the Logger
   a. For production-quality logging, configuration should be done using the logging.properties file that comes with Java. Details about this are available in the setup folder for this course in the directory logging. (This is an FPP topic.)
   b. Log configuration can also be done in code. For this course, the logger can be configured using logsetup.jar. The global logger can be configured using this jar file with a call (in application startup):

      ```
      LogSetup.setup();
      ```

      The setup method does the following:

      i. Provides simple output messages to the console. You can create one of these messages with these calls:

         ```
         LOG.config(<message>), LOG.info(<message>),
         LOG.warning(<message>), LOG.severe(<message>)
         ```

      ii. Provides XML-formatted messages to a log file, placed at the top level of your src directory: src\logs

Demos: lesson10.lecture.logging.defaultlogging, lesson10.lecture.logging.defaultlogging2

Tricky try/catch/finally Situations (Optional Module #2)

1. *Avoid memory leaks.* When your application uses external resources, like files or a database connection, it is important to close the connections after your application has finished using them. Typically, using these connections involves checked exceptions; but whether or not an exception is thrown, your application must disconnect from the resource, or there can be a memory leak, causing memory to fill up.

2. *Do clean-up in a finally block.* The usual way to clean up resources is in a finally block, which will execute whether or not an exception is thrown.

3. But, what if closing a resource is also capable of throwing a checked exception? How should this be handled?

Demo: lesson10.lecture.trickycatch1

Tricky try/catch/finally Situations (Optional Module #2) (cont.)

Problems with the lesson10.lecture.trickycatch1
Solution:

1. It's messy
2. It is possible to make it more readable by separating the part that does interesting work (opening and reading a file) from the part that handles exceptions.

See lesson10.lecture.trickycatch2

In this approach, the code is separated into inner and outer try blocks. Inner try/finally block does file processing and outer try/catch block takes care of exception-handling.

Tricky try/catch/finally Situations (Optional Module #2) (cont.)

Problems with Both Solutions:

1. In both solutions, an IOException could be thrown for two different reasons. The first of these would indicate a difficulty in finding or reading the file; the second would arise because the readers could not be closed. Only the first is of any interest, but if both are triggered, only the second one is thrown.

2. In Java 7, a new feature was added that allows you to add "suppressed" exceptions to a main exception, and then access them from the main exception as desired. The examples in lesson10.lecture.trickycatch3_suppressed show how this can be done here. Note that the code is rather complicated!