

14

Exception Handling

Objectives

- To understand exception and error handling.
- To be able to use **try** blocks to delineate code in which an exception may occur.
- To be able to **throw** exceptions.
- To use **catch** blocks to specify exception handlers.
- To use the **finally** block to release resources.
- To understand the Java exception hierarchy.
- To create programmer-defined exceptions.

It is common sense to take a method and try it. If it fails, admit it frankly and try another. But above all, try something.
Franklin Delano Roosevelt

*O! throw away the worser part of it,
And live the purer with the other half.*
William Shakespeare

*If they're running and they don't look where they're going
I have to come out from somewhere and catch them.*
Jerome David Salinger

*And oftentimes excusing of a fault
Doth make the fault the worse by the excuse.*
William Shakespeare

I never forget a face, but in your case I'll make an exception.
Groucho (Julius Henry) Marx



Outline

- 14.1 Introduction**
- 14.2 When Exception Handling Should Be Used**
- 14.3 Other Error-Handling Techniques**
- 14.4 Basics of Java Exception Handling**
- 14.5 `try` Blocks**
- 14.6 Throwing an Exception**
- 14.7 Catching an Exception**
- 14.8 Exception-Handling Example: Divide by Zero**
- 14.9 Rethrowing an Exception**
- 14.10 `throws` Clause**
- 14.11 Constructors, Finalizers and Exception Handling**
- 14.12 Exceptions and Inheritance**
- 14.13 `finally` Block**
- 14.14 Using `printStackTrace` and `getMessage`**

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

14.1 Introduction

In this chapter, we introduce *exception handling*. An *exception* is an indication that a problem occurred during the program's execution. The extensibility of Java can increase the number and types of errors that can occur. Every new class can add its own error possibilities. The features presented here enable programmers to write clearer, more robust, more fault-tolerant programs. We also consider when exception handling should not be used.

The style and details of exception handling in Java as presented in this chapter are based in part on the work of Andrew Koenig and Bjarne Stroustrup as presented in their paper, "Exception Handling for C++ (revised)," published in the *Proceedings of the USENIX C++ Conference* held in San Francisco in April 1990. Their work forms the basis of C++ exception handling. Java's designers chose to implement an exception handling mechanism similar to that used in C++.

Error-handling code varies in nature and quantity among software systems depending on the application and whether the software is a product for release. Products tend to contain far more error-handling code than does "casual" software.

There are many popular means for dealing with errors. Most commonly, error-handling code is interspersed throughout a system's code. Errors are dealt with at the places in the code where the errors can occur. The advantage to this approach is that a programmer reading code can see the error processing in the immediate vicinity of the code and determine if the proper error checking has been implemented.

The problem with this scheme is that the code can become "polluted" with the error processing. It becomes more difficult for a programmer concerned with the application itself to read the code and determine if it is functioning correctly. This can make understanding and maintaining the application difficult.

Some common examples of exceptions are an out-of-bounds array subscript, arithmetic overflow (i.e., a value outside the representable range of values), division by zero, invalid method parameters and memory exhaustion.



Good Programming Practice 14.1

Using Java exception handling enables the programmer to remove the error-handling code from the “main line” of the program’s execution. This improves program clarity and enhances modifiability.

Exception handling is provided to enable programs to catch and handle errors rather than letting them occur and suffering the consequences. Exception handling is designed for dealing with *synchronous errors* such as an attempt to divide by zero (that occurs as the program executes the divide instruction). Exception handling is not designed to deal with *asynchronous* events such as disk I/O completions, network message arrivals, mouse clicks, keystrokes and the like; these are best handled through other means, such as Java event listeners.

Java exception handling enables a program to catch all exceptions, all exceptions of a certain type or all exceptions of related types. This flexibility makes programs more robust by reducing the likelihood that programs will not process problems during program execution.

Exception handling is used in situations in which the system can recover from the malfunction that caused the exception. The recovery procedure is called an *exception handler*. The exception handler can be defined in the method that may cause an exception or in a calling method.



Software Engineering Observation 14.1

Use exceptions for malfunctions that must be processed in a different method from where they are detected. Use conventional error-handling techniques for local error processing in which a method is able to deal with its own exceptions.

Exception handling is designed to process exceptional conditions—problems that do not happen frequently, but *can* happen. It is possible that the exception-handling code may not be optimized to the same performance levels as other language elements.



Performance Tip 14.1

When an exception does not occur, little or no overhead is imposed by the presence of exception handling code. When exceptions happen, they do incur execution-time overhead.



Testing and Debugging Tip 14.1

Exception handling helps improve a program’s fault tolerance.



Good Programming Practice 14.2

Using Java’s standardized exception handling rather than having programmers use a diversity of “home-grown” techniques improves program clarity on large projects.

We will see that exceptions are objects of classes derived from superclass **Exception**. We will show how to deal with “uncaught” exceptions. We will consider how unexpected exceptions are handled by Java. We will show how related exception types can be represented by exception subclasses that are derived from a common exception superclass.

Exception handling can be viewed as another means of returning control from a method or exiting a block of code. Normally, when an exception occurs, the exception is

handled by a caller of the method generating the exception, by a caller of that caller, or however far back in the call stack it becomes necessary to go to find a handler for that exception.



Software Engineering Observation 14.2

Exception handling is particularly well-suited to systems of separately developed components. Such systems are typical of real-world software. Exception handling makes it easier to combine the components and have them work together effectively.



Software Engineering Observation 14.3

With other programming languages that do not support exception handling, programmers often delay writing error-processing code, and sometimes programmers simply forget to include it. This results in less-robust, and thus inferior, software products. Java forces the programmer to deal with exception handling from the inception of a project. Still, the programmer must put considerable effort into incorporating an exception-handling strategy into software projects.



Software Engineering Observation 14.4

It is best to incorporate your exception-handling strategy into a system from the inception of the design process. It is difficult to add effective exception handling after a system has been implemented.

14.2 When Exception Handling Should Be Used

Exception handling should be used

- to process exceptional situations where a method is unable to complete its task for reasons it cannot control,
- to process exceptions from program components that are not geared to handling those exceptions directly, or
- on large projects to handle exceptions in a uniform manner project wide.



Software Engineering Observation 14.5

The client of a library class will likely have unique error processing in mind for an exception generated in the library class. It is unlikely that a library class will perform error processing that would meet the unique needs of all clients. Exceptions are an appropriate means for dealing with errors produced by library classes.

14.3 Other Error-Handling Techniques

We have presented various ways of dealing with exceptional situations prior to this chapter. A program can ignore some exception types. This can be devastating for software products released to the general public, or for special-purpose software needed for mission-critical situations. But for software developed for your own purposes, it is common to ignore many kinds of errors. A program could be directed to abort upon encountering an exceptional situation. This prevents a program from running to completion and producing incorrect results. For many types of errors this is a good strategy. Such a strategy is inappropriate for mission-critical applications. Resource issues also are important here. If a program obtains a resource, the program should return that resource before program termination.



Common Programming Error 14.1

Aborting a program could leave a resource in a state in which other programs would not be able to acquire the resource; hence, we would have a so-called “resource leak.”



Good Programming Practice 14.3

If your method is capable of handling a given type of exception, then handle it rather than passing the exception on to other regions of your program. This makes programs clearer.



Performance Tip 14.2

If an error can be processed locally instead of throwing an exception, do so. This will improve program execution speed. Exception handling is slow compared to local processing.

14.4 Basics of Java Exception Handling

In this section, we overview the Java exception-handling process. Throughout the chapter, we present detailed discussions of the steps discussed here.

Java exception handling is geared to situations in which the method that detects an error is unable to deal with it. Such a method will *throw an exception*. There is no guarantee that there will be “anything out there” (i.e., an *exception handler*—code that executes when the program detects an exception) to process that kind of exception. If there is, the exception will be *caught* and *handled*. The following *Testing and Debugging Tip* describes what happens if no appropriate exception handler can be found.



Testing and Debugging Tip 14.2

All Java applets and certain Java applications are GUI-based. Some Java applications are not GUI-based; these are often called *command-line applications* (or *console applications*). When an exception is not caught in a command-line application, the program terminates (i.e., Java exits) after the default exception handler runs. When an exception is not caught in an applet or a GUI-based application, the GUI remains displayed and the user can continue using the applet or application even after the default exception handler runs. However, the program may be in an inconsistent state and may produce incorrect results.

