Part

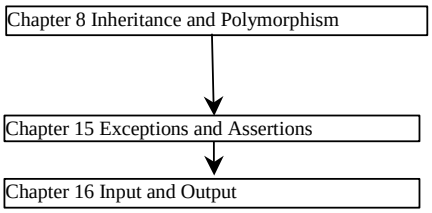
IV

Exception Handling and IO

This part introduces to use exception handling and assertions to make your program robust and correct, use input and output to manage and process a large quantity of data.

***PD: Put this in the center of the back of the Part-opening page. AU

Prerequisites for Part IV



CHAPTER

15

Exceptions and Assertions

Objectives

[BL]To understand the concept of exception handling.

[BL]To distinguish exception types: Error (fatal) vs. Exception (non-fatal), and checked vs. uncheck exceptions.

[BL]To declare exceptions in the method header.

[BL]To throw exceptions out of a method.

[BL]To write a try-catch block to handle exceptions.

[BL]To explain how an exception is propagated.

[BL]To rethrow exceptions in a try-catch block.

[BL]To use the finally clause in a try-catch block.

[BL]To declare custom exception classes.

[BL]To apply assertions to help ensure program correctness.

[BL]To know when to use exceptions and assertions.

15.1 Introduction

Section 2.16, “Programming Errors,” introduced three categories of errors: syntax errors, runtime errors, and logic errors. *Syntax errors* arise because the rules of the

language have not been followed. They are detected by the compiler. *Runtime errors* occur while the program is running if the environment detects an operation that is impossible to carry out. *Logic errors* occur when a program doesn't perform the way it was intended to. In general, syntax errors are easy to find and easy to correct because the compiler indicates where they came from and why they occurred. You can use the debugging techniques introduced in Section 2.17, “Debugging,” to find logic errors. This chapter introduces using exception handling to deal with runtime errors and using assertions to help ensure program correctness.

15.2 Exceptions and Exception Types

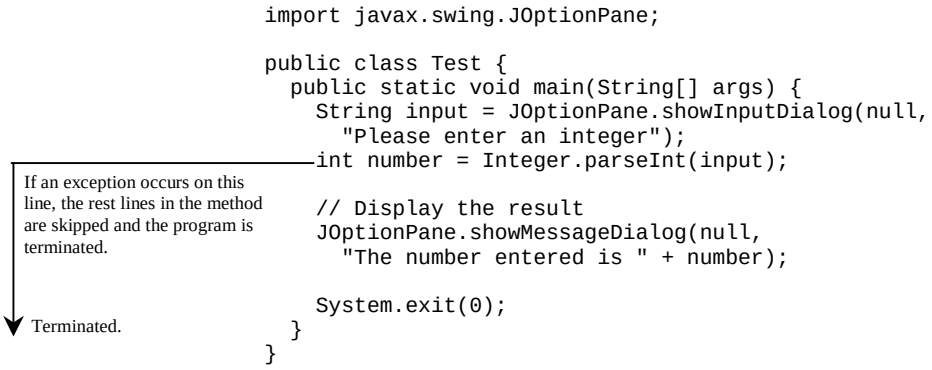
Runtime errors cause *exceptions*, which are events that occur during the execution of a program and disrupt the normal flow of control. A program that does not provide code to handle exceptions may terminate abnormally, causing serious problems. For example, if your program attempts to transfer money from a savings account to a checking account, but because of a runtime error is terminated *after* the money is drawn from the savings account and *before* the money is deposited in the checking account, the customer will lose money.

Runtime errors occur for various reasons. The user may enter an invalid input, for example, or the program may attempt to open a file that doesn't exist, or the network connection may hang up, or the program may attempt to access an out-of-bounds array element. When a runtime error occurs, Java raises an exception.

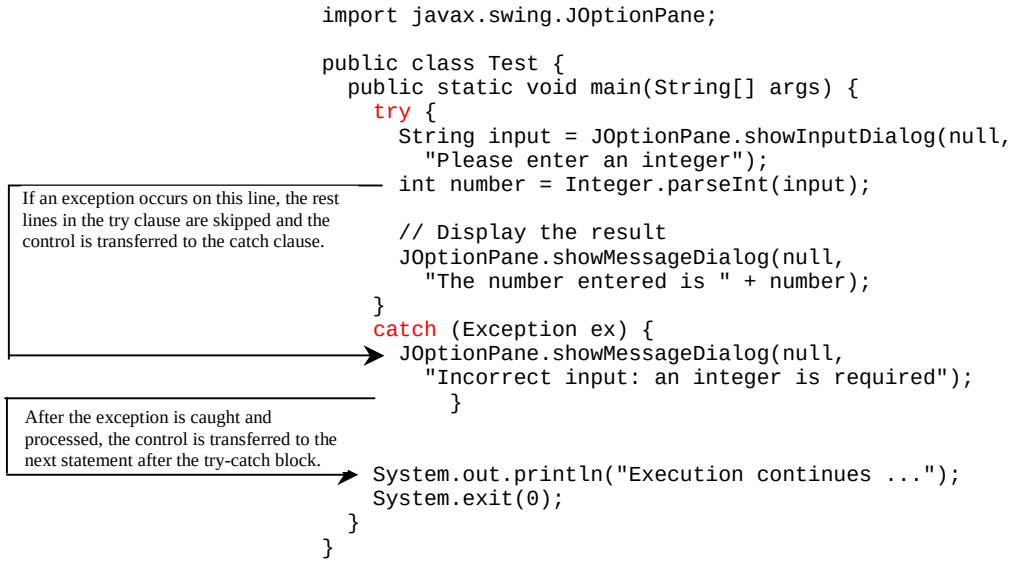
Exceptions are handled differently from the events of GUI programming. (In Chapter 12, "Event-Driven Programming," you learned the events used in GUI programming.) An *event* may be ignored in GUI programming, but an *exception* cannot be ignored. In GUI programming, a listener must register with the source object. External user action on the source object generates an event, and the source object notifies the listener by invoking the handlers implemented by the listener. If no listener is registered with the source object, the event is ignored. When an exception occurs, however, the program will terminate if the exception is not caught by the program.

Java provides programmers with the capability to elegantly handle runtime errors. With this capability, referred to as *exception handling*, you can develop robust programs for mission-critical computing.

Here is an example. The following program terminates abnormally if you entered a floating-point value instead of an integer.



Java allows the programmer to catch this error when it occurs, and perform some specific actions, including choosing whether to halt the program or not. You can handle this error in the following code, using a new construct called the *try-catch block* to enable the program to catch the error and continue to execute.



15.2.1 Exception Classes

A Java exception is an instance of a class derived from Throwable. The Throwable class is contained in the java.lang package, and subclasses of Throwable are contained in various packages. Errors related to GUI components are included in the java.awt package; numeric exceptions are included in the java.lang package because they are related to the java.lang.Number class. You can create your own exception classes by extending Throwable or a subclass of Throwable. Figure 15.1 shows some of Java's predefined exception classes.

