AUG 29, 2019 11:40 AM PDT

ADVANCED JAVA LANGUAGE FEATURES

Exceptions in Java, Part 1: Exception handling basics

Throwing, trying, catching, and cleaning up after Java exceptions

Page 2 of 3

Checked exception controversy

The `throws` clause and checked exceptions are controversial. Many developers hate being forced to specify `throws` or handle the checked exception(s). Learn more about this from my Are checked exceptions good or bad? blog post.

It's necessary to inform the compiler that a checked `ClassNotFoundException` is being thrown by attaching a `throws ClassNotFoundException` clause to the invoking `main()` method's header. After all, the exception isn't handled in this method. When the exception is thrown to the JVM, it will note this. In this case, because there is no parent method of `main()`, it will terminate with a message.

You'll see additional examples of `throws` later on. For now, keep these rules in mind for working with `throws` clauses:

- If at all possible, don't include the names of unchecked exception classes (such as `ArithmeticException`) in a `throws` clause. These names don't need to be included because `throws` clauses are for checked exceptions only. Including unchecked class names only clutters the source code.
- You can append a `throws` clause to a constructor and throw a checked exception from the constructor when something goes wrong while the constructor is executing. The resulting object will not be created.

- If a superclass method declares a `throws` clause, the overriding subclass method doesn't have to declare a `throws` clause. However, if the subclass method declares a `throws` clause, the clause *must not* include the names of checked exception classes that are not also included in the superclass method's `throws` clause--unless they are the names of exception subclasses. For example, given superclass method `void open(String name) throws IOException {}`, the overriding subclass method could be declared as `void open(String name) {}`, `void open(String name) throws IOException {}`, or `void open(String name) throws FileNotFoundException {}`--`FileNotFoundException` subclasses `IOException`. However, you couldn't specify, in the subclass, `void open(String name) throws ClassNotFoundException`, because `ClassNotFoundException` doesn't appear in the superclass's `throws` clause.
- A checked exception class name doesn't need to appear in a `throws` clause when the name of its superclass appears. For example, you don't need to specify `throws FileNotFoundException, IOException`. Only `throws IOException` is necessary.
- The compiler reports an error when a method throws a checked exception and doesn't also handle the exception or list the exception in its `throws` clause.
- You can declare a checked exception class name in a method's `throws` clause without throwing an instance of this class from the method. (Perhaps the method has yet to be fully coded.) However, Java requires that you provide code to handle this exception, even though it isn't thrown.

Keeping checked exceptions readable

There is one more rule to consider: Sometimes, a method throws many checked exceptions, and it's tempting to specify a `throws Exception` clause to save keystrokes. Although specifying only `Exception` saves time, it makes the source code less readable. Someone who is reading your code might have difficulty identifying all of the checked exceptions that are being passed on to the caller. I suggest following this practice for private, throwaway code only.

Trying exceptions

Java provides the `try` block to delimit a sequence of statements that may throw exceptions. A `try` block has the following syntax:

```
try
{
   // one or more statements that might throw exceptions
}
```

The statements in a `try` block serve a common purpose and might directly or indirectly throw an exception. Consider the following example:

```
FileInputStream fis = null;
FileOutputStream fos = null;
try
{
   fis = new FileInputStream(args[0]);
   fos = new FileOutputStream(args[1]);
   int c;
   while ((c = fis.read()) != -1)
      fos.write(c);
}
```

This example excerpts a larger Java Copy application (see this article's code archive) that copies a source file to a destination file. It uses the `java.io` package's `FileInputStream` and `FileOutputStream` classes (introduced later in the article) for this purpose. Think of `FileInputStream` as a way to read an input stream of bytes from a file, and `FileOutputStream` as a way to write an output stream of bytes to a file.

The `FileInputStream(String filename)` constructor creates an input stream to the file identified by `filename`. This constructor throws `FileNotFoundException` when the file doesn't exist, refers to a directory, or another related problem occurs. The `FileOutputStream(String filename)` constructor creates an output stream to the file identified by `filename`. It throws `FileNotFoundException` when the file exists but refers to a directory, doesn't exist and cannot be created, or another related problem occurs.

