Chapter 11
Exception Handling: A Deeper Look

Java™ How to Program, 9/e

OBJECTIVES

In this chapter you'll learn:

- What exceptions are and how they're handled.

- When to use exception handling.

- To use `try` blocks to delimit code in which exceptions might occur.

- To `throw` exceptions to indicate a problem.

- To use `catch` blocks to specify exception handlers.

- To use the `finally` block to release resources.

- The exception class hierarchy.

- To create user-defined exceptions.

11.1 Introduction

- Exception handling
- Exception—an indication of a problem that occurs during a program's execution.
  - The name "exception" implies that the problem occurs infrequently.
- With exception handling, a program can continue executing (rather than terminating) after dealing with a problem.
  - Mission-critical or business-critical computing.
  - Robust and fault-tolerant programs (i.e., programs that can deal with problems as they arise and continue executing).

11.1 Introduction (Cont.)

- `ArrayIndexOutOfBoundsException` occurs when an attempt is made to access an element past either end of an array.

- `ClassCastException` occurs when an attempt is made to cast an object that does not have an *is-a* relationship with the type specified in the cast operator.

- A `NullPointerException` occurs when a `null` reference is used where an object is expected.

- Only classes that extend `Throwable` (package `java.lang`) directly or indirectly can be used with exception handling.

11.2 Example: Divide by Zero without Exception Handling

- Exceptions are thrown (i.e., the exception occurs) when a method detects a problem and is unable to handle it.
- Stack trace—information displayed when an exception occurs and is not handled.
- Information includes:
  - The name of the exception in a descriptive message that indicates the problem that occurred
  - The method-call stack (i.e., the call chain) at the time it occurred. Represents the path of execution that led to the exception method by method.
- This information helps you debug the program.

11.2 Example: Divide by Zero without Exception Handling (Cont.)

- Java does not allow division by zero in integer arithmetic.
  - Throws an ArithmeticException.
  - Can arise from a several problems, so an error message (e.g., "/ 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
  - Floating-point value that displays as Infinity or -Infinity.
  - If 0.0 is divided by 0.0, the result is NaN (not a number), which is represented as a floating-point value that displays as NaN.

```
1    // Fig. 11.1: DivideByZeroNoExceptionHandling.java
2    // Integer division without exception handling.
3    import java.util.Scanner;
4
5    public class DivideByZeroNoExceptionHandling
6    {
7       // demonstrates throwing an exception when a divide-by-zero occurs
8       public static int quotient( int numerator, int denominator )
9       {
10          return numerator / denominator; // possible division by zero
11       } // end method quotient
12
```

Fig. 11.1 | Integer division without exception handling. (Part 1 of 3.)

```
13      public static void main( String[] args )
14      {
15         Scanner scanner = new Scanner( System.in ); // scanner for input
16
17         System.out.print( "Please enter an integer numerator: " );
18         int numerator = scanner.nextInt();
19         System.out.print( "Please enter an integer denominator: " );
20         int denominator = scanner.nextInt();
21
22         int result = quotient( numerator, denominator );
23         System.out.printf(
24            "\nResult: %d / %d = %d\n", numerator, denominator, result );
25      } // end main
26   } // end class DivideByZeroNoExceptionHandling
```

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

Result: 100 / 7 = 14
```

Fig. 11.1 | 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:10)
        at DivideByZeroNoExceptionHandling.main(
            DivideByZeroNoExceptionHandling.java:22)
```

```
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:20)
```

Fig. 11.1 | Integer division without exception handling. (Part 3 of 3.)

11.2 Example: Divide by Zero without Exception Handling (Cont.)

- Last "at" line in the stack trace started the call chain.
- Each line contains the class name and method followed by the file name and line number.
- The top "at" line of the call chain indicates the throw point—the initial point at which the exception occurs.
- As you read a stack trace top to bottom, the first "at" line that contains your class name and method name is typically the point in the program that led to the exception.

11.2 Example: Divide by Zero without Exception Handling (Cont.)

- Prior examples that read numeric values from the user assumed that the user would input a proper integer value.

- 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.

- If a stack trace contains "`Unknown Source`" for a particular method, the debugging symbols for that method's class were not available to the JVM—this is typically the case for the classes of the Java API.