******Same as Fig 15.1 in introjb3e p657***

***AU: Mark checked and unchecked exceptions. 11/12/03

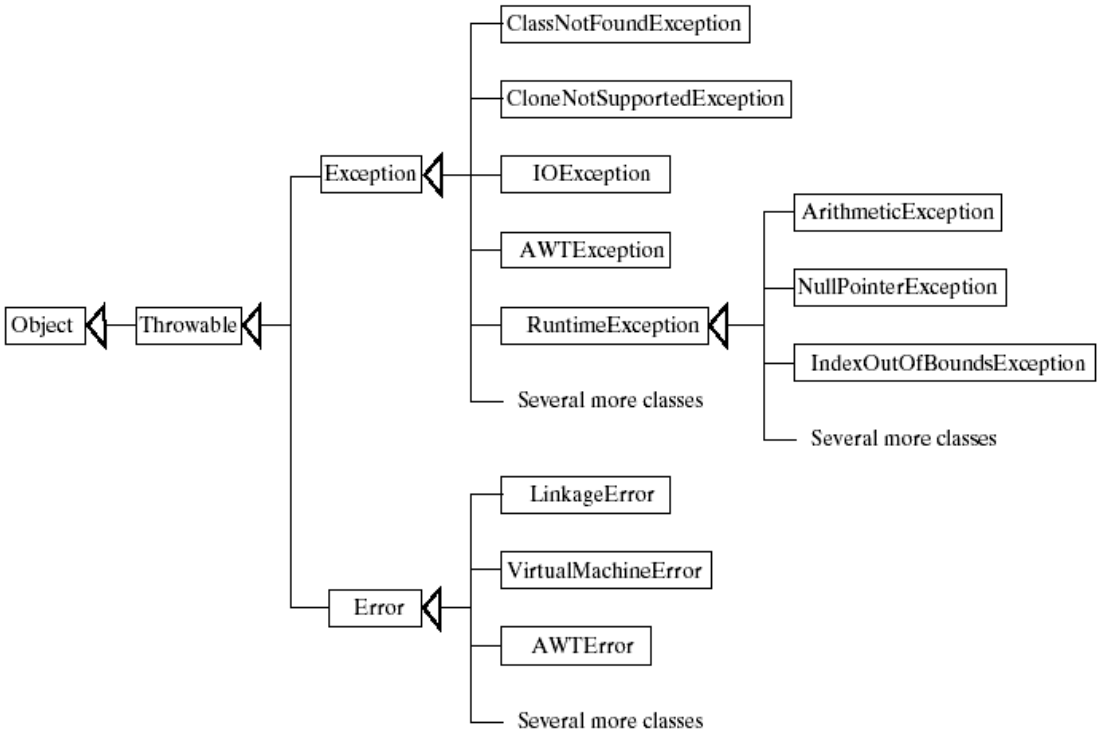


Figure 15.1

Exceptions thrown are instances of the classes shown in this diagram, or a subclass of one of these classes.

NOTE

The class names Error, Exception, and RuntimeException are somewhat confusing. All these classes are exceptions. Exception is just one of these classes, and all the errors discussed here occur at runtime.

The exception classes can be classified into three major types: system errors, exceptions, and runtime exceptions.

- *System errors* are thrown by JVM and represented in the Error class. The Error class describes internal system errors. Such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully. Examples of subclasses of Error are listed in Table 15.1.

- *Exceptions* are represented in the Exception class that describes errors caused by your program and external circumstances. These errors can be caught and handled

by your program. Examples of subclasses of Exception are listed in Table 15.2.

- *Runtime exceptions* are represented in the RuntimeException class that describes programming errors, such as bad casting, accessing an out-of-bounds array, and numeric errors. Runtime exceptions are generally thrown by JVM. Examples of subclasses are listed in Table 15.3.

Table 15.1

Examples of Subclasses of Error

Class	**Possible Exception Reason**
LinkageError	A class has some dependency on another class, but that the latter class has changed incompatibly after the compilation of the former class.
VirtualMachineError	The JVM is broken or has run out of the resources necessary for it to continue operating.
AWTError	A fatal error in the GUI runtime system.
AssertionError	An assertion has failed. Assertions will be introduced in Section 15.8, "Assersions."

Table 15.2

Examples of Subclasses of Exception

Class	**Possible Exception Reason**
ClassNotFoundException	Attempt to use a class that does not exist. This exception would occur, for example, if you tried to run a nonexistent class using the **java** command, or your program was composed of, say, three class files, only two of which could be found.
CloneNotSupportedException	Attempt to clone an object whose defining class does not implement the Cloneable interface. Cloning objects were introduced in Chapter 9, "Abstract Classes and Interfaces."
IOException	Related to input/output operations, such as invalid input, reading past the end of a file, and opening a nonexistent

	file. Examples of subclasses of IOException are InterruptedIOException, EOFException (EOF short for End Of File), and FileNotFoundException.
AWTException	Exceptions in GUI components.

Table 15.3

Examples of Subclasses of RuntimeException

Class	Possible Exception Reason
ArithmeticException	Dividing an integer by zero. Note that Floating-point arithmetic does not throw exceptions.
NullPointerException	Attempt to access an object through a null reference variable.
IndexOutOfBoundsException	Index to an array is out of range.
IllegalArgumentException	A method is passed an argument that is illegal or inappropriate.

15.2.2 Checked and Unchecked Exceptions

RuntimeException, Error, and their subclasses are known as *unchecked exceptions*. All other exceptions are known as *checked exceptions*, meaning that the compiler forces the programmer to check and deal with them.

In most cases, unchecked exceptions reflect programming logic errors that are not recoverable. For example, a NullPointerException is thrown if you access an object through a reference variable before an object is assigned to it; an IndexOutOfBoundsException is thrown if you access an element in an array outside the bounds of the array. These are logic errors that should be corrected in the program. Unchecked exceptions can occur anywhere in a program. To avoid cumbersome overuse of try-catch blocks, Java does not mandate that you write code to catch unchecked exceptions.

15.3 Understanding Exception Handling

Java's exception-handling model is based on three operations: *declaring an exception*, *throwing an exception*, and *catching an exception*, as shown in Figure 15.2.

******Same as Fig 15.2 in introjb3e p659***

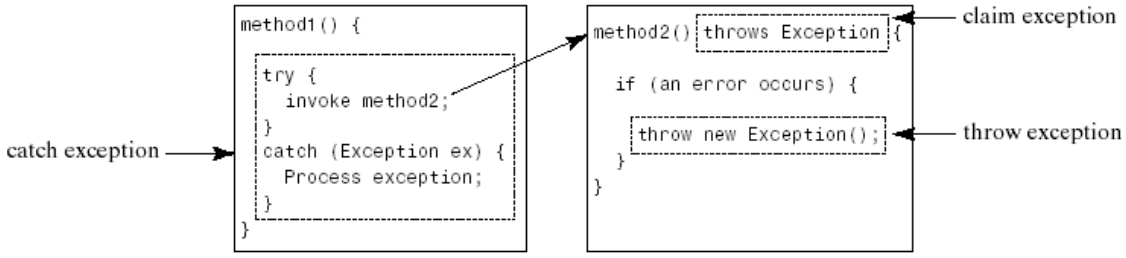


Figure 15.2

Exception handling in Java consists of declaring exceptions, throwing exceptions, and catching and processing exceptions.

15.3.1 Declaring Exceptions

In Java, the statement currently being executed belongs to a method. The Java interpreter invokes the main method for a Java application, and the Web browser invokes the applet's default constructor and then the init method for a Java applet. Every method must state the types of unchecked exceptions it might throw. This is known as *declaring exceptions*. Because system errors and runtime errors can happen to any code, Java does not require that you declare Error and RuntimeException (unchecked exceptions) explicitly in the method. However, all other exceptions must be explicitly declared in the method declaration if they are thrown by the method so that the caller of the method is informed of the exception.

To declare an exception in a method, use the throws keyword in the method declaration, as in this example:

```
public void myMethod() throws IOException
```

The throws keyword indicates that myMethod might throw an IOException. If the method might throw multiple exceptions, you can add a list of the exceptions, separated by commas, after throws:

```
public void myMethod()
  throws Exception1, Exception2, ..., ExceptionN
```

15.3.2 Throwing Exceptions

A program that detects an error can create an instance of an appropriate exception type and throw it. This is known as *throwing an exception*. Here is an example: Suppose the program detected that an argument passed to the method violates the method contract (e.g., the argument must be nonnegative, but a negative argument is passed); the program can create an instance of IllegalArgumentException and throw it, as follows:

```
IllegalArgumentException ex =
  new IllegalArgumentException("Wrong Argument");
throw ex;
```

Or if you prefer, you can use the following:

```
throw new IllegalArgumentException("Wrong Argument");
```

NOTE

The keyword to declare an exception is throws, and the keyword to throw an exception is throw.

