CHAPTER

11 Exception Handling

Learning Objectives

After reading this chapter, you will be able to

- understand the concepts relating to exception and exception handling
- learn the different types of exceptions
- know about the methods for dealing with exceptions
- learn how a programmer can create his/her own exceptions
- know about multiple type–catch blocks introduced in Java SE 7
- understand the concept relating to rethrowing exceptions
- know the basics of Throws clause
- write programs for managing different types of exceptional conditions using various keywords, for making use of try–catch block, nested try–catch block, rethrowing exceptions, and so on, and using user defined exceptions

11.1 Introduction

There is no guarantee that a perfectly compiled and running program will not crash. An exceptional situation may arise due to human or machine error. Some of the examples of exceptions are as follows:

1. When user enters a negative number as the size of an array
2. When the user provides a negative number as argument in the method for finding square root
3. When the program has to execute division by zero
4. When there is insufficient memory space
5. When the requisite file is not found
6. When a network connection is lost during the communication process
7. When the user tries to open a non-existing file

In fact, there are a large number of such exceptions that may come up during the running of a program.

In Java, an exception is an *object* of a relevant exception class. When an error occurs in a method, it throws out an exception object that contains the information about where the error occurred and the type of error. The error (exception) object is passed on to the runtime system, which searches for the appropriate code that can handle the exception. The event handling code is called *exception handler*. Exception handling is a mechanism that is used to handle runtime errors such as classNotFound and IO. This ensures that the normal flow

of application is not disrupted and program execution proceeds smoothly. The exception handling code handles only the type of exception that is specified for it to handle. The appropriate code is the one whose specified type matches the type of exception thrown by the method. If the runtime system finds such a handler, it passes on the exception object to the handler. If an appropriate handler is not found, the program terminates.

An exception is an undesirable event that may occur during the execution of the program and may lead to termination of the program if it is not handled properly.

The method that creates an exception may itself provide a code to deal with it. The `try` and `catch` blocks are used to deal with exceptions. The code that is likely to create an exception is kept in the `try` block, which may include statements that may create or throw exceptions. The exception object has a data member that keeps information about the *type* of exception and it becomes an argument for another block of code that is meant to deal with the exception, catch block, which follows immediately after the try block. There may be more than one catch blocks. The exception is caught by a catch block whose *type* matches the *type* of exception thrown.

There are basically two models of exception handling. The exception handling facility supported in Java is based on the termination model. According to this model, when the method encounters an exception, further processing in that method is terminated and control is transferred from the point where an exception occurs to the point where its nearest matching exception handler (i.e., the catch block) is located. Under this approach, the control is not transferred to the point of exception occurrence and the program execution is not resumed from there, even if the cause of exception is rectified by the exception handler. In this case, the runtime stack is usually unwound. This causes all the statements and method invocations to terminate abruptly. However, the program execution may not necessarily terminate. This model is analogous to the real-life situation when the gas in the cylinder gets over while cooking. Under the termination model, the cooking process will stop, whereas in resumption model, you would change the cylinder with a new one and continue with the cooking process.

Thus, the alternative approach is based on the resumption model wherein the exception handler tries to rectify the exception situation and resume the program. Resumption model was mostly used in earlier languages including PL/I, Mesa, and BETA. The implementation of resumption model in contemporary languages such as C++ and Java is rarely found as the effort to implement this model is quite high. This is because the program code becomes quite cumbersome and difficult to understand, and further, it is more error prone. However, there are situations where the resumption model is quite useful. For instance, when OutOfMemory error arises during the program execution, it could be solved in certain cases by freeing data structures (written within the handler) that occupied the memory space. Therefore, by using resumption code, the program execution would resume from the place where an exception is raised. According to various researchers (Stroustrup, 2000, Koenig and Stroustrup, 1990), the resuming exception handlers are very much similar to function calls. In order to apply resumption model, the existing exception handling mechanism available in Java needs to be extended. One of the methods involves including throw statement containing one or more accept clauses.

11.2 Hierarchy of Standard Exception Classes

In Java, exceptions are instances of classes derived from the class `Throwable` which in turn is derived from class Object. Whenever an exception is thrown, it implies that an object is thrown. However, it does not mean that any object can be thrown. Only objects belonging to class that is derived from class Throwable can be thrown as exceptions. The hierarchy of the classes is shown in Fig. 11.1. The next level of derived classes comprises two classes: the class `Error` and class `Exception`. Error class involves errors that are mainly caused by the environment in which an application is running. All errors in Java happen during runtime, and therefore,