11.3 Example: Handling `ArithmeticExceptions` and `InputMismatchExceptions`

- The application in Fig. 11.2 uses exception handling to process any `ArithmeticException`s and `InputMismatchException`s that arise.

- If the user makes a mistake, the program catches and handles (i.e., deals with) the exception—in this case, allowing the user to try to enter the input again.

```
 1    // Fig. 11.2: 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       } // end method quotient
14
15       public static void main( String[] args )
16       {
17          Scanner scanner = new Scanner( System.in ); // scanner for input
18          boolean continueLoop = true; // determines if more input is needed
19
```

Fig. 11.2 | Handling ArithmeticExceptions and InputMismatchExceptions.
(Part 1 of 4.)

```
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       } // end try
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.println(
40             "You must enter integers. Please try again.\n" );
41       } // end catch
```

Fig. 11.2 | Handling ArithmeticExceptions and InputMismatchExceptions.
(Part 2 of 4.)

```
42              catch ( ArithmeticException arithmeticException )
43              {
44                 System.err.printf( "\nException: %s\n", arithmeticException );
45                 System.out.println(
46                    "Zero is an invalid denominator. Please try again.\n" );
47              } // end catch
48          } while ( continueLoop ); // end do...while
49       } // end main
50   } // end class DivideByZeroWithExceptionHandling
```

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

Result: 100 / 7 = 14
```

Fig. 11.2 | Handling ArithmeticExceptions and InputMismatchExceptions.
(Part 3 of 4.)

```
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. 11.2 | Handling ArithmeticExceptions and InputMismatchExceptions.
(Part 4 of 4.)

- try block encloses
  - code that might `throw` an exception
  - code that should not execute if an exception occurs.
- Consists of the keyword `try` followed by a block of code enclosed in curly braces.

Software Engineering Observation 11.1

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

11.3 Example: Handling `ArithmeticExceptions` and `InputMismatchExceptions` (Cont.)

- catch block (also called a catch clause or exception handler) catches and handles an exception.
  - Begins with the keyword `catch` and is followed by an exception parameter in parentheses and a block of code enclosed in curly braces.
- At least one `catch` block or a finally block (Section 11.6) must immediately follow the `try` block.
- The exception parameter identifies the exception type the handler can process.
  - The parameter's name enables the `catch` block to interact with a caught exception object.

- 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.
- Use the System.err (standard error stream) object to output error messages.
  - By default, displays data to the command prompt.

Common Programming Error 11.1

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

Common Programming Error 11.2
Each `catch` block can have only a single parameter—specifying a comma-separated list of exception parameters is a syntax error.

11.3 Example: Handling `ArithmeticExceptions` and `InputMismatchExceptions` (Cont.)

- Uncaught exception—one for which there are no matching `catch` blocks.
- Recall that previous uncaught exceptions caused the application to terminate early.
  - This does not always occur as a result of uncaught exceptions.
- Java uses a multithreaded model of program execution.
  - Each thread is a parallel 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 where the exception occurred.

- If an exception occurs in a `try` block, the `try` block terminates immediately and program control transfers to the first matching `catch` block.

- After the exception is handled, control resumes after the last `catch` block.

- 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.

Good Programming Practice 11.1

Using an exception-parameter name that reflects the parameter's type promotes clarity by reminding you of the type of exception being handled.

- If no exceptions are thrown in a `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.
- The `try` block and its corresponding `catch` and/or `finally` blocks form a try statement.

11.3 Example: Handling ArithmeticExceptions and InputMismatchExceptions (Cont.)

- When a `try` block terminates, local variables declared in the block go out of scope.
  - 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) also go out of scope.
- Any remaining `catch` blocks in the `try` statement are ignored, and execution resumes at the first line of code after the `try…catch` sequence.
  - A `finally` block, if one is present.

- **throws clause**—specifies the exceptions a method throws.
  - Appears after the method's parameter list and before the method's body.
  - Contains a comma-separated list of the exceptions that the method will throw if various problems occur.
    - May be thrown by statements in the method's body or by methods called from the body.
  - Method can throw exceptions of the classes listed in its `throws` clause or of their subclasses.
  - Clients of a method with a `throws` clause are thus informed that the method may throw exceptions.

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 a method throws an exception, the method terminates and does not return a value, and its local variables go out of scope.
  - If the local variables were references to objects and there were no other references to those objects, the objects would be available for garbage collection.

