

Part IV: Developing Comprehensive Projects

This part of the book is devoted to several advanced features of Java programming. The subjects treated include the use of exception handling to make programs robust, the use of internationalization support to develop projects for international audiences, the use of multithreading to make programs more responsive and interactive, the incorporation of sound and images to make programs user-friendly, the use of input and output to manage and process large quantities of data, and the creation of client/server applications with Java networking support. You will learn how to use these features to develop comprehensive programs.

Chapter 13	Exception Handling
Chapter 14	Internationalization
Chapter 15	Multithreading
Chapter 16	Multimedia
Chapter 17	Input and Output
Chapter 18	Networking
Chapter 19	Java Data Structures

Exception Handling

Objectives

- Understand the concept of exception handling.
- Become familiar with exception types.
- Claim exceptions in a method.
- Throw exceptions in a method.
- Use the try-catch block to handle exceptions.
- Create your own exception classes.
- Rethrow exceptions in a try-catch block.
- Use the finally clause in a try-catch block.
- Know when to use exceptions.

Introduction

Programming errors are unavoidable, even for experienced programmers. In Chapter 2, "Primitive Data Types and Operations," you learned that there are three categories of errors: compilation errors, runtime errors, and logic errors. *Compilation 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 Chapter 2 to find logic errors. This chapter deals with runtime errors.

Runtime errors cause *exceptions*: events that occur during the execution of a program and disrupt the normal flow of control. A program that does not provide the 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.

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

For example, the following program terminates abnormally, because the divisor is 0, which causes a numerical error.

```
public class Test
{
    public static void main(String[] args)
    {
        System.out.println(3/0);
    }
}
```

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.

```
public class Test
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println(3/0);
        }
        catch (Exception ex)
        {
            System.out.println("Error: " + ex.getMessage());
        }

        System.out.println("Execution continues");
    }
}
```

This chapter introduces Java's exception-handling model. The chapter covers exception types, claiming exceptions, throwing exceptions, catching exceptions, creating exception classes, rethrowing exceptions, and the finally clause.

Exceptions and Exception Types

Runtime errors occur for various reasons. For example, the user may enter an invalid input, 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 graphics programming. (In Chapter 10, "Getting Started with Graphics Programming," you learned the events used in

graphics.) An event may be ignored in graphics programming, but an exception cannot be ignored. In graphics 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. However, when an exception occurs, the program may terminate if no handler can be used to deal with the exception.

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 graphics 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 13.1 shows some of Java's predefined exception classes.

*****Insert Figure 13.1 (Same as Figure 11.1 in the 3rd Edition, p480)**

Figure 13.1

The exceptions are instances of the classes shown in this diagram.

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

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 LinkageError, VirtualMachineError, and AWTError. Subclasses of LinkageError indicate that a class has some dependency on another class, but that the latter class has changed incompatibly after the compilation of the former class. Subclasses of VirtualMachineError indicate that the Java Virtual Machine is broken or has run out of the resources necessary for it to continue operating. AWTError is caused by a fatal error in the graphics programs.

The Exception class describes errors caused by your program and external circumstances. These errors can be caught and handled by your program. Exception has many subclasses.

Examples are ClassNotFoundException, CloneNotSupportedException, IOException, RuntimeException, and AWTException,

The ClassNotFoundException is raised if you attempt to use a class that does not exist. It would occur, for example, if you tried to run a nonexistent class using the **java** command. The CloneNotSupportedException is raised if you attempt to clone an object whose defining class does not implement the Cloneable interface. Cloning objects were introduced in Chapter 8, "Class Inheritance and Interfaces."

The RuntimeException class describes programming errors, such as bad casting, accessing an out-of-bound array, and numeric errors. Examples of subclasses of RuntimeException are ArithmeticException, NullPointerException, and IndexOutOfBoundsException.

ArithmeticException is for integer arithmetic. Java deals with integer arithmetic differently from floating-point arithmetic. Dividing by zero or modulus by zero is invalid for integer arithmetic and throws ArithmeticException. Floating-point arithmetic does not throw exceptions. For floating-point arithmetic, dividing by zero overflows to infinity See Appendix I, "Special Floating-Point Values," for a discussion on special values for floating-point arithmetic.

