11.6 `finally` Block

Programs that obtain certain resources must return them to the system to avoid so-called **resource leaks.** In programming languages such as C and C++, the most common resource leak is a *memory leak*. Java performs automatic *garbage collection* of memory no longer used by programs, thus avoiding most memory leaks. However, other types of resource leaks can occur. For example, files, database connections and network connections that are not closed properly after they're no longer needed might not be available for use in other programs.

Error-Prevention Tip 11.4

A subtle issue is that Java does not entirely eliminate memory leaks. Java will not garbage-collect an object until there are no remaining references to it. Thus, if you erroneously keep references to unwanted objects, memory leaks can occur.

The optional `finally` block (sometimes referred to as the `finally` **clause**) consists of the `finally` keyword, followed by code enclosed in curly braces. If it's present, it's placed after the last `catch` block. If there are no `catch` blocks, the `finally` block is required and immediately

follows the `try` block.

When the `finally` Block Executes

The `finally` block will execute *whether or not* an exception is thrown in the corresponding `try` block. The `finally` block also will execute if a `try` block exits by using a `return`, `break` or `continue` statement or simply by reaching its closing right brace. The one case in which the `finally` block will *not* execute is if the application *exits early* from a `try` block by calling method `System.exit`. This method, which we demonstrate in Chapter 15, *immediately* terminates an application.

If an exception that occurs in a `try` block cannot be caught by one of that `try` block's `catch` handlers, the program skips the rest of the `try` block and control proceeds to the finally block. Then the program passes the exception to the next outer `try` block—normally in the calling method—where an associated `catch` block might catch it. This process can occur through many levels of `try` blocks. Also, the exception could go *uncaught* (as we discussed in Section 11.3).

If a `catch` block throws an exception, the `finally` block still executes. Then the exception is passed to the next outer `try` block—again, normally in the calling method.

Releasing Resources in a `finally` Block

Because a `finally` block always executes, it typically contains *resource-release code*. Suppose a resource is allocated in a `try` block. If no exception occurs, the `catch` blocks are *skipped* and control proceeds to the `finally` block, which frees the resource. Control then proceeds to the first statement after the `finally` block. If an exception occurs in the `try` block, the `try` block *terminates*. If the program catches the exception in one of the corresponding `catch` blocks, it processes the exception, then the `finally` block *releases the resource* and control proceeds to the first statement after the `finally` block. If the program doesn't catch the exception, the `finally` block *still* releases the resource and an attempt is made to catch the exception in a calling method.

Error-Prevention Tip 11.5

The `finally` block is an ideal place to release resources acquired in a `try` block (such as opened files), which helps eliminate resource leaks.

Performance Tip 11.1

Always release a resource explicitly and at the earliest possible moment at which it's no longer needed. This makes resources available for reuse as early as possible, thus improving resource utilization and program performance.

Demonstrating the `finally` Block

Figure 11.5 demonstrates that the `finally` block executes even if an exception is *not* thrown in the corresponding `try` block. The program contains `static` methods `main` (lines 5–14), `throwException` (lines 17–35) and `doesNotThrowException` (lines 38–50). Methods `throwException` and `doesNotThrowException` are declared static, so main can call them directly without instantiating a `UsingExceptions` object.

```
1   // Fig. 11.5: UsingExceptions.java
2   // try…catch…finally exception handling mechanis
3
4   public class UsingExceptions {
5      public static void main(String[] args) {
6         try {
7            throwException();
8         }
9         catch (Exception exception) { // exception
10           System.err.println("Exception handled i
11        }
```

```
                    12
      13          doesNotThrowException();
                  14      }
                    15
    16      // demonstrate try…catch…finally
17      public static void throwException() throws Ex
18         try { // throw an exception and immediatel
19            System.out.println("Method throwExcepti
20            throw new Exception(); // generate exce
                21         }
22         catch (Exception exception) { // catch exc
      23            System.err.println(
24               "Exception handled in method throwEx
25            throw exception; // rethrow for further
                    26
27            // code here would not be reached; woul
                    28
                29         }
30         finally { // executes regardless of what o
31            System.err.println("Finally executed in
                32         }
                    33
34         // code here would not be reached; would c
                  35      }
                    36
37      // demonstrate finally when no exception occu
38      public static void doesNotThrowException() {
39         try { // try block does not throw an excep
40            System.out.println("Method doesNotThrow
                41         }
42         catch (Exception exception) { // does not
   43            System.err.println(exception);
                  44         }
45         finally { // executes regardless of what o
46            System.err.println("Finally executed in
                47         }
                    48
49         System.out.println("End of method doesNotT
                  50      }
                    51   }
```

```
                Method throwException
      Exception handled in method throwException
          Finally executed in throwException
               Exception handled in main
             Method doesNotThrowException
      Finally executed in doesNotThrowException
         End of method doesNotThrowException
```

Fig. 11.5

`try`…`catch`…`finally` exception-handling mechanism.