The programmer encloses in a **try block** the code that may generate an exception and any code that should not execute if an exception occurs. The **try** block is followed by zero or more **catch blocks**. Each **catch** block specifies the type of exception it can catch and contains an exception handler. After the last **catch** block, an optional **finally** block provides code that always executes, regardless of whether an exception occurs. As we will see, the **finally** block is an ideal location for code that releases resources to prevent “resource leaks.” The **try** block must be followed by a **catch** block or a **finally** block.

When a method throws an exception, program control leaves the **try** block and continues execution at the first **catch** block. The program searches the **catch** blocks in order looking for an appropriate handler. (We will soon discuss what makes a handler “appropriate”.) If the type of the thrown exception matches the parameter type in one of the **catch** blocks, the code for that **catch** block executes. If a **try** block completes successfully without throwing any exceptions, the program skips the exception handlers for that block and resumes execution after the last **catch** block. If a **finally** block appears after the last **catch** block, the **finally** block executes regardless of whether an exception occurs.

In a method definition, a **throws clause** specifies the exceptions the method throws. This clause appears after the parameter list and before the method body. The clause con-

tains a comma-separated list of potential exceptions the method will throw if a problem occurs while the method executes. Such exceptions may be thrown by statements in the method's body, or they may be thrown by methods called in the body. The point at which the **throw** occurs is called the *throw point*.

When an exception occurs, the block in which the exception occurred expires (terminates)—program control cannot return directly to the throw point. Java uses the *termination model of exception handling* rather than the *resumption model of exception handling*. In the resumption model, control would return to the point at which the exception occurred and resume execution.

When an exception occurs, it is possible to communicate information to the exception handler from the vicinity in which the exception occurred. That information is the type of thrown exception object or information harvested from the vicinity in which the exception occurred and placed into the thrown object.

14.5 try Blocks

An exception that occurs in a **try** block normally is caught by an exception handler specified by a **catch** block immediately following that **try** block as in

```
try {
    statements that may throw an exception
}
catch( ExceptionType exceptionReference ) {
    statements to process an exception
}
```

A **try** block can be followed by zero or more **catch** blocks. If a **try** block executes and no exceptions are thrown, all the exception handlers are skipped and control resumes with the first statement after the last exception handler. If a **finally** block (presented in Section 14.13) follows the last **catch** block, the code in the **finally** block executes regardless of whether an exception is thrown. Note that an exception handler cannot access objects defined in the corresponding **try** block, because the **try** block expires before the handler begins executing.



Common Programming Error 14.2

It is a syntax error to separate with other code the **catch** handlers that correspond to a particular **try** block.

14.6 Throwing an Exception

The **throw** statement is executed to indicate that an exception has occurred (i.e., a method could not complete successfully). This is called *throwing an exception*. A **throw** statement specifies an object to be thrown. The operand of a **throw** can be of any class derived from class **Throwable** (package **java.lang**). The two immediate subclasses of class **Throwable** are **Exception** and **Error**. **Errors** are particularly serious system problems that generally should not be caught. **Exceptions** are caused by problems that should be caught and processed during program execution to make a program more robust. If the operand of the **throw** is an object of class **Exception**, it is called an *exception object*.



Testing and Debugging Tip 14.3

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.



Testing and Debugging Tip 14.4

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

When an exception is thrown, control exits the current **try** block and proceeds to an appropriate **catch** handler (if one exists) after that **try** block. It is possible that the **throw** point could be in a deeply nested scope within a **try** block; control will still proceed to the **catch** handler. It is also possible that the **throw** point could be in a deeply nested method call; still, control will proceed to the **catch** handler.

A **try** block may appear to contain no error checking and include no **throw** statements, but methods called from the **try** block may throw exceptions. Also, statements in a **try** block that do not invoke methods may cause exceptions. For example, a statement that performs array subscripting on an array object throws an **ArrayIndexOutOfBoundsException** if the statement specifies an invalid array subscript. Any method call can invoke code that might **throw** an exception or call another method that throws an exception.

14.7 Catching an Exception

Exception handlers are contained in **catch** blocks. Each **catch** block starts with the keyword **catch** followed by parentheses containing a class name (specifying the type of exception to be caught) and a parameter name. The handler can reference the thrown object through this parameter. This is followed by a block delineating the exception-handling code. When a handler catches an exception, the code in the **catch** block executes.



Common Programming Error 14.3

Logic errors can occur if you assume that after an exception is processed, control will return to the first statement after the **throw**. Program control continues with the first statement after the **catch** handlers.



Common Programming Error 14.4

Specifying a comma-separated list of **catch** parameters is a syntax error. A **catch** can have only a single parameter.



Common Programming Error 14.5

It is a syntax error to **catch** the same type in two different **catch** blocks associated with a particular **try** block.

A **catch** that catches a **Throwable** object

```
catch( Throwable throwable )
```

catches all exceptions and errors. Similarly, a **catch** that catches an **Exception** object

```
catch( Exception exception )
```

catches all exceptions. In general, programs do not define exception handlers for type `Throwable`, because **Errors** normally should not be caught in a program.



Common Programming Error 14.6

Placing **`catch(Exception exception)`** before other **`catch`** blocks that catch specific types of exceptions would prevent those blocks from executing; an exception handler that catches type **`Exception`** must be placed last in the list of exception handlers following a **`try`** block, or a syntax error occurs.

It is possible that a **`try`** block will not have a corresponding **`catch`** handler that matches a particular thrown object. This causes the search for a matching **`catch`** handler to continue in the next enclosing **`try`** block. As this process continues, eventually the program may determine that there is no handler on the execution stack that matches the type of the thrown object. In this case, a non-GUI-based application terminates—applets and GUI-based applications return to their regular event processing. Although applets and GUI-based applications continue to execute, they may execute incorrectly.



Software Engineering Observation 14.6

If you know that a method may throw an exception, include appropriate exception-handling code in your program. This will make your program more robust.



Good Programming Practice 14.4

Read the online API documentation for a method before using that method in a program. The documentation specifies the exceptions thrown by the method (if any) and indicates reasons why such exceptions may occur.



Good Programming Practice 14.5

Read the online API documentation for an exception class before writing exception-handling code for that type of exception. The documentation for an exception class typically contains potential reasons that such exceptions may occur during program execution.

It is possible that several exception handlers will provide an acceptable match to the type of the exception. This can happen for several reasons: There can be a “catch-all” handler **`catch(Exception exception)`** that will catch any exception. Also, inheritance relationships enable a subclass object to be caught either by a handler specifying the subclass type, or by handlers specifying the types of any of that class’s superclasses. The first exception handler that matches the exception type executes—all other exception handlers for the corresponding **`try`** block are ignored.



Software Engineering Observation 14.7

If several handlers match the type of an exception, and if each of these handles the exception differently, then the order of the handlers will affect the manner in which the exception is handled.



Common Programming Error 14.7

It is a syntax error if a **`catch`** that catches a superclass object is placed before a **`catch`** for that class’s subclass types.

Sometimes a program may process many closely related types of exceptions. Instead of providing separate **`catch`** handlers for each, a programmer can provide a single **`catch`** handler for a group of exceptions.

What happens when an exception occurs in an exception handler? The **try** block that noticed the exception expires before the exception handler begins running, so exceptions occurring in an exception handler are processed by **catch** handlers for an enclosing **try** block. The enclosing **try** block watches for errors occurring in the original **try** block's **catch** handlers. An enclosing **try** block is either a **try** block that contains a complete **try/catch** sequence or a **try** block in a calling method.

Exception handlers can be written a variety of ways. They can rethrow an exception (as we will see in Section 14.9). They can convert one type of exception into another by throwing a different type of exception. They can perform any necessary recovery and resume execution after the last exception handler. They can look at the situation causing the error, remove the cause of the error and retry by calling the original method that caused an exception. They can return a status value to their environment, etc.

It is not possible to return to the **throw** point by issuing a **return** statement in a **catch** handler. Such a **return** simply returns control to the method that called the method containing the **catch** block. Again, the **throw** point is in a block that has expired, so returning to that point via a **return** statement would not make sense.



