Chapter 11
Exception Handling: A Deeper Look

Java How to Program, 11/e, Global Edition

Questions? E-mail paul.deitel@deitel.com

OBJECTIVES

In this chapter you'll:

- Learn why exception handling is an effective mechanism for responding to runtime problems.
- Use `try` blocks to delimit code in which exceptions might occur.
- Use `throw` to indicate a problem.
- Use `catch` blocks to specify exception handlers.
- Learn when to use exception handling.

OBJECTIVES (cont.)

- Understand the exception class hierarchy.
- Use the `finally` block to release resources.
- Chain exceptions by catching one exception and throwing another.
- Create user-defined exceptions.
- Use the debugging feature `assert` to state conditions that should be true at a particular point in a method.
- Learn how `try`-with-resources can automatically release a resource when the `try` block terminates.

OUTLINE

OUTLINE (cont.)

Software Engineering Observation 11.1

In industry, you're likely to find that companies have strict design, coding, testing, debugging and maintenance standards. These often vary among companies. Companies often have their own exception-handling standards that are sensitive to the type of application, such as real-time systems, high-performance mathematical calculations, big data, network-based distributed systems, etc. This chapter's tips provide observations consistent with these types of industry policies.

Chapter	Sample of exceptions used
Chapter 7	`ArrayIndexOutOfBoundsException`
Chapters 8–10	`IllegalArgumentException`
Chapter 11	`ArithmeticException`, `InputMismatchException`
Chapter 15	`SecurityException`, `FileNotFoundException`, `IOException`, `ClassNotFoundException`, `IllegalStateException`, `FormatterClosedException`, `NoSuchElementException`
Chapter 16	`ClassCastException`, `UnsupportedOperationException`, `NullPointerException`, custom exception types
Chapter 20	`ClassCastException`, custom exception types

Fig. 11.1 | Various exception types that you'll see throughout this book

Chapter	Sample of exceptions used
Chapter 21	`IllegalArgumentException`, custom exception types
Chapter 23	`InterruptedException`, `IllegalMonitorStateException`, `ExecutionException`, `CancellationException`
Chapter 24	`SQLException`, `IllegalStateException`, `PatternSyntaxException`
Chapter 28	`MalformedURLException`, `EOFException`, `SocketException`, `InterruptedException`, `UnknownHostException`
Chapter 31	`SQLException`

Fig. 11.1 | Various exception types that you'll see throughout this book

```
1  // Fig. 11.2: DivideByZeroNoExceptionHandling.java
2  // Integer division without exception handling.
3  import java.util.Scanner;
4
5  public class DivideByZeroNoExceptionHandling {
6     // demonstrates throwing an exception when a divide-by-zero occurs
7     public static int quotient(int numerator, int denominator) {
8        return numerator / denominator; // possible division by zero
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 numerator: ");
15       int numerator = scanner.nextInt();
```

Fig. 11.2 | Integer division without exception handling. (Part 1 of 3.)

```
16          System.out.print("Please enter an integer denominator: ");
17          int denominator = scanner.nextInt();
18
19          int result = quotient(numerator, denominator);
20          System.out.printf(
21             "%nResult: %d / %d = %d%n", numerator, denominator, result);
22       }
23    }
```

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

Result: 100 / 7 = 14
```

Fig. 11.2 | Integer division without exception handling. (Part 2 of 3.)

```
Please enter an integer numerator: 100
Please enter an integer denominator: 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        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.InputMismatchException
        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. (Part 3 of 3.)

```
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       return numerator / denominator; // possible division by zero
12    }
13
```

Fig. 11.3 | Handling `ArithmeticExceptions` and `InputMismatchExceptions`. (Part 1 of 5.)

```
14      public static void main(String[] args) {
15         Scanner scanner = new Scanner(System.in);
16         boolean continueLoop = true; // determines if more input is needed
17
18         do {
19            try { // read two numbers and calculate quotient
20               System.out.print("Please enter an integer numerator: ");
21               int numerator = scanner.nextInt();
22               System.out.print("Please enter an integer denominator: ");
23               int denominator = scanner.nextInt();
24
25               int result = quotient(numerator, denominator);
26               System.out.printf("%nResult: %d / %d = %d%n", numerator,
27                  denominator, result);
28               continueLoop = false; // input successful; end looping
29            }
```

