Three Rules for Effective Exception Handling

December 4, 2003

Jim Cushing

Exceptions in Java provide a consistent mechanism for identifying and responding to error conditions. Effective exception handling will make your programs more robust and easier to debug. Exceptions are a tremendous debugging aid because they help answer these three questions:

- What went wrong?
- Where did it go wrong?
- Why did it go wrong?

When exceptions are used effectively, *what* is answered by the type of exception thrown, *where* is answered by the exception stack trace, and *why* is answered by the exception message. If you find your exceptions aren't answering all three questions, chances are they aren't being used effectively. Three rules will help you make the best use of exceptions when debugging your programs. These rules are: be specific, throw early, and catch late.

To illustrate these rules of effective exception handling, this article discusses a fictional personal finance manager called JCheckbook. JCheckbook can be used to record and track bank account activity, such as deposits, withdrawals, and checks written. The initial version of JCheckbook runs as a desktop application, but future plans call for an HTML client and a client/server applet implementation.

Be Specific

Java defines an exception class hierarchy, starting with `Throwable`, which is extended by `Error` and `Exception`, which is then extended by `RuntimeException`. These are illustrated in Figure 1.

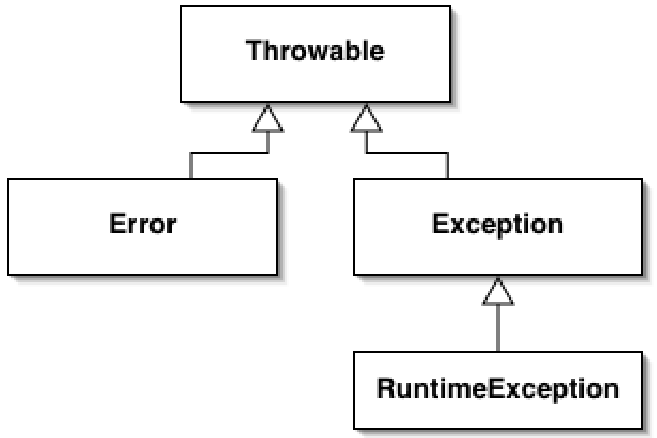

Figure 1. Java exception hierarchy

These four classes are generic and they don't provide much information about what went wrong. While it is legal to instantiate any of these classes (e.g., `new Throwable()`), it is best to think of them as abstract base classes, and work with more specific subclasses. Java provides a substantial number of exception subclasses, and you may define your own exception classes for additional specificity.

For example, the `java.io` package defines the `IOException` subclass, which extends `Exception`. Even more specific are `FileNotFoundException`, `EOFException`, and `ObjectStreamException`, all subclasses of `IOException`. Each one describes a particular type of I/O-related failure: a missing file, an unexpected end-of-file, or a corrupted serialized object stream, respectively. The more specific the exception, the better our program answers *what* went wrong.

It is important to be specific when catching exceptions, as well. For example, JCheckbook may respond to a`FileNotFoundException` by asking the user for a different file name. In the case of an `EOFException`, it may be able to continue with just the information it was able to read before the exception was thrown. If an`ObjectStreamException` is thrown, the program may need to inform the user that the file has been corrupted, and that a backup or a different file needs to be used.

Java makes it fairly easy to be specific when catching exceptions because we can specify multiple `catch` blocks for a single `try` block, each handling a different type of exception in an appropriate manner.

```
File prefsFile = new File(prefsFilename);

try
{
    readPreferences(prefsFile);
}
catch (FileNotFoundException e)
{
    // alert the user that the specified file
    // does not exist
}
catch (EOFException e)
{
    // alert the user that the end of the file
    // was reached
}
catch (ObjectStreamException e)
{
    // alert the user that the file is corrupted
}
catch (IOException e)
{
    // alert the user that some other I/O
    // error occurred
}
```

JCheckbook uses multiple `catch` blocks in order to provide the user with specific information about the type of exception that was caught. For instance, if a `FileNotFoundException` was caught, it can instruct the user to specify a different file. The extra coding effort of multiple catch blocks may be an unnecessary burden in some cases, but in this example, it does help the program respond in a more user-friendly manner.