Software Engineering Observation 14.8

Another reason not to use exceptions for conventional flow of control is that these “additional” exceptions can “get in the way” of genuine error-type exceptions. It becomes more difficult for the programmer to keep track of the larger number of exception cases. Exceptional situations should be rare, not commonplace.



Common Programming Error 14.8

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

14.8 Exception-Handling Example: Divide by Zero

Now let us consider a simple example of exception handling. The application of Fig. 14.1 and Fig. 14.2 uses **try**, **throw** and **catch** to detect, indicate and handle exceptions. The application displays two **JTextField**s in which the user can type integers. When the user presses the *Enter* key in the second **JTextField**, the program calls method **actionPerformed** to read the two integers from the **JTextField**s and pass the integers to method **quotient**, which calculates the quotient of the two values and returns a **double** result. If the user types 0 in the second **JTextField**, the program uses an exception to indicate that the user is attempting to divide by zero. Also, if the user types a noninteger value in either **JTextField**, a **NumberFormatException** occurs. In prior examples that read numeric values from the user, we simply assumed that the user would input a proper integer value. However, users sometimes make mistakes. This program demonstrates how to catch the **NumberFormatException** that occurs when a program attempts to convert a **String** that does not represent an integer value to an **int** value with **Integer** method **parseInt**.

Before we discuss the program, consider the sample executions shown in the five output windows of Fig. 14.2. The first window shows a successful execution. The user typed the values 100 and 7. The third **JTextField** shows the result of the division performed by method **quotient**. In the second output window, the user entered the string “hello” in the second **JTextField**. When the user presses *Enter* in the second **JText-**

Field, an error-message dialog is displayed indicating that an integer must be entered. In the last two windows, a zero denominator is entered and the program detects the problem, throws an exception and issues an appropriate diagnostic message.

Now let's discuss the program beginning with class **DivideByZeroException** of Fig. 14.1. Java can test for division by zero when the values in the division are both integers. If Java discovers an attempt to divide by zero in integer arithmetic, Java throws an **ArithmeticException**. However, our program performs a floating-point division of two integers by casting the first integer to a **double** before performing the calculation. Java allows floating-point division by zero. The result is positive or negative infinity. (Classes **Float** and **Double** of package **java.lang** each provide constants that represent these values.) Even though Java allows floating-point division by zero, we would like to use exception handling in this example to indicate to the user of our program that they are attempting to divide by zero.

As we will see, method **quotient** throws an exception when it receives zero as its second argument. A method can throw an exception of any existing type in the Java API or of a type specific to the program. In this example, we demonstrate defining a new exception type. Actually, we could use class **ArithmeticException** (package **java.lang**) with a customized error message in this program.



Good Programming Practice 14.6

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



Software Engineering Observation 14.9

If possible, use an existing exception type, rather than creating a new class. The Java API contains many exception types that may be suitable for your program.

```

1  // Fig. 14.1: DivideByZeroException.java
2  // Definition of class DivideByZeroException.
3  // Used to throw an exception when a
4  // divide-by-zero is attempted.
5  public class DivideByZeroException extends ArithmeticException {
6
7      // no-argument constructor specifies default error message
8      public DivideByZeroException()
9      {
10         super( "Attempted to divide by zero" );
11     }
12
13     // constructor to allow customized error message
14     public DivideByZeroException( String message )
15     {
16         super( message );
17     }
18
19 } // end class DivideByZeroException

```

Fig. 14.1 Exception class **DivideByZeroException**.

Class **DivideByZeroException** extends class **ArithmeticException**. We chose to extend class **ArithmeticException** because dividing by zero occurs during arithmetic. Like any other class, an exception class can contain instance variables and methods. A typical exception class contains only two constructors—one that takes no arguments and specifies a default exception message and one that receives a customized exception message as a **String**. The default constructor (lines 8–11) specifies the string **"Attempted to divide by zero"** as the exception message that indicates what went wrong. This string is passed to the superclass constructor to initialize the error message associated with the exception object. The other constructor (lines 14–17) passes its argument—a customized error-message string—to the superclass constructor.



Software Engineering Observation 14.10

*When defining your own exception type, subclass an existing related exception type in the Java API. This requires you to investigate the existing exceptions in the Java API. If the existing types are not appropriate for subclassing, the new exception class should extend **Exception** if the client of your code should be required to handle the exception, or should extend **RuntimeException** if the client of your code should have the option of ignoring the exception.*

Now consider the **DivideByZeroTest** application (Fig. 14.2). The application's constructor (lines 21–51) builds a graphical user interface with three **JLabels** (all right aligned) and three **JTextFields** and registers the **DivideByZeroTest** object as the **ActionListener** for **JTextField** **inputField2**.

```

1  // Fig. 14.2: DivideByZeroTest.java
2  // A simple exception handling example.
3  // Checking for a divide-by-zero-error.
4
5  // Java core packages
6  import java.awt.*;
7  import java.awt.event.*;
8  import java.text.DecimalFormat;
9
10 // Java extension packages
11 import javax.swing.*;
12
13 public class DivideByZeroTest extends JFrame
14     implements ActionListener {
15
16     private JTextField inputField1, inputField2, outputField;
17     private int number1, number2;
18     private double result;
19
20     // set up GUI
21     public DivideByZeroTest()
22     {
23         super( "Demonstrating Exceptions" );
24     }

```

Fig. 14.2 A simple exception-handling example with divide by zero (part 1 of 3).

```

25      // get content pane and set its layout
26      Container container = getContentPane();
27      container.setLayout( new GridLayout( 3, 2 ) );
28
29      // set up label and inputField1
30      container.add(
31          new JLabel( "Enter numerator ", SwingConstants.RIGHT ) );
32      inputField1 = new JTextField( 10 );
33      container.add( inputField1 );
34
35      // set up label and inputField2; register listener
36      container.add(
37          new JLabel( "Enter denominator and press Enter ",
38              SwingConstants.RIGHT ) );
39      inputField2 = new JTextField( 10 );
40      container.add( inputField2 );
41      inputField2.addActionListener( this );
42
43      // set up label and outputField
44      container.add(
45          new JLabel( "RESULT ", SwingConstants.RIGHT ) );
46      outputField = new JTextField();
47      container.add( outputField );
48
49      setSize( 425, 100 );
50      setVisible( true );
51  }
52
53  // process GUI events
54  public void actionPerformed((ActionEvent event)
55  {
56      DecimalFormat precision3 = new DecimalFormat( "0.000" );
57
58      outputField.setText( "" ); // clear outputField
59
60      // read two numbers and calculate quotient
61      try {
62          number1 = Integer.parseInt( inputField1.getText() );
63          number2 = Integer.parseInt( inputField2.getText() );
64
65          result = quotient( number1, number2 );
66          outputField.setText( precision3.format( result ) );
67      }
68
69      // process improperly formatted input
70      catch ( NumberFormatException numberFormatException ) {
71          JOptionPane.showMessageDialog( this,
72              "You must enter two integers",
73              "Invalid Number Format",
74              JOptionPane.ERROR_MESSAGE );
75      }
76

```

Fig. 14.2 A simple exception-handling example with divide by zero (part 2 of 3).

```

77      // process attempts to divide by zero
78      catch ( ArithmeticException arithmeticException ) {
79          JOptionPane.showMessageDialog( this,
80              arithmeticException.toString(),
81              "Arithmetic Exception",
82              JOptionPane.ERROR_MESSAGE );
83      }
84  }
85
86  // method quotient demonstrated throwing an exception
87  // when a divide-by-zero error occurs
88  public double quotient( int numerator, int denominator )
89      throws DivideByZeroException
90  {
91      if ( denominator == 0 )
92          throw new DivideByZeroException();
93
94      return ( double ) numerator / denominator;
95  }
96
97  // execute application
98  public static void main( String args[] )
99  {
100      DivideByZeroTest application = new DivideByZeroTest();
101
102      application.setDefaultCloseOperation(
103          JFrame.EXIT_ON_CLOSE );
104  }
105
106 } // end class DivideByZeroTest

