11.3 Example: Handling `ArithmeticExceptions` and `InputMismatchExceptions`

The application in Fig. 11.3, which is based on Fig. 11.2, uses *exception handling* to process any `ArithmeticExceptions` and `InputMistmatchExceptions` that arise. The application still prompts the user for two integers and passes them to method `quotient`, which calculates the quotient and returns an `int` result. This version of the application uses exception handling so that if the user makes a mistake, the program catches and handles (i.e., deals with) the exception—in this case, allowing the user to re-enter the input.

```
// Fig. 11.3: DivideByZeroWithExceptionHandling.
// Handling ArithmeticExceptions and InputMismat
import java.util.InputMismatchException;
import java.util.Scanner;

public class DivideByZeroWithExceptionHandling
{
   // demonstrates throwing an exception when a
   public static int quotient(int numerator, int
         throws ArithmeticException{
      return numerator / denominator; // possibl
   }

   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);
      boolean continueLoop = true; // determines

      do {
         try { // read two numbers and calculate
            System.out.print("Please enter an in
            int numerator = scanner.nextInt();
            System.out.print("Please enter an in
            int denominator = scanner.nextInt();

            int result = quotient(numerator, den
            System.out.printf("%nResult: %d / %d
               denominator, result);
            continueLoop = false; // input succe
         }
         catch (InputMismatchException inputMism
            System.err.printf("%nException: %s%n
               inputMismatchException);
            scanner.nextLine(); // discard input
            System.out.printf(
               "You must enter integers. Please
         }
         catch (ArithmeticException arithmeticEx
            System.err.printf("%nException: %s%n
            System.out.printf(
               "Zero is an invalid denominator.
         }
      } while (continueLoop);
   }
}
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 7
```

```
Result: 100 / 7 = 14
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 0

Exception: java.lang.ArithmeticException: / by zero
Zero is an invalid denominator. Please try again.

Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: hello

Exception: java.util.InputMismatchException
You must enter integers. Please try again.

Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

Fig. 11.3

Handling `ArithmeticExceptions` and `InputMismatchExceptions`.

The first sample execution in Fig. 11.3 does *not* encounter any problems. In the second execution the user enters a *zero denominator*, and an `ArithmeticException` exception occurs. In the third execution the user enters the string `"hello"` as the denominator, and an `InputMismatchException` occurs. For each exception, the user is informed of the mistake and asked to try again, then is prompted for two new integers. In each sample execution, the program runs to completion successfully.

Class `InputMismatchException` is imported in line 3. Class `ArithmeticException` does not need to be imported because it's in package `java.lang`. Line 16 creates the `boolean` variable `continueLoop`, which is `true` if the user has *not* yet entered valid input. Lines 18–42 repeatedly ask users for input until a *valid* input is received.

Enclosing Code in a `try` Block

Lines 19–29 contain a `try` **block**, which encloses the code that *might* `throw` an exception and the code that should *not* execute if an exception occurs (i.e., if an exception occurs, the remaining code in the `try` block will be skipped). A `try` block consists of the keyword `try` followed by a block of code enclosed in curly braces. [*Note:* The term "`try` block" sometimes refers only to the block of code that follows the `try` keyword (not including the try keyword itself). For simplicity, we use the term “`try` block” to refer to the block of code that follows the `try` keyword, as well as the `try` keyword.] The statements that read the integers from the keyboard (lines 21 and 23) each use method `nextInt` to read an `int` value. Method `nextInt` throws an `InputMismatchException` if the value read in is *not* an integer.

The division that can cause an `ArithmeticException` is not performed in the `try` block. Rather, the call to method `quotient` (line 25) invokes the code that attempts the division (line 11); the JVM *throws* an `ArithmeticException` object when the denominator is zero.