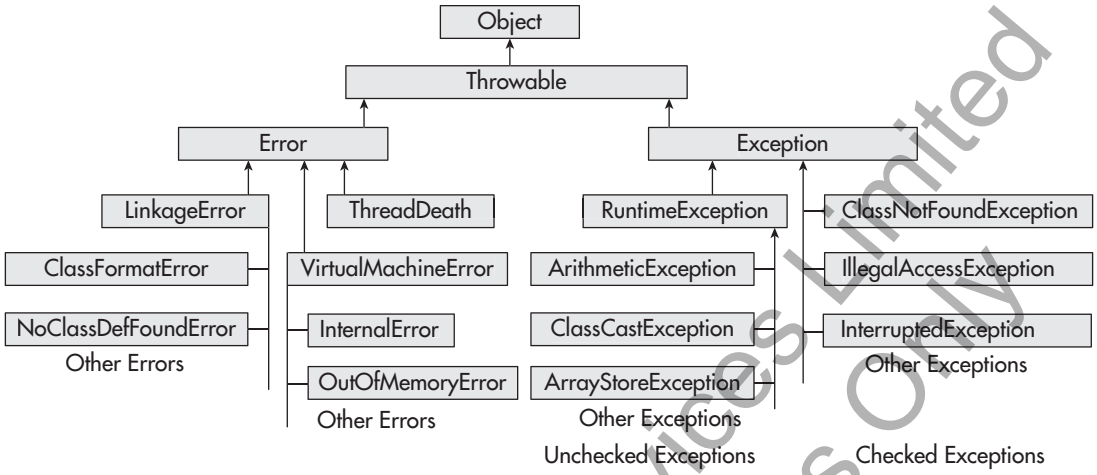


Fig. 11.1 Illustration of hierarchy of error and exception classes

they are of unchecked type. Some of the examples including OutOfMemoryError occurs when the JVM runs out of memory and StackOverflowError occurs when the stack overflows. On the other hand, Exception class represents exceptions that are mainly caused by the application itself—for instance, `ArithmeticException` and `NullPointerException`. It is not possible to recover from an error using try–catch blocks. The only option available is to terminate the execution of the program and recover from exceptions using either the try–catch block or throwing an exception. The exception class has several subclasses that deal with the exceptions that are caught and dealt with by the user's program.

The `Error` class includes such errors over which a programmer has less control. There are three subclasses of `Error` class, that is `LinkageError`, `VirtualMachineError`, and `ThreadDeath`. Then, there are several subclasses of `VirtualMachineError class` and `LinkageError class`. A programmer cannot do anything about these errors except to get the error message and check the program code.

A programmer can have control over the exceptions (errors) defined by several subclasses of class `Exception`. These are also called *Built-in-Exceptions* in Java. The subclasses of `Exception` class are broadly subdivided into two categories.

Unchecked exceptions These are subclasses of class `RuntimeException`, which are derived from `Exception` class. For these exceptions, the compiler does not check whether the method that throws these exceptions has provided any exception handler code or not.

Checked exceptions These are direct subclasses of the `Exception` class and are not subclasses of the class `RuntimeException`. These are called so because the compiler ensures (checks) that the methods that throw checked exceptions deal with them. This can be done in two ways:

1. The method provides the exception handler in the form of appropriate try–catch blocks.
2. The method may simply declare the list of exceptions that the method may throw with `throws` clause in the header of the method. In this case, the method need not provide any exception handler. It is a reminder for those who use the method to provide the appropriate exception handler.

If a method does not follow either of the aforementioned ways, the compilation will result in an error.

The compiler checks if a method that calls another method can throw exceptions that either provide for appropriate catch blocks for dealing with the exception or declare it in the throws part of its header.

The class `Throwable` defines several methods. The three commonly used methods for getting information about the exceptions are described in Table11.1.

Table 11.1 Methods defined in class `Throwable`

Method name	*Description*
`getMessage ()`	It returns a string that gives information about the current exception and consists of a fully qualified name of the exception class and a relevant brief description.
`toString()`	The `class Throwable` overrides the method `toString()` of Object class for displaying messages on screen.
`printStackTrace()`	It traces and displays the hierarchy of method calls that resulted in the exception. The information will be displayed on the screen in the case of a console program.

The application of `toString()` is illustrated in Program 11.1.

Program 11.1: Illustration of `toString()`in which object `e` is converted to a string

```
1   class  TryCatch1 {
2     public static void main (String Str[])
3      {int i = 6, j = 0, k;
4   try {
5   System.out.println ("Entered try block.");
6      k = i/j;
7   System.out.println ("Exiting try block.");
8   }
9   catch (ArithmeticException e)
10   {  System.out.println ("e= " + e);}
11  }
12  }
```

Output

```
Entered try block.
e = java.lang.ArithmeticException: / by zero
```

Explanation

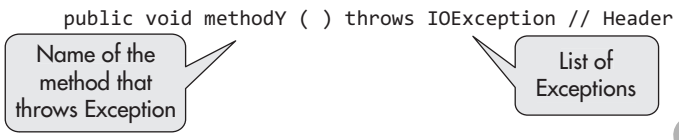
The aim of the program is to display the string representation of the exception. The try block is given in lines 4–8 followed by catch block in lines 9–11. The exception object `e` is converted into a string in line 10 and is displayed as the second line in the output.

11.3 Keywords throws and throw

If a method can cause one or more checked exceptions directly or indirectly by calling other methods and throw exceptions and does not deal with them, it must declare the list of exceptions it can throw by using the *throws* clause. By doing this, the caller of the method can ensure that appropriate arrangements are made to deal with the exceptions, that is, keep the method in a try block, and provide suitable catch blocks. The program will not compile if this is not done. The throws clause comprises the keyword `throws` followed by the list of exceptions that the method is likely to throw separated by commas. The method declaration with throws clause is

```
type Method_identifier(type parameters) throws list-of-Exceptions
{ /* Body of Method*/ }
```

The following example illustrates it.




```
{/*Body of method */}
```

However, if a method throws an unchecked exception, the method need not declare these in the throws clause in the header of the method. The keyword `throw` is used to throw an exception object from within a method. Using `throw` keyword, checked and unchecked exceptions can be thrown. The word throw is followed by the exception class. When the `throw` statement is executed, the execution of the current method is stopped and the control goes to the calling method. It is illustrated as follows:

```
public void methodX() {
int n;
if(n < 0)
throw illegalArgumentException;
double s = sqrt(n);
}
```

11.4 try, catch, and finally Blocks

For managing different types of exceptional conditions, the five keywords `try`, `catch`, `finally`, `throw`, and `throws` are used. The first four are commonly used and they also control the program flow. The part of code that is suspected to create an exceptional situation is placed in a `try` block. It has the code to throw one or more types of exceptions that are likely to occur. The `try` block is immediately followed by one or more `catch` blocks of statements to deal with the exceptions thrown in the `try` block. No other code is allowed between a `try` block and `catch` block.

A `catch` block catches only a single *type* of exception. A catch block may deal with more than one type if they are connected by Boolean operator `OR`.