```

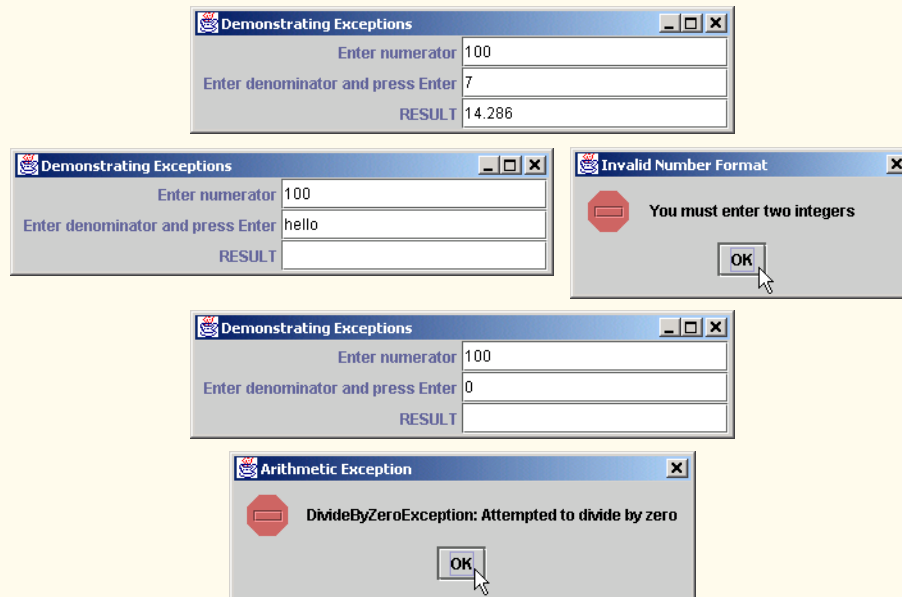


Fig. 14.2 A simple exception-handling example with divide by zero (part 3 of 3).

When the user inputs the denominator and presses the *Enter* key, the program calls method **actionPerformed** (lines 54–84). Next, method **actionPerformed** proceeds with a **try** block (lines 61–67), which encloses the code that may **throw** an exception and any code that should not be executed if an exception occurs. The statements that read the integers from the **JTextFields** (lines 62–63) each use method **Integer.parseInt** to convert **Strings** to **int** values. Method **parseInt** throws a **NumberFormatException** if its **String** argument is not a valid integer. The division that can cause the divide-by-zero error is not performed explicitly in the **try** block. Rather, the call to method **quotient** (line 65) invokes the code that attempts the division. Method **quotient** (lines 89–96) **throws** the **DivideByZeroException** object, as we will see momentarily. In general, errors may surface through explicitly mentioned code in a **try** block, through calls to a method or even through deeply nested method calls initiated by code in a **try** block.

The **try** block in this example is followed by two **catch** blocks—lines 70–75 contain the exception handler for the **NumberFormatException** and lines 78–83 contain the exception handler for the **DivideByZeroException**. In general, when the program detects an exception while executing a **try** block, the program catches the exception in a **catch** block that specifies an appropriate exception type (i.e., the type in the **catch** matches the thrown exception type exactly or is a superclass of the thrown exception type). In Fig. 14.2, the first **catch** block specifies that it will catch exception objects of type **NumberFormatException** (this type matches the exception object type thrown in method **Integer.parseInt**) and the second **catch** block specifies that it will catch exception objects of type **ArithmeticException** (this type is a superclass of the exception object type thrown in method **quotient**). Only the matching **catch** handler executes when an exception occurs. Both our exception handlers simply display an error-message dialog, but exception handlers can be more elaborate than this. After executing an exception handler, program control proceeds to the first statement after the last **catch** block (or in the **finally** block, if one is present).

If the code in the **try** block does not **throw** an exception, then the **catch** handlers are skipped and execution resumes with the first line of code after the **catch** handlers (or in the **finally** block, if one is present). In Fig. 14.2, method **actionPerformed** simply returns, but the program could continue executing more statements after the **catch** blocks.



Testing and Debugging Tip 14.5

With exception handling, a program can continue executing after dealing with a problem. This helps ensure robust applications that contribute to what is called mission-critical computing or business-critical computing.

Now let us examine method **quotient** (lines 89–96). When the **if** structure determines that **denominator** is zero, the body of the **if** executes a **throw** statement that creates and **throws** a new **DivideByZeroException** object. This object will be caught by the **catch** block (lines 78–83) specifying type **ArithmeticException** after the **try** block. The **catch** block specifies parameter name **arithmeticException** to receive the thrown exception object. The **ArithmeticException** handler converts the exception to a **String** via **toString** and passes this **String** as the message to display in an error-message dialog.

If **denominator** is not zero, **quotient** does not throw an exception. Rather, **quotient** performs the division and returns the result of the division to the point of invocation

of method **quotient** in the **try** block (line 65). Line 66 displays the result of the calculation in the third **JTextField**. In this case, the **try** block completes successfully, so the program skips the **catch** blocks and the **actionPerformed** method completes execution normally.

Note that when **quotient** throws the **DivideByZeroException**, **quotient**'s block expires (i.e., the method terminates). This would cause any of its local variables to be destroyed—objects that were referenced by local variables in the block would have their reference counts decremented accordingly (and are possibly marked for garbage collection). Also, the **try** block from which the method was called expires before line 66 can execute. Here, too, if there were local variables created in the **try** block prior to the exception being thrown, these variables would be destroyed.

If a **NumberFormatException** is generated by lines 62–63, the **try** block expires and execution continues with the exception handler at line 70, which displays an error message to tell the user to input integers. Then the **actionPerformed** method continues with the next valid statement after the **catch** blocks (i.e., the method terminates in this example).

14.9 Rethrowing an Exception

It is possible that the **catch** handler that catches an exception may decide it cannot process the exception, or it may want to let some other **catch** handler handle the exception. In this case, the handler that received the exception can rethrow the exception with the statement

```
throw exceptionReference;
```

where *exceptionReference* is the parameter name for the exception in the **catch** handler. Such a **throw** rethrows the exception to the next enclosing **try** block.

Even if a handler can process an exception, and regardless of whether it does any processing on that exception, the handler still can rethrow the exception for further processing outside the handler. A rethrown exception is detected by the next enclosing **try** block and is handled by an exception handler listed after that enclosing **try** block.

14.10 throws Clause

A **throws** clause lists the exceptions that can be thrown by a method as in

```
int functionName( parameterList )
    throws ExceptionType1, ExceptionType2, ExceptionType3, ...
{
    // method body
}
```

The types of exceptions that are thrown by a method are specified in the method definition with a comma-separated list in the **throws** clause. A method can **throw** objects of the indicated classes, or it can **throw** objects of their subclasses.

Some exceptions can occur at any point during the execution of the program. Many of these exceptions can be avoided by coding properly. These are run-time exceptions, and they derive from class **RuntimeException**. For example, if your program attempts to access an out-of-range array subscript, an exception of type **ArrayIndexOutOfBoundsException** (derived from **RuntimeException**) occurs. Your program clearly can avoid such a problem; hence, it is a run-time exception.

Another run-time exception occurs when your program creates an object reference, but has not yet created an object and assigned it to the reference. Attempting to use such a **null** reference causes a **NullPointerException** to be thrown. Clearly, your program can avoid this circumstance; hence, it is a run-time exception. Another run-time exception is an invalid cast, which throws a **ClassCastException**.

There are a variety of exceptions that are not **RuntimeExceptions**. Two of the most common are **InterruptedExceptions** (see Chapter 15, “Multithreading”) and **IOExceptions** (see Chapter 16, “Files and Streams”).

Not all errors and exceptions that can be thrown from a method are required to be listed in the **throws** clause. **Errors** do not need to be listed, nor do **RuntimeExceptions** (avoidable exceptions). **Errors** are serious system problems that can occur almost anywhere, and most programs will not be able to recover from them. Methods should process **RuntimeExceptions** caught in their bodies directly rather than passing them on to other program components. If a method throws any non-**RuntimeExceptions**, it *must* specify those exception types in its **throws** clause.