Software Engineering Observation 11.2

Exceptions may surface through explicitly mentioned code in a `try` *block, through deeply nested method calls initiated by code in a* `try` *block or from the Java Virtual Machine as it executes Java bytecodes.*

Catching Exceptions

The `try` block in this example is followed by two `catch` blocks—one that handles an `InputMismatchException` (lines 30–36) and one that handles an `ArithmeticException` (lines 37–41). A `catch` **block** (also called a `catch` **clause** or **exception handler**) *catches* (i.e., receives) and *handles* an exception. A `catch` block begins with the keyword `catch` followed by a parameter in parentheses (called the *exception parameter*, discussed shortly) and a block of code enclosed in curly braces.

At least one `catch` block or a `finally` **block** (discussed in Section 11.6) *must* immediately follow the `try` block. Each `catch` block specifies in parentheses an **exception parameter** that identifies the exception type the handler can process. When an exception occurs in a `try` block, the `catch` block that executes is the *first* one whose type matches the type of the exception that occurred (i.e., the type in the `catch` block matches the thrown exception type exactly or is a direct or indirect superclass of it). The exception parameter's name enables the `catch` block to interact with a caught exception object—e.g., to implicitly invoke the caught exception's `toString` method (as in lines 31–32 and 38), which displays basic information about the exception. Notice that we use the `System.err` **(standard error stream) object** to output error messages. By default, `System.err`'s print methods, like those of `System.out`, display data to the *command prompt*.

Line 33 in the first `catch` block calls `Scanner` method `nextLine`. Because an `InputMismatchException` occurred, the call to method `nextInt` never successfully read in the user’s data—so we read that input with a call to method `nextLine`. We do not do anything with the input at this point, because we know that it’s *invalid*. Each `catch` block displays an error message and asks the user to try again. After either `catch` block terminates, the user is prompted for input. We’ll soon take a deeper look at how this flow of control works in exception handling.

Common Programming Error 11.1

It’s a syntax error to place code between a `try` *block and its corresponding* `catch` *blocks.*

Multi-`catch`

It’s relatively common for a `try` block to be followed by several `catch` blocks to handle various types of exceptions. If the bodies of several `catch` blocks are identical, you can use the **multi-**`catch` feature to catch those exception types in a *single* `catch` handler and perform the same task. The syntax for a *multi*-`catch` is:

```
catch (Type1 | Type2 | Type3 e)
```

Each exception type is separated from the next with a vertical bar (`|`). The preceding line of code indicates that *any* of the types (or their subclasses) can be caught in the exception handler. Any number of `Throwable` types can be specified in a multi-`catch`. In this case, the exception parameter's type is the common superclass of the specified types.

Uncaught Exceptions

An **uncaught exception** is one for which there are no matching `catch` blocks. You saw uncaught exceptions in the second and third outputs of Fig. 11.2. Recall that when exceptions occurred in that example, the application terminated early (after displaying the exception's *stack trace*). This does not always occur as a result of uncaught exceptions. Java uses a "multithreaded" model of program execution—each **thread** is a *concurrent activity*. One program can have many threads. If a program has only *one* thread, an uncaught exception will cause the program to terminate. If a program has *multiple* threads, an uncaught exception will terminate *only* the thread in which the exception occurred. In such programs, however, certain threads may rely on others, and if one thread terminates due to an uncaught exception, there may be adverse effects on the rest of the program. Chapter 23, Concurrency, discusses these issues in depth.

Termination Model of

Exception Handling

If an exception occurs in a `try` block (such as an `InputMismatchException` being thrown as a result of the code at line 23 of Fig. 11.3), the `try` block *terminates* immediately and program control transfers to the *first* of the following `catch` blocks in which the exception parameter's type matches the thrown exception's type. In Fig. 11.3, the first `catch` block catches `InputMismatchExceptions` (which occur if invalid input is entered) and the second `catch` block catches `ArithmeticExceptions` (which occur if an attempt is made to divide by zero). After the exception is handled, program control does *not* return to the throw point, because the `try` block has *expired* (and its *local variables* have been *lost*). Rather, control resumes after the last `catch` block. This is known as the **termination model of exception handling**. Some languages use the **resumption model of exception handling**, in which, after an exception is handled, control resumes just after the *throw point*.