A method can always throw an unchecked exception. If a method throws a checked exception, the exception must be declared in the method declaration.

15.3.3 Catching Exceptions

You now know how to declare an exception and how to throw an exception. When an exception is thrown, it can be caught and handled in a try-catch block, as follows:

```
try {
  statements;  // Statements that may throw exceptions
}
catch (Exception1 exVar1) {
  handler for exception1;
}
catch (Exception2 exVar2) {
  handler for exception2;
}
...
catch (ExceptionN exVar3) {
  handler for exceptionN;
}
```

If no exceptions arise during the execution of the try clause, the catch clauses are skipped.

If one of the statements inside the try block throws an exception, Java skips the remaining statements in the try block and starts the process of finding the code to handle the exception. The code that handles the exception is called the *exception handler*; it is found by propagating the exception backward through a chain of method calls, starting from the current method. Each catch clause is examined in turn, from first to last, to see whether the type of the exception object is an instance of the exception class in the catch clause. If so, the exception object is assigned to the variable declared and the code in the catch clause is executed. If no handler is found, Java exits this method, passes the exception to the method that invoked the method, and continues the same process to find a handler. If no handler is found in the chain of methods being invoked, the program terminates and prints an error message on the console. The process of finding a handler is called *catching an exception*.

Suppose the main method invokes method1, method1 invokes method2, method2 invokes method3, and an exception occurs in method3, as shown in Figure 15.3. Consider the following scenario:

- If method3 cannot handle the exception, method3 is aborted and the control is returned to method2. If the exception type is Exception3, it is caught by the catch clause for handling exception ex3 in method2. statement5 is skipped, and statement6 is executed.

- If the exception type is Exception2, method2 is aborted, the control is returned to method1, and the exception is caught by the catch clause for handling exception ex2 in method1. statement3 is skipped, and statement4 is executed.

- If the exception type is Exception1, method1 is aborted, the control is returned to the main method, and the exception is caught by the catch clause for handling exception ex1 in the main method. statement1 is skipped, and statement2 is executed.

- If the exception type is not Exception1, Exception2, or Exception3, the exception is not caught and the program terminates. statement1 and statement2 are not executed.

Same as Fig 15.3 in introjb3e p661

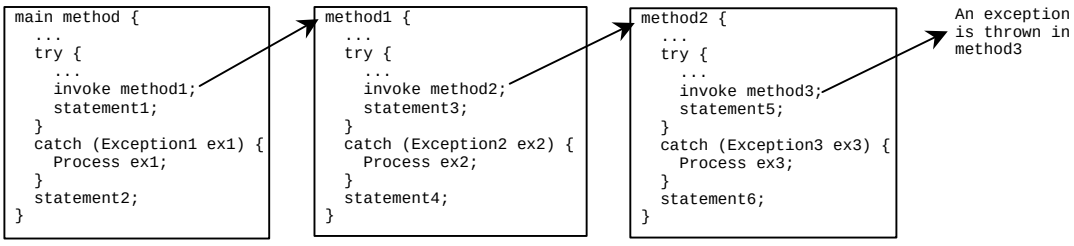


Figure 15.3

If an exception is not caught in the current method, it is passed to its caller. The process is repeated until the exception is caught or passed to the main method.

An exception object contains valuable information about the exception. You may use the following instance methods in the java.lang.Throwable class to get information regarding the exception.

[BL]public String getMessage()
Returns the detailed message of the Throwable object.

[BL]public String toString()
Returns the concatenation of three strings: (1) the full name of the exception class; (2) ": " (a colon and a space); (3) the getMessage() method.

[BL]public void printStackTrace()
Prints the Throwable object and its trace information on the console.

NOTE

Various exception classes can be derived from a common superclass. If a catch clause catches exception objects of a superclass, it can catch all the exception objects of the subclasses of that superclass.

NOTE

The order in which exceptions are specified in catch clauses is important. A compilation error will result if a catch clause for a superclass type appears before a catch clause for a subclass type. For example, the following ordering is erroneous, since RuntimeException is a subclass of Exception:

```
try {
  ...
}
catch (Exception ex) {
  ...
}
catch (RuntimeException ex) {
  ...
}
```

The correct ordering should be:

```
try {
  ...
}
catch (RuntimeException ex) {
  ...
}
catch (Exception ex) {
  ...
}
```

******End of NOTE***

NOTE

Java forces you to deal with checked exceptions. If a method declares a checked exception (i.e., an exception other than Error or

RuntimeException), you must place it in a try statement and handle it in order to avoid abnormal termination.

Example 15.1

Declaring, Throwing, and Catching Exceptions

Problem

This example demonstrates declaring, throwing, and catching exceptions by modifying the setRadius method in the Circle class defined in Example 6.5, "Using Instance and Static Variables and Methods." The new setRadius method throws an exception if the radius is negative.

Solution

Name the new circle class CircleWithException, which is the same except that the setRadius(double newRadius) method throws an IllegalArgumentException if the argument newRadius is negative.

******PD: Please add line numbers in the following code******

```
// CircleWithException.java: setRadius throws an exception
public class CircleWithException {
  /** The radius of the circle */
  private double radius;

  /** The number of the objects created */
  private static int numOfObjects = 0;

  /** Default constructor */
  public CircleWithException() {
    this(1.0);
  }

  /** Construct a circle with a specified radius */
  public CircleWithException(double newRadius) {
    setRadius(newRadius);
    numOfObjects++;
  }

  /** Return radius */
  public double getRadius() {
    return radius;
  }

  /** Set a new radius */
  public void setRadius(double newRadius)
    throws IllegalArgumentException {
    if (newRadius >= 0)
      radius =  newRadius;
    else
      throw new IllegalArgumentException(
        "Radius cannot be negative");
  }

  /** Return numOfObjects */
  public static int getNumOfObjects() {
    return numOfObjects;
  }
```

```
  /** Return the area of this circle */
  public double findArea() {
    return radius * radius * 3.14159;
  }
}
```

A test program that uses the new Circle class is given below. Figure 15.4 shows a sample run of the test program.

PD: Please add line numbers in the following code

```
// TestCircleWithException.java: Test exception handling
public class TestCircleWithException {
  /** Main method */
  public static void main(String[] args) {
    try {
      Circle c1 = new Circle(5);
      Circle c2 = new Circle(-5);
      Circle c3 = new Circle(0);
    }
    catch (IllegalArgumentException ex) {
      System.out.println(ex);
    }

    System.out.println("Number of objects created: " +
      Circle.getNumOfObjects());
  }
}
```

***Insert Figure 15.4

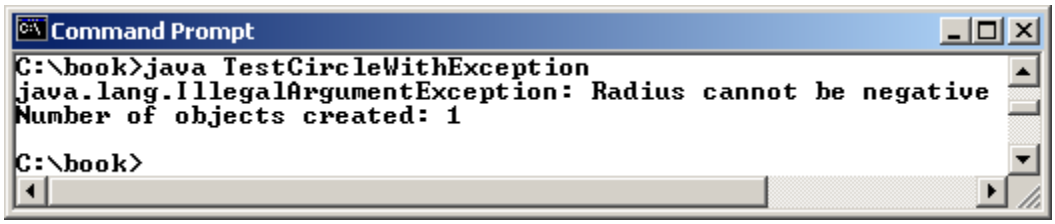


Figure 15.4

The exception is thrown when the radius is negative.

Review

The original CircleWithStaticVariable class remains intact except that the class name is changed to CircleWithException, a new constructor CircleWithException(newRadius) is added, and the setRadius method now declares an exception and throws it if the radius is negative.