11.4 When to Use Exception Handling

- Exception handling is designed to process synchronous errors, which occur when a statement executes.
- Common examples in this book:
  - out-of-range array indices
  - arithmetic overflow
  - division by zero
  - invalid method parameters
  - thread interruption
  - unsuccessful memory allocation

11.4 When to Use Exception Handling (Cont.)

- Exception handling is not designed to process problems associated with asynchronous events
  - disk I/O completions
  - network message arrivals
  - mouse clicks and keystrokes

Software Engineering Observation 11.2

Incorporate your exception-handling strategy into your system from the inception of the design process. Including exception handling after a system has been implemented can be difficult.

Software Engineering Observation 11.3

Exception handling provides a single, uniform technique for processing problems. This helps programmers working on large projects understand each other's error-processing code.

11.5 Java Exception Hierarchy

- Exception classes inherit directly or indirectly from class Exception, forming an inheritance hierarchy.
  - Can extend this hierarchy with your own exception classes.
- Figure 11.3 shows a small portion of the inheritance hierarchy for class Throwable (a subclass of `Object`), which is the superclass of class `Exception`.
  - Only `Throwable` objects can be used with the exception-handling mechanism.
- Class `Throwable` has two subclasses: `Exception` and `Error`.

11.5 Java Exception Hierarchy (Cont.)

- Class `Exception` and its subclasses represent exceptional situations that can occur in a Java program
  - These can be caught and handled by the application.
- Class Error and its subclasses represent abnormal situations that happen in the JVM.
  - `Error`s happen infrequently.
  - These should not be caught by applications.
  - Applications usually cannot recover from `Error`s.
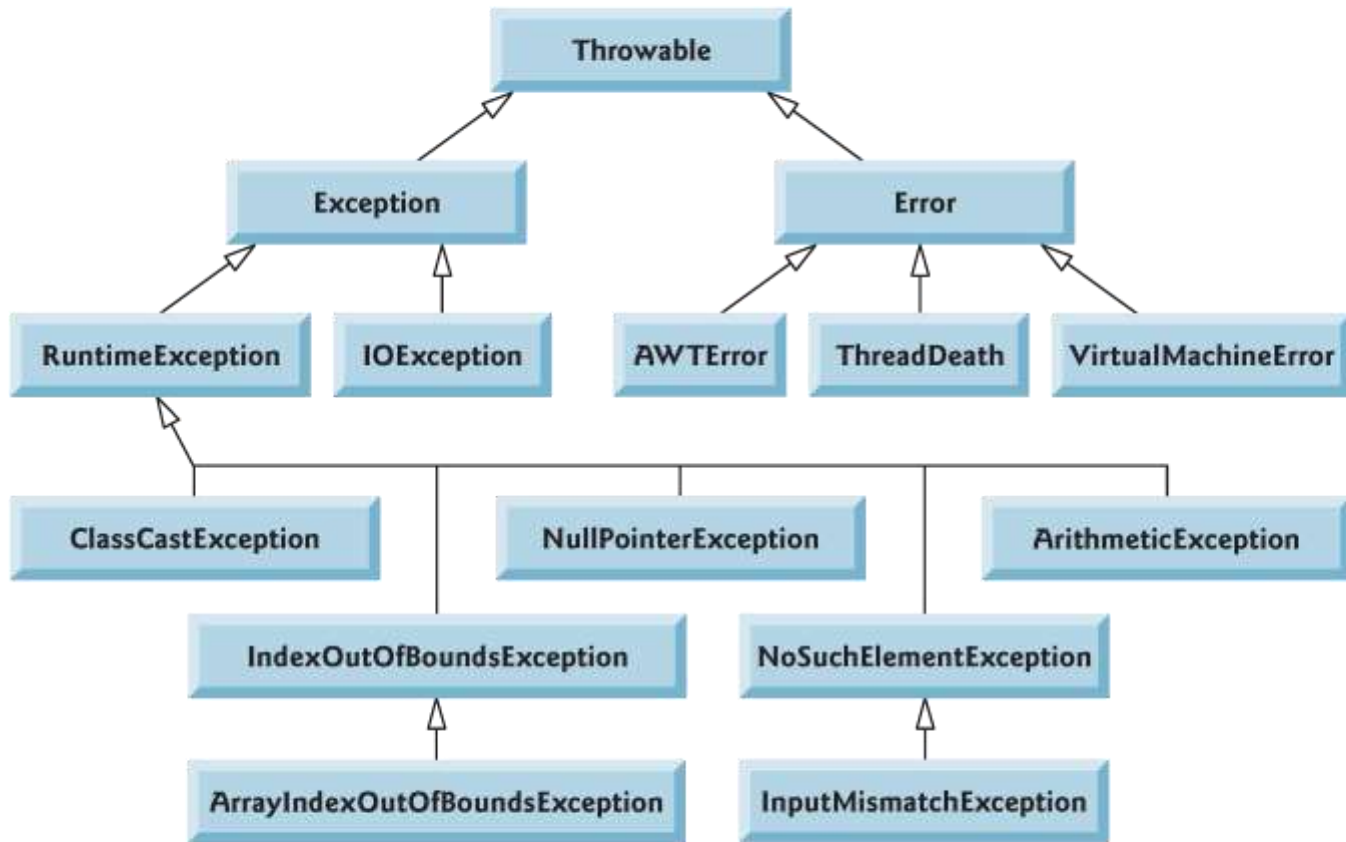
Fig. 11.3 | Portion of class `Throwable`'s inheritance hierarchy.

