Chapter 12

EXCEPTIONS

In this chapter, you will learn about exceptions. Exceptions are abnormal conditions occuring in a program. When such exceptions occur, the program aborts or hangs, leaving the user without any information. This chapter explains how to manage such situations.

A Java program, compiled and error-free, will execute and complete the task for which the program has been developed. However, there are occasions in which a program running smoothly in normal conditions may encounter errors in abnormal conditions such as divide by zero. Such errors occurring in abnormal conditions are called exceptions. The errors may occur internally in the program code or through the resources the program is trying to access. The sources of errors may be user-input errors such as giving a wrong URL, device errors such as printer not ready, physical limitations such as disk full or memory not enough and code errors like array index out of bound, trying to access an empty stack and divide by zero error. If such exceptions are not caught, the program aborts at the point of occurring of the exception. Java provides appropriate mechanisms to handle such exceptions. The objective of handling exceptions is to inform the user about the error or to take an alternative path to overcome the problem and complete the remaining task. This helps the user to know what is happening inside the program. In the absence of such messages, the user may end up with unexpected output.

12.1 Types of Exceptions

All exceptions in Java are handled by a superclass **Throwable** defined in java.lang package. The **Throwable** class has two subclasses, **Exception** and **Error**. **Exception** again has two subclasses, **IOException** and **RuntimeException**. The Exception hierarchy is given in fig. 12.1.

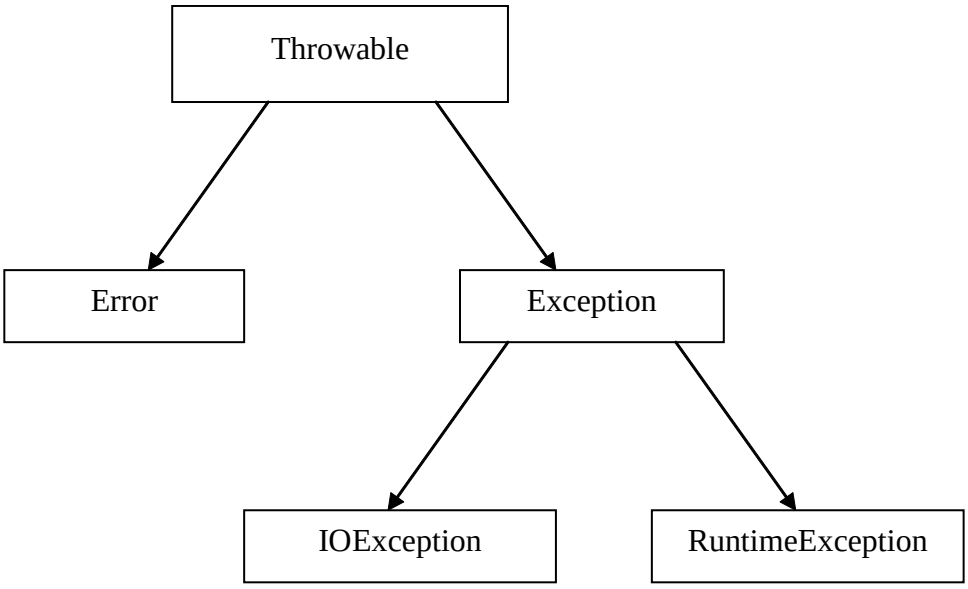

Fig.12.1 The Exception Hierarchy

RuntimeException

Exception occurring in the program code at runtime are handled in this class. Divide by zero error, array index out of bound, wrong cast and null pointer access are of this type. These types of errors could have been avoided, if the programmer had taken care to write the program. These exceptions can be caught and handled by Java.

IOException

Exceptions occurring while accessing I/O devices are handled in this class. File not found, end of file encountered are of this type.

Exception

In this class, a user can create one's own Exception and use it in Java program. All Exceptions under this class are to be caught and handled.

Error

Errors which are beyond the control of the programmers are dealt in this class. Disk full and memory not enough are of this type. Java does not provide any mechanism to handle them and should not be caught.

12.2 Catching Exceptions

Java developers have identified commonly occurring exceptions and they are specified in the Exception. When such exceptions appear in a program at runtime, they are to be caught and handled. The Exception is caught by **try...catch** mechanism. The general form of the **try...catch** block is:

```
try    { ...
         statements that may give
         exceptions
       } catch (ExceptionType e1){
         statements to handle
         Exception Type1
       } catch (ExceptionType e2) {
         statements to handle ExceptionType2
       } finally {
         statements to be executed
         whether an exception occurs or not
       }
```

The terms **try**, **catch** and **finally** are Java keywords. e1 and e2 are errors. In the **try** block, the statements that are suspected to cause exceptions are placed. ExceptionType1 and ExceptionType2 are the different exception types. The **catch** block contains statements that are to be executed in the event of occurring of an **Exception**.

Multiple **catch** blocks for single **try** block can be set up, each **catch** block dealing one specific type of exception out of several occurring inside the **try** block. The **try** block may be constructed to trap any number of exceptions. If an exception occur in the **try** block and if there is no matching **catch** block, the program is aborted. In case no **Exception** occurs inside the **try** block, all **catch** block statements will be skipped and only statements in the **finally** block will be executed. This **finally** block will be executed irrespective of whether an **Exception** occurs or not. The **catch** block statements are operative only for the preceeding **try** block. The **finally** block is optional.