The setRadius method declares to throw IllegalArgumentException in the method declaration (Lines 33-34 in CircleWithException.java). The CircleWithException class would still compile if the throws IllegalArgumentException clause were removed from the method declaration, since it is a subclass

of RuntimeException and every method can throw RuntimeException (unchecked exception) regardless of whether it is declared in the method header.

The test program creates three CircleWithException objects, c1, c2, and c3, to test how to handle exceptions. Invoking new CircleWithException(-5) (Line 9 in TestCircleWithException) causes the setRadius method to be invoked, which throws an IllegalArgumentException, because the radius is negative. In the catch clause, the type of the object ex is IllegalArgumentException, which matches the exception object thrown by the setRadius method. So this exception is caught by the catch clause.

The exception handler simply prints a short message, ex.toString() (Line 13), about the exception, using System.out.println(ex).

Note that the execution continues in the event of the exception. If the handlers had not caught the exception, the program would have abruptly terminated.

The test program would still compile if the try statement were not used, because the method throws an instance of IllegalArgumentException, a subclass of RuntimeException (unchecked exception). If a method throws an exception other than RuntimeException and Error, the method must be invoked within a try-catch block.

******End of Example***

Methods are executed on threads. If an exception occurs on a thread, the thread is terminated if the exception is not handled. However, the other threads in the application are not affected. There are several threads running to support a GUI application. A thread is launched to execute an event handler (e.g., the actionPerformed method for the ActionEvent). If an exception occurs during the execution of a GUI event handler, the thread is terminated if the exception is not handled. Interestingly, Java prints the error message on the console, but does not terminate the application. The program goes back to its user-interface-processing loop to run continuously. The following example demonstrates this.

Example 15.2

Exceptions in GUI Applications

Problem

Write a program that creates a user interface to perform integer divisions, as shown in Figure 15.5. The user enters two numbers in the text fields Number 1 and Number 2. The division of Number 1 and Number 2 is displayed in the Result field when the Divide button is clicked.

Solution

Complete the program as follows:

******PD: Please add line numbers in the following code******

```
// IntegerDivision.java: Perform division on two integers
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class IntegerDivision extends JApplet
    implements ActionListener {
  // Text fields for Number 1, Number 2, and Result
  private JTextField jtfNum1, jtfNum2, jtfResult;

  // Create the "Divide" button
  private JButton jbtDiv = new JButton("Divide");

  // Default Constructor
  public IntegerDivision() {
    // Panel p1 to hold text fields and labels
    JPanel p1 = new JPanel();
    p1.setLayout(new FlowLayout());
    p1.add(new JLabel("Number 1"));
    p1.add(jtfNum1 = new JTextField(3));
    p1.add(new JLabel("Number 2"));
    p1.add(jtfNum2 = new JTextField(3));
    p1.add(new JLabel("Result"));
    p1.add(jtfResult = new JTextField(4));
    jtfResult.setEditable(false);
    jtfResult.setHorizontalAlignment(SwingConstants.RIGHT);

    getContentPane().add(p1, BorderLayout.CENTER);
    getContentPane().add(jbtDiv, BorderLayout.SOUTH);

    // Register listener
    jbtDiv.addActionListener(this);
  }

  /** Handle ActionEvent from the Divide button */
  public void actionPerformed(ActionEvent e) {
    if (e.getSource() == jbtDiv) {
      // Get numbers
      int num1 = Integer.parseInt(jtfNum1.getText().trim());
      int num2 = Integer.parseInt(jtfNum2.getText().trim());

      int result = num1 / num2;

      // Set result in jtfResult
      jtfResult.setText(String.valueOf(result));
    }
  }
}
```

Run the program and enter any number in the Number 1 field and 0 in the Number 2 field; then click the Divide button (see Figure 15.5). You will see nothing in the Result field, but an error message will appear in the Output window, as shown in Figure 15.6. The GUI application continues.

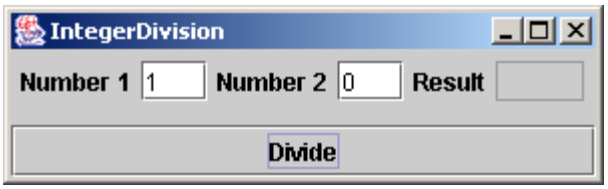


Figure 15.5

Since the divisor is 0 in the Number 2 field, a RuntimeException is thrown when the Divide button is clicked.

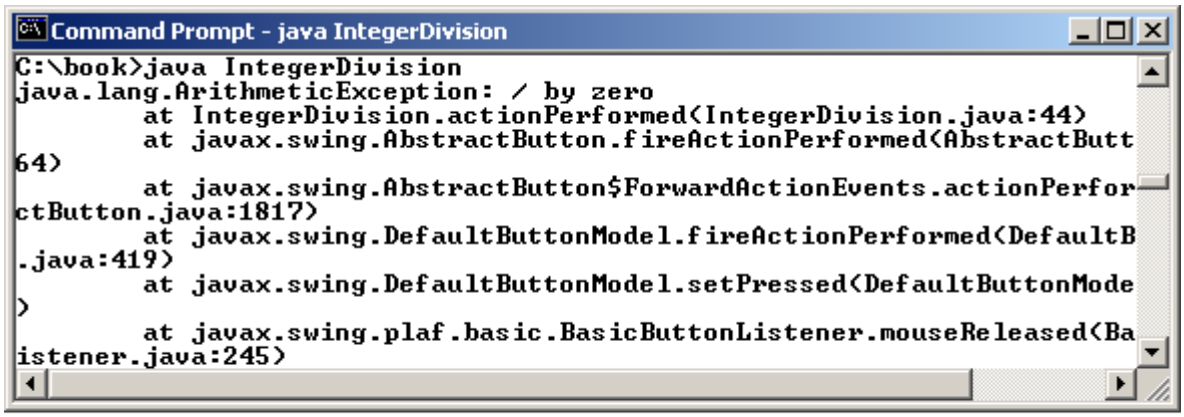


Figure 15.6

In GUI programs, if an exception is not caught, an error message appears in the console window.

Review

An ArithmeticException occurred during the execution of the actionPerformed method. The thread on which the method is executed is terminated, but the program continues to run.

If you add a try-catch block around the code in Lines 42-45, as shown below, the program will display a message dialog box in the case of a numerical error, as shown in Figure 15.7. No errors are reported because they are handled in the program.

```
        try {
            int result = num1 / num2;

            // Set result in jtfResult
            jtfResult.setText(String.valueOf(result));
```

```
    }
    catch (RuntimeException ex) {
      JOptionPane.showMessageDialog(this, ex.getMessage(),
        "Operation error", JOptionPane.ERROR_MESSAGE);
    }
```

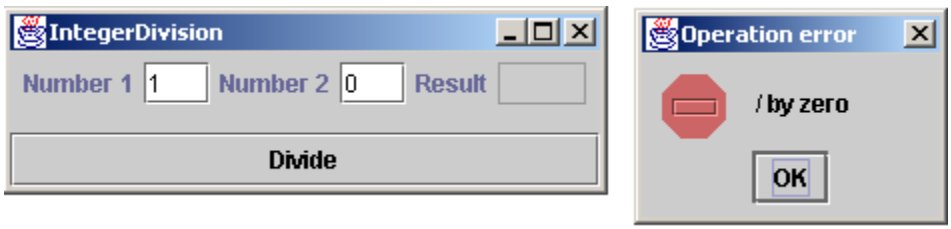


Figure 15.7

When you click the Divide button to divide a number by 0, a numerical exception occurs. The exception is displayed in the message dialog box.

15.4 Rethrowing Exceptions

When an exception occurs in a method, the method exits immediately if it does not catch the exception. If the method is required to perform some task before exiting, you can catch the exception in the method and then rethrow it to the real handler in a structure like the one given below:

```
try {
  statements;
}
catch (TheException ex) {
  perform operations before exits;
  throw ex;
}
```

The statement `throw ex` rethrows the exception so that other handlers get a chance to process the exception `ex`.