Software Engineering Observation 14.11

*If a non-**RuntimeException** is thrown by a method, or if that method calls methods that throw non-**RuntimeExceptions**, each of those exceptions must be declared in the **throws** clause of that method or caught in a **try/catch** in that method.*

Java distinguishes *checked Exceptions* versus *unchecked RuntimeExceptions* and **Errors**. A method’s checked exceptions need to be listed in that method’s **throws** clause. **Errors** and **RuntimeExceptions** can be thrown from almost any method, so it would be cumbersome for programmers to be required to list them for every method definition. Such exceptions and errors are not required to be listed in a method’s **throws** clause and, hence, are said to be “unchecked” by the compiler. All non-**RuntimeExceptions** a method can **throw** must be listed in that method’s **throws** clause and, hence, are said to be “checked” by the compiler. If a non-**RuntimeException** is not listed in the **throws** clause, the compiler will issue an error message indicating that the exception must be caught (with a **try/catch** in the body of the method) or declared (with a **throws** clause).



Common Programming Error 14.9

*It is a syntax error if a method throws a checked exception not in that method’s **throws** clause.*



Common Programming Error 14.10

*Attempting to throw a checked exception from a method that has no **throws** clause is a syntax error.*



Software Engineering Observation 14.12

*If your method calls other methods that explicitly **throw** checked exceptions, those exceptions must be listed in the **throws** clause of your method, unless your method catches those exceptions. This is Java’s “**catch-or-declare**” requirement.*



Common Programming Error 14.11

*If a subclass method overrides a superclass method, it is an error for the subclass method to list more exceptions in its **throws** list than the overridden superclass method does. A subclass’s **throws** list can contain a subset of a superclass’s **throws** list.*

Java's **catch-or-declare requirement** demands that the programmer either **catch** each checked exception or place it in the **throws** clause of a method. Of course, placing a checked exception in the **throws** clause would force other methods to process the checked exception as well. If a programmer feels a particular checked exception is unlikely to occur, the programmer might elect to catch that checked exception and do nothing with it to avoid being forced to deal with it later. This can of course come back to haunt you, because as a program evolves, it may become important to deal with this checked exception.



Testing and Debugging Tip 14.6

Do not try to circumvent Java's **catch-or-declare requirement** by simply catching exceptions and doing nothing with them. Exceptions are generally of a serious enough nature that they need to be processed rather than suppressed.



Testing and Debugging Tip 14.7

The Java compiler, through the **throws** clause used with exception handling, forces programmers to process the exceptions that can be thrown from each method a program calls. This helps avoid bugs that arise in programs when programmers ignore the fact that problems occur and make no provisions for these problems.



Software Engineering Observation 14.13

Subclass methods that do not override their corresponding superclass methods exhibit the same exception-handling behavior of the inherited superclass methods. The **throws** clause of a subclass method that overrides a superclass method may contain the same list of exceptions as the overridden superclass method or a subset of that list.



Common Programming Error 14.12

The Java compiler requires that a method either catch any checked exceptions thrown in the method (either directly from the method's body or indirectly through called methods) or declare checked **Exceptions** the method can **throw** to other methods; otherwise, the Java compiler issues a syntax error.



Testing and Debugging Tip 14.8

Suppose a method **throws** all subclasses of a particular exception superclass. You may be tempted to list only the superclass in the **throws** clause. Instead, explicitly list all the subclasses. This focuses the programmer's attention on the specific **Exceptions** that may occur and will often help avoid bugs caused by performing general processing for a category of exception types.

Figure 14.3–Fig. 14.8 list many of Java's **Errors** and **Exceptions** hierarchically for the packages **java.lang**, **java.util**, **java.io**, **java.awt** and **java.net**. In these tables, a class indented under another class is a subclass. The exception and error classes for the other packages of the Java API can be found in the Java online documentation. The online documentation for each method in the API specifies whether that method throws exceptions and what the exceptions are that can be thrown. We show a portion of Java's **Error** hierarchy in Fig. 14.3. Most Java programmers ignore **Errors**. They are serious but rare events.

Figure 14.4 is particularly important because it lists many of Java's **Runtime-Exceptions**. Although Java programmers are not required to declare these exceptions in **throws** clauses, these are the exceptions that commonly will be caught and handled in Java applications.

The java.lang package errors

Error (all in `java.lang` except for `AWTError`, which is in `java.awt`)

- LinkageError**
 - ClassCircularityError**
 - ClassFormatError**
 - ExceptionInInitializerError**
 - IncompatibleClassChangeError**
 - AbstractMethodError**
 - IllegalAccessError**
 - InstantiationError**
 - NoSuchFieldError**
 - NoSuchMethodError**
 - NoClassDefFoundError**
 - UnsatisfiedLinkError**
 - VerifyError**
- ThreadDeath**
- VirtualMachineError** (Abstract class)
 - InternalError**
 - OutOfMemoryError**
 - StackOverflowError**
 - UnknownError**
- AWTError** (package `java.awt`)

Fig. 14.3 The `java.lang` package errors .

The java.lang package exceptions

Exception

- ClassNotFoundException**
- CloneNotSupportedException**
- IllegalAccessException**
- InstantiationException**
- InterruptedException**
- NoSuchFieldException**
- NoSuchMethodException**

Fig. 14.4 The `java.lang` package exceptions (part 1 of 2).

The java.lang package exceptions

```
RuntimeException
    ArithmeticException
    ArrayStoreException
    ClassCastException
    IllegalArgumentException
        IllegalThreadStateException
    NumberFormatException
    IllegalMonitorStateException
    IllegalStateException
    IndexOutOfBoundsException
        ArrayIndexOutOfBoundsException
        StringIndexOutOfBoundsException
    NegativeArraySizeException
    NullPointerException
    SecurityException
```

Fig. 14.4 The **java.lang** package exceptions (part 2 of 2).

Figure 14.5 lists Java's other three **RuntimeException**s data types. We will encounter these exceptions in Chapter 20 when we study the **Vector** class. A **Vector** is a dynamic array that can grow and shrink to accommodate a program's varying storage requirements.

Figure 14.6 lists Java's **IOException**s. These are all checked exceptions that can occur during input/output and file processing.

Figure 14.7 lists the **java.awt** package's only checked **Exception**, the **AWTException**. This is a checked exception that is thrown by various abstract windowing toolkit methods.

The java.util package exceptions

```
Exception
    RuntimeException
        EmptyStackException
        MissingResourceException
        NoSuchElementException
        TooManyListenersException
```

Fig. 14.5 The **java.util** package exceptions.

The `java.io` package exceptions

```
Exception
  IOException
    CharConversionException
    EOFException
    FileNotFoundException
    InterruptedIOException
    ObjectStreamException
      InvalidClassException
      InvalidObjectException
      NotActiveException
      NotSerializableException
      OptionalDataException
      StreamCorruptedException
      WriteAbortedException
    SyncFailedException
    UnsupportedCodingException
    UTFDataFormatException
```

Fig. 14.6 The `java.io` package exceptions .

The `java.awt` package exceptions

```
Exception
  AWTException
  RuntimeException
    IllegalStateException
      IllegalComponentStateException
```

Fig. 14.7 The `java.awt` package exceptions .

Figure 14.8 lists the `IOExceptions` of the `java.net` package. These are all checked **Exceptions** that indicate various networking problems.

Most packages in the Java API define **Exceptions** and **Errors** specific to the package. For a complete list of these types, see the online API documentation for the package.

The `java.net` package exceptions

```
Exception
  IOException
    BindException
    MalformedURLException
    ProtocolException
    SocketException
      ConnectException
      NoRouteToHostException
    UnknownHostException
    UnknownServiceException
```

Fig. 14.8 The `java.net` package exceptions.

14.11 Constructors, Finalizers and Exception Handling

First, let us deal with an issue we have mentioned, but that has yet to be resolved satisfactorily. What happens when an error is detected in a constructor? The problem is that a constructor cannot return a value, so how do we let the program know that an object has not been constructed properly? One scheme is to return the improperly constructed object and hope that anyone using the object would perform appropriate tests to determine that the object is in fact bad. However, this directly contradicts discussions in Chapter 8 in which we indicated that you should maintain an object in a consistent state at all times. Another scheme is to set some instance variable with an error indicator outside the constructor, but this is a poor programming practice. In Java, the typical (and proper) mechanism is to throw an exception from the constructor to the code creating the object. The thrown object contains the information about the failed constructor call and the caller is responsible for handling the failure.