Fig. 11.3 | Handling `ArithmeticExceptions` and `InputMismatchExceptions`. (Part 2 of 5.)

```
30            catch (InputMismatchException inputMismatchException) {
31               System.err.printf("%nException: %s%n",
32                  inputMismatchException);
33               scanner.nextLine(); // discard input so user can try again
34               System.out.printf(
35                  "You must enter integers. Please try again.%n%n");
36            }
37            catch (ArithmeticException arithmeticException) {
38               System.err.printf("%nException: %s%n", arithmeticException);
39               System.out.printf(
40                  "Zero is an invalid denominator. Please try again.%n%n");
41            }
42         } while (continueLoop);
43      }
44   }
```

Fig. 11.3 | Handling `ArithmeticExceptions` and `InputMismatchExceptions`. (Part 3 of 5.)

```
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
```

Fig. 11.3 | Handling `ArithmeticExceptions` and `InputMismatchExceptions`. (Part 4 of 5.)

```
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 5 of 5.)

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.

Common Programming Error 11.1

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

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.

Software Engineering Observation 11.3

Incorporate your exception-handling and error-recovery strategy into your system from the inception of the design process—including these after a system has been implemented can be difficult.

Software Engineering Observation 11.4

Exception handling provides a single, uniform technique for documenting, detecting and recovering from errors. This helps programmers working on large projects understand each other's error-processing code.

Software Engineering Observation 11.5

A great variety of situations can generate exceptions—some exceptions are easier to recover from than others.

Software Engineering Observation 11.6

Sometimes you can prevent an exception by validating data first. For example, before you perform integer division, you can ensure that the denominator is not zero, which prevents the `ArithmeticException` that occurs when you divide by zero.

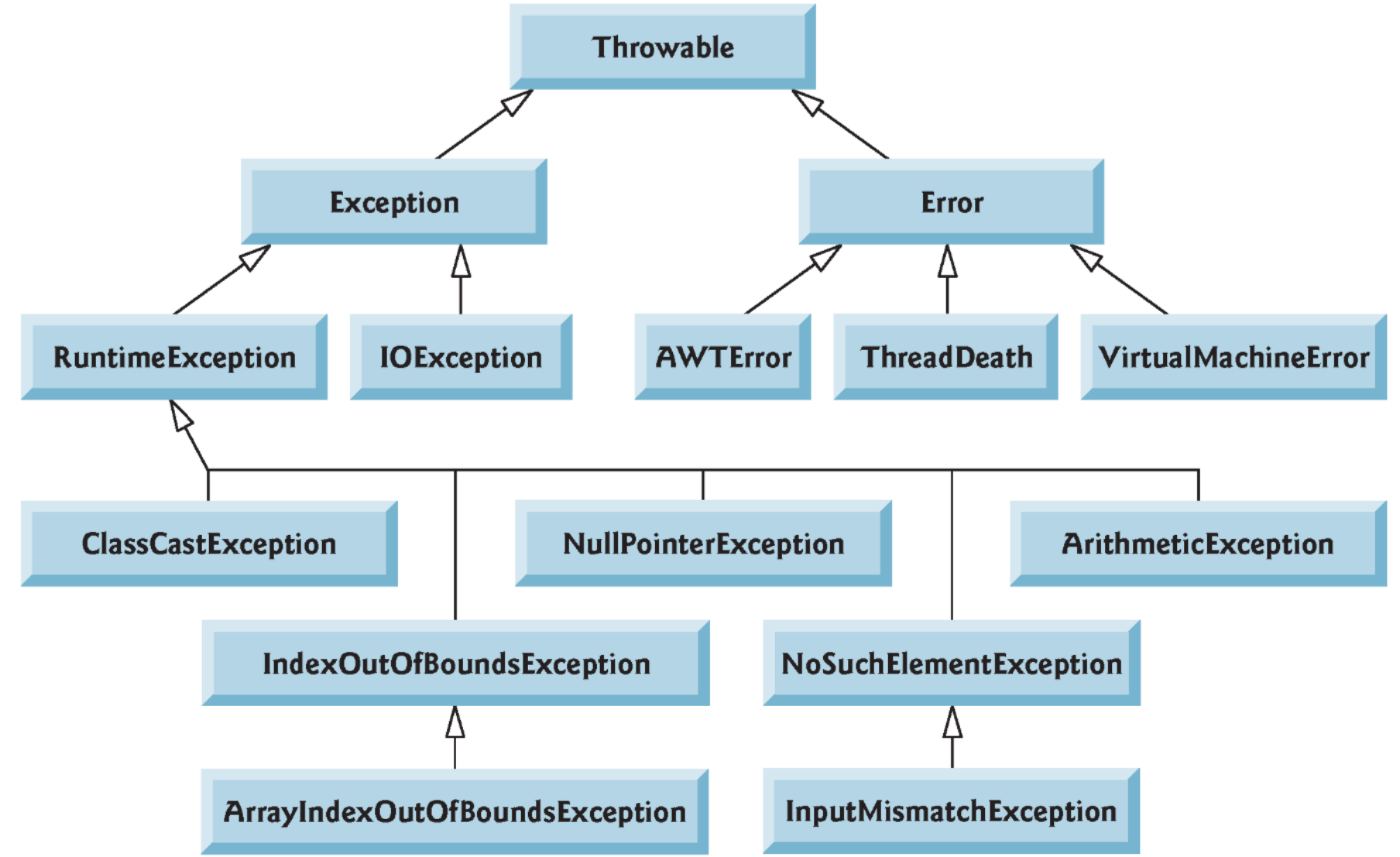

Fig. 11.4 | Portion of class `Throwable`'s inheritance hierarchy.

Error-Prevention Tip 11.2

You must deal with checked exceptions. This results in more robust code than would be created if you were able to simply ignore them.

Common Programming Error 11.2