The IOException class describes errors 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, and FileNotFoundException.

The AWTException class describes exceptions in graphics programming.

Understanding Exception Handling

Java's exception-handling model is based on three operations: *claiming an exception*, *throwing an exception*, and *catching an exception*, as shown in Figure 13.2.

*****Insert Figure 13.2 (Same as Figure 11.2 in the 3rd Edition, p481)**

Figure 13.2 *The exception handling in Java consists of claiming exception, throwing exception, and catching and processing exceptions.*

In Java, the statement currently being executed belongs either to the main method or to a method invoked by another method. The Java interpreter invokes the main method for a Java application, and the Web browser invokes the init method for a Java applet. In general, every method must state the types of exceptions it might encounter. This process is called *claiming an exception*, which simply tells the compiler what might go wrong.

When a statement causes errors, the method containing the statement creates an exception object and passes it to the system. The exception object contains information about the exception, including its type and the state of the program when the error occurred. This process is called *throwing an exception*.

After a method throws an exception, the Java runtime system begins the process of finding the code to handle the error. The code that handles the error is called the *exception handler*; it is found by searching backward through a chain of method calls, starting from the current method. The handler must match the type of exception thrown. If no handler is found, the program terminates. The process of finding a handler is called *catching an exception*.

Claiming Exceptions

To claim an exception is to declare in a method what might go wrong when the method is executing. Because system errors and runtime errors can happen to any code, Java does not require you to claim Error and RuntimeException explicitly in the method. However, all the other exceptions must be explicitly claimed in the method declaration if they are thrown by the method.

To claim an exception in a method, you 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:

```
MethodDeclaration throws Exception1, Exception2, ..., ExceptionN
```

Throwing Exceptions

In the method that has claimed the exception, you can throw an object of the exception if the exception arises. The following is the syntax to throw an exception:

```
throw new TheException();
```

Or if you prefer, you can use the following:

```
TheException ex = new TheException();  
throw ex;
```

NOTE: The keyword to claim an exception is throws, and the keyword to throw an exception is throw.

A method can only throw the exceptions claimed in the method declaration or throw Error, RuntimeException, or subclasses of Error and RuntimeException. For example, the method cannot throw IOException if it is not claimed in the method declaration, but a method can always throw RuntimeException or a subclass of it even if it is not claimed by the method.

Catching Exceptions

You now know how to claim an exception and how to throw an exception. Next, you will learn how to handle exceptions. When calling a method that explicitly claims an exception, you must use the try-catch block to wrap the statement, as shown in the next few lines:

```
try  
{  
    statements; // Statements that may throw exceptions  
}  
catch (Exception1 ex)  
{  
    handler for exception1;  
}  
catch (Exception2 ex)  
{  
    handler for exception2;  
}  
...  
catch (ExceptionN ex)  
{  
    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 and starts to search for a handler for the exception. If the exception type matches one listed in a catch clause, the code in the catch clause is executed. If the exception type does not match any exception in the catch clauses, Java exits this method, passes the exception to the method that invoked this 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.

Consider the scenario in Figure 13.3. Suppose that an exception occurs in the try-catch block that contains a call to method3. If the exception type is Exception3, it is caught by the catch clause for handling exception ex3. If the exception type is Exception2, it is caught by the catch clause for handling exception ex2. If the exception type is Exception1, it is caught by the catch clause for handling exception ex1 in the main method. If the exception type is not Exception1, Exception2, or Exception3, the program immediately terminates.

*****Insert Figure 13.3 (Same as Figure 11.3 in the 3rd Edition, p483)**

Figure 13.3

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.

If the exception type is Exception3, statement3 is skipped. If the exception type is Exception2, statement2 and statement3 are skipped. If the exception type is Exception1, statement1, statement2, and statement3 are skipped.

NOTE: If an exception of a subclass of Exception occurs in a graphics program, 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.

The 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.

- public String getMessage()

This returns the detailed message of the Throwable object.

- public String toString()

This returns a short description of the Throwable object, whereas getMessage() returns a detailed message.

- public String getLocalizedMessage()

This returns a localized description of the Throwable object. Subclasses of Throwable can override this method in order to produce a locale-specific message. For subclasses that do not override this method, the default implementation returns the same result as getMessage(). Locale-specific issues are addressed in Chapter 14, "Internationalization."

- public void printStackTrace()

This 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.

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 the exception object of the superclass of that class.

NOTE: Java forces you to deal with exceptions. A method that claims an exception other than Error or RuntimeException must be placed in a try statement and be handled to avoid abnormal termination.

Example 13.1 Claiming, Throwing, and Catching Exceptions

This example demonstrates claiming, throwing, and catching exceptions by modifying the Rational class defined in Example 9.2, "Using the Rational Class" (see Chapter 9, "Object-Oriented Software Development") so that it can handle the zero-denominator exception.

You create a new Rational class, which is the same except that the divide method throws a zero-denominator exception if the client attempts to call the method with a zero denominator. The new Rational class is shown below:

```
// Rational.java: Define a rational number and its associated
// operations such as add, subtract, multiply, and divide
// the divide method throws an exception
package chapter13;

public class Rational extends Number implements Comparable
{
```

```

// Data fields for numerator and denominator
private long numerator = 0;
private long denominator = 1;

/**Default constructor*/
public Rational()
{
    this(0, 1);
}

/**Construct a rational with specified numerator and denominator*/
public Rational(long numerator, long denominator)
{
    long gcd = gcd(numerator, denominator);
    this.numerator = numerator/gcd;
    this.denominator = denominator/gcd;
}

/**Find GCD of two numbers*/
private long gcd(long n, long d)
{
    long t1 = Math.abs(n);
    long t2 = Math.abs(d);
    long remainder = t1%t2;

    while (remainder != 0)
    {
        t1 = t2;
        t2 = remainder;
        remainder = t1%t2;
    }

    return t2;
}

/**Return numerator*/
public long getNumerator()
{
    return numerator;
}

/**Return denominator*/
public long getDenominator()
{
    return denominator;
}

/**Add a rational number to this rational*/
public Rational add(Rational secondRational)
{
    long n = numerator*secondRational.getDenominator() +
        denominator*secondRational.getNumerator();
    long d = denominator*secondRational.getDenominator();
    return new Rational(n, d);
}

/**Subtract a rational number from this rational*/
public Rational subtract(Rational secondRational)
{
    long n = numerator*secondRational.getDenominator()
        - denominator*secondRational.getNumerator();
    long d = denominator*secondRational.getDenominator();
    return new Rational(n, d);
}

/**Multiply a rational number to this rational*/
public Rational multiply(Rational secondRational)
{
    long n = numerator*secondRational.getNumerator();
    long d = denominator*secondRational.getDenominator();
    return new Rational(n, d);
}

/**Divide a rational number from this rational*/
public Rational divide(Rational secondRational)
    throws RuntimeException
{

```

```

        if (secondRational.getNumerator() == 0)
            throw new RuntimeException("Denominator cannot be zero");

        long n = numerator*secondRational.getDenominator();
        long d = denominator*secondRational.getNumerator();
        return new Rational(n, d);
    }

    /**Override the toString() method*/
    public String toString()
    {
        if (denominator == 1)
            return numerator + "";
        else
            return numerator + "/" + denominator;
    }

    /**Override the equals method*/
    public boolean equals(Object parm1)
    {
        /**@todo: Override this java.lang.Object method*/
        if ((this.subtract((Rational) (parm1))).getNumerator() == 0)
            return true;
        else
            return false;
    }

    /**Override the intValue method*/
    public int intValue()
    {
        /**@todo: implement this java.lang.Number abstract method*/
        return (int)doubleValue();
    }

    /**Override the floatValue method*/
    public float floatValue()
    {
        /**@todo: implement this java.lang.Number abstract method*/
        return (float)doubleValue();
    }

    /**Override the doubleValue method*/
    public double doubleValue()
    {
        /**@todo: implement this java.lang.Number abstract method*/
        return numerator*1.0/denominator;
    }

    /**Override the longValue method*/
    public long longValue()
    {
        /**@todo: implement this java.lang.Number abstract method*/
        return (long)doubleValue();
    }

    /**Override the compareTo method*/
    public int compareTo(Object o)
    {
        /**@todo: Implement this java.lang.Comparable method*/
        if ((this.subtract((Rational)o)).getNumerator() > 0)
            return 1;
        else if ((this.subtract((Rational)o)).getNumerator() < 0)
            return -1;
        else
            return 0;
    }
}

```

A test program that uses the new Rational class is given as follows. Figure 13.4 shows a sample run of this test program.