Generally, separate catch blocks are provided for catching different *types* of exceptions thrown by `try` block. If a `try` block throws two *types* of exceptions, then two separate `catch` blocks are provided to deal with them. However, at a time, only one exception can be handled because after an exception is thrown, the program flow goes out of the `try` block and searches for the appropriate catch block. The subsequent exceptions in the `try` block do not get a chance to be thrown. Figure 11.2 illustrates the processes involved in dealing with exceptional conditions. The code in the `finally block` is always executed irrespective of whether an exception is thrown or not or whether it is handled or not. Different blocks are explained in the following sections.

11.4.1 try {} Block

The program code that is most likely to create exceptions is kept in the `try` block, which is followed by the `catch` block to handle the exception. In normal execution, the statements are executed and if there are no exceptions, the program flow goes to the code line after the `catch` blocks. However, if there is an exception, an exception object is thrown from the `try` block. Its data members keep the information about the *type* of exception thrown. The program flow comes out of the `try` block and searches for an appropriate `catch` block with the same *type* as its argument.

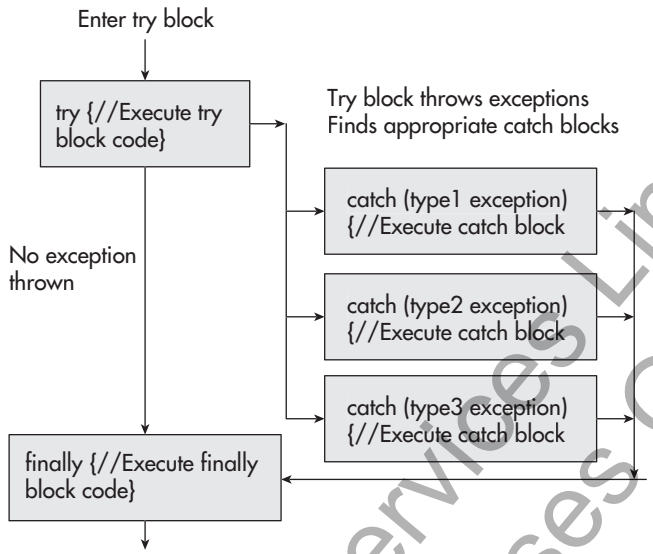


Fig. 11.2 Illustration of try, catch, and finally blocks

Since Java 7 *try-with-resources* has been introduced for automatically closing the sources opened in try block. This is discussed in Section 11.9.

11.4.2 `catch {}` Block

A catch block is meant to catch the exception if the *type* of its argument matches with the *type* of exception thrown. If the *type* of exception does not match the *type* of the first catch block, the program flow checks the other catch blocks one by one (refer to Fig. 11.2). If the *type* of a catch block matches, its statements are executed. If none matches, the program flow records the *type* of exception, executes the `finally` block, and terminates the program.

If a programmer has provided several catch blocks that match the exception object thrown, only the first `catch` block will be executed and others will be ignored. A `catch` block for an exception class will also catch the subclass of the exception class.

Thus, if a programmer provides a `catch` block for a super class exception followed by a `catch` block with subclass exception, only the first will be executed and the second `catch` block will be ignored. Therefore, the `catch` block for a subclass should be provided before the super class `catch` block. A `catch` block can catch more than one type of exception if the argument of `catch` block contains exception classes connected by the Boolean operator `OR` represented by a vertical bar (|).

Figure 11.3 illustrates the sequence of try and catch blocks with analogy from cricket. The `catch` block has to be placed immediately after the `try` block. The figure shows that in the game of cricket, when the bowler bowls, the ball might be caught by the fielder. Similarly, in try–catch block of Java, when an exception arises, it is caught by the catch block. Different catch blocks can be used for handling different types of exceptions.

11.4.3 `finally {}` Block

This is the block of statements that is always executed even when there is an exceptional condition, which may or may not have been caught or dealt with. Only in rare cases, it is bypassed. Thus, `finally` block can be used as a tool for the clean up operations and for recovering the memory resources. For this, the resources

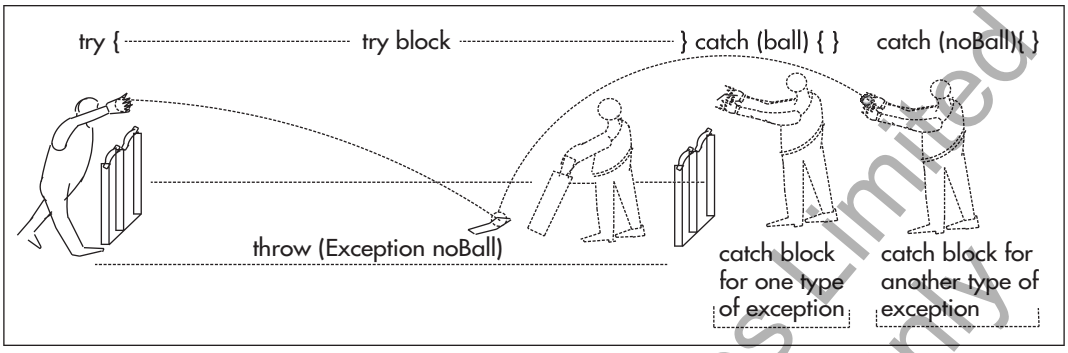


Fig. 11.3 Illustration of sequence of `try` and `catch` blocks

should be closed in the `finally` block. This will also guard against situations when the closing operations are bypassed by statements such as continue, break, or return.