If a subclass method overrides a superclass method, it's an error for the subclass method to list more exceptions in its `throws` clause than the superclass method does. However, a subclass's `throws` clause can contain a subset of a superclass's `throws` clause.

Software Engineering Observation 11.7

If your method calls other methods that throw checked exceptions, those exceptions must be caught or declared. If an exception can be handled meaningfully in a method, the method should catch the exception rather than declare it.

Software Engineering Observation 11.8

Checked exceptions represent problems from which programs often can recover, so programmers are required to deal with them.

Software Engineering Observation 11.9

Although the compiler does not enforce the `catch`-or-declare requirement for unchecked exceptions, provide appropriate exception-handling code when it's known that such exceptions might occur. For example, a program should process the `NumberFormatException` from `Integer` method `parseInt`, even though `NumberFormatException` is an indirect subclass of `RuntimeException` (and thus an unchecked exception). This makes your programs more robust.

Common Programming Error 11.3

Placing a `catch` block for a superclass exception type before other `catch` blocks that catch subclass exception types would prevent those `catch` blocks from executing, so a compilation error occurs.

Error-Prevention Tip 11.3

Catching subclass types individually is subject to error if you forget to test for one or more of the subclass types explicitly; catching the superclass guarantees that objects of all subclasses will be caught. Positioning a `catch` block for the superclass type after all other subclass `catch` blocks ensures that all subclass exceptions are eventually caught.

Software Engineering Observation 11.10

In industry, throwing or catching type `Exception` is discouraged—we use it in this chapter simply to demonstrate exception-handling mechanics. In subsequent chapters, we generally throw and catch more specific exception types.

Error-Prevention Tip 11.4

A subtle issue is that Java does not entirely eliminate memory leaks. Java will not garbage-collect an object until there are no remaining references to it. Thus, if you erroneously keep references to unwanted objects, memory leaks can occur.

Error-Prevention Tip 11.5

The `finally` block is an ideal place to release resources acquired in a `try` block (such as opened files), which helps eliminate resource leaks.

Performance Tip 11.1

Always release a resource explicitly and at the earliest possible moment at which it's no longer needed. This makes resources available for reuse as early as possible, thus improving resource utilization and program performance.

