KIT101 Notes — 16 — Error Handling with Exceptions

James Montgomery

Revised: 2017-04-24

References: L&L 11.1–11.5

Program errors & how to read exception output

There are three broad categories of error that can occur in a program:

1. Syntactical
    - Compiler cannot interpret what we mean
    - Discovered at *compile time*
2. Logical
    - Wrong behaviour but it *is* what the code says to do
    - Discovered at *runtime*
3. Runtime errors
    - Errors that cause program execution to terminate unexpectedly
    - Represented by `Exception` and `Error` objects
    - And can result in a long, unfriendly list of method names and line numbers

Exceptions (and Errors) interrupt the normal flow of program execution. For instance, if we had the line:

```
int a = 4 / 0; //<--imagine this is at line 16 of Divide.java
```

in a program then at runtime the program would end when it attempted to evaluate the expression and then generate an `ArithmeticException` (because it is not possible to divide by zero). Example exception output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Divide.main(Divide.java:16)
```

which indicates that the *kind* of exception is an `ArithmeticException`, and its detailed message is “/ by zero” (that is, divide by zero). The next line reports in which file and on which line the error occurred.

Tip: Even if you don't understand what the exception error message means at first, you can use the clues in the error output to help you identify the cause of the error. The class name of the exception and additional details can provide some insight, as will the line where the error occurred in your code.

```
                                type of exception        additional
details
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Divide.main(Divide.java:16)
       class  method  source  line
                              number
```

Terminology: Exceptions are *thrown* (by the system or by application code in response to unusual situations). Handling an exception is referred to as *catching* the exception.

Types of exception

Exceptions fall into three categories which are reflected in their class hierarchy. Showing just down to the 'top-level' exception classes of `Error`, `Exception` and `RuntimeException`:

- `Object`
  - `Throwable`
    - `Error`
    - `Exception`
      - `RuntimeException`

Errors indicate an unrecoverable situation and should not be caught (because there is generally nothing that can be done to salvage the situation). Examples include `StackOverflowError` (when there have been two many method calls so there's no more room in stack memory to hold them) and `OutOfMemoryError` (when the virtual machine cannot allocate any more memory to a program).

Exceptions indicate either an unusual or erroneous condition. They can be 'caught' and handled by another part of the program. Catching exceptions allows a program's code (within a single method) to be separated into a part representing normal execution flow and another for exception execution flow.

Checked versus Unchecked exceptions

Subclasses of `RuntimeException` are known as *unchecked* exceptions, while all other subclasses of `Exception` are known as *checked* exceptions (for reasons that will be shown below).

In most cases, unchecked exceptions represent unexpected erroneous conditions that the programmer can probably avoid with carefully written code. For this reason the compiler does not check for these and does not force the programmer to write error handling code for them. Examples include `ArrayIndexOutOfBoundsException`, `ArithmeticException` and `NumberFormatException`.

In contrast, *checked* exceptions typically represent unusual situations that the programmer cannot control but which *will* happen eventually, such as `IOException` (input/output exception), which can occur when reading or writing data from/to a file or network connection. The compiler checks for these exceptions and forces the programmer to write code to handle them.

Handling exceptions

There are three approaches to handling exceptions:

1. Ignore it
   - exception propagated to top level (`main()`)
   - program will terminate and produce an appropriate (but ugly) message
   - an appropriate choice during development because you can use the error message to identify the location of the fault and fix it so it doesn't happen again
2. Handle it where it occurs
   - uses a **`try-catch`** block (see next) to *try* the risky code
   - and *catch* any exception that occurs
3. handle it another place in the program
   - a variant of (2) where the method header declares that it **`throws`** *`ExceptionClassName`*

The manner in which an exception is processed is an important design consideration when programming, but a full investigation of the implications is beyond the scope of this introductory unit.

try-catch

The general approach to truly handling an exception is to (1) optimistically *try* the code that might produce an exception and (2) if an exception of some class occurs, jump immediately to the **`catch`** clause and execute its code.

```
try {
    Code that might produce an exception
} catch (ExceptionClass ex) {
    Code to handle the exception
}
```

where *`ExceptionClass`* is the name of the type of exception you are expecting to occur, and which you wish to handle. The handling code can do a variety of things:

- do nothing

- throw/re-throw exception
- report error to the user
- execute ‘correcting’ code
- abort the program

See the lecture on this topic and accompanying code for examples.