An illustration of codes in `try`, `catch`, and `finally` blocks is given.

```
int a = 7, b = 0, d = 0;
try {  d = a/b; }
catch(ArithmeticException ae)
    {   d = 0; }
finally {
System.out.println("The finally block is always executed."); }
```

Program 11.2 illustrates the sequence of `try`, `catch`, and `finally` blocks of statements. The exception is thrown when an attempt is made to divide by zero, which throws `ArithmeticException`. The catch block has `ArithmeticException` as its argument.

```
catch(ArithmeticException e) { /* body */ }
```

Program 11.2 illustrates the try and catch blocks of statements. The exception is created due to the division by zero.

Program 11.2: Illustration of try{} and catch{} blocks for dealing with exceptions

```
1   class  TryCatch
2    {public static void main (String Str[])
3    { int i = 6, j = 0, k;
4
5   try {
6   System.out.println ("Entered try block.");
7     k = i/j;
8   System.out.println ("Exiting try block.");
9   }     // end of try block.
10          //Below catch block starts
11  catch (ArithmeticException e)
12  { System.out.println ("Arithmetic Exception caught.");
13  System.out.println ("Exiting catch block.");
14  } // end of catch block
15    j = 2;          // value of j changed
16  System.out.println ("When j = 2 the i/j = " + i/j);
17    }
18  }
```

```
33      }
34      }
35    }
```

Output

```
Contents of myFile are:
Java is a purely object oriented language.
```

Explanation

The program illustrates the conventional way of closing sources that has already been explained in the program. Note that the close operation in `finally` block may also throw exception, which is managed within the `finally` block.

In Program 11.15, the try-with-source is illustrated. According to new addition of try-with-source in Java 7, the code of the program for writing to the file is simplified as

```
byte [] bstr = str.getBytes();
            // opens resource as argument of try
try(OutputStream fileOUT = new FileOutputStream("myFile");)
{ for (int k=0; k<bstr.length; k++)
fileOUT.write(bstr [k]);
}
```

No statement is needed for closing because it automatically closes the resource when try block ends. This happens because `OutputStream` implements the `AutoCloseable` interface and the file source is opened as argument of `try`. A similar code is created for opening the file for reading.

Program 11.15: Illustration of try-with-resources

```
1    import java.io.*;
2    public class TryWithResource {
3      public static void main (String arg[])
4       throws IOException
5
6    {String str = "Java is a purely object oriented language.";
7     byte [] bstr = str.getBytes();
8
9    try(OutputStream fileOUT = new FileOutputStream("myFile");)
10   {
11      for (int k=0; k<bstr.length; k++)
12   fileOUT.write(bstr [k]);
13   }
14   try(InputStream fileIn = new FileInputStream("myFile");)
15   {
16    System.out.println ("Contents of myFile are:");
17   for (int i =0; i<= bstr.length; i++)
18   System.out.print((char) fileIn.read());
19     System.out.println();
20     }
21   }}
```

Output

```
Contents of myFile are:
Java is a purely object oriented language.
```

Explanation

For writing to the file, the output stream object is created as argument of `try` in code line 9. Therefore, no close statement is required. Similarly, for reading from file the `InputStream` object `fileIn` is created in code line 14 as argument of `try`. Therefore, no close statement is needed for this as well.

11.10 Catching Subclass Exception

A programmer may define some method in super class. When this method is overridden in subclass with exception handling, then there are some points that need to be taken into consideration. If the super class method does not declare any exception, then subclass overridden method cannot declare checked exceptions but it can declare unchecked exceptions. This is illustrated in Program 11.16.

Program 11.16: Illustration of subclass overridden method declaring checked exception

```
1    package sub;
2
3    import java.io.*;
4
5    class A
6    {
7        void display()
8        {
9            System.out.println("Super class display method");
10       }
11   }
12
13
14   public class Sub extends A {
15
16       void display() throws IOException
17       {
18           System.out.println("Sub class display method");
19
20       }
21       public static void main(String[] args) {
22           A obj1 = new A();
23           obj1.display();
24           Sub obj2 = new Sub();
25           obj2.display();
26       }
27
28   }
```

Output

```
Super class display method
Exception in thread "main" java.lang.RuntimeException.
```

Explanation

In the aforementioned example, method `display()` does not throw any exception in super class. Therefore, its overridden version cannot throw any checked exception. Program 11.17 illustrates the case when the subclass overridden method throws unchecked exception.

Program 11.17: Illustration of overridden method declaring unchecked exception

```
1    package sub;
2
3    import java.io.*;
4
5    class A
6    {
7        void display()
8        {
9            System.out.println("Super class display method");
10       }
```

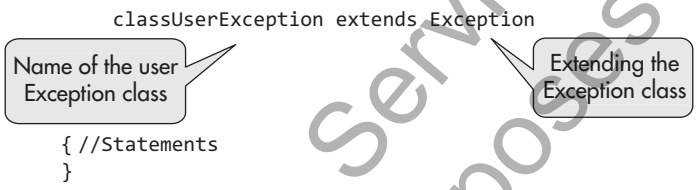
```
29            catch(Exception e){}
30        }
31    }
```

Output
```
Super class display method
```

11.11 Custom Exceptions

It is also called as user defined exception. A programmer may create his/her own exception class by extending the exception class and can customize the exception according to his/her needs. Using Java custom exception, the programmer can write their own exceptions and messages. For writing custom exceptions, the following steps need to be considered:

1. Extend the 'Exception' class as



2. Define constructor of user exception class and this constructor takes a String argument.
3. Write the code for the class containing the main class. The try–catch block is written within this class (see Program 11.9 for an illustration).

Program 11.19: Illustration of `getMessage()` and user's own exception class

```
1    class UserException extends Exception
2     {
3    UserException (String Message)
4     { super (Message);
5     } }                  // End of class UserException
6     class ExceptionEX
7     {public static void main(String args[])
8     { byte a = 4, b = 9 ;
9
10   try
11    {if(a/b== 0)
12   throw new UserException ("It is integer division.");
13    }
14   catch (UserException Ue)
15      {System.out.println("The exception has been caught.");
16   System.out.println(Ue.getMessage());}
17     }}
```

Output
```
The exception has been caught.
It is integer division.
```

Explanation

The program defines a user exception in lines 3–5, which is linked to an integer division `a/b` kept in the try block. The integer division a/b results in 0, and thus, exception occurs. The message is displayed in the last line of output.