11.5 Java Exception Hierarchy (Cont.)

- Checked exceptions vs. unchecked exceptions.
  - Compiler enforces a catch-or-declare requirement for checked exceptions.
- An exception's type determines whether it is checked or unchecked.
- Direct or indirect subclasses of class RuntimeException (package `java.lang`) are *unchecked* exceptions.
  - Typically caused by defects in your program's code (e.g., `ArrayIndexOutOfBoundsException`s).
- Subclasses of `Exception` but not `RuntimeException` are *checked* exceptions.
  - Caused by conditions that are not in the control of the program—e.g., in file processing, the program can't open a file because the file does not exist.

11.5 Java Exception Hierarchy (Cont.)

- Classes that inherit from class `Error` are considered to be *unchecked.*
- The compiler *checks* each method call and method declaration to determine whether the method throws checked exceptions.
  - If so, the compiler verifies that the checked exception is caught or is declared in a `throws` clause.
- `throws` clause specifies the exceptions a method throws.
  - Such exceptions are typically not caught in the method's body.

11.5 Java Exception Hierarchy (Cont.)

- To satisfy the *catch* part of the *catch-or-declare requirement*, the code that generates the exception must be wrapped in a `try` block and must provide a `catch` handler for the checked-exception type (or one of its superclasses).

- To satisfy the *declare* part of the *catch-or-declare requirement*, the method must provide a `throws` clause containing the checked-exception type after its parameter list and before its method body.

- If the catch-or-declare requirement is not satisfied, the compiler will issue an error message indicating that the exception must be caught or declared.

Software Engineering Observation 11.4

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

Common Programming Error 11.3

A compilation error occurs if a method explicitly attempts to throw a checked exception (or calls another method that throws a checked exception) and that exception is not listed in that method's **throws** clause.

Common Programming Error 11.4

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 overridden superclass method does. However, a subclass's **throws** clause can contain a subset of a superclass's **throws** list.

Software Engineering Observation 11.5

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

11.5 Java Exception Hierarchy (Cont.)

- The compiler does not check the code to determine whether an unchecked exception is caught or declared.
  - These typically can be prevented by proper coding.
  - For example, an `ArithmeticException` can be avoided if a method ensures that the denominator is not zero before attempting to perform the division.
- Unchecked exceptions are not required to be listed in a method's `throws` clause.
  - Even if they are, it's not required that such exceptions be caught by an application.

Software Engineering Observation 11.6

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 (an indirect subclass of RuntimeException) is an unchecked exception type. This makes your programs more robust.

11.5 Java Exception Hierarchy (Cont.)

▸ A `catch` parameter of a superclass-type can also catch all of that exception type's subclass types.

- Enables `catch` to handle related er-rors with a concise notation
- Allows for polymorphic processing of related exceptions
- Catching related exceptions in one `catch` block makes sense only if the handling behavior is the same for all subclasses.

▸ You can also catch each subclass type individually if those exceptions require different processing.

11.5 Java Exception Hierarchy (Cont.)

- If there multiple `catch` blocks match a particular exception type, only the first matching `catch` block executes.