15.5 The `finally` Clause

Occasionally, you may want some code to be executed regardless of whether an exception occurs or is caught. Java has a `finally` clause that can be used to accomplish this objective. The syntax for the `finally` clause might look like this:

```
try {
  statements;
}
catch (TheException ex) {
  handling ex;
}
finally {
  finalStatements;
}
```

The code in the `finally` block is executed under all circumstances, regardless of whether an exception occurs in the `try` block or is caught. Consider three possible cases:

[BL]If no exception arises in the try block, finalStatements is executed, and the next statement after the try statement is executed.

[BL]If one of the statements causes an exception in the try block that is caught in a catch clause, the other statements in the try block are skipped, the catch clause is executed, and the finally clause is executed. If the catch clause does not rethrow an exception, the next statement after the try statement is executed. If it does, the exception is passed to the caller of this method.

[BL]If one of the statements causes an exception that is not caught in any catch clause, the other statements in the try block are skipped, the finally clause is executed, and the exception is passed to the caller of this method.

NOTE

The catch clause may be omitted when the finally clause is used.

15.6 When to Use Exceptions

Exception handling separates error-handling code from normal programming tasks, thus making programs easier to read and to modify. Be aware, however, that exception handling usually requires more time and resources because it requires instantiating a new exception object, rolling back the call stack, and propagating the exception through the chain of method invoked to search for the handler.

An exception occurs in a method. If you want the exception to be processed by its caller, you should create an exception object and throw it. If you can handle the exception in the method where it occurs, there is no need to throw it.

In general, common exceptions that may occur in multiple classes in a project are candidates for exception classes. Simple errors that may occur in individual methods are best handled locally without throwing out exceptions.

When should you use a try-catch block in the code? Use it when you have to deal with unexpected error conditions. Do not use a try-catch block to deal with simple, expected situations. For example, the following code

```
try {
  System.out.println(refVar.toString());
}
catch (NullPointerException ex) {
  System.out.println("refVar is null");
}
```

is better replaced by

```
if (refVar != null)
   System.out.println(refVar.toString());
else
   System.out.println("refVar is null");
```

Which situations are exceptional and which are expected is sometimes difficult to decide. The point is not to abuse exception handling as a way to deal with a simple logic test.

15.7 Creating Custom Exception Classes (Optional)

Java provides quite a few exception classes. Use them whenever possible instead of creating your own exception classes. However, if you run into a problem that cannot be adequately described by the predefined exception classes, you can create your own exception class, derived from Exception or from a subclass of Exception, such as IOException. This section shows how to create your own exception class.

Example 15.3

Creating Your Own Exception Classes

Problem

Create a Java applet for handling account transactions. The applet displays the account ID and balance, and lets the user deposit to or withdraw from the account. For each transaction, a message is displayed to indicate the status of the transaction: successful or failed. In case of failure, the reason for the failure is reported. A sample run of the program is shown in Figure 15.8.

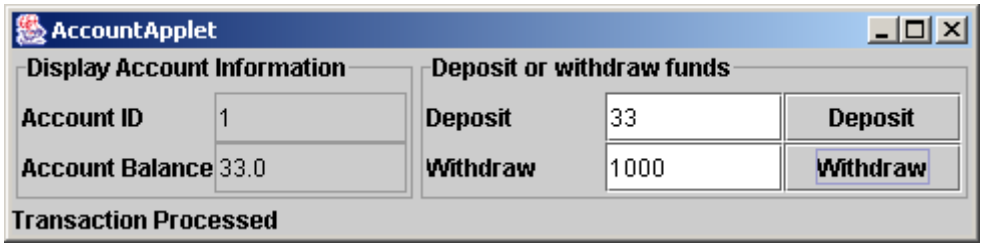


Figure 15.8

The program lets you deposit and withdraw funds, and displays the transaction status on the label.

If a transaction amount is negative, the program raises a negative-amount exception. If the account's

balance is less than the requested transaction amount, an insufficient-funds exception is raised.

The example consists of four classes: Account, NegativeAmountException, InsufficientAmountException, and AccountApplet. The Account class provides information and operations pertaining to the account. NegativeAmountException and InsufficientAmountException are the exception classes that deal with transactions of negative or insufficient amounts. The AccountApplet class utilizes all these classes to perform transactions, transferring funds among accounts. The relationships among these classes are shown in Figure 15.9.

***Same as Fig 15.9 in introjb3e p670

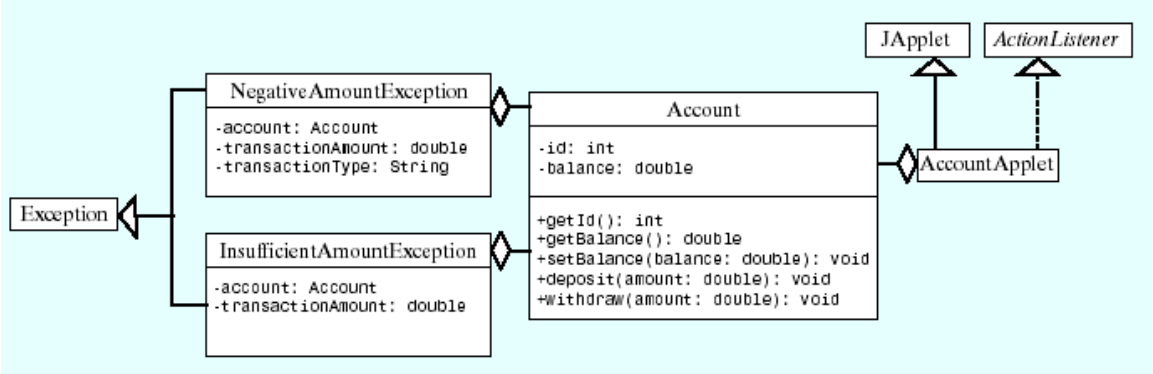


Figure 15.9

NegativeAmountException and InsufficientAmountException are subclasses of Exception that contain the account information, transaction amount, and transaction type for the failed transaction.

Solution

The code for the Account class follows. This class contains two data fields: id (for account ID) and balance (for current balance). The methods for Account are deposit and withdraw. Both methods will throw NegativeAmountException if the transaction amount is negative. The withdraw method will also throw InsufficientFundException if the current balance is less than the requested transaction amount.

PD: Please add line numbers in the following code

```
// Account.java: The class for describing an account
public class Account {
  // Two data fields in an account
  private int id;
  private double balance;

  /** Construct an account with specified id and balance */
  public Account(int id, double balance) {
    this.id = id;
    this.balance = balance;
  }

  /** Return id */
  public int getId() {
    return id;
  }

  /** Setter method for balance */
  public void setBalance(double balance) {
    this.balance = balance;
  }

  /** Return balance */
  public double getBalance() {
    return balance;
  }

  /** Deposit an amount to this account */
  public void deposit(double amount)
    throws NegativeAmountException {
    if (amount < 0)
      throw new NegativeAmountException
        (this, amount, "deposit");
    balance = balance + amount;
  }

  /** Withdraw an amount from this account */
  public void withdraw(double amount)
    throws NegativeAmountException, InsufficientFundException {
    if (amount < 0)
      throw new NegativeAmountException
        (this, amount, "withdraw");
    if (balance < amount)
      throw new InsufficientFundException(this, amount);
    balance = balance - amount;
  }
}
```

The NegativeAmountException exception class follows. It contains information about the attempted transaction type (deposit or withdrawal), the account, and the negative amount passed from the method.