```

// TestRationalException.java: Catch and handle exceptions
package chapter13;

```

```

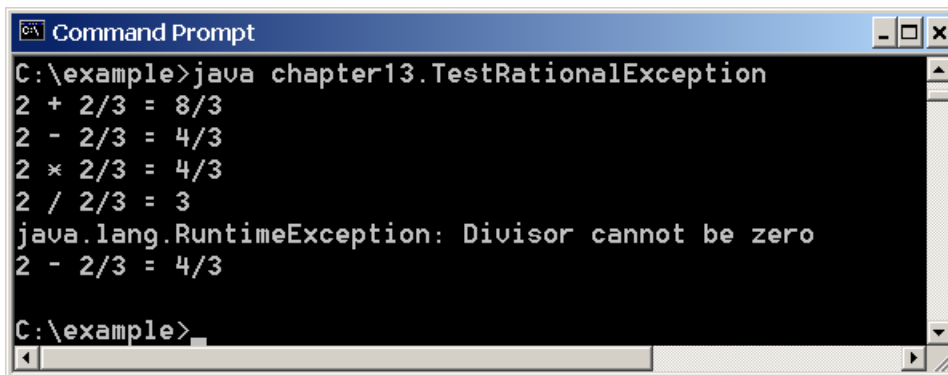
public class TestRationalException
{
    // Main method
    public static void main(String[] args)
    {
        // Create three rational numbers
        Rational r1 = new Rational(4, 2);
        Rational r2 = new Rational(2, 3);
        Rational r3 = new Rational(0, 1);

        try
        {
            System.out.println(r1+" + " + r2 +" = " + r1.add(r2));
            System.out.println(r1+" - " + r2 +" = " + r1.subtract(r2));
            System.out.println(r1+" * " + r2 +" = " + r1.multiply(r2));
            System.out.println(r1+" / " + r2 +" = " + r1.divide(r2));
            System.out.println(r1+" / " + r3 +" = " + r1.divide(r3));
            System.out.println(r1+" + " + r2 +" = " + r1.add(r2));
        }
        catch (Exception ex)
        {
            System.out.println(ex);
        }

        // Display the result
        System.out.println(r1 + " - " + r2 + " = " + r1.subtract(r2));
    }
}

```

***Insert Figure 13.4



```

C:\example>java chapter13.TestRationalException
2 + 2/3 = 8/3
2 - 2/3 = 4/3
2 * 2/3 = 4/3
2 / 2/3 = 3
2 / 2/3 = 3
java.lang.RuntimeException: Divisor cannot be zero
2 - 2/3 = 4/3
C:\example>

```

Figure 13.4

The exception is raised when the divisor is zero.

Example Review

The original Rational class remains intact except for the divide method. The divide method now claims an exception and throws it if the divisor is zero.

The divide method claims the exception to be an instance of RuntimeException by using throws RuntimeException in the method signature. The method throws the exception by using the following statement:

```
throw new RuntimeException("Denominator cannot be zero");
```

The program creates two Rational numbers, r1 and r2, to test numeric methods (add, subtract, multiply, and divide) on rational numbers.

Invoking the divide method with divisor 0 causes the method to throw an exception object. In the catch clause, the type of the object ex is RuntimeException, which matches the object thrown by the divide method. So this exception is caught by the catch clause.

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

Note that the execution continues in the event of the zero denominator. 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 divide method throws RuntimeException. If a method throws an exception other than RuntimeException and Error, the method must be invoked within a try statement.

Example 13.2 Exceptions in GUI Applications

Here Example 11.11, "Using Menus" (from Chapter 11, "Creating User Interfaces"), is used to demonstrate the effect of exceptions in GUI applications. 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 13.5). You will see nothing in the Result field, but an error message will appear on the console, as shown in Figure 13.6. The GUI application continues.

***Insert Figure 13.5 (Same as Figure 11.5 in the 3rd Edition, p487)

Figure 13.5

In GUI programs, if an exception of the Exception class is not caught, it is ignored, and the program continues.

***Insert Figure 13.6 (Same as Figure 11.6 in the 3rd Edition, p488)

Figure 13.6

In GUI programs, if an exception of the Exception class is not caught, an error message appears on the console.

Example Review

If an exception of the Exception type is not caught when a Java graphics program is running, an error message is displayed on the console, but the program continues to run.

If you rewrite the calculate method in the MenuDemo program of Example 11.11 with a try-catch block to catch RuntimeException as follows, the program will display a message dialog box in the case of a numerical error, as shown in Figure 13.7. No errors are shown on the console because they are handled in the program.