When an exception occurs in a constructor, other objects created by that constructor are marked for eventual garbage collection. Before each object is garbage collected, its **finalize** method will be called.

14.12 Exceptions and Inheritance

Various exception classes can be derived from a common superclass. If a **catch** is written to catch exception objects of a superclass type, it can also catch all objects of subclasses of that superclass. This can allow for polymorphic processing of related exceptions.

Using inheritance with exceptions enables an exception handler to catch related errors with a concise notation. One could certainly catch each subclass exception object individually if those exceptions require different processing, but it is more concise to catch the superclass exception object. Of course, this makes sense only if the handling behavior would be the same for all subclasses. Otherwise, catch each subclass exception individually.



Testing and Debugging Tip 14.9

Catching subclass exception objects individually is subject to error if the programmer forgets to test for one or more of the subclass types explicitly; catching the superclass guarantees that objects of all subclasses will be caught. Often, a catch of the superclass type follows all other subclass exception handlers to ensure that all exceptions are processed properly.

14.13 finally Block

Programs that obtain certain types of resources must return those resources to the system explicitly to avoid so-called *resource leaks*. In programming languages like C and C++, the most common kind of resource leak is a memory leak. Java performs automatic garbage collection of memory no longer used by programs, thus avoiding most memory leaks. However, other types of resource leaks can occur in Java.



Software Engineering Observation 14.14

*A **finally** block typically contains code to release resources acquired in its corresponding **try** block; this is an effective way to eliminate resource leaks. For example, the **finally** block should close any files opened in the **try** block.*



Testing and Debugging Tip 14.10

Actually, Java does not eliminate memory leaks completely. There is a subtle issue here. Java will not garbage collect an object until there are no more references to the object. Thus, memory leaks can occur, but only if programmers erroneously keep references to unwanted objects. Most memory leak problems are solved by Java's garbage collection.

The **finally** block is optional. If it is present, it is placed after the last of a **try** block's **catch** blocks, as follows:

```
try {
    statements
    resource-acquisition statements
}
catch ( AKindOfException exception1 ) {
    exception-handling statements
}
catch ( AnotherKindOfException exception2 ) {
    exception-handling statements
}
finally {
    statements
    resource-release statements
}
```

Java guarantees that a **finally** block (if one is present) will execute regardless of whether any exception is thrown in the corresponding **try** block or any of its corresponding **catch** blocks. Java also guarantees that a **finally** block (if one is present) will execute if a **try** block exits via a **return**, **break** or **continue** statement.

Resource-release code is placed in a **finally** block. Suppose a resource is allocated in a **try** block. If no exception occurs, the **catch** handlers are skipped and 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 program skips the rest of the **try** block. If the program catches the exception in one of the **catch** handlers, the program processes the exception. Then the **finally** block releases the resource, and control then proceeds to the first statement after the **finally** block.

If an exception that occurs in the **try** block cannot be caught by one of the **catch** handlers, the program skips the rest of the **try** block and control proceeds to the **finally** block, which releases the resource. Then the program passes the exception up the call chain until some calling method chooses to **catch** it. If no method chooses to deal with it, a non-GUI-based application terminates.

If a **catch** handler throws an exception, the **finally** block still executes. Then the exception is passed up the call chain for a calling method to **catch** and handle.

The Java application of Fig. 14.9 demonstrates that the **finally** block (if one is present) executes even if an exception is not thrown in the corresponding **try** block. The program contains methods **main** (lines 7–21), **throwException** (lines 24–50) and **doesNotThrowException** (lines 53–75). Methods **throwException** and **doesNotThrowException** are declared **static** so **main** (another **static** method) can call them directly.

Method **main** begins executing, enters its **try** block and immediately calls **throwException** (line 11). Method **throwException** throws an **Exception** (line 29), catches it (line 33) and rethrows it (line 37). The rethrown exception will be handled in **main**, but first the **finally** block (lines 44–47) executes. Method **main** detects the rethrown exception in the **try** block in **main** (lines 10–12) and handles it by the **catch** block (lines 15–18). Next, **main** calls method **doesNotThrowException** (line 20). No exception is thrown in **doesNotThrowException**'s **try** block, so the program skips the **catch** block (lines 61–64), but the **finally** block (lines 68–71) nevertheless executes. Control proceeds to the statement after the **finally** block. Then control returns to **main** and the program terminates.

```

1  // Fig. 14.9: UsingExceptions.java
2  // Demonstration of the try-catch-finally
3  // exception handling mechanism.
4  public class UsingExceptions {
5
6      // execute application
7      public static void main( String args[] )
8      {
9          // call method throwException
10         try {
11             throwException();
12         }
13
14         // catch Exceptions thrown by method throwException
15         catch ( Exception exception )
16         {
17             System.err.println( "Exception handled in main" );
18         }
19     }

```

Fig. 14.9 Demonstration of the **try-catch-finally** exception-handling mechanism (part 1 of 3).

```
20     doesNotThrowException();
21 }
22
23 // demonstrate try/catch/finally
24 public static void throwException() throws Exception
25 {
26     // throw an exception and immediately catch it
27     try {
28         System.out.println( "Method throwException" );
29         throw new Exception(); // generate exception
30     }
31
32     // catch exception thrown in try block
33     catch ( Exception exception )
34     {
35         System.err.println(
36             "Exception handled in method throwException" );
37         throw exception; // rethrow for further processing
38
39         // any code here would not be reached
40     }
41
42     // this block executes regardless of what occurs in
43     // try/catch
44     finally {
45         System.err.println(
46             "Finally executed in throwException" );
47     }
48
49     // any code here would not be reached
50 }
51
52 // demonstrate finally when no exception occurs
53 public static void doesNotThrowException()
54 {
55     // try block does not throw an exception
56     try {
57         System.out.println( "Method doesNotThrowException" );
58     }
59
60     // catch does not execute, because no exception thrown
61     catch( Exception exception )
62     {
63         System.err.println( exception.toString() );
64     }
65
66     // this block executes regardless of what occurs in
67     // try/catch
68     finally {
69         System.err.println(
70             "Finally executed in doesNotThrowException" );
71     }
```

Fig. 14.9 Demonstration of the **try-catch-finally** exception-handling mechanism (part 2 of 3).

```

72
73     System.out.println(
74         "End of method doesNotThrowException" );
75     }
76
77 } // end class UsingExceptions

```

```

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. 14.9 Demonstration of the **try-catch-finally** exception-handling mechanism (part 3 of 3).

The Java application in Fig. 14.10 demonstrates that when an exception thrown in a **try** block is *not* caught in a corresponding **catch** block, the exception will be detected in the next outer **try** block and handled by an appropriate **catch** block (if one is present) associated with that outer **try** block.

```

1  // Fig. 14.10: UsingExceptions.java
2  // Demonstration of stack unwinding.
3  public class UsingExceptions {
4
5      // execute application
6      public static void main( String args[] )
7      {
8          // call throwException to demonstrate stack unwinding
9          try {
10             throwException();
11         }
12
13         // catch exception thrown in throwException
14         catch ( Exception exception ) {
15             System.err.println( "Exception handled in main" );
16         }
17     }
18
19     // throwException throws an exception that is not caught in
20     // the body of this method
21     public static void throwException() throws Exception
22     {
23         // throw an exception and catch it in main
24         try {
25             System.out.println( "Method throwException" );
26             throw new Exception(); // generate exception
27         }

```

Fig. 14.10 Demonstration of stack unwinding (part 1 of 2).