******PD: Please add line numbers in the following code******

```
// NegativeAmountException.java: Negative amount exception
public class NegativeAmountException extends Exception {
  /** Account information to be passed to the handlers */
  private Account account;
  private double amount;
  private String transactionType;

  /** Construct an negative amount exception */
  public NegativeAmountException(Account account,
                                 double amount,
                                 String transactionType) {
    super("Negative amount");
    this.account = account;
    this.amount = amount;
    this.transactionType = transactionType;
  }
}
```

The InsufficientFundException exception class follows. It contains information about the account and the amount passed from the method.

PD: Please add line numbers in the following code

```
// InsufficientFundException.java: An exception class for describing
// insufficient fund exception
public class InsufficientFundException extends Exception {
  /** Information to be passed to the handlers */
  private Account account;
  private double amount;

  /** Construct an insufficient exception */
  public InsufficientFundException(Account account, double amount) {
    super("Insufficient amount");
    this.account = account;
    this.amount = amount;
  }

  /** Override the "toString" method */
  public String toString() {
    return "Account balance is " + account.getBalance();
  }
}
```

The AccountApplet class is given as follows:

PD: Please add line numbers in the following code

```
// AccountApplet.java: Use custom exception classes
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;

public class AccountApplet extends JApplet
  implements ActionListener {
  // Declare text fields
  private JTextField jtfID, jtfBalance, jtfDeposit, jtfWithdraw;

  // Declare Deposit and Withdraw buttons
  private JButton jbtDeposit, jbtWithdraw;

  // Create an account with initial balance $1000
  private Account account = new Account(1, 1000);

  // Create a label for showing status
  private JLabel jlblStatus = new JLabel();

  /** Initialize the applet */
  public void init() {
    // Panel p1 to group ID and Balance labels and text fields
    JPanel p1 = new JPanel();
    p1.setLayout(new GridLayout(2, 2));
    p1.add(new JLabel("Account ID"));
    p1.add(jtfID = new JTextField(4));
    p1.add(new JLabel("Account Balance"));
    p1.add(jtfBalance = new JTextField(4));
    jtfID.setEditable(false);
    jtfBalance.setEditable(false);
    p1.setBorder(new TitledBorder("Display Account Information"));

    // Panel p2 to group deposit amount and Deposit button and
    // withdraw amount and Withdraw button
    JPanel p2 = new JPanel();
    p2.setLayout(new GridLayout(2, 3));
    p2.add(new JLabel("Deposit"));
    p2.add(jtfDeposit = new JTextField(4));
    p2.add(jbtDeposit = new JButton("Deposit"));
    p2.add(new JLabel("Withdraw"));
    p2.add(jtfWithdraw = new JTextField(4));
    p2.add(jbtWithdraw = new JButton("Withdraw"));
    p2.setBorder(new TitledBorder("Deposit or withdraw funds"));
```

```
    // Place panels p1, p2, and label in the applet
    this.getContentPane().add(p1, BorderLayout.WEST);
    this.getContentPane().add(p2, BorderLayout.CENTER);
    this.getContentPane().add(jlblStatus, BorderLayout.SOUTH);

    // Refresh ID and Balance fields
    refreshFields();

    // Register listener
    jbtDeposit.addActionListener(this);
    jbtWithdraw.addActionListener(this);
  }

  /** Handle ActionEvent */
  public void actionPerformed(ActionEvent e) {
    if (e.getSource() == jbtDeposit) {
      try {
        double depositValue = (Double.valueOf(
          jtfDeposit.getText().trim())).doubleValue();
        account.deposit(depositValue);
        refreshFields();
        jlblStatus.setText("Transaction Processed");
      }
      catch (NegativeAmountException ex) {
        jlblStatus.setText("Negative Amount");
      }
    }
    else if (e.getSource() == jbtWithdraw) {
      try {
        double withdrawValue = (Double.valueOf(
          jtfWithdraw.getText().trim())).doubleValue();
        account.withdraw(withdrawValue);
        refreshFields();
        jlblStatus.setText("Transaction Processed");
      }
      catch (NegativeAmountException ex) {
        jlblStatus.setText("Negative Amount");
      }
      catch (InsufficientFundException ex) {
        jlblStatus.setText("Insufficient Funds");
      }
    }
  }

  /** Update the display for account balance */
  public void refreshFields() {
    jtfID.setText(String.valueOf(account.getId()));
    jtfBalance.setText(String.valueOf(account.getBalance()));
  }
}
```

Review

In the Account class, the deposit method (Line 29) throws NegativeAmountException (Line 32–33) if the amount to be deposited is less than 0. The withdraw method (Line 38) throws a NegativeAmountException if the amount to be withdrawn is less than 0 (Lines 41–42), and an InsufficientFundException if the amount to be withdrawn is less than the current balance (Line 44).

The user-defined exception class always extends Exception or a subclass of Exception. Therefore, both NegativeAmountException and InsufficientFundException extend Exception. In the Java API, each exception class has at least two constructors: a default constructor and a constructor with a string parameter for a detailed message. Line 12 in InsufficientFundException and Line 14 in NegativeAmountException invoke the constructor in

Exception using super("Insufficient amount") and super("Negative amount").

Storing relevant information in the exception object is useful, because the handler can then retrieve the information from the exception object. For example, NegativeAmountException contains the account, the amount, and the transaction type.

The AccountApplet class creates an applet with two panels (p1 and p2) (Lines 24, 36) and a label that displays messages. Panel p1 contains account ID and balance; panel p2 contains the action buttons for depositing and withdrawing funds.

With a click of the Deposit button, the amount in the Deposit text field is added to the balance. With a click of the Withdraw button, the amount in the Withdraw text field is subtracted from the balance.

For each successful transaction, the message Transaction Processed is displayed. For a negative amount, the message Negative Amount is displayed; for insufficient funds, the message Insufficient Funds is displayed.

15.8 Assertions

An *assertion* is a Java statement that enables you to assert an assumption about your program. An assertion contains a Boolean expression that should be true during program execution. Assertions can be used to ensure program correctness and avoid logic errors.

15.8.1 Declaring Assertions

An *assertion* is declared using the new Java keyword assert in JDK 1.4, as follows:

assert *assertion;* or

assert *assertion : detailMessage;*

where *assertion* is a Boolean expression and *detailMessage* is a primitive-type or an Object value.

When an assertion statement is executed, Java evaluates the assertion. If it is false, an AssertionError will be thrown. The AssertionError class has a default constructor and seven overloaded single-parameter constructors of type int, long, float, double, boolean, char, and Object. For the first assert statement with no detailed message, the default constructor of AssertionError is used. For the second assert statement with a detailed message, an appropriate

AssertionError constructor is used to match the data type of the message. Since AssertionError is a subclass of Error, when an assertion becomes false, the program displays a message on the console and exits.

Here is an example of using assertions.

PD: Please add line numbers in the following code

```
public class AssertionDemo {
  public static void main(String[] args) {
    int i; int sum = 0;
    for (i = 0; i < 10; i++) {
      sum += i;
    }
    assert i == 10;
    assert sum > 10 && sum < 5 * 10 : "sum is " + sum;
  }
}
```

The statement assert i == 10 asserts that i is 10 when the statement is executed. If i is not 10, an AssertionError is thrown. The statement assert sum > 10 && sum < 5 * 10 : "sum is " + sum asserts that sum > 10 and sum < 5 * 10. If false, an AssertionError with message "sum is " + sum is thrown.