11.12 Nested try and catch Blocks

In nested try–catch blocks, one try–catch block can be placed within another try's body. Nested try block is used in cases where a part of block may cause one error and the entire block may cause another error. In such cases, exception handlers are nested.

If a try block does not have a catch handler for a particular exception, the next try block's catch handlers are inspected for a match. If no catch block matches, then the Java runtime system handles the exception.

The try and catch block may be nested as

```
try{// main try block
    //statements
    try {   // try block 1
    //statements
    try {   // try block 2
    /* statement */}
    catch(Exception  e1)
    {// statements. Innermost catch block}
    }
    catch(Exception e2)
    {/*statements. Middle catch block.*/}
    }
catch()
    {/* Statements. outer most catch block*/}
```

In order to understand better, each try block is given a name such as `try block 1` and `try block 2`. From the aforementioned statements, it can be seen that `try block 2` is placed inside the `try block 1`, which is inside the main `try block`.

Program 11.20 presents an illustration.

Program 11.20: Illustration of nested try–catch blocks

```
1   class NestedTry{
2   public static void main (String args[])
3   {int [] array = {6, 7};
4                   //outer try block
5   try {
6   System.out.println("Entered outer try block.");
7      {
8                   // inner try block
9   try {
10  System.out.println ("Entered inner try block.");
11     for ( inti = 0; i<= 2; i++)
12  System.out.println ("Array Element["+i+"]= " + array[i]);
13  System.out.println ("Exiting try block.");
14     }
15                  // inner catch block
16  catch (ArrayIndexOutOfBoundsException e)
17  {System.out.println ("ArrayIndexOutOfBoundsException caught");
18  System.out.println ("Exiting inner catch block.");
19  } }
20  int n = 5, j=2 ;
21       for (j =2; j>=0; j--)
22  System.out.println ("n/j = " + n/j);
23  }
24  // outer catch block
25  catch (ArithmeticException E)
```

```
26      {System.out.println ("Arithmetic Exception
      caught");
27       }}
28    }
```

Output

```
Entered outer try block.
Entered inner try block.
Array Element[0]= 6
Array Element[1]= 7
ArrayIndexOutOfBoundsException caught
Exiting inner catch block.
n/j = 2
n/j = 5
Arithmetic Exception caught
```

Explanation

Two try and catch blocks are included. One try block is nested in another. There are two *for* loops, one in the inner try block and another in the outer try block. The flow control enters outer try block, and then the inner try block. The `ArrayIndexOutOfBoundsException` is encountered. It enters the inner catch block, catches exception, and then, gets out of the inner catch block. In the outer block loop, exception due to division by 0 occurs. This exception is caught in outer catch block.

11.13 Rethrowing Exception

An exception may be thrown and caught and also partly dealt within a catch block, and then, rethrown. In many cases, the exception has to be fully dealt within another catch block, and throw the same exception again or throw another exception from within the catch block.

When an exception is rethrown and handled by the catch block, the compiler verifies that the type of rethrown exception is meeting the conditions that the try block is able to throw and there are no other preceding catch blocks that can handle it.

Program 11.21 illustrates one such case.

Program 11.21: Illustration of rethrowing of exceptions

```
1    class Rethrow{
2     public static void main (String Str[]){
3    int [] array = {6, 7};
4    try {
5    System.out.println ("Entered inner try block.");
6       for(inti = 0; i<= 2; i++)
7    System.out.println ("Array Element["+i+"]= " + array[i]);
8    System.out.println ("Exiting try block.");
9        }
10   catch (ArrayIndexOutOfBoundsException e)
11       {System.out.println ("ArrayIndexOutOfBounds
     Exception caught");
12   System.out.println ("Throwing e and exiting inner
     catch block.");
13   throw e;
14        } }
15   }
```

Output

```
Entered inner try block.
Array Element[0]= 6
Array Element[1]= 7
ArrayIndexOutOfBoundsException caught
Throwing e and exiting inner catch block.
```

In the *for* loop of the method, compute an exception (division by 0) that can occur. Therefore, in the main method, the compute method is called in a try block that is followed by a catch block, which catches user defined exception. The output is given.

11.15 Application Program

The main applications, which are design of cellular mobile systems and spindle speeds of machine tools, are explained in the following sections.

11.15.1 Design of Cellular Mobile System

In cellular mobile system, a particular geographical area is covered by a number of cells. Each cell is assumed to be having equal size and hexagonal in shape. A group of cells using a different set of frequencies in each cell is called a cluster. It is to be noted that only a selected number of cells can form a cluster, and therefore, the cluster size could be 4, 7, 9, 12, and so on. In each cell, a group of frequency channels are allocated in such a way so as to prevent interference between the channels of one cell and another cell. In Program 11.23, the coverage area of the cell and number of channels per cell is evaluated.