- It's a compilation error to catch the exact same type in two different `catch` blocks associated with a particular `try` block.

Error-Prevention Tip 11.2

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.*

Common Programming Error 11.5

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.*

11.6 finally Block

- Programs that obtain resources must return them to the system explicitly to avoid so-called resource leaks.
  - In programming languages such as C and C++, the most common kind of resource leak is a memory leak.
  - Java automatically garbage collects memory no longer used by programs, thus avoiding most memory leaks.
  - Other types of resource leaks can occur.
    - Files, database connections and network connections that are not closed properly might not be available for use in other programs.
- The finally block is used for resource deallocation.
  - Placed after the last catch block.

Error-Prevention Tip 11.3

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. To help avoid this problem, set reference-type variables to `null` *when they're no longer needed.*

11.6 finally Block (Cont.)

- finally block will execute whether or not an exception is thrown in the corresponding try block.
- finally block will execute if a try block exits by using a return, break or continue statement or simply by reaching its closing right brace.
- finally block will *not* execute if the application terminates immediately by calling method System.exit.

11.6 finally Block (Cont.)

- Because a `finally` block almost always executes, it typically contains resource-release code.
- Suppose a resource is allocated in a `try` block.
  - If no exception occurs, control proceeds to the `finally` block, which frees the resource. Control then proceeds to the first statement after the `finally` block.
  - If an exception occurs, the `try` block terminates. The program catches and processes the exception in one of the corresponding `catch` blocks, then the `finally` block releases the resource and control proceeds to the first statement after the `finally` block.
  - If the program doesn't catch the exception, the `finally` block still releases the resource and an attempt is made to catch the exception in a calling method.

Error-Prevention Tip 11.4
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.

11.6 finally Block (Cont.)

- If an exception that occurs in a `try` block cannot be caught by one of that `try` block's `catch` handlers, control proceeds to the `finally` block.

- Then the program passes the exception to the next outer `try` block—normally in the calling method—where an associated `catch` block might catch it.
  - This process can occur through many levels of `try` blocks.
  - The exception could go uncaught.

- If a `catch` block throws an exception, the `finally` block still executes.
  - Then the exception is passed to the next outer `try` block—again, normally in the calling method.

```java
1   // Fig. 11.4: UsingExceptions.java
2   // try...catch...finally exception handling mechanism.
3
4   public class UsingExceptions
5   {
6      public static void main( String[] args )
7      {
8         try
9         {
10            throwException(); // call method throwException
11         } // end try
12         catch ( Exception exception ) // exception thrown by throwException
13         {
14            System.err.println( "Exception handled in main" );
15         } // end catch
16
17         doesNotThrowException();
18      } // end main
19
20      // demonstrate try...catch...finally
21      public static void throwException() throws Exception
22      {
```

Fig. 11.4 | try...catch...finally exception-handling mechanism. (Part 1 of 4.)

```
23        try // throw an exception and immediately catch it
24        {
25           System.out.println( "Method throwException" );
26           throw new Exception(); // generate exception
27        } // end try
28        catch ( Exception exception ) // catch exception thrown in try
29        {
30           System.err.println(
31              "Exception handled in method throwException" );
32           throw exception; // rethrow for further processing
33
34           // code here would not be reached; would cause compilation errors
35
36        } // end catch
37        finally // executes regardless of what occurs in try...catch
38        {
39           System.err.println( "Finally executed in throwException" );
40        } // end finally
41
42        // code here would not be reached; would cause compilation errors
43
44     } // end method throwException
45
```

Fig. 11.4 | try...catch...finally exception-handling mechanism. (Part 2 of 4.)

```
46      // demonstrate finally when no exception occurs
47      public static void doesNotThrowException()
48      {
49         try // try block does not throw an exception
50         {
51            System.out.println( "Method doesNotThrowException" );
52         } // end try
53         catch ( Exception exception ) // does not execute
54         {
55            System.err.println( exception );
56         } // end catch
57         finally // executes regardless of what occurs in try...catch
58         {
59            System.err.println(
60               "Finally executed in doesNotThrowException" );
61         } // end finally
62
63         System.out.println( "End of method doesNotThrowException" );
64      } // end method doesNotThrowException
65   } // end class UsingExceptions
```

Fig. 11.4 | 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. 11.4 | try...catch...finally exception-handling mechanism. (Part 4 of 4.)