```
// Calculate and show the result in jtfResult
private void calculate(char operator)
{
    // Obtain Number 1 and Number 2
    int num1 = (Integer.parseInt(jtfNum1.getText().trim()));
    int num2 = (Integer.parseInt(jtfNum2.getText().trim()));
    int result = 0;

    try
    {
        // Perform selected operation
        switch (operator)
        {
            case '+': result = num1 + num2;
                      break;
            case '-': result = num1 - num2;
                      break;
            case '*': result = num1 * num2;
                      break;
            case '/': 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);
    }
}
```

*****Insert Figure 13.7 (Same as Figure 11.7 in the 3rd Edition, p489)**

Figure 13.7

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.

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.
The finally Clause

Occasionally, you may want some code to be executed regardless of whether the exception occurs or whether it 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 whether it 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.

[BX] 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.

Cautions When Using 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 errors to the calling methods.

Exception handling should not be used to replace simple tests. You should test simple exceptions whenever possible, and let exception handling deal with circumstances that cannot be handled with if statements. Do not use exception handling to validate user input. The input can be validated with simple if statements.

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 13.3 Creating Your Own Exception Classes

This program creates 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 failure reason is reported. A sample run of the program is shown in Figure 13.8.

******Insert Figure 13.8***

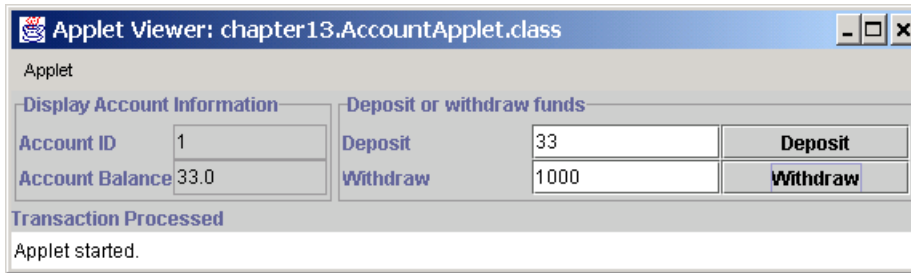


Figure 13.8

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 the 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 13.9.

*****Insert Figure 13.9**

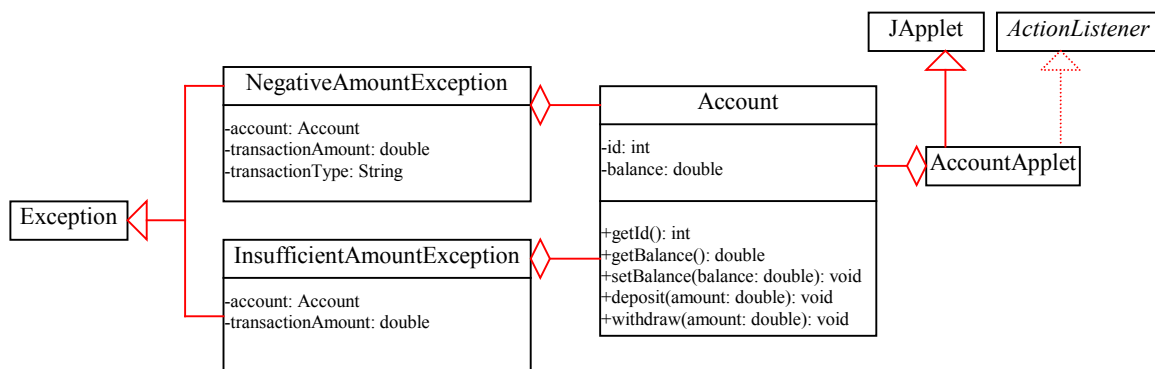


Figure 13.9

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

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.