`System.out` and `System.err` are **streams**—sequences of bytes. While `System.out` (known as the **standard output stream**) displays a program's output, `System.err` (known as the **standard error stream**) displays a program's errors. Output from these streams can be *redirected* (i.e., sent to somewhere other than the *command prompt*, such as to a *file*). Using two different streams enables you to easily *separate* error messages from other output. For example, data output from `System.err` could be sent to a log file, while data output from `System.out` can be displayed on the screen. For simplicity, this chapter will *not* redirect output from `System.err` but will display such messages to the *command prompt*. You'll learn more about input/output streams in Chapter 15.

Throwing Exceptions Using the `throw` Statement

Method `main` (Fig. 11.5) begins executing, enters its `try` block and immediately calls method `throwException` (line 7). Method `throwException` throws an `Exception`. The statement at line 20 is known as a `throw` **statement**—it's executed to indicate that an exception has occurred. So far, you've caught only exceptions thrown by called methods. You can throw exceptions yourself by using the `throw` statement. Just as with exceptions thrown by the Java API's methods, this indicates to client applications that an error has occurred. A `throw` statement specifies an object to be thrown. The operand of a `throw` can be of any class derived from class `Throwable`.

Software Engineering Observation 11.11

When `toString` *is invoked on any* `Throwable` *object, its resulting* `String` *includes the descriptive* `string` *that was supplied to the constructor, or simply the class name if no* `string was supplied.`

Software Engineering Observation 11.12

An exception can be thrown without containing information about the problem that occurred. In this case, simply knowing that an exception of a particular type occurred may provide sufficient information for the handler to process the problem correctly.

Software Engineering Observation 11.13

Throw exceptions from constructors to indicate that the constructor parameters are not valid—this prevents an object from being created in an invalid state.

Rethrowing Exceptions

Line 25 of Fig. 11.5 **rethrows the exception**. Exceptions are rethrown when a `catch` block, upon receiving an exception, decides either that it cannot process that exception or that it can only partially process it. Rethrowing an exception defers the exception handling (or perhaps a portion of it) to another `catch` block associated with an outer `try` statement. An exception is rethrown by using the `throw` **keyword**, followed

by a reference to the exception object that was just caught. Exceptions cannot be rethrown from a `finally` block, as the exception parameter (a local variable) from the `catch` block no longer exists.

When a rethrow occurs, the *next enclosing* `try` *block* detects the rethrown exception, and that `try` block's `catch` blocks attempt to handle it. In this case, the next enclosing `try` block is found at lines 6–8 in method `main`. Before the rethrown exception is handled, however, the `finally` block (lines 30–32) executes. Then method `main` detects the rethrown exception in the `try` block and handles it in the `catch` block (lines 9–11).

Next, `main` calls method `doesNotThrowException` (line 13). No exception is thrown in `doesNotThrowException`'s `try` block (lines 39–41), so the program skips the `catch` block (lines 42–44), but the `finally` block (lines 45–47) nevertheless executes. Control proceeds to the statement after the `finally` block (line 49). Then control returns to `main` and the program terminates.

Common Programming Error 11.4

If an exception has not been caught when control enters a `finally` *block and the* `finally` *block throws an exception that's not caught in the* `finally` *block,* the first exception

will be lost *and the exception from the* `finally` *block will be returned to the calling method.*

 Error-Prevention Tip 11.6

Avoid placing in a `finally` *block code that can* `throw` *an exception. If such code is required, enclose the code in a* `try`…`catch` *within the* `finally` *block.*

 Common Programming Error 11.5

Assuming that an exception thrown from a `catch` *block will be processed by that* `catch` *block or any other* `catch` *block associated with the same* `try` *statement can lead to logic errors.*

 Good Programming Practice 11.1

Exception handling removes error-processing code from the main line of a program's code to improve program clarity. Do

not place `try`…`catch`… `finally` *around every statement that may throw an exception. This decreases readability. Rather, place one* `try` *block around a significant portion of your code, follow the* `try` *with* `catch` *blocks that handle each possible exception and follow the* `catch` *blocks with a single* `finally` *block (if one is required).*