Should an `IOException` other than those specified by the first three `catch` blocks be thrown, the last `catch` block will handle it by presenting the user with a somewhat more generic error message. This way, the program can provide specific information when possible, but still handle the general case should an unanticipated file-related exception "slip by."

Sometimes, developers will catch a generic `Exception` and then display the exception class name or stack trace, in order to "be specific." Don't do this. Seeing `java.io.EOFException` or a stack trace printed to the screen is likely to confuse, rather than help, the user. Catch specific exceptions and provide the user with specific information in English (or some other human language). Do, however, include the exception stack trace in your log file. Exceptions and stack traces are meant as an aid to the developer, not to the end user.

Finally, notice that instead of catching the exception in the `readPreferences()` method, JCheckbook defers catching and handling the exception until it reaches the user interface level, where it can alert the user with a dialog box or in some other fashion. This is what is meant by "catch late," as will be discussed later in this article.

Throw Early

The exception stack trace helps pinpoint *where* an exception occurred by showing us the exact sequence of method calls that lead to the exception, along with the class name, method name, source code filename, and line number for each of these method calls. Consider the stack trace below:

```
java.lang.NullPointerException
    at java.io.FileInputStream.open(Native Method)
    at java.io.FileInputStream.<init>(FileInputStream.java:103)
    at jcheckbook.JCheckbook.readPreferences(JCheckbook.java:225)
    at jcheckbook.JCheckbook.startup(JCheckbook.java:116)
    at jcheckbook.JCheckbook.<init>(JCheckbook.java:27)
    at jcheckbook.JCheckbook.main(JCheckbook.java:318)
```

This shows that the `open()` method of the `FileInputStream` class threw a `NullPointerException`. But notice that`FileInputStream.close()` is part of the standard Java class library. It is much more likely that the problem that is causing the exception to be thrown is within our own code, rather than the Java API. So the problem must have occurred in one of the preceding methods, which fortunately are also displayed in the stack trace.

What is not so fortunate is that `NullPointerException` is one of the least informative (and most frequently encountered and frustrating) exceptions in Java. It doesn't tell us what we really want to know: exactly *what* is null. Also, we have to backtrack a few steps to find out *where* the error originated.

By stepping backwards through the stack trace and investigating our code, we determine that the error was caused by passing a null filename parameter to the `readPreferences()` method. Since `readPreferences()` knows it cannot proceed with a null filename, it checks for this condition immediately:

```
public void readPreferences(String filename)
    throws IllegalArgumentException
{
    if (filename == null)
    {
        throw new IllegalArgumentException
                            ("filename is null");
    }  //if

    //...perform other operations...

    InputStream in = new FileInputStream(filename);

    //...read the preferences file...
}
```

By throwing an exception early (also known as "failing fast"), the exception becomes both more specific and more accurate. The stack trace immediately shows what went wrong (an illegal argument value was supplied), why this is an error (`null` is not allowed for `filename`), and where the error occurred (early in the `readPreferences()` method). This keeps our stack trace honest:

```
java.lang.IllegalArgumentException: filename is null
    at jcheckbook.JCheckbook.readPreferences(JCheckbook.java:207)
    at jcheckbook.JCheckbook.startup(JCheckbook.java:116)
    at jcheckbook.JCheckbook.<init>(JCheckbook.java:27)
    at jcheckbook.JCheckbook.main(JCheckbook.java:318)
```

In addition, the inclusion of an exception message ("filename is null") makes the exception more informative by answering specifically what was null, an answer we don't get from the `NullPointerException` thrown by the earlier version of our code.

Failing fast by throwing exceptions as soon as an error is detected can eliminate the need to construct objects or open resources, such as files or network connections, that won't be needed. The clean-up effort associated with opening these resources is also eliminated.

Catch Late

A common mistake of many Java developers, both new and experienced, is to catch an exception before the program can handle it in an appropriate manner. The

Java compiler reinforces this behavior by insisting that checked exceptions either be caught or declared. The natural tendency is to immediately wrap the code in a `try` block and catch the exception to stop the compile from reporting errors.