`FileInputStream` provides an `int read()` method to read one byte and return it as a 32-bit integer. This method returns -1 on end-of-file. `FileOutputStream` provides a `void write(int b)` method to write the byte in the lower 8 bits of b. Either method throws `IOException` when something goes wrong.

The bulk of the example is a `while` loop that repeatedly `read()`s the next byte from the input stream and `write()`s that byte to the output stream, until `read()` signals end-of-file.

The `try` block's file-copy logic is easy to follow because this logic isn't combined with exception-checking code (`if` tests and related `throw` statements hidden in the constructors and methods), exception-handling code (which is executed in one or more associated `catch` blocks), and cleanup code (for closing the source and destination files; this code is relegated to an associated `finally` block). In contrast, C's lack of a similar exception-oriented framework results in more verbose code, as illustrated by the following excerpt from a larger C `cp` application (in this article's code archive) that copies a source file to a destination file:

```c
if ((fpsrc = fopen(argv[1], "rb")) == NULL)
{
   fprintf(stderr, "unable to open %s for reading\n", argv[1]);
   return;
}

if ((fpdst = fopen(argv[2], "wb")) == NULL)
{
   fprintf(stderr, "unable to open %s for writing\n", argv[1]);
   fclose(fpsrc);
   return;
}

while ((c = fgetc(fpsrc)) != EOF)
   if (fputc(c, fpdst) == EOF)
   {
      fprintf(stderr, "unable to write to %s\n", argv[1]);
      break;
   }
```

In this example, the file-copy logic is harder to follow because the logic is intermixed with exception-checking, exception-handling, and cleanup code:

- The two `== NULL` and one `== EOF` checks are the equivalent of the hidden `throw` statements and related checks.
- The three `fprintf()` function calls are the exception-handling code whose Java equivalent would be executed in one or more `catch` blocks.

- The `fclose(fpsrc);` function call is cleanup code whose Java equivalent would be executed in a `finally` block.

Following try with catch or finally

A `try` block must be followed by a `catch` block or a `finally` block. Otherwise, the compiler will report an error. This rule doesn't apply to the `try`-with-resources statement, discussed in the second half of this article.

Catching exceptions

Java's exception-handling capability is based on `catch` blocks. This section introduces `catch` and various `catch` blocks.

The catch block

Java provides the `catch` block to delimit a sequence of statements that handle an exception. A `catch` block has the following syntax:

```
catch (throwableType throwableObject)
{
   // one or more statements that handle an exception
}
```

The `catch` block is similar to a constructor in that it has a parameter list. However, this list consists of only one parameter, which is a throwable type (`Throwable` or one of its subclasses) followed by an identifier for an object of that type.

When an exception occurs, a throwable is created and thrown to the JVM, which searches for the closest `catch` block whose parameter type directly matches or is the supertype of the thrown throwable object. When it finds this block, the JVM passes the throwable to the parameter and executes the `catch` block's statements, which can interrogate the passed throwable and otherwise handle the exception. Consider the following example:

```
catch (FileNotFoundException fnfe)
{
   System.err.println(fnfe.getMessage());
}
```

This example (which extends the previous `try` block example) describes a `catch` block that catches and handles throwables of type `FileNotFoundException`. Only throwables matching this type or a subtype are caught by this block.

Tip

I've found that using an acronym based on the exception type for the parameter name (such as `fnfe`) makes the handler code easier to read.

Suppose the `FileInputStream(String filename)` constructor throws `FileNotFoundException`. The JVM checks the `catch` block following `try` to see if its parameter type matches the throwable type. Detecting a match, the JVM passes the throwable's reference to `fnfe` and transfers execution to the block. The block responds by invoking `getMessage()` to retrieve the exception's message, which it then outputs.