```
// Account.java: The class for describing an account
package chapter13;

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.

```
// NegativeAmountException.java: Negative amount exception
package chapter13;

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.

```
// InsufficientFundException.java: An exception class for describing
// insufficient fund exception
package chapter13;

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:

```
// AccountApplet.java: Use custom exception classes
package chapter13;

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 evt)
    {
        String actionCommand = evt.getActionCommand();
        if (evt.getSource() instanceof JButton)
            if ("Deposit".equals(actionCommand))
            {
                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 ("Withdraw".equals(actionCommand))
            {
                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()));
}
}

```

Example Review

In the Account class, the deposit method throws NegativeAmountException if the amount to be deposited is less than 0. The withdraw method throws a NegativeAmountException if the amount to be withdrawn is less than 0, and throws an InsufficientFundException if the amount to be withdrawn is less than the current balance.

The user-defined exception class always extends Exception or a subclass of Exception. Therefore, both NegativeAmountException and InsufficientFundException extend Exception.

Storing relevant information in the exception object is useful, enabling the handler to 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) 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.

Chapter Summary

In this chapter, you learned how Java handles exceptions. 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. You can create your own exception classes by extending Throwable or a subclass of Throwable. The Java system provides a number of predefined exception classes, such as Error, Exception, RuntimeException, and IOException. You can also define your own exception class.

Exceptions occur during the execution of a method. When defining the method, you have to claim an exception if the method might throw that exception, thus telling the compiler what can go wrong.

To use the method that claims exceptions, you need to enclose the method call in the try statement. When the exception occurs during the execution of the method, the catch clause catches and handles the exception.

Exception handling takes time because it requires the instantiation of a new exception object. Exceptions are not meant to substitute for simple tests. Avoid using exception handling if an alternative solution can be found. Using an alternative would significantly improve the program's performance.

Review Questions

- 13.1 Describe the Java Throwable class, its subclasses, and the types of exceptions.
- 13.2 What is the purpose of claiming exceptions? How do you claim an exception, and where? Can you claim multiple exceptions in a method declaration?
- 13.3 How do you throw an exception? Can you throw multiple exceptions in one throw statement?
- 13.4 What is the keyword throw used for? What is the keyword throws used for?

- 13.5 What does the Java runtime system do when an exception occurs?
- 13.6 How do you catch an exception?
- 13.7 Does the presence of the try-catch block impose overhead when no exception occurs?
- 13.8 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?

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

- 13.9 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?

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

13.10 What is wrong in the following program?

```
class TestRationalWithException
{
    public static void main(String[] args)
    {
        Rational r1 = new Rational(4,2);
        Rational r2 = new Rational(2,3);
        Rational r3 = new Rational(0,1);

        try
        {
            System.out.println(
                r1 + " + " + r2 + " = " + r1.add(r2));
            System.out.println(
                r1 + " - " + r2 + " = " + r1.subtract(r2));
            System.out.println(
                r1 + " * " + r2 + " = " + r1.multiply(r2));
            System.out.println(
                r1 + " / " + r2 + " = " + r1.add(r2));
        }
    }
}
```

13.11 What is displayed on the console when the following program is run?

```
class Test
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println("Welcome to Java");
        }
        finally
        {
            System.out.println("The finally clause is executed");
        }
    }
}
```

13.12 What is displayed on the console when the following program is run?

```
class Test
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println("Welcome to Java");
            return;
        }
        finally
        {
        }
    }
}
```



```

        System.out.println("The finally clause is executed");
    }
}

```

13.13 What is displayed on the console when the following program is run?

```

class Test
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println("Welcome to Java");
            int i = 0;
            int y = 2/i;
            System.out.println("Welcome to HTML");
        }
        finally
        {
            System.out.println("The finally clause is executed");
        }
    }
}

```

13.14 What is displayed on the console when the following program is run?

```

class Test
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println("Welcome to Java");
            int i = 0;
            double y = 2.0/i;
            System.out.println("Welcome to HTML");
        }
        finally
        {
            System.out.println("The finally clause is executed");
        }
    }
}