Program 11.23: Illustration of cellular mobile system design

```
1   import java.util.Scanner;
2       import java. util.*;
3
4    class MobileException extends Exception {
5
6   public MobileException(String message)
7   {
8   super (message);
9   }
10  }
11  public class MobileApp
12  {
13  public void mobileCalc() throws MobileException
14  {double geoArea, r;
15  int totalChannels;
16  Scanner sc = new Scanner(System.in);
17  System.out.println("Enter the geographical area");
18
19  geoArea = sc.nextDouble();
20  if (geoArea <= 0)
21  throw new MobileException ("Do not enter zero geographical area");
22
23      System.out.println("Enter number of channels available");
24
25
26  totalChannels = sc.nextInt();
27  if (totalChannels <= 1)
28  throw new MobileException ("Number of channels
    should be greater than one");
29
30  System.out.println("Enter radius of a given cell in Km");
31
32      r = sc.nextDouble();
33      if (r < 1)
```

```
34        throw new MobileException ("Radius of cell should be positive value");
35
36
37        double cellArea = r*r*3*(Math.sqrt(3))/2;
38    // calculating coverage area of each cell
39    System.out.println("Coverage area of each
      mobile cell in Km square is:"+ cellArea);
40
41     double ncells = geoArea/cellArea;
42    // calculating number of cells in a particular geographical area
43
44
45
46    System.out.println("Enter cluster
      size, choose one of the values " + "as cluster
      size 3 or 4 or 7 or 9 or 12");
47        int clusterSize = sc.nextInt();
48            int chlpercell = (totalChannels/clusterSize);
49
50            System.out.println("Number of
      channels available per cell" + chlpercell);
51
52    }
53       public static void main (String args[])
54    {
55    try{
56    new MobileApp().mobileCalc();    }
57    catch(MobileException e)
58    {
59        System.out.println
60    ("Exception caught");
61    }
62       }
63    }
```

Output

```
1500
Enter number of channels available
40
Enter radius of a given cell in Km
5
Coverage area of each mobile cell in Km square
is:64.9519052838329
Enter cluster size, choose one of the values as
cluster size 3 or 4 or 7 or 9 or 12
7
Number of channels available per cells
```

When entering incorrect data

```
Enter the geographical area
0
mobileapp1.MobileException: Do not enter zero
geographical area
```

Explanation

In line 4, `MobileException` class extends the class `Exception`. The custom exception is created by extending the `Exception` class. In line 6, constructor of `MobileException` class is defined that takes a String argument. String message is passed to super class that is the Exception class. In line 11, public class `MobileApp` is declared. In line 13, method `mobileCalc()` is defined that also contains the "`throws MobileException`" clause in the method signature. This implies that the method may throw an exception. In line 16, an object of Scanner class is created. In lines 20–21, if `geoArea` is less than 1, an exception is thrown with a message "`Do not enter zero geographical area`". Similarly, for lines 26–34. In line 53, main method is declared. Within the

Another incorrect data

```
Enter the geographical area
1500
Enter number of channels available
0
mobileapp1.MobileException: Number of channels
should be greater than one
```

Another incorrect data

```
Enter the geographical area
1500
Enter number of channels available
40
Enter radius of a given cell in Km
0
mobileapp1.MobileException: Radius of cell
should be positive value
```

try block in lines 55–56, the method `mobileCalc()` of the class `MobileApp` is accessed. If exception occurs, it is caught in line 57. The mobile cell and channels per cell is calculated and displayed in lines 30–50.

11.15.2 Design of Spindle Speeds of Machine Tools

The rotational speed, that is, revolutions per minute, of spindles of machine tools such as drilling machine or a lathe is designed in geometric progression. Let us assume S_1 is the minimum speed, S_2 the maximum speed, and n the number of speeds. The ratio of any two consecutive speeds is the constant geometric ratio R. Thus, the n speeds may be written as

$$S_1, S_1 \times R, S_1 \times R^2, S_1 \times R^3, \ldots, S_1 \times R^{n-1}$$

The highest speed $S_2 = S_1 \times R^{n-1}$

Therefore, $\frac{S_2}{S^1} = R^{n-1}$

From this, we can find the ratio R as

$$R = \left(\frac{S_2}{S^1}\right)^{1/(n-1)}$$

The individual speeds may be found as $S_1, S_1 \times R, S_1 \times R^2, \ldots$

In Program 11.24, the user has to enter the values of S_1 = minspeed, S_2 = maxspeed, and n. The user may make a mistake while entering the values; for example, user may enter zero or less than zero for S_1 or S_2 or may enter $n = 1$ or less than one. The program takes care of these by throwing appropriate exceptions. Since the exceptions are the programmer's exception, we define the `class MyException` that extends the standard exception class and defines the exception.

Program 11.24: Designing spindle speeds of a machine tool

```
1   import java.util.Scanner;
2   class MyException extends Exception
3   {MyException (String Message)
4    {super (Message);}
5     }    // the class MyException ends
6
7   public class AppException  {
8     public void Compute() throws MyException
```

(ii) When max speed entered is 0 or less than 0.

```
Enter the maximum speed
0
Exception caught.
MyException: Maximum speed entered is zero.
```

(iii) When minimum speed entered is 0.

```
Enter the maximum speed
900
Enter the minimum speed
0
Exception caught.
```

11.16 Best Practices for Dealing with Exceptions

For writing a robust application program, it is important to give serious attention to exceptions, their causes as well as handling, so that the program can gracefully handle them. Many expert programmers have come up with the best practices that lead to robust programs. The following are some of the important ones.

1. The exceptions are costly in computer time. Therefore, exceptions should be thrown judicially; otherwise, throwing too many exceptions would not only make the program difficult to understand but also slow execution.
2. It is better to use built-in exceptions rather than using custom exception. This will make the program easily understandable because most of the programmers are aware of standard exceptions and know their behaviour.
3. Do not use Exception or Throwable as argument of catch block because they will catch any subclass exception and the programmer would not know the exact cause of exception.
4. Do not throw any exception from `finally` block because other executable codes will get ignored. If some process throws an exception, better deal with it within the `finally` block.
5. Catch only those exceptions that you can handle. Others may be placed in throws clause.
6. Never use exceptions for flow control. This makes the program difficult to understand and ugly.
7. It is better to do clean up after using exceptions. It is good practice to open resources in `try` block and clean up in `finally` block.
8. For declaring a list of exceptions in the header of a method, better use subclasses rather than the Exception or Throwable class.
9. While designing custom exception, it is a better practice to wrap up standard exception because these are well known to programmers and the track trace is not lost.
10. For debugging, it helps to know the exact cause of exception. Therefore, make use of method `getCause()` to find the exact cause of exception.
11. The empty `catch` blocks should be avoided because you lose the chance to get the stack trace of exception and it may leave your object in corrupt state.
12. It is better to use try-with-resource rather than simple try so that you are sure that the resources will be automatically closed.