The following program 12.1 contains an array out of bounds Exception that is caused during the runtime and aborted without completing the program.

Program 12.1

```
// This program illustrates the Exception in a program.
class ExceptionDemo1
    {
    public static void main(String args [])
        {
```

```
        int mat [] = new int[10];
        mat[10] = 25;
        System.out.println("Last element of mat = " +mat[9]);
        }
    }
```

The above program gives the following error message and aborts the execution:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
at ExceptionDemo1.main(ExceptionDemo1.java:5)
```

The above error message indicates that an **ArrayIndexOutOfBoundsException** as defined in java.lang package has occurred in the main method of the program ExceptionDemo1.java at line 5. In program 12.1, the array index for the mat array can take values from 0 to 9 only. However the program tried to access an element at index 10, which is out of bounds.

Now, we will make use of an exception-handling feature of Java to handle the Exception using **try** and **catch**. The following program 12.2, uses the **try**...**catch** structure to handle the array out of bounds Exception:

Program 12.2

```
// This program illustrates the catch of an  Exception
// in a program.
class CatchException1
    {
    public static void main(String args [])
        {
        int mat [] = new int[10];
        try
            {
            mat[10] = 25;
            }
        catch (ArrayIndexOutOfBoundsException e)
            {
            System.out.println("Array index out of bounds in
                mat array");
            mat[9] = 25;
            }
      System.out.println("Beyond the exception point ");
      System.out.println("Last element of mat = " + mat[9]);
        }
    }
```

The above program gives the following output:

```
Array index out of bounds in mat array
Beyond the exception point
Last element of mat = 25
```

The output of the above program 12.2 shows that the program crossed the array out of bounds exception and completed the whole program. In the **catch** block, any statement that can provide remedy to the statement causing the exception may be placed.

In a **try** block, when an exception occurs, the control skips all statements from that point till the end of **try** block. Therefore, care must be taken to include a statement in a **try** block. In case, an exception occurs in a code and if it is not going to cause any problem to your results, then an empty block{} can be placed after the **catch** block. The following program 12.3 illustrates the above two concepts in a **try. . . catch** block overlapping with a for loop:

Program 12.3

```
// Program to illutrate the behavior of a try block
class TryBehave
    {
    public static void main(String args [])
        {
        int i, x, con = 4;
        for (i = 1; i < 8; i++)
            try
                {
                x = 25 / (con - i);

                System.out.println("\n quot =" + x);
                }
            catch (ArithmeticException e)
                {
                System.out.println("\n Divide by zero error
                                    for i =" + i);
                }
            catch (ArrayIndexOutOfBoundsException e)
                {
                }
        }
    }
```

The above program gives the following output:

```
quot =8
quot =12
quot =25
Divide by zero error for i =4
```

```
quot =-25
quot =-12
quot =-8
```

In a try ... catch block, when an exception occurs, all statements between the statement causing the exception and the end of the try block will be skipped. Hence a programmer must decide which statements are to be placed inside a try block.

12.2.1 Nested try Blocks

Nested **try** block, one **try** block enclosing another **try** block, is permitted in a Java program. The **catch** statements are operative for the corresponding statement blocks defined by { and }. The following program 12.4, illustrates the nested **try** block:

Program 12.4

```
// This program illustrates the nested Try block.
class NestedTry
    {
    public static void main(String args [])
        {
        int vec [] =  { 3, 5, 4, 10, 2 };
        int nmr = 24;
        int quot, sum = 0;
        for (int i = 0; i <= 5; i++)
            {
            try
                {
                quot = nmr / (2 - i);

                try
                    {
                    sum += vec[i];
                    }
                catch (ArrayIndexOutOfBoundsException e)
                    {
                    System.out.println("\n Array index out of
                     bounds at index(inner try)  = " + i);
                    }
                catch (ArithmeticException e)
                    {
                    System.out.println("\n Divide by zero
                          error at i (inner try)  = " + i);
                    }
```

```
                System.out.println("\n Quotient = " + quot);
                }
            catch (ArrayIndexOutOfBoundsException e)
                {
                System.out.println("\n Array index out of
                     bounds at index(outer try)  = " + i);
                }
            catch (ArithmeticException e)
                {
                System.out.println("\n Divide by zero error
                     at i (outer try)  = " + i);
                }
            }
        System.out.println("\n Sum of numbers = " + sum);
        }
}
```

The above program gives the following output:

```
Quotient  = 12
Quotient  = 24
Divide by zero error at i (outer try)  = 2
Quotient  = -24
Quotient  = -12
Array index out of bounds at index (inner try)  = 5
Quotient  = -8
Sum of numbers = 20
```

The above program finds quotient of integer division of the variable nmr by i - 2 and finds the sum of the array vec with five elements. When the for loop is repeated, divide by zero error occurs and the control skips all the statements till the end of outer try block and looks for the matching catch statement. Thus the catch statement in the outer try block for **ArithmeticException** is caught and the **catch** block statements are executed. Since the control has skipped all statements between the occurrence of the exception and the corresponding }, of the corresponding **try** block, sum += vec[i] for i=2 is not executed. Hence, the final value of sum gives only 20 instead of 24. When the array index i takes the value 5, **ArrayIndexOutOfBoundsException