```

28
29      // catch is incorrect type, so Exception not caught
30      catch( RuntimeException runtimeException ) {
31          System.err.println(
32              "Exception handled in method throwException" );
33      }
34
35      // finally block always executes
36      finally {
37          System.err.println( "Finally is always executed" );
38      }
39  }
40
41  } // end class UsingExceptions

```

Method `throwException`
 Finally is always executed
 Exception handled in main

Fig. 14.10 Demonstration of stack unwinding (part 2 of 2).

When method `main` executes, line 10 in the `try` block calls `throwException` (lines 21–39). In the `try` block of method `throwException`, line 26 throws an `Exception`. This terminates the `try` block immediately and control proceeds to the `catch` handler at line 30. The type being caught (`RuntimeException`) is not an exact match with the thrown type (`Exception`) and is not a superclass of the thrown type, so the exception is not caught in method `throwException`. The exception must be handled before normal program execution can continue. Therefore, method `throwException` terminates (but not until its `finally` block executes) and returns control to the point from which it was called in the program (line 10). Line 10 is in the enclosing `try` block. If the exception has not yet been handled, the `try` block terminates and an attempt is made to catch the exception at line 14. The type being caught (`Exception`) matches the thrown type. Therefore, the `catch` handler processes the exception and the program terminates at the end of `main`.

As we have seen, a `finally` block executes for a variety of reasons, such as a `try` completing successfully, handling of an exception in a local `catch`, an exception being thrown for which no local `catch` is available or execution of a program control statement like a `return`, `break` or `continue`. Normally, the `finally` block executes, then behaves appropriately (we will call this `finally`'s "continuation action") depending on the reason program control entered the block. For example, if an exception is thrown in the `finally` block, the continuation action will be for *that* exception to be processed in the next enclosing `try` block. Unfortunately, if there was an exception that had not yet been caught, that exception is lost and the more recent exception is processed. This is dangerous.



Common Programming Error 14.13

If an exception is thrown for which no local `catch` is available, when control enters the local `finally` block, the `finally` block could also `throw` an exception. If this happens, the first exception will be lost.

**Testing and Debugging Tip 14.11**

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

**Good Programming Practice 14.7**

Java's exception-handling mechanism is intended to remove 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 makes programs difficult to read. Rather, place one **try** block around a significant portion of your code, follow that **try** block with **catch** blocks that handle each possible exception and follow the **catch** blocks with a single **finally** block (if one is required).

**Performance Tip 14.3**

As a rule, resources should be released as soon as it is apparent that they are no longer needed. This makes these resources immediately available for reuse and can improve program performance.

**Software Engineering Observation 14.15**

If a **try** block has a corresponding **finally** block, the **finally** block will execute even if the **try** block exits with **return**, **break** or **continue**; then the effect of the **return**, **break** or **continue** will occur.

14.14 Using printStackTrace and getMessage

Exceptions derive from class **Throwable**. Class **Throwable** offers a **printStackTrace** method that prints the method call stack. By calling this method for a **Throwable** object that has been caught, a program can print the method call stack. Often, this is helpful in testing and debugging. Two other overloaded versions of **printStackTrace** enable the program to direct the stack trace to a **PrintStream** or **PrintWriter** stream. In this section, we consider an example that exercises the **printStackTrace** method and another useful method, **getMessage**.

**Testing and Debugging Tip 14.12**

All **Throwable** objects contain a **printStackTrace** method that prints a stack trace for the object.

**Testing and Debugging Tip 14.13**

An exception that is not caught eventually causes Java's default exception handler to run. This displays the name of the exception, the optional character string that was supplied when the exception was constructed and a complete execution stack trace. The stack trace shows the complete method call stack. This lets the programmer see the path of execution that led to the exception file-by-file (and thus class-by-class) and method-by-method. This information is helpful in debugging a program.

There are two constructors for class **Throwable**. The first constructor

```
public Throwable()
```

takes no arguments. The second constructor

```
public Throwable( String informationString )
```

takes the argument **informationString**, which is descriptive information about the **Throwable** object. The **informationString** stored in the **Throwable** object may be obtained with method **getMessage**.



Testing and Debugging Tip 14.14

Throwable classes have a constructor that accepts a **String** argument. Using this form of the constructor is helpful in determining the source of the exception via method **getMessage**.

Figure 14.11 demonstrates **getMessage** and **printStackTrace**. Method **getMessage** returns the descriptive **String** stored in an exception. Method **printStackTrace** outputs to the standard error stream (normally, the command line or console) an error message with the class name of the exception, the descriptive **String** stored in the exception and a list of the methods that had not completed execution when the exception was thrown (i.e., all methods currently residing on the method call stack).

In the program, **main** invokes **method1**, **method1** invokes **method2** and **method2** invokes **method3**. At this point, the method call stack for the program is

```
method3
method2
method1
main
```

with the last method called (**method3**) at the top and the first method called (**main**) at the bottom. When **method3** throws an **Exception** (line 36), a stack-trace message is generated and stored in the **Exception** object. The stack trace reflects the throw point in the code (i.e., line 36). Then, the stack unwinds to the first method in the method call stack in which the exception can be caught (i.e., **main** because it contains a **catch** handler for **Exception**). The **catch** handler then uses **getMessage** and **printStackTrace** on the **Exception** object **exception** to produce the output. Notice that the line numbers in the output window correspond to the line numbers in the program.

```
1 // Fig. 14.11: UsingExceptions.java
2 // Demonstrating the getMessage and printStackTrace
3 // methods inherited into all exception classes.
4 public class UsingExceptions {
5
6     // execute application
7     public static void main( String args[] )
8     {
9         // call method1
10        try {
11            method1();
12        }
13
14        // catch Exceptions thrown from method1
15        catch ( Exception exception ) {
16            System.err.println( exception.getMessage() + "\n" );
17            exception.printStackTrace();
18        }
19    }
20 }
```

Fig. 14.11 Using **getMessage** and **printStackTrace** (part 1 of 2).

```

20
21 // call method2; throw exceptions back to main
22 public static void method1() throws Exception
23 {
24     method2();
25 }
26
27 // call method3; throw exceptions back to method1
28 public static void method2() throws Exception
29 {
30     method3();
31 }
32
33 // throw Exception back to method2
34 public static void method3() throws Exception
35 {
36     throw new Exception( "Exception thrown in method3" );
37 }
38
39 } // end class Using Exceptions

```

Exception thrown in method3