Common Programming Errors and Tips

1. The catch block should immediately follow the try block. No other code is allowed between the try and the catch block. Inserting any other code between the two will cause error.
2. If a statement or code block or method is likely to throw an exception, it should be placed in a try block. The try block should be followed by the catch block.

3. If the try block throws more than one type of exception, the catch blocks must be provided for each type of exception.
4. The finally block is always executed whether an exception is thrown or not thrown, whether it is caught or not caught.

SUMMARY

- An exception is an undesirable event that may occur during the execution of the program and that may lead to the termination of the program if it is not handled properly.
- In Java, an exception is an *object* of a relevant exception class. When an error occurs in a method, it throws out an exception object that contains the information about where the error occurred and the type of error.
- Exception handling is a mechanism that is used to handle runtime errors such as classNotFound and IO. This ensures that the normal flow of application is not disrupted and program execution proceeds smoothly.
- Java language has a number of built in types of exceptions, which form a hierarchy of classes. On the top of the hierarchy is the class Throwable, which is a subclass of class Object.
- The exceptions are instances of classes derived from the class Throwable. The next level of derived classes comprises two classes, that is, the class Error and class Exception.
- The subclasses to Exception class are broadly subdivided into two categories: checked exception classes and unchecked exception classes
- The unchecked exception classes are subclasses of class RuntimeException, which is derived from Exception class.
- The checked exceptions are direct subclasses of Exception class and are not subclasses of class RuntimeException. These are called so because the compiler ensures (checks) that the methods that throw checked exceptions deal with them.
- The RunTimeExceptions, Error, and their subclasses are called unchecked exceptions because the compiler does not check whether a method is dealing with these exceptions or not. All other exceptions are called checked exceptions because the compiler ensures that the methods that can throw checked exceptions are supposed to deal with them.
- A method can cause one or more checked exceptions directly or indirectly by calling other methods that throw exceptions and does not deal with them. Then it must declare the list of exceptions it can throw using the throws clause.
- The keyword throw is used to throw an exception object from within a method. Using throw keyword, checked and unchecked exceptions can be thrown.
- For managing different types of exceptional conditions, the five keywords, namely try, catch, finally, throw, and throws are used.
- The part of code that is suspected to create an exceptional situation is placed in a try block. It has the code to throw one or more types of exceptions that are likely to occur.
- Catch block follows immediately after the try block. There can be more than one catch blocks. The exception is caught by a catch block whose type matches the type of exception thrown.
- RunTime Exceptions form a subgroup of classes that are subclasses of class Exception.
- A programmer may create his/her own exception class by extending the exception class.
- An exception may be thrown and caught and also partly dealt with a catch block and then rethrown; this is called rethrowing an exception.

GLOSSARY

Catch The try block is immediately followed by one or more `catch` blocks of statements to deal with the exceptions thrown in try block.

Checked exceptions These are the exceptions that are *checked* by a compiler and these are also called compile time exceptions.

Custom exceptions These are also called user defined exceptions. A programmer may create his/her own exception class by extending the exception class and can customize the exception according to his/her needs.

Finally It is a keyword. The code in the finally block is always executed irrespective of whether an exception is thrown or not, or whether it is handled or not.

Nested try–catch block Here, try–catch block can be placed within another try block's body.

Rethrowing exception An exception may be thrown and caught and also partly dealt in a catch block and then rethrown.
Runtime exceptions These exceptions form a subgroup of classes that are subclasses of class Exception. These are also called unchecked Exceptions.

Try The part of code that is suspected to create an exceptional situation is placed in a `try` block.
Unchecked exceptions For these exceptions, the compiler does not check whether the method that can throw these exceptions has provided any exception handler code or not.

EXERCISES

Multiple-choice Questions

1. Which of the following keywords is not a part of exception handling?
   (a) thrown (c) try
   (b) catch (d) finally
2. Which of the following statements are correct regarding the finally block of statements?
   (a) It is a block of statements that must be executed even if exception occurs.
   (b) It is the block that has a constant number of statements.
   (c) It is the final block for dealing with rethrown exceptions.
   (d) None of these is true.
3. The exception class belongs to which of the following packages?
   (a) java.io (c) java.file
   (b) java.lang (d) java.util
4. Which of the following keywords is used when a method can throw an exception?
   (a) finally (c) throw
   (b) throws (d) catch
5. Which keyword is used to monitor a statement for exception?
   (a) catch (c) throw
   (b) try (d) throws
6. Which of the following blocks gets executed compulsorily whether the exception is caught or not?
   (a) throws (c) finally
   (b) catch (d) throw
7. In order to create customer exceptions class, we have to
   (a) create our own try and catch block
   (b) extend exception class
   (c) use finally block
   (d) use throws keyword
8. Which of the following is correct regarding the exception created by attempt to divide by zero?
   (a) IndexOutOfBoundsException
   (b) ArithmeticException
   (c) IllegalArgumentException
   (d) MethodNotFoundException
9. Which of the following is the super class of all exception classes?
   (a) Throwable
   (b) RuntimeException
   (c) Exception
   (d) IOException
10. Which of the following is true for multiple catch blocks?
   (a) The superclass exception cannot be caught first.
   (b) Either the super or subclass can be caught first.
   (c) The superclass exception must be caught first.
   (d) None of these
11. Which of the following exceptions is thrown when an array element is accessed beyond the array size?
   (a) ArrayIndexOutOfBoundsException
   (b) ArrayElementOutOfLimit
   (c) ArrayIndexOutOfBounds
   (d) ArrayElementOutOfBounds
12. Which of the following statements are correct regarding ArrayStoreException?
   (a) This exception arises because of lack of memory to store more elements.
   (b) This exception arises due to index value getting more than declared.
   (c) This exception arises because the *type* of the new element to be added is different from the declared *type* of the array.
   (d) None of these is true.
13. Which exception will be thrown by the following code?

```
int Array = {5,4,6};
Math.sqrt(Array[3]);
```

   (a) IllegalArgumentException
   (b) ArithmeticException
   (c) ArrayStoreExcepting
   (d) ArrayIndexOutOfBoundsException
14. Which of the following errors are not supposed to be dealt with as exceptions?

