Obtaining Data from an Exception Object

All exceptions derive from class `Throwable`, which has a **`printStackTrace`** method that outputs to the standard error stream the *stack trace* (discussed in Section 11.2). Often this is helpful in testing and debugging. Class `Throwable` also provides a **`getStackTrace`** method that retrieves the stack-trace information that might be printed by `printStackTrace`. Class `Throwable`'s **`getMessage`** method returns the descriptive string stored in an exception.

Error-Prevention Tip 11.7

An exception that's not caught in an application causes Java's default exception handler *to run. This displays the name of the exception, a descriptive message that indicates the problem that occurred and a complete execution stack trace. In an application with a single thread of execution, the application terminates. In an application with multiple threads, the thread that caused the exception terminates. We discuss multithreading in Chapter 23.*

Error-Prevention Tip 11.8

`Throwable` method `toString` (inherited by all `Throwable` subclasses) returns a `String` containing the name of the exception's class and a descriptive message.

The `catch` handler in Fig. 11.6 (lines 12–31) demonstrates `getMessage`, `printStackTrace` and `getStackTrace`. If we wanted to output the stack-trace information to streams other than the standard error stream, we could use the information returned from `getStackTrace` and output it to another stream or use one of the overloaded versions of method `printStackTrace`. Sending data to other streams is discussed in Chapter 15.

Line 14 invokes the exception's `getMessage` method to get the *exception description*. Line 15 invokes the exception's `printStackTrace` method to output the *stack trace* that indicates where the exception occurred. Line 18 invokes the exception's `getStackTrace` method to obtain the stack-trace information as an array of **`StackTraceElement`** objects. Lines 24–30 get each `StackTraceElement` in the array and invoke its methods **`getClassName`**, **`getFileName`**, **`getLineNumber`** and **`getMethodName`** to get the class name, filename, line number and method name, respectively, for that `StackTraceElement`. Each `StackTraceElement` represents *one* method call on the *method-call stack*.

The program's output shows that output of `printStackTrace` follows the pattern: *className.methodName*(*fileName*:*lineNumber*), where *className*, *methodName* and *fileName* indicate the names of the class, method and file in which the exception occurred, respectively, and the *lineNumber* indicates where in the file the exception occurred. You saw this in the output for Fig. 11.2. Method `getStackTrace` enables custom processing of the exception information. Compare the output of `printStackTrace` with the output created from the `StackTraceElements` to see that both contain the same stack-trace information.

Software Engineering Observation 11.11

Occasionally, you might want to ignore an exception by writing a `catch` handler with an empty body. Before doing so, ensure that the exception doesn't indicate a condition that code higher up the stack might want to know about or recover from.

11.8 Chained Exceptions

Sometimes a method responds to an exception by throwing a different exception type that's specific to the current application. If a `catch` block throws a new exception, the orig-

inal exception's information and stack trace are *lost*. Earlier Java versions provided no mechanism to wrap the original exception information with the new exception's information to provide a complete stack trace showing where the original problem occurred. This made debugging such problems particularly difficult. **Chained exceptions** enable an exception object to maintain the complete stack-trace information from the original exception. Figure 11.7 demonstrates chained exceptions.

```
1  // Fig. 11.7: UsingChainedExceptions.java
2  // Chained exceptions.
3
4  public class UsingChainedExceptions
5  {
6     public static void main(String[] args)
7     {
8        try
9        {
10          method1();
11       }
12       catch (Exception exception) // exceptions thrown from method1
13       {
14          exception.printStackTrace();
15       }
16    }
17
18    // call method2; throw exceptions back to main
19    public static void method1() throws Exception
20    {
21       try
22       {
23          method2();
24       } // end try
25       catch (Exception exception) // exception thrown from method2
26       {
27          throw new Exception("Exception thrown in method1", exception);
28       }
29    }
30
31    // call method3; throw exceptions back to method1
32    public static void method2() throws Exception
33    {
34       try
35       {
36          method3();
37       }
38       catch (Exception exception) // exception thrown from method3
39       {
40          throw new Exception("Exception thrown in method2", exception);
41       }
42    }
43
```

Fig. 11.7 | Chained exceptions. (Part 1 of 2.)

```
44      // throw Exception back to method2
45      public static void method3() throws Exception
46      {
47         throw new Exception("Exception thrown in method3");
48      }
49   } // end class UsingChainedExceptions
```

```
java.lang.Exception: Exception thrown in method1
        at UsingChainedExceptions.method1(UsingChainedExceptions.java:27)
        at UsingChainedExceptions.main(UsingChainedExceptions.java:10)
Caused by: java.lang.Exception: Exception thrown in method2
        at UsingChainedExceptions.method2(UsingChainedExceptions.java:40)
        at UsingChainedExceptions.method1(UsingChainedExceptions.java:23)
        ... 1 more
Caused by: java.lang.Exception: Exception thrown in method3
        at UsingChainedExceptions.method3(UsingChainedExceptions.java:47)
        at UsingChainedExceptions.method2(UsingChainedExceptions.java:36)
        ... 2 more
```

Fig. 11.7 | Chained exceptions. (Part 2 of 2.)

Program Flow of Control

The program consists of four methods—`main` (lines 6–16), `method1` (lines 19–29), `method2` (lines 32–42) and `method3` (lines 45–48). Line 10 in method `main`'s `try` block calls `method1`. Line 23 in `method1`'s `try` block calls `method2`. Line 36 in `method2`'s `try` block calls `method3`. In `method3`, line 47 throws a new `Exception`. Because this statement is not in a `try` block, `method3` terminates, and the exception is returned to the calling method (`method2`) at line 36. This statement *is* in a `try` block; therefore, the `try` block terminates and the exception is caught at lines 38–41. Line 40 in the `catch` block throws a new exception. In this case, the `Exception` constructor with *two* arguments is called. The second argument represents the exception that was the original cause of the problem. In this program, that exception occurred at line 47. Because an exception is thrown from the `catch` block, `method2` terminates and returns the new exception to the calling method (`method1`) at line 23. Once again, this statement is in a `try` block, so the `try` block terminates and the exception is caught at lines 25–28. Line 27 in the `catch` block throws a new exception and uses the exception that was caught as the second argument to the `Exception` constructor. Because an exception is thrown from the `catch` block, `method1` terminates and returns the new exception to the calling method (`main`) at line 10. The `try` block in `main` terminates, and the exception is caught at lines 12–15. Line 14 prints a stack trace.

Program Output

Notice in the program output that the first three lines show the most recent exception that was thrown (i.e., the one from `method1` at line 27). The next four lines indicate the exception that was thrown from `method2` at line 40. Finally, the last four lines represent the exception that was thrown from `method3` at line 47. Also notice that, as you read the output in reverse, it shows how many more chained exceptions remain.