

a number of different problems, so the extra data (“/ by zero”) provides more specific information. Java *does* allow division by zero with floating-point values. Such a calculation results in the value positive or negative infinity, which is represented in Java as a floating-point value (but displays as the string `Infinity` or `-Infinity`). If 0.0 is divided by 0.0, the result is NaN (not a number), which is also represented in Java as a floating-point value (but 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`.

Starting from the last line of the stack trace, we see that the exception was detected in line 22 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:22`). Moving up the stack trace, we see that the exception occurs in line 10, 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 10 of method `quotient`.

#### *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 20 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.

### 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 `InputMismatchExceptions` 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 ex-

ception 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.

---

```

1 // Fig. 11.3: DivideByZeroWithExceptionHandling.java
2 // Handling ArithmeticExceptions and InputMismatchExceptions.
3 import java.util.InputMismatchException;
4 import java.util.Scanner;
5
6 public class DivideByZeroWithExceptionHandling
7 {
8     // demonstrates throwing an exception when a divide-by-zero occurs
9     public static int quotient(int numerator, int denominator)
10     throws ArithmeticException
11     {
12         return numerator / denominator; // possible division by zero
13     }
14
15     public static void main(String[] args)
16     {
17         Scanner scanner = new Scanner(System.in);
18         boolean continueLoop = true; // determines if more input is needed
19
20         do
21         {
22             try // read two numbers and calculate quotient
23             {
24                 System.out.print("Please enter an integer numerator: ");
25                 int numerator = scanner.nextInt();
26                 System.out.print("Please enter an integer denominator: ");
27                 int denominator = scanner.nextInt();
28
29                 int result = quotient(numerator, denominator);
30                 System.out.printf("\nResult: %d / %d = %d\n", numerator,
31                                 denominator, result);
32                 continueLoop = false; // input successful; end looping
33             }
34             catch (InputMismatchException inputMismatchException)
35             {
36                 System.err.printf("\nException: %s\n",
37                                 inputMismatchException);
38                 scanner.nextLine(); // discard input so user can try again
39                 System.out.printf(
40                     "You must enter integers. Please try again.\n\n");
41             }
42             catch (ArithmeticException arithmeticException)
43             {
44                 System.err.printf("\nException: %s\n", arithmeticException);
45                 System.out.printf(
46                     "Zero is an invalid denominator. Please try again.\n\n");
47             }
48         } while (continueLoop);
49     }
50 } // end class DivideByZeroWithExceptionHandling

```

---

**Fig. 11.3** | Handling ArithmeticExceptions and InputMismatchExceptions. (Part 1 of 2.)

```
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. (Part 2 of 2.)

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 18 creates the boolean variable `continueLoop`, which is `true` if the user has *not* yet entered valid input. Lines 20–48 repeatedly ask users for input until a *valid* input is received.

### *Enclosing Code in a try Block*

Lines 22–33 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 25 and 27) 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 29) invokes the code that attempts the division (line 12); the JVM *throws* an `ArithmeticException` object when the denominator is zero.



### Software Engineering Observation 11.1

*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 34–41) and one that handles an `ArithmeticException` (lines 42–47). 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. [Note: The term “catch clause” is sometimes used to refer to the keyword `catch` followed by a block of code, whereas the term “catch block” refers to only the block of code following the `catch` keyword, but not including it. For simplicity, we use the term “catch block” to refer to the block of code following the `catch` keyword, as well as the keyword itself.]

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 37 and 44), 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 38 of 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 (introduced in Java SE 7) 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.

**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 25 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 48 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 32, 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 9–13), 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 the online API documentation for a method before using it in a program. The documentation specifies the 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 12 executes, if the denominator is zero, the JVM throws an `ArithmeticException` object. This object will be caught by the `catch` block at lines 42–47, 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 29). Lines 30–31 display the result of the calculation and line 32 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 48, 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 30–32 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.