Suppose you typed i < 100 instead of i < 10 by mistake in Line 4, the following AssertionError would be thrown:

```
Exception in thread "main" java.lang.AssertionError
        at AssertionDemo.main(AssertionDemo.java:7)
```

Suppose you typed sum += 1 instead of sum += i by mistake in Line 5, the following AssertionError would be thrown:

```
Exception in thread "main" java.lang.AssertionError: sum is 10
        at AssertionDemo.main(AssertionDemo.java:8)
```

15.8.2 Compiling Programs with Assertions

Since assert is a new Java keyword introduced in JDK 1.4, you have to compile the program using a JDK 1.4 compiler. Furthermore, you need to include the switch -source 1.4 in the compiler command, as follows:

```
javac -source 1.4 AssertionDemo.java
```

15.8.3 Running Programs with Assertions

By default, assertions are disabled at runtime. To enable them, use the switch -enableassertions, or -ea for short, as follows:

```
java -ea AssertionDemo
```

Assertions can be selectively enabled or disabled at class level or package level. The disable switch is -disableassertions, or -da for short. For example, the following command enables assertions in package package1 and disables assertions in class Class1.

```
java -ea:package1 -da:Class1 AssertionDemo
```

NOTE: In JBuilder, if you compile the program with the *Enable assert keyword* option checked in the Project Properties, the program will be executed with assertions.

15.8.4 Using Exception Handling or Assertions

Assertion should not be used to replace exception handling. Exception handling deals with unusual circumstances during program execution. Assertions are intended to ensure the correctness of the program. Exception handling addresses robustness whereas assertion addresses correctness. Like exception handling, assertions are not used for normal tests, but for internal consistency and validity checks. Assertions are checked at runtime and can be turned on or off at startup time.

Do not use assertions for argument checking in public methods. Valid arguments that may be passed to a public method are considered to be part of the method's contract. The contract must always be obeyed whether assertions are enabled or disabled. For example, the following code should be rewritten using exception handling as shown in Lines 28-35 in Circle.java in Example 15.1, "Declaring, Throwing, and Catching Exceptions."

```
public void setRadius(double newRadius) {
  assert newRadius >= 0;
  radius =  newRadius;
}
```

Use assertions to reaffirm assumptions. This will increase your confidence in the program's correctness. A common use of assertions is to replace assumptions with assertions in the code. For example, the following code

```
if (even) {
  ...
}
else { // even is false
  ...
}
```

can be replaced by

```
if (even) {
  ...
}
else {
  assert !even;
  ...
}
```

The following code

```
if (numOfDollars > 1) {
  ...
}
else if (numOfDollars == 1) {
  ...
}
```

can be replaced by

```
if (numOfDollars > 1) {
  ...
}
else if (numOfDollars == 1) {
  ...
}
else
  assert false : numOfDollars;
```

Another good use of assertions is to place them in a switch statement without a default case. For example,

```
switch (month) {
  case 1: ... ; break;
  case 2: ... ; break;
  ...
  case 12: ... ; break;
  default: assert false : "Invalid month: " + month
}
```

Key Classes and Methods

- **java.lang.Throwable** The root class for exceptions. The getMessage() method returns the detailed message of the exception, toString() returns the combination of the exception class name, colon (:), and getMessage(), and printStackTrace() displays track information on the console.
- **java.lang.Error** The base class for internal system errors.
- **java.lang.AssertionError** A subclass of Error thrown when an assertion failed.
- **java.lang.Exception** The base class for errors caused by the program and external circumstance.
- **java.lang.ClassNotFoundException** Thrown when a dependent class is not found at runtime.
- **java.lang.CloneNotSupportedException** Thrown if the object being cloned is not an instance of java.lang.Cloneable.
- **java.lang.RuntimeException** The base class for programming errors such as bad casting, array index error, and numeric error.
- **java.lang.NullPointerException** A runtime exception thrown when accessing an object through a null reference.
- **java.lang.IndexOutOfBoundsException** A runtime exception thrown when accessing an array object through a null reference.

Key Terms

- **exception** An unexpected event indicating that a program has failed in some way. Exceptions are represented by exception objects in Java. Exceptions can be handled in a try-catch block.
- **unchecked exception** Instances of RuntimeException and Error.
- **checked exception** Exceptions other than RuntimeException and Error.
- **assertion** A Java statement that enables you to assert an assumption about your program. An assertion contains a Boolean expression that should be true during program execution. Assertions can be used to ensure program correctness.

Chapter Summary

- When an exception occurs, Java creates an object that contains the information for the exception. You can use the information to handle the exception.

- A Java exception is an instance of a class derived from java.lang.Throwable. Java provides a number of predefined exception classes, such as Error, Exception, RuntimeException, ClassNotFoundException, NullPointerException, and ArithmeticException. You can also define your own exception class by extending Exception.

- Exceptions occur during the execution of a method. RuntimeException and Error are unchecked exceptions, and all other exceptions are checked exceptions.

- When declaring a method, you have to declare a checked exception if the method might throw that checked exception, thus telling the compiler what can go wrong.

- The keyword to declare an exception is throws, and the keyword to throw an exception is throw.

- To invoke the method that declares checked exceptions, you must enclose the method call in a try statement. When an exception occurs during the execution of the method, the catch clause catches and handles the exception.

- If an exception is not caught in the current method, it is passed to its caller. The process is repeated until the exception is caught or passed to the main method.

- If an exception of a subclass of Exception occurs in a GUI component, Java prints the error message on the console, but the program goes back to its user-interface-processing loop to run continuously. The exception is ignored.

- Various exception classes can be derived from a common superclass. If a catch clause catches exception objects of a superclass, it can catch all the exception objects of the subclasses of that superclass.

- The order in which the exceptions are specified in a catch clause is important. A compilation error will result if you do not specify an exception object of a class before an exception object of the superclass of that class.

- When an exception occurs in a method, the method exits immediately if it does not catch the exception. If the method is required to perform some task before exiting, you can catch the exception in the method and then rethrow it to the real handler.

- The code in the finally block is executed under all circumstances, regardless of whether an exception occurs in the try block or is caught.

- Exception handling separates error-handling code from normal programming tasks, thus making programs easier to read and to modify.

- Exception handling should not be used to replace simple tests. You should test simple exceptions whenever possible, and reserve exception handling for dealing with situations that cannot be handled with if statements.

- Exceptions address robustness, whereas assertions address correctness. Exceptions and assertions are not meant to substitute for simple tests. Avoid using exception handling if a simple if statement is sufficient. Never use assertions to check normal conditions.

REVIEW QUESTIONS

NOTE

In the following questions, assume that the divide method in Rational in Example 10.2, "The Rational Class," is modified as follows:

```
public Rational divide(Rational secondRational) throws Exception {
  if (secondRational.getNumerator() == 0)
    throw new Exception("Divisor cannot be zero");

  long n = numerator * secondRational.getDenominator();
  long d = denominator * secondRational.getNumerator();
```

```
    return new Rational(n, d);
  }
```

The divide method in the Rational class throws Exception if the divisor is 0.

******End of NOTE***

Sections 15.2 - 15.3

15.1
Describe the Java Throwable class, its subclasses, and the types of exceptions.

15.2
What is the purpose of declaring exceptions? How do you declare an exception, and where? Can you declare multiple exceptions in a method declaration?

15.10
What is a checked exception, and what is an unchecked exception?

15.4
How do you throw an exception? Can you throw multiple exceptions in one throw statement?

15.5
What is the keyword throw used for? What is the keyword throws used for?

15.6
What does the Java runtime system do when an exception occurs?

15.7
How do you catch an exception?

15.8
Suppose that statement2 causes an exception in the following try-catch block:

```
try {
  statement1;
  statement2;
  statement3;
}
catch (Exception1 ex1) {
}
catch (Exception2 ex2) {
}

statement4;
```

Answer the following questions:

[BL]Will statement3 be executed?

[BL] If the exception is not caught, will statement4 be executed?

[BL] If the exception is caught in the catch clause, will statement4 be executed?

[BL] If the exception is passed to the caller, will statement4 be executed?