(a) The sqrt() method of Math class has a negative number as argument.
(b) Linkage Error
(c) Error due to division by zero
(d) VitualMachineError

15. Indicate the output of the following code:

```
class ExceptionDemo {
public static void main (String args[]){
try{
int a = 0;
int b = 6/a;
System.out.print(" OutputA");
}
catch (ArithmeticException e) {
System.out.print("OutputB");
}
}
```

(a) OutputA
(b) OutputB
(c) Compilation error
(d) Runtime error

Review Exercises

1. What are exceptions?
2. Are the exceptions methods or objects?
3. What is dealt with in Error class and in Exception class?
4. Illustrate the code for making your own Exception class.
5. What are the checked and unchecked exceptions? Give one example of each.
6. Explain the functions of try{}, catch{}, and finally {} blocks.
7. Give an example of code of nested try blocks and catch blocks.
8. What is ArrayStoreException?
9. Explain the code that can give rise to IllegalAccessException.
10. What does finally block do?
11. What will happen if an exception is thrown and there is no catch block to deal with it?
12. Why is an exception rethrown?

Programming Exercises

1. Write a program to illustrate the output when a negative number is placed as argument for sqrt() method of Math class.
2. Write a program to illustrate the throwing of ArrayIndexOutOfBoundsException exception.
3. Write a program to illustrate a method that throws exceptions.
4. Write a program to illustrate the nested try and catch blocks.
5. Write a program to illustrate the IllegalAccessException.
6. Write a program that illustrates rethrowing of an exception.
7. Write a program that illustrates use of a user defined exception.
8. Write a program that throws InterruptedException.
9. Write a program that throws StringIndexOutOf BoundsException.
10. Write a program that throws `ArithmeticException`.
11. Write a program that illustrates the use of try-with-source.
12. Write a program in which `AutoClosable` interface is implemented.
13. Write a program in which the area of a room is calculated and the cost of white washing is also evaluated. Further, include the provisions for window on any of the walls. The inputs regarding the parameters including length, breadth, and height of the room are taken through Command line. If there is a window, then its parameters including length and breadth are also taken through Command line. If these input parameters are below 1, then raise an exception; otherwise, calculate the area and cost and display the result. (Note that in order to calculate the area of the room to be painted the area of window must be deducted from the total of the room.)

Debugging Exercises

1. Debug the following program code and run the program.

```
class TryCatch;
{public static void main (String Str[])
{
    inti = 6, j = 0, k;
    {try
    System.out.println ("Entered try
    block." );
```

```
 k = i/j;
System.out.println (“Exiting try
block.” );}
catch  { (ArithmeticException e)
System.out.println (“Arithmetic
Exception caught.”);
System.out.println (“Exiting catch
block.”); }
j = 2;
System.out.println (“i/j = ” + i/j);
}}
```

2. Debug the following program code and run the program.

```
class  DemoTest
   {public static void main (String Str[])
{ int a = 8, b = 0, m;
try {
System.out.println (“Hello, it is try
block”);
  m = a/b;
System.out.println (“Exiting try block”);
catch (ArithmeticException)
{ System.out.println (“Arithmetic Exception
caught.”);
}
    b = 5;
    System.out.println (“The result is a/b
    = ” + a/b);
  }
}
```

3. Debug the following program code and run the program.

```
class Example1 {
public static void main (String args[])
try {
System.out.println (“Entered try block.”);
int num = 43/0;
System.out.println (num);
}
catch(ArrayIndexOutOfBoundsException e){
System.out.println
(“ArrayIndexOutOfBoundsException ”);}
Finally {
System.out.println (“Entered finally
block1.”);
System.out.println (“Exiting finally
block1.”); }
}
Finally{
System.out.println (“Entered finally
block2.”);
System.out.println (“Exiting finally
block2.”); }
  }
    }
```

4. Debug the following program code and run the program.

```
class ExceptionCustom
  {
ExceptionCustom (String Message)
{ super (Message);
  } }
class Exception1
 {public static void main(String args[])
{ byte i = 3, j = 6 ;

try
 {if(i/j== 0)
throw new ExceptionCustom (“Exception
Occurred”);

catch (ExceptionCustom Se)
    {System.out.println(“The exception has
been caught.”);
System.out.println(Se.getMessage());}
 }}
```

5. Debug the following program code and run the program.

```
public class UserException Extends
Exception
 {
Public UserException (Message)
{ super (Message);
 } }
public class TestException
 {public static void main(String args[])
{ TestException ex = new TestException();
ex.giveNumbers();
}
public void giveNumbers() throw
UserException
{
for(int n = 0; n<10; i++);
{ System.out.println(“n = ” +n);
if(n == 4)
throws new UserException (“Exception is
caught.”);
 }
    }}
```

6. Debug the following program code and run the program.

```
import java.io.*;
public class DemoException{
void myVote(int age) throw IOException,
ClassNotFoundException{
if (age<18)
throws new IOException (“Exception Message
1”);
else
throws new ClassNotFoundException
```