Throwing exceptions from catch blocks

A `catch` block might be unable to fully handle an exception--perhaps it needs to access information provided by some ancestor method in the method-call stack. If it can partly handle the exception, the `catch` block should conclude by rethrowing the exception so that an ancestor handler can finish handling it. Another possibility is to log the exception (for later analysis) and then rethrow it. That technique is demonstrated here:

```
catch (FileNotFoundException fnfe)
{
   logger.log(fnfe);
   throw fnfe;
}
```

Specifying multiple catch blocks

You can specify multiple `catch` blocks after a `try` block. For example, consider this larger excerpt from the aforementioned `Copy` application:

```
FileInputStream fis = null;
FileOutputStream fos = null;
{
   fis = new FileInputStream(args[0]);
   fos = new FileOutputStream(args[1]);
   int c;
   while ((c = fis.read()) != -1)
      fos.write(c);
}
catch (FileNotFoundException fnfe)
{
   System.err.println(fnfe.getMessage());
}
catch (IOException ioe)
{
   System.err.println("I/O error: " + ioe.getMessage());
}
```

The first `catch` block handles `FileNotFoundExceptions` thrown from either constructor. The second `catch` block handles `IOExceptions` thrown from the `read()` and `write()` methods.

When specifying multiple `catch` blocks, don't specify a `catch` block with a supertype before a `catch` block with a subtype. For example, don't place `catch (IOException ioe)` before `catch (FileNotFoundException fnfe)`. If you do, the compiler will report an error because `catch (IOException ioe)` would also handle `FileNotFoundExceptions`, and `catch (FileNotFoundException fnfe)` would never have a chance to execute.

Likewide, don't specify multiple `catch` blocks with the same throwable type. For example, don't specify two `catch (IOException ioe) {}` blocks. Otherwise, the compiler reports an error.

Cleaning up with finally blocks

Whether or not an exception is handled, you may need to perform cleanup tasks, such as closing an open file. Java provides the `finally` block for this purpose.

The `finally` block consists of keyword `finally` followed by a brace-delimited sequence of statements to execute. It may appear after the final `catch` block or after the `try` block.

Cleaning up in a try-catch-finally context

When resources must be cleaned up and an exception isn't being thrown out of a method, a `finally` block is placed after the final `catch` block. This is demonstrated by the following `Copy` excerpt:

```
FileInputStream fis = null;
FileOutputStream fos = null;
try
{
   fis = new FileInputStream(args[0]);
   fos = new FileOutputStream(args[1]);
   int c;
   while ((c = fis.read()) != -1)
      fos.write(c);
}
catch (FileNotFoundException fnfe)
{
   System.err.println(fnfe.getMessage());
}
catch (IOException ioe)
{
   System.err.println("I/O error: " + ioe.getMessage());
}
finally
{
   if (fis != null)
      try
      {
         fis.close();
      }
      catch (IOException ioe)
      {
         // ignore exception
      }

   if (fos != null)
      try
      {
         fos.close();
      }
      catch (IOException ioe)
      {
         // ignore exception
      }
}
```

If the `try` block executes without an exception, execution passes to the `finally` block to close the file input/output streams. If an exception is thrown, the `finally` block executes after the appropriate `catch` block.

`FileInputStream` and `FileOutputStream` inherit a `void close()` method that throws `IOException` when the stream cannot be closed. For this reason, I've wrapped each of `fis.close();` and `fos.close();` in a `try` block. I've left the associated `catch` block empty to illustrate the common mistake of ignoring an exception.

An empty `catch` block that's invoked with the appropriate throwable has no way to report the exception. You might waste a lot of time tracking down the exception's cause, only to discover that you could have detected it sooner if the empty `catch` block had reported the exception, even if only in a log.

Cleaning up in a try-finally context

When resources must be cleaned up and an exception is being thrown out of a method, a `finally` block is placed after the `try` block: there are no `catch` blocks. Consider the following excerpt from a second version of the `Copy` application:

Page 2 of 3