```

java.lang.Exception: Exception thrown in method3
    at UsingExceptions.method3(UsingExceptions.java:36)
    at UsingExceptions.method2(UsingExceptions.java:30)
    at UsingExceptions.method1(UsingExceptions.java:24)
    at UsingExceptions.main(UsingExceptions.java:11)

```

Fig. 14.11 Using `getMessage` and `printStackTrace` (part 2 of 2).

SUMMARY

- Some common examples of exceptions are memory exhaustion, an out-of-bounds array subscript, arithmetic overflow, division by zero and invalid method parameters.
- Exception handling is designed for dealing with synchronous malfunctions (i.e., those that occur as the result of a program's execution).
- Exception handling is used in situations in which a malfunction will be dealt with in a different scope from that which detected the malfunction.
- Exception handling should be used to process exceptions from software components such as methods, libraries and classes that are likely to be widely used and where it does not make sense for those components to handle their own exceptions.
- Exception handling should be used on large projects to handle error processing in a standardized manner for the entire project.
- Java exception handling is geared to situations in which the method that detects an error is unable to deal with it. Such a method will **throw** an exception. If the exception matches the type of the parameter in one of the **catch** blocks, the code for that **catch** block executes.
- The programmer encloses in a **try** block the code that may generate an exception. The **try** block is followed by one or more **catch** blocks. Each **catch** block specifies the type of exception it can catch and handle. Each **catch** block is an exception handler.

- If no exceptions are thrown in the **try** block, the exception handlers for that block are skipped. Then the program resumes execution after the last **catch** block, after executing a **finally** block if one is provided.
- Exceptions are thrown in a **try** block in a method or from a method called directly or indirectly from the **try** block.
- The operand of a **throw** can be of any class derived from **Throwable**. The immediate subclasses of **Throwable** are **Error** and **Exception**.
- **RuntimeExceptions** and **Errors** are said to be “unchecked.” Non-**RuntimeExceptions** are said to be “checked.” The checked exceptions thrown by a particular method must be specified in that method’s **throws** clause.
- Exceptions are caught by the closest exception handler (for the **try** block from which the exception was thrown) specifying an appropriate type.
- An exception terminates the block in which the exception occurred.
- A handler may rethrow the object to an outer **try** block.
- **catch(Exception exception)** catches all **Exceptions**.
- **catch(Throwable throwable)** catches all **Exceptions** and **Errors**.
- If no handler matches a particular thrown object, the search for a match continues in an enclosing **try** block.
- Exception handlers are searched in order for an appropriate match based on type. The first handler that matches is executed. When that handler finishes executing, control resumes with the first statement after the last **catch** block.
- The order of the exception handlers affects how an exception is handled.
- A subclass object can be caught either by a handler specifying that subclass type or by handlers specifying the types of any direct or indirect superclasses of that subclass.
- If no handler is found for an exception, a non-GUI-based application terminates; an applet or a GUI-based application will return to its regular event handling.
- An exception handler cannot access variables in the scope of its **try** block because by the time the exception handler begins executing, the **try** block has expired. Information the handler needs is normally passed in the thrown object.
- Exception handlers can rethrow an exception. They can convert one type of exception into another by throwing a different exception. They can perform any necessary recovery and resume execution after the last exception handler. They can look at the situation causing the error, remove the cause of the error and retry by calling the original method that caused an exception. They can simply return some status value to their environment.
- A handler that catches a subclass object should be placed before a handler that catches a superclass object. If the superclass handler were first, it would catch superclass objects and the objects of subclasses of that superclass.
- When an exception is caught, it is possible that resources may have been allocated, but not yet released in the **try** block. A **finally** block should release these resources.
- It is possible that the handler that catches an exception may decide it cannot process the exception. In this case, the handler can simply rethrow the exception. A **throw** followed by the exception object name rethrows the exception.
- Even if a handler can process an exception, and regardless of whether it does any processing on that exception, the handler can rethrow the exception for further processing outside the handler. A rethrown exception is detected by the next enclosing **try** block (normally in a calling method) and is

handled by an appropriate exception handler (if there is one) listed after that enclosing **try** block.

- A **throws** clause lists the checked exceptions that may be thrown from a method. A method may **throw** the indicated exceptions, or it may **throw** subclass types. If a checked exception not listed in the **throws** clause is thrown, a syntax error occurs.
- A powerful reason for using inheritance with exceptions is to **catch** a variety of related errors easily with concise notation. One could certainly **catch** each type of subclass exception object individually, but it is more concise to simply **catch** the superclass exception object.

TERMINOLOGY

ArithmeticException

array exceptions

ArrayIndexOutOfBoundsException

business-critical computing

catch a group of exceptions

catch all exceptions

catch an exception

catch block

catch(Exception e)

catch-or-declare requirement

checked **Exceptions**

ClassCastException

declare exceptions that can be thrown

default exception handler

EmptyStackException

Error class

Error class hierarchy

error handling

exception

Exception class

Exception class hierarchy

exception handler

exception handling

exception object

fault tolerance

FileNotFoundException

finally block

getMessage method of **Throwable** class

handle an exception

IllegalAccessError

IncompatibleClassChangeException

instanceof operator

InstantiationException

InternalException

InterruptedException

IOException

library exception classes

memory exhaustion

mission-critical computing

NegativeArraySizeException

NoClassDefFoundException

non-run-time exception

null reference

NullPointerException

OutOfMemoryError

printStackTrace method (**Throwable**)

resource leak

resumption model of exception handling

rethrow an exception

RuntimeException

stack unwinding

synchronous error

termination model of exception handling

throw an exception

throw point

throw statement

Throwable class

throws clause

try block

unchecked **Exceptions**

UnsatisfiedLinkException

SELF-REVIEW EXERCISES

14.1 List five common examples of exceptions.

14.2 Why should exception-handling techniques not be used for conventional program control?

14.3 Why are exceptions particularly appropriate for dealing with errors produced by library classes and methods?

14.4 What is a “resource leak?”

- 14.5** If no exceptions are thrown in a **try** block, where does control proceed to when the **try** block completes execution?
- 14.6** What happens if an exception occurs and an appropriate exception handler cannot be found?
- 14.7** Give a key advantage of using **catch(Exception e)**.
- 14.8** Should a conventional applet or application catch **Error** objects?
- 14.9** What happens if several handlers match the type of the thrown object?
- 14.10** Why would a programmer specify a superclass type as the type of a **catch** handler, then throw objects of subclass types?
- 14.11** How might a **catch** handler be written to process related types of errors without using inheritance among exception classes?
- 14.12** What is the key reason for using **finally** blocks?
- 14.13** Does throwing an **Exception** have to cause program termination?
- 14.14** What happens when a **catch** handler throws an **Exception**?
- 14.15** What happens to a local reference in a **try** block when that block throws an **Exception**?

ANSWERS TO SELF-REVIEW EXERCISES

- 14.1** Memory exhaustion, array subscript out of bounds, arithmetic overflow, division by zero, invalid method parameters.
- 14.2** (a) **Exception** handling is designed to handle infrequently occurring situations that often result in program termination, so compiler writers are not required to implement exception handling to perform optimally. (b) Flow of control with conventional control structures is generally clearer and more efficient than with exceptions. (c) Problems can occur because the stack is unwound when an exception occurs and resources allocated prior to the exception may not be freed. (d) The “additional” exceptions can get in the way of genuine error-type exceptions. It becomes more difficult for the programmer to keep track of the larger number of exception cases.
- 14.3** It is unlikely that library classes and methods could perform error processing that would meet the unique needs of all users.
- 14.4** A resource leak occurs when an executing program does not properly release a resource when the resource is no longer needed. If the program attempts to use the resource again in the future, the program may not be able to access the resource.
- 14.5** The exception handlers (in the **catch** blocks) for that **try** block are skipped, and the program resumes execution after the last **catch** block. If there is a **finally** block, it is executed and the program resumes execution after the **finally** block.
- 14.6** A non-GUI-based application terminates; an applet or a GUI-based application resumes regular event processing.
- 14.7** The form **catch(Exception e)** catches any type of exception thrown in a **try** block. An advantage is that no thrown **Exception** can slip by.
- 14.8** **Errors** are usually serious problems with the underlying Java system; most programs will not want to catch **Errors**.
- 14.9** The first matching **Exception** handler after the **try** block is executed.
- 14.10** This is a nice way to catch related types of exceptions, but it should be used carefully.

14.11 Provide a single **Exception** subclass and **catch** handler for a group of exceptions. As each exception occurs, the exception object can be created with different instance data. The **catch** handler can examine this data to distinguish the type of the **Exception**.

14.12 The **finally** block is the preferred means for preventing resource leaks.

14.13 No, but it does terminate the block in which the **Exception** is thrown.

14.14 The exception will be processed by a **catch** handler (if one exists) associated with the **try** block (if one exists) enclosing the **catch** handler that caused the exception.

14.15 The reference is removed from memory, and the reference count for the referenced object is decremented. If the reference count is zero, the object is marked for garbage collection.

EXERCISES

14.16 Under what circumstances would you use the following **catch** handler?

```
catch ( Exception exception ) {  
    throw exception;  
}
```

14.17 List the benefits of exception handling over conventional means of error processing.

14.18 Describe an object-oriented technique for handling related exceptions.

14.19 Until this chapter, we have found that dealing with errors detected by constructors is a bit awkward. Explain why exception handling is an effective means for dealing with constructor failure.

14.20 Suppose a program throws an exception and the appropriate exception handler begins executing. Now suppose that the exception handler itself throws the same exception. Does this create an infinite recursion? Explain your answer.

14.21 Use inheritance to create an exception superclass and various exception subclasses. Write a program to demonstrate that the **catch** specifying the superclass catches subclass exceptions.

14.22 Write a Java program that shows that not all finalizers for objects constructed in a block are necessarily called after an exception is thrown from that block.

14.23 Write a Java program that demonstrates how various exceptions are caught with

```
catch ( Exception exception )
```

14.24 Write a Java program that shows that the order of exception handlers is important. If you try to catch a superclass exception type before a subclass type, the compiler should generate errors. Explain why these errors occur.

14.25 Write a Java program that shows a constructor passing information about constructor failure to an exception handler after a **try** block.

14.26 Write a Java program that illustrates rethrowing an exception.

14.27 Write a Java program that shows that a method with its own **try** block does not have to catch every possible error generated within the **try**. Some exceptions can slip through to, and be handled in, other scopes.