11.2 Example: Divide by Zero without Exception Handling

First we demonstrate what happens when errors arise in an application that does not use exception handling. Figure 11.2 prompts the user for two integers and passes them to method `quotient`, which calculates the integer quotient and returns an `int` result. In this example, you'll see that exceptions are **thrown** (i.e., the exception occurs) by a method when it detects a problem and is unable to handle it.

```
1   // Fig. 11.2: DivideByZeroNoExceptionHandling.ja
2   // Integer division without exception handling.
      3   import java.util.Scanner;
                      4
5   public class DivideByZeroNoExceptionHandling {
6      // demonstrates throwing an exception when a
7      public static int quotient(int numerator, int
8         return numerator / denominator; // possibl
                  9      }
                     10
11     public static void main(String[] args) {
12        Scanner scanner = new Scanner(System.in);
                     13
14        System.out.print("Please enter an integer
  15        int numerator = scanner.nextInt();
16        System.out.print("Please enter an integer
17        int denominator = scanner.nextInt();
                     18
```

```
19          int result = quotient(numerator, denominat
           20          System.out.printf(
21              "%nResult: %d / %d = %d%n", numerator,
                      22       }
                        23    }
```

```
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 in thread "main" java.lang.ArithmeticExcept
        at DivideByZeroNoExceptionHandling.quotient(
           DivideByZeroNoExceptionHandling.java:8)
         at DivideByZeroNoExceptionHandling.main(
           DivideByZeroNoExceptionHandling.java:19)
```

```
Please enter an integer numerator: 100
   Please enter an integer denominator: hello
Exception in thread "main" java.util.InputMismatchExc
        at java.util.Scanner.throwFor(Unknown Source)
        at java.util.Scanner.next(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
         at DivideByZeroNoExceptionHandling.main(
            DivideByZeroNoExceptionHandling.java:17)
```

Fig. 11.2

Integer division without exception handling.

Stack Trace

The first sample execution in Fig. 11.2 shows a successful division. In the second execution, the user enters the value `0` as the denominator. Several lines of information are displayed in response to this invalid input. This information is known as a **stack trace**, which includes the name of the exception (`java.lang.ArithmeticException`) in a descriptive message that indicates the problem and the method-call stack (i.e., the call chain) at the time the problem occurred. The stack trace includes the path of execution that led to the exception method by method. This helps you debug the program. Even if a problem has not occurred, you can see the stack trace any time by calling `Thread.dumpStack()`.

Stack Trace for an `ArithmeticException`

The first line specifies that an `ArithmeticException` has occurred. The text after the name of the exception (“`/ by zero`”) indicates that this exception occurred as a result of an attempt to divide by zero. Java does not allow division by zero in integer arithmetic. When this occurs, Java throws an

`ArithmeticException`. `ArithmeticExceptions` can arise from a number of different problems, so the extra data ("`/by zero`") provides more specific information.

Starting from the last line of the stack trace, we see that the exception was detected in line 19 of method `main`. Each line of the stack trace contains the class name and method (e.g., `DivideByZeroNoExceptionHandling.main`) followed by the filename and line number (e.g., `DivideByZeroNoExceptionHandling.java:19`). Moving up the stack trace, we see that the exception occurs in line 8, in method `quotient`. The top row of the call chain indicates the **throw point**—the initial point at which the exception occurred. The throw point of this exception is in line 8 of method `quotient`.

Side Note Regarding Floating-Point Arithmetic

Java *does* allow division by zero with floating-point values. Such a calculation results in positive or negative infinity, which is represented as a floating-point value that displays as "`Infinity`" or "`-Infinity`". If you divide 0.0 by 0.0, the result is `NaN` (not a number), which is represented as a floating-point value that displays as `"NaN"`. If you need to compare a floating-point value to `NaN`, use the method `isNaN` of class Float (for `float` values) or of class `Double` (for `double` values). Classes `Float` and `Double` are in package

`java.lang`.

Stack Trace for an `InputMismatchException`

In the third execution, the user enters the string `"hello"` as the denominator. Notice again that a stack trace is displayed. This informs us that an `InputMismatchException` has occurred (package `java.util`). Our prior examples that input numeric values assumed that the user would input a proper integer value. However, users sometimes make mistakes and input noninteger values. An **`InputMismatchException`** occurs when `Scanner` method `nextInt` receives a `string` that does not represent a valid integer. Starting from the end of the stack trace, we see that the exception was detected in line 17 of method main. Moving up the stack trace, we see that the exception occurred in method `nextInt`. Notice that in place of the filename and line number, we're provided with the text `Unknown Source`. This means that the so-called *debugging symbols* that provide the filename and line number information for that method's class were not available to the JVM—this is typically the case for the classes of the Java API. Many IDEs have access to the Java API source code and will display filenames and line numbers in stack traces.

Program Termination

In the sample executions of Fig. 11.2 when exceptions occur and stack traces are displayed, the program also *exits*. This does not always occur in Java. Sometimes a program may continue even though an exception has occurred and a stack trace has been printed. In such cases, the application may produce unexpected results. For example, a graphical user interface (GUI) application will often continue executing. In Fig. 11.2 both types of exceptions were detected in method `main`. In the next example, we'll see how to *handle* these exceptions so that you can enable the program to run to normal completion.