Catching Subclass Exceptions

If a `catch` handler is written to catch *superclass* exception objects, it can also catch all objects of that class's *subclasses*. This enables `catch` to handle related exceptions *polymorphically*. You can catch each subclass individually if those exceptions require different processing.

Only the First Matching `catch` Executes

If *multiple* `catch` blocks match a particular exception type, only the *first* matching `catch` block executes when an exception of that type occurs. It's a compilation error to catch the *exact same type* in two different `catch` blocks associated with a particular `try` block. However, there can be several `catch` blocks that match an exception—i.e., several `catch` blocks whose types are the same as the exception type or a superclass of that type. For instance, we could follow a `catch` block for type `ArithmeticException` with a `catch` block for type `Exception`—both would match `ArithmeticExceptions`, but only the first matching `catch` block would execute.

Common Programming Error 11.3

Placing a `catch` block for a superclass exception type before other `catch` blocks that catch subclass exception types would prevent those `catch` blocks from executing, so a compilation error occurs.

Error-Prevention Tip 11.3

Catching subclass types individually is subject to error if you forget to test for one or more of the subclass types explicitly; catching the superclass guarantees that objects of all subclasses will be caught. Positioning a `catch` block for the superclass type after all other subclass `catch` blocks ensures that all subclass exceptions are eventually caught.

Software Engineering Observation 11.7

In industry, throwing or catching type `Exception` is discouraged—we use it here simply to demonstrate exception-handling mechanics. In subsequent chapters, we generally throw and catch more specific exception types.

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 `finally` block (which consists of the `finally` keyword, followed by code enclosed in curly braces), sometimes referred to as the **`finally` clause**, is optional. If it's

present, it's placed after the last `catch` block. If there are no `catch` blocks, the `finally` block, if present, 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 6–18), `throwException` (lines 21–44) and `doesNotThrowException` (lines 47–64). 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 mechanism.
3
4  public class UsingExceptions
5  {
6     public static void main(String[] args)
7     {
8        try
9        {
10          throwException();
11       }
12       catch (Exception exception) // exception thrown by throwException
13       {
14          System.err.println("Exception handled in main");
15       }
16
17       doesNotThrowException();
18    }
19
20    // demonstrate try...catch...finally
21    public static void throwException() throws Exception
22    {
23       try // throw an exception and immediately catch it
24       {
25          System.out.println("Method throwException");
26          throw new Exception(); // generate exception
27       }
28       catch (Exception exception) // catch exception thrown in try
29       {
30          System.err.println(
31             "Exception handled in method throwException");
32          throw exception; // rethrow for further processing
33
34          // code here would not be reached; would cause compilation errors
35
36       }
37       finally // executes regardless of what occurs in try...catch
38       {
39          System.err.println("Finally executed in throwException");
40       }
41
42       // code here would not be reached; would cause compilation errors
43
44    }
45
46    // demonstrate finally when no exception occurs
47    public static void doesNotThrowException()
48    {
49       try // try block does not throw an exception
50       {
51          System.out.println("Method doesNotThrowException");
52       }
```

Fig. 11.5 | `try`...`catch`...`finally` exception-handling mechanism. (Part 1 of 2.)

```
53          catch (Exception exception) // does not execute
54          {
55             System.err.println(exception);
56          }
57          finally // executes regardless of what occurs in try...catch
58          {
59             System.err.println(
60                "Finally executed in doesNotThrowException");
61          }
62
63          System.out.println("End of method doesNotThrowException");
64       }
65    } // end class UsingExceptions
```

```
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. (Part 2 of 2.)

`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 instance, 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 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 10). Method `throwException` throws an `Exception`. The statement at line 26 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 11.8

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 11.9

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 11.10

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 32 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 8–11 in method `main`. Before the rethrown exception is handled, however, the `finally` block (lines 37–40) executes. Then method `main` detects the rethrown exception in the `try` block and handles it in the `catch` block (lines 12–15).

Next, `main` calls method `doesNotThrowException` (line 17). No exception is thrown in `doesNotThrowException`'s `try` block (lines 49–52), so the program skips the `catch` block (lines 53–56), but the `finally` block (lines 57–61) nevertheless executes. Control proceeds to the statement after the `finally` block (line 63). 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).