```
1  // Fig. 11.5: UsingExceptions.java
2  // try...catch...finally exception handling mechanism.
3
4  public class UsingExceptions {
5     public static void main(String[] args) {
6        try {
7           throwException();
8        }
9        catch (Exception exception) { // exception thrown by throwException
10          System.err.println("Exception handled in main");
11       }
12
13       doesNotThrowException();
14    }
15
```

Fig. 11.5 | `try...catch...finally` exception-handling mechanism. (Part 1 of 4.)

```
16      // demonstrate try...catch...finally
17      public static void throwException() throws Exception {
18         try { // throw an exception and immediately catch it
19            System.out.println("Method throwException");
20            throw new Exception(); // generate exception
21         }
22         catch (Exception exception) { // catch exception thrown in try
23            System.err.println(
24               "Exception handled in method throwException");
25            throw exception; // rethrow for further processing
26
27            // code here would not be reached; would cause compilation errors
28
29         }
30         finally { // executes regardless of what occurs in try...catch
31            System.err.println("Finally executed in throwException");
32         }
33
34         // code here would not be reached; would cause compilation errors
35      }
```

Fig. 11.5 | `try...catch...finally` exception-handling mechanism. (Part 2 of 4.)

```
36
37      // demonstrate finally when no exception occurs
38      public static void doesNotThrowException() {
39         try { // try block does not throw an exception
40            System.out.println("Method doesNotThrowException");
41         }
42         catch (Exception exception) { // does not execute
43            System.err.println(exception);
44         }
45         finally { // executes regardless of what occurs in try...catch
46            System.err.println("Finally executed in doesNotThrowException");
47         }
48
49         System.out.println("End of method doesNotThrowException");
50      }
51   }
```

Fig. 11.5 | `try...catch...finally` exception-handling mechanism. (Part 3 of 4.)

```
Method throwException
Exception handled in method throwException
Finally executed in throwException
Exception handled in main
Method doesNotThrowException
Finally executed in doesNotThrowException
End of method doesNotThrowException
```

Fig. 11.5 | `try...catch...finally` exception-handling mechanism. (Part 4 of 4.)

Software Engineering Observation 11.11

When `toString` is invoked on any `Throwable` object, its resulting `String` includes the descriptive `string` that was supplied to the constructor, or simply the class name if no `string` was supplied.

Software Engineering Observation 11.12

An exception can be thrown without containing information about the problem that occurred. In this case, simply knowing that an exception of a particular type occurred may provide sufficient information for the handler to process the problem correctly.

Software Engineering Observation 11.13

Throw exceptions from constructors to indicate that the constructor parameters are not valid—this prevents an object from being created in an invalid state.

Common Programming Error 11.4

If an exception has not been caught when control enters a `finally` block and the `finally` block throws an exception that's not caught in the `finally` block, the first exception will be lost and the exception from the `finally` block will be returned to the calling method.

Error-Prevention Tip 11.6

Avoid placing in a `finally` block code that can `throw` an exception. If such code is required, enclose the code in a `try...catch` within the `finally` block.

Common Programming Error 11.5

Assuming that an exception thrown from a `catch` block will be processed by that `catch` block or any other `catch` block associated with the same `try` statement can lead to logic errors.

Good Programming Practice 11.1

Exception handling removes error-processing code from the main line of a program's code to improve program clarity. Do not place `try...catch... finally` around every statement that may throw an exception. This decreases readability. Rather, place one `try` block around a significant portion of your code, follow the `try` with `catch` blocks that handle each possible exception and follow the `catch` blocks with a single `finally` block (if one is required).

```
1  // Fig. 11.6: UsingExceptions.java
2  // Stack unwinding and obtaining data from an exception object.
3
4  public class UsingExceptions {
5     public static void main(String[] args) {
6        try {
7           method1();
8        }
9        catch (Exception exception) { // catch exception thrown in method1
10          System.err.printf("%s%n%n", exception.getMessage());
11          exception.printStackTrace();
12
13          // obtain the stack-trace information
14          StackTraceElement[] traceElements = exception.getStackTrace();
15
```

Fig. 11.6 | Stack unwinding and obtaining data from an exception object. (Part 1 of 4.)

```
16            System.out.printf("%nStack trace from getStackTrace:%n");
17            System.out.println("Class\t\tFile\t\t\tLine\tMethod");
18
19            // loop through traceElements to get exception description
20            for (StackTraceElement element : traceElements) {
21               System.out.printf("%s\t", element.getClassName());
22               System.out.printf("%s\t", element.getFileName());
23               System.out.printf("%s\t", element.getLineNumber());
24               System.out.printf("%s%n", element.getMethodName());
25            }
26         }
27      }
28
```

Fig. 11.6 | Stack unwinding and obtaining data from an exception object. (Part 2 of 4.)

```
29      // call method2; throw exceptions back to main
30      public static void method1() throws Exception {
31         method2();
32      }
33
34      // call method3; throw exceptions back to method1
35      public static void method2() throws Exception {
36         method3();
37      }
38
39      // throw Exception back to method2
40      public static void method3() throws Exception {
41         throw new Exception("Exception thrown in method3");
42      }
43   }
```

Fig. 11.6 | Stack unwinding and obtaining data from an exception object. (Part 3 of 4.)