15.9

What is displayed when the following program is run?

```
class Test {
  public static void main(String[] args) {
    try {
      Rational r1 = new Rational(3, 4);
      Rational r2 = new Rational(0, 1);
      Rational x = r1.divide(r2);

      int i = 0;
      int y = 2 / i;
      System.out.println("Welcome to Java");
    }
    catch (RuntimeException ex) {
      System.out.println("Integer operation error");
    }
    catch (Exception ex) {
      System.out.println("Rational operation error");
    }
  }
}
```

15.10

What is displayed when the following program is run?

```
class Test {
  public static void main(String[] args) {
    try {
      method();
      System.out.println("After the method call");
    }
    catch (RuntimeException ex) {
      System.out.println("Integer operation error");
    }
    catch (Exception e) {
      System.out.println("Rational operation error");
    }
  }

  static void method() throws Exception {
    Rational r1 = new Rational(3, 4);
    Rational r2 = new Rational(0, 1);
    Rational x = r1.divide(r2);

    int i = 0;
    int y = 2 / i;
    System.out.println("Welcome to Java");
  }
}
```

15.11

What is displayed when the following program is run?

```
class Test {
  public static void main(String[] args) {
    try {
      method();
      System.out.println("After the method call");
    }
    catch (RuntimeException ex) {
      System.out.println("Integer operation error");
    }
    catch (Exception ex) {
      System.out.println("Rational operation error");
    }
  }

  static void method() throws Exception {
    try {
      Rational r1 = new Rational(3, 4);
      Rational r2   = new Rational(0, 1);
      Rational x = r1.divide(r2);

      int i = 0;
      int y = 2 / i;
      System.out.println("Welcome to Java");
    }
    catch (RuntimeException ex) {
      System.out.println("Integer operation error");
    }
    catch (Exception ex) {
      System.out.println("Rational operation error");
```

```
    }
  }
}
```

15.12
If an exception were not caught in a non-GUI application, what would happen? If an exception were not caught in a GUI application, what would happen?

15.13
What does the method printStackTrace do?

15.14
Does the presence of a try-catch block impose overhead when no exception occurs?

Sections 15.4-15.5

15.15
Suppose that statement2 causes an exception in the following statement:

```
try {
  statement1;
  statement2;
  statement3;
}
catch (Exception1 ex1) {
}
catch (Exception2 ex2) {
}
catch (Exception3 ex3) {
  throw ex3;
}
finally {
  statement4;
};
statement5;
```

Answer the following questions:

[BL] Will statement5 be executed if the exception is not caught?

[BL]If the exception is of type Exception3, will statement4 be executed, and will statement5 be executed?

15.16
What is displayed when the following program is run?

```
class Test {
  public static void main(String[] args) {
    try {
      method();
      System.out.println("After the method call");
    }
    catch (RuntimeException ex) {
      System.out.println("Integer operation error");
    }
    catch (Exception ex) {
      System.out.println("Rational operation error");
    }
  }

  static void method() throws Exception {
    try {
      Rational r1 = new Rational(3, 4);
      Rational r2   = new Rational(0, 1);
      Rational x = r1.divide(r2);

      int i = 0;
      int y = 2 / i;
      System.out.println("Welcome to Java");
    }
    catch (RuntimeException ex) {
      System.out.println("Integer operation error");
    }
```

```
    catch (Exception ex) {
      System.out.println("Rational operation error");
      throw ex;
    }
  }
}
```

Section 15.8 Assertions

15.17
What is assertion for? How do you declare assertions? How do you compile code with assertions? How do you run programs with assertions?

15.18
What happens when you run the following code?

```
public class Test {
  public static void main(String[] args) {
    int i; int sum = 0;
    for (i = 0; i < 11; i++) {
      sum += i;
    }
    assert i == 10: "i is " + i;
  }
}
```

PROGRAMMING EXERCISES

Sections 15.2 – 15.3

15.1

(Revising Example 7.5 "Passing Command-Line Arguments") Example 7.5 is a simple command-line calculator. Note that the program terminates if any operand is non-numeric. Write a program with an exception handler that deals with non-numeric operands; then write another program without using an exception handler to achieve the same objective. Your program should display a message that informs the user of the wrong operand type before exiting (see Figure 15.10).

******Same as Fig 15.11 in introjb3e p683***

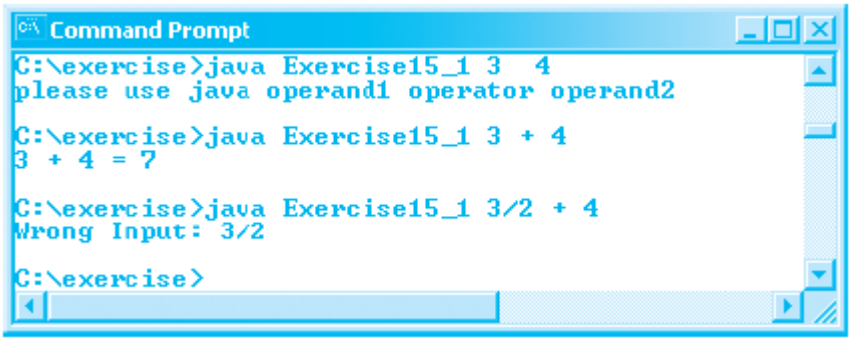


Figure 15.10

The program performs arithmetic operations and detects input errors.

15.2

(Revising Example 15.2 "Exceptions in GUI Applications") Example 15.2 is a GUI calculator. Note that if Number 1 or Number 2 were a non-numeric string, the program would report exceptions. Modify the program with an exception handler to catch ArithmeticException (e.g., divided by 0) and NumberFormatException (e.g., input is not an integer), and display the errors in a message dialog box, as shown in Figure 15.11.

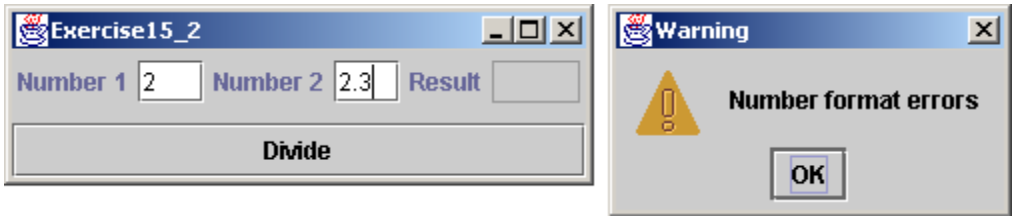


Figure 15.11

The program displays an error message in the dialog box if the number is not well formatted.

15.**3**

(Handling ArrayIndexOutBoundsException) Write a program that meets the following requirements:

[BL] Create an array with one hundred randomly chosen integers.

[BL] Create a text field to enter an array index and another text field to display the array element at the specified index (see Figure 15.12).

[BL] Create a Show Element button to cause the array element to be displayed. If the specified index is out of bounds, display the message Out of Bound.

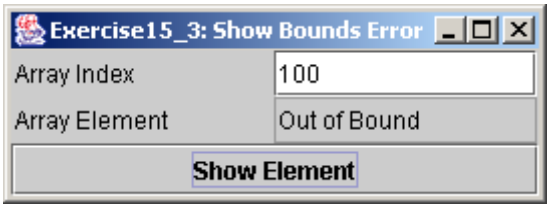


Figure 15.12

The program displays the array element at the specified index or displays the message Out of Bound *if the index is out of bounds.*

15.**4**

(Revising Example 6.7 "The Loan Class") Modify the Loan class in Example 6.7 to throw IllegalArgumentException if the loan amount, interest rate, or number of years is less than or equal to zero.

15.**5**

(The IllegalTriangleException class) Exercise 8.1 defined the Triangle class with three sides. In a triangle, the sum of any two sides is greater than the other side. The Triangle class must adhere to this rule. Create the IllegalTriangleException class, and modify the constructor to throw an IllegalTriangleException object if a triangle is created with sides that violate the rule, as follows:

```
/** Construct a triangle with the specified sides */
public Triangle(double side1, double side2, double side3)
  throws IllegalTriangleException {
  // Implement it
}
```