The question is, what to do with an exception after it is caught? The absolute worst thing to do is nothing. An empty`catch` block swallows the exception, and all information about what, where, and why something went wrong is lost forever. Logging the exception is slightly better, since there is at least a record of the exception. But we can hardly expect that the user will read or even understand the log file and stack trace. It is not appropriate for`readPreferences()` to display a dialog with an error message, because while JCheckbook currently runs as a desktop application, we also plan to make it an HTML-based web application. In that case, displaying an error dialog is not an option. Also, in both the HTML and the client/server versions, the preferences would be read on the server, but the error needs to be displayed in the web browser or on the client. The `readPreferences()` method should be designed with these future needs in mind. Proper separation of user interface code from program logic increases the reusability of our code.

Catching an exception too early, before it can properly be handled, often leads to further errors and exceptions. For example, had the `readPreferences()` method shown earlier immediately caught and logged the`FileNotFoundException` that could be thrown while calling the `FileInputStream` constructor, the code would look something like this:

```
public void readPreferences(String filename)
{
    //...

    InputStream in = null;

    // DO NOT DO THIS!!!
    try
    {
        in = new FileInputStream(filename);
    }
    catch (FileNotFoundException e)
    {
        logger.log(e);
    }

    in.read(...);

    //...
}
```

This code catches `FileNotFoundException`, when it really cannot do anything to recover from the error. If the file is not found, the rest of the method certainly cannot read from the file. What would happen should`readPreferences()` be called with the name of file that doesn't exist? Sure, the `FileNotFoundException` would be logged, and if we happened to be looking at the log file at the time, we'd be aware of this. But what happens when the program tries to read data from the file? Since the file doesn't exist, `in` is `null`, and a `NullPointerException`gets thrown.

When debugging a program, instinct tells us to look at the latest information in the log. That's going to be the`NullPointerException`, dreaded because it is so unspecific. The stack trace lies, not only about what went wrong (the real error is a `FileNotFoundException`, not a `NullPointerException`), but also about where the error originated. The problem occurred several lines of code away from where the `NullPointerException` was thrown, and it could have easily been several method calls and classes removed. We end up wasting time chasing red herrings that distract our attention from the true source of the error. It is not until we scroll back in the log file that we see what actually caused the program to malfunction.

What should `readPreferences()` do instead of catching the exceptions? It may seem counterintuitive, but often the best approach is to simply let it go; don't catch the exception immediately. Leave that responsibility up to the code that calls `readPreferences()`. Let that code determine the appropriate way to handle a missing preferences file, which could mean prompting the user for another file, using default values, or, if no other approach works, alerting the user of the problem and exiting the application.

The way to pass responsibility for handling exceptions further up the call chain is to declare the exception in the`throws` clause of the method. When declaring which exceptions may be thrown, remember to be as specific as possible. This serves to document what types of exceptions a program calling your method should anticipate and be ready to handle. For example, the "catch late" version of the `readPreferences()` method would look like this:

```
public void readPreferences(String filename)
    throws IllegalArgumentException,
           FileNotFoundException, IOException
{
    if (filename == null)
    {
        throw new IllegalArgumentException
                        ("filename is null");
    }  //if

    //...

    InputStream in = new FileInputStream(filename);

    //...
}
```

Technically, the only exception we need to declare is `IOException`, but we document our code by declaring that the method may specifically throw a `FileNotFoundException`. `IllegalArgumentException` need not be declared, because it is an unchecked exception (a subclass of `RuntimeException`). Still, including it serves to document our code (the exceptions should also be noted in the JavaDocs for the method).

Of course, eventually, your program needs to catch exceptions, or it may terminate unexpectedly. But the trick is to catch exceptions at the proper layer, where your program can either meaningfully recover from the exception and continue without causing further errors, or provide the user with specific information, including instructions on how to recover from the error. When it is not practical for a method to do either of these, simply let the exception go so it can be caught later on and handled at the appropriate level.

Conclusion

Experienced developers know that the hardest part of debugging usually is not fixing the bug, but finding where in the volumes of code the bug hides. By following the three rules in this article, you can help exceptions help you track down and eradicate bugs and make your programs more robust and user-friendly.