```
Exception thrown in method3

java.lang.Exception: Exception thrown in method3
        at UsingExceptions.method3(UsingExceptions.java:41)
        at UsingExceptions.method2(UsingExceptions.java:36)
        at UsingExceptions.method1(UsingExceptions.java:31)
        at UsingExceptions.main(UsingExceptions.java:7)

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. (Part 4 of 4.)

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.

```
1  // Fig. 11.7: UsingChainedExceptions.java
2  // Chained exceptions.
3
4  public class UsingChainedExceptions {
5     public static void main(String[] args) {
6        try {
7           method1();
8        }
9        catch (Exception exception) { // exceptions thrown from method1
10          exception.printStackTrace();
11       }
12    }
13
```

Fig. 11.7 | Chained exceptions. (Part 1 of 3.)

```
14      // call method2; throw exceptions back to main
15      public static void method1() throws Exception {
16         try {
17            method2();
18         }
19         catch (Exception exception) { // exception thrown from method2
20            throw new Exception("Exception thrown in method1", exception);
21         }
22      }
23
24      // call method3; throw exceptions back to method1
25      public static void method2() throws Exception {
26         try {
27            method3();
28         }
29         catch (Exception exception) { // exception thrown from method3
30            throw new Exception("Exception thrown in method2", exception);
31         }
32      }
```

Fig. 11.7 | Chained exceptions. (Part 2 of 3.)

```
33
34      // throw Exception back to method2
35      public static void method3() throws Exception {
36         throw new Exception("Exception thrown in method3");
37      }
38   }
```

```
java.lang.Exception: Exception thrown in method1
        at UsingChainedExceptions.method1(UsingChainedExceptions.java:17)
        at UsingChainedExceptions.main(UsingChainedExceptions.java:7)
Caused by: java.lang.Exception: Exception thrown in method2
        at UsingChainedExceptions.method2(UsingChainedExceptions.java:27)
        at UsingChainedExceptions.method1(UsingChainedExceptions.java:17)
        ... 1 more
Caused by: java.lang.Exception: Exception thrown in method3
        at UsingChainedExceptions.method3(UsingChainedExceptions.java:36)
        at UsingChainedExceptions.method2(UsingChainedExceptions.java:27)
        ... 2 more
```

Fig. 11.7 | Chained exceptions. (Part 3 of 3.)

Good Programming Practice 11.2

Associating each type of serious execution-time malfunction with an appropriately named `Exception` class improves program clarity.

Software Engineering Observation 11.15

Most programmers will not need to declare their own exception classes. Before defining your own, study the existing ones in the Java API and try to choose one that already exists. If there is not an appropriate existing class, try to extend a related exception class. For example, if you're creating a new class to represent when a method attempts a division by zero, you might extend class `ArithmeticException` because division by zero occurs during arithmetic. If the existing classes are not appropriate superclasses for your new exception class, decide whether your new class should be a checked or an unchecked exception class. If clients should be required to handle the exception, the new exception class should be a checked exception (i.e., extend `Exception` but not `RuntimeException`). The client application should be able to reasonably recover from such an exception. If the client code should be able to ignore the exception (i.e., the exception is an unchecked one), the new exception class should extend `RuntimeException`.

Good Programming Practice 11.3

By convention, all exception-class names should end with the word `Exception`.

```java
// Fig. 11.8: AssertTest.java
// Checking with assert that a value is within range
import java.util.Scanner;

public class AssertTest {
   public static void main(String[] args) {
      Scanner input = new Scanner(System.in);

      System.out.print("Enter a number between 0 and 10: ");
      int number = input.nextInt();

      // assert that the value is >= 0 and <= 10
      assert (number >= 0 && number <= 10) : "bad number: " + number;

      System.out.printf("You entered %d%n", number);
   }
}
```

Fig. 11.8 | Checking with `assert` that a value is within range. (Part 1 of 2.)

```
Enter a number between 0 and 10: 5
You entered 5
```

```
Enter a number between 0 and 10: 50
Exception in thread "main" java.lang.AssertionError: bad number: 50
        at AssertTest.main(AssertTest.java:13)
```

Fig. 11.8 | Checking with `assert` that a value is within range. (Part 2 of 2.)

Software Engineering Observation 11.16

Users shouldn't encounter `AssertionErrors`—these should be used only during program development. For this reason, you shouldn't catch `AssertionErrors`. Instead, allow the program to terminate, so you can see the error message, then locate and fix the source of the problem. You should not use `assert` to indicate runtime problems in production code (as we did in Fig. 11.8 for demonstration purposes)—use the exception mechanism for this purpose.