11.7 Stack Unwinding and Obtaining Information from an Exception

When an exception is thrown but *not caught* in a particular method, the method-call stack is “unwound,” and an attempt is made to `catch` the exception in the next outer `try` block. This process is called **stack unwinding**. Unwinding the method-call stack means that the method in which the exception was not caught *terminates*, all local variables in that method *go out of scope* and control returns to the statement that originally invoked that method. If a `try` block encloses that statement, an attempt is made to `catch` the exception. If a `try` block does not enclose that statement or if the exception is not caught, stack unwinding occurs again. Figure 11.6 demonstrates stack unwinding, and the exception handler in `main` shows how to access the data in an exception object.

```java
// Fig. 11.6: UsingExceptions.java
// Stack unwinding and obtaining data from an ex

public class UsingExceptions {
   public static void main(String[] args) {
      try {
         method1();
      }
      catch (Exception exception) { // catch exc
         System.err.printf("%s%n%n",exception.ge
         exception.printStackTrace();

         // obtain the stack-trace information
         StackTraceElement[] traceElements = exc

         System.out.printf("%nStack trace from g
         System.out.println("Class\t\tFile\t\t\t

         // loop through traceElements to get ex
         for (StackTraceElement element : traceE
            System.out.printf("%s\t", element.ge
            System.out.printf("%s\t", element.ge
            System.out.printf("%s\t", element.ge
            System.out.printf("%s%n", element.ge
         }
      }
   }

   // call method2; throw exceptions back to mai
   public static void method1() throws Exception
      method2();
   }

   // call method3; throw exceptions back to met
   public static void method2() throws Exception
      method3();
   }

   // throw Exception back to method2
   public static void method3() throws Exception
      throw new Exception("Exception thrown in m
   }
}
```

```
Exception thrown in method3

java.lang.Exception: Exception thrown in method3
       at UsingExceptions.method3(UsingExceptions.ja
       at UsingExceptions.method2(UsingExceptions.ja
       at UsingExceptions.method1(UsingExceptions.ja
       at UsingExceptions.main(UsingExceptions.java:

        Stack trace from getStackTrace:
Class           File                 Line Method
UsingExceptions UsingExceptions.java 41   method3
UsingExceptions UsingExceptions.java 36   method2
UsingExceptions UsingExceptions.java 31   method1
 UsingExceptions UsingExceptions.java 7    main
```

Fig. 11.6

Stack unwinding and obtaining data from an exception object.

Stack Unwinding

In `main`, the `try` block (lines 6–8) calls `method1` (declared at lines 30–32), which in turn calls `method2` (declared at lines 35–37), which in turn calls `method3` (declared at lines 40–42). Line 41 in `method3` throws an `Exception` object —this is the *throw point*. Because the `throw` statement is *not* enclosed in a `try` block, *stack unwinding* occurs—`method3` terminates at line 41, then returns control to the statement in `method2` that invoked `method3`(i.e., line 36). Because *no* `try` block encloses line 36, *stack unwinding* occurs again —`method2` terminates at line 36 and returns control to the statement in `method1` that invoked `method2` (i.e., line 31). Because *no* `try` block encloses line 31, *stack unwinding* occurs one more time—`method1` terminates at line 31 and returns control to the statement in `main` that invoked `method1` (i.e., line 7). The `try` block at lines 6–8 encloses this statement. The exception has not been handled, so the `try` block terminates and the first matching `catch` block (lines 9–26) catches and processes the exception. If there were no matching `catch` blocks, and the exception is *not declared* in each method that throws it, a compilation error would occur—main does not have a `throws` clause because `main` catches the exception. Remember that this is not always the case—for *unchecked* exceptions, the application will compile, but it will run with unexpected results.

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 (inherited by all `Throwable` subclasses) returns the descriptive string stored in an exception. `Throwable` method `toString` (also inherited by all `Throwable` subclasses) returns a `String` containing the name of the exception’s class and a descriptive message.

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.

The `catch` handler in Fig. 11.6 (lines 9–26) demonstrates `getMessage`, `printStack-Trace` 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 10 invokes the exception's `getMessage` method to get the *exception description*. Line 11 invokes the exception's `printStackTrace` method to output the *stack trace* that indicates where the exception occurred. Line 14 invokes the exception's `getStackTrace` method to obtain the stack-trace information as an array of `StackTraceElement` objects. Lines 20–25 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 the 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 11.14

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.*

Java SE 9: Stack-Walking

API

`Throwable` methods `printStackTrace` and `getStackTrace` each process the entire method-call stack. When debugging, this can be inefficient—for example, you may be interested only in stack frames corresponding to methods of a specific class. Java SE 9 introduces the **Stack-Walking API** (class `StackWalker` in package `java.lang`), which uses lambdas and streams (introduced in Chapter 17) to access method-call-stack information in a more efficient manner. You can learn more about this API at:

```
http://openjdk.java.net/jeps/259
```