Notice that we name our exception parameters (`inputMismatchException` and `arithmeticException`) based on their type. Java programmers often simply use the letter `e` as the name of their exception parameters.

After executing a `catch` block, this program's flow of control proceeds to the first statement after the last `catch` block (line 42 in this case). The condition in the `do`…`while` statement is `true` (variable `continueLoop` contains its initial value of `true`), so control returns to the beginning of the loop and the user is again prompted for input. This control statement will loop until *valid* input is entered. At that point, program control reaches line 28, which assigns `false` to variable `continueLoop`. The `try` block then *terminates*. If no exceptions are thrown in the `try` block, the `catch` blocks are *skipped* and control continues with the first statement after the `catch` blocks (we'll learn about another possibility when we discuss the `finally` block in Section 11.6). Now the condition for the `do`…`while` loop is `false`, and method `main` ends.

The `try` block and its corresponding `catch` and/or `finally` blocks form a `try` **statement**. Do not confuse the terms "`try` block" and "`try` statement"—the latter includes the `try` block as well as the following `catch` blocks and/or `finally` block.

As with any other block of code, when a `try` block terminates, *local variables* declared in the block *go out of scope* and are no longer accessible; thus, the local variables of a `try` block are not accessible in the corresponding `catch` blocks. When a `catch` block *terminates*, *local variables* declared within the `catch` block (including the exception parameter of that `catch` block) also *go out of scope* and are *destroyed*. Any remaining `catch` blocks in the `try` statement are *ignored*, and execution resumes at the first line of code after the `try…catch` sequence—this will be a `finally` block, if one is present.

Using the `throws` Clause

In method `quotient` (Fig. 11.3, lines 9–12), line 10 is known as a `throws` **clause**. It specifies the exceptions the method *might* throw if problems occur. This clause, which must appear after the method’s parameter list and before the body, contains a comma-separated list of the exception types. Such exceptions may be thrown by statements in the method’s body or by methods called from there. We’ve added the `throws` clause to this application to indicate that this method might throw an `ArithmeticException`. Method `quotient`’s callers are thus informed that the method might throw an `ArithmeticException`. Some exception types, such as `ArithmeticException`, are not required to be listed in the `throws` clause. For those that are, the method can throw exceptions that have the *is-a* relationship with the classes listed in the `throws` clause. You’ll learn more about this in Section 11.5.

Error-Prevention Tip 11.1

Read a method’s online API documentation before using it in a program. The documentation specifies exceptions thrown by the method (if any) and indicates reasons why such exceptions may occur. Next, read the online API documentation for the specified exception classes. The documentation for an *exception class typically contains potential reasons that such exceptions occur. Finally, provide for handling those exceptions in your program.*

When line 11 executes, if the `denominator` is zero, the JVM throws an `ArithmeticException` object. This object will be caught by the `catch` block at lines 37–41, which displays basic information about the exception by *implicitly* invoking the exception's `toString` method, then asks the user to try again.

If the `denominator` is not zero, method `quotient` performs the division and returns the result to the point of invocation of method `quotient` in the `try` block (line 25). Lines 26–27 display the result of the calculation and line 28 sets `continueLoop` to `false`. In this case, the `try` block completes successfully, so the program skips the `catch` blocks and fails the condition at line 42, and method `main` completes execution normally.

When `quotient` throws an `ArithmeticException`, `quotient` *terminates* and does *not* return a value, and `quotient`'s *local variables go out of scope* (and are destroyed). If `quotient` contained local variables that were references to objects and there were no other references to those objects, the objects would be marked for *garbage collection*. Also, when an exception occurs, the `try` block from which `quotient` was called *terminates* before lines 26–28 can execute. Here, too, if local variables were created in the `try` block prior to the exception's being thrown, these variables would go out of scope.

If an `InputMismatchException` is generated by lines 21 or 23, the `try` block *terminates* and execution *continues* with the `catch` block at lines 30–36. In this case, method `quotient` is not called. Then method `main` continues after the last `catch` block.