```

13.15 What is displayed on the console when the following program is run?

```

class Test
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println("Welcome to Java");
            int i = 0;
            int y = 2/i;
            System.out.println("Welcome to HTML");
        }
        catch (RuntimeException ex)
        {
            System.out.println("RuntimeException caught");
        }
        finally
        {
            System.out.println("The finally clause is executed");
        }
    }
}

```

13.16 What is displayed on the console when the following program is run?

```
class Test
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println("Welcome to Java");
            int i = 0;
            int y = 2/i;
            System.out.println("Welcome to HTML");
        }
        catch (RuntimeException ex)
        {
            System.out.println("RuntimeException caught");
        }
        finally
        {
            System.out.println("Finally clause is executed");
        }

        System.out.println("End of the block");
    }
}
```

13.17 What is displayed on the console when the following program is run?

```
class Test
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println("Welcome to Java");
            int i = 0;
            int y = 2/i;
            System.out.println("Welcome to HTML");
        }
        finally
        {
            System.out.println("The finally clause is executed");
        }

        System.out.println("End of the block");
    }
}
```

In the following questions, assume that the modified Rational given in Example 13.1 is used.

13.18 What is wrong with the following code?

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

```

    }
    catch (RuntimeException ex)
    {
        System.out.println("Integer operation error");
    }
}

```

13.19 What is displayed on the console 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");
        }
    }
}

```

13.20 What is displayed on the console 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");
    }
}

```

13.21 What is displayed on the console 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");
        }
    }
}

```

13.22 What is displayed on the console 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;
    }
}
}

```

13.23 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?

13.24 What does the method `printStackTrace` do?

Programming Exercises

13.1 Example 7.5, "Using Command-Line Parameters," in Chapter 7, "Strings," is a simple command-line calculator. Note that the program terminates if any operand is non-numeric. Write a program with an exception handler to deal with non-numeric operands; then write another program without using an exception handler to achieve the same objective. Your program should display a message to inform the user of the wrong operand type before exiting (see Figure 13.10).

***Insert Figure 13.10

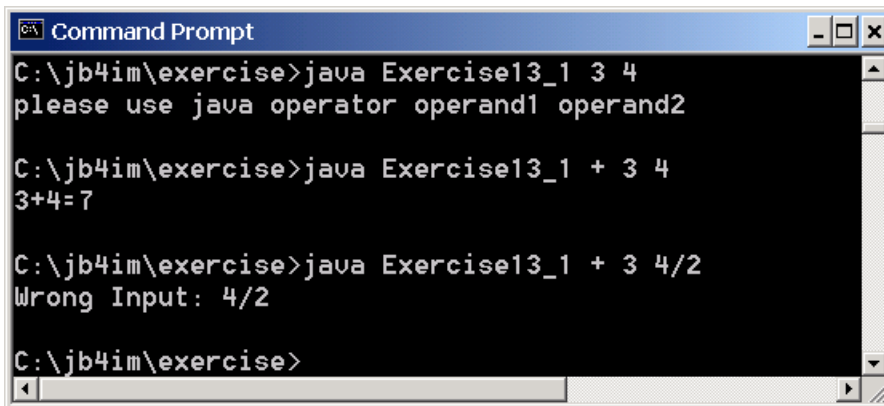


Figure 13.10

The program performs arithmetic operations and detects input errors.

- 11.2 Example 11.11, "Using Menus," is a GUI calculator. Note that if Number 1 or Number 2 were a non-numeric string, the program would display errors on the console. Modify the program with an exception handler to catch ArithmeticException (i.e., divided by 0) and NumberFormatException (i.e., input is not an integer) and display the errors in a message dialog box, as shown in Figure 13.11.

*****Insert Figure 13.11 (Same as Figure 11.11 in the 3rd Edition, p504)**

Figure 13.11

The program displays an error message in the dialog box if the divisor is 0.

- 13.3. Write a program that meets the following requirements:

- [BL] Create an array with 100 elements that are randomly chosen.
- [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 13.12).
- [BL] Create a Show button to cause the array element to be displayed. If the specified index is out of bounds, display the message **Out of Bound**.

*****Insert Figure 13.12 (Same as Figure 11.12 in the 3rd Edition, p504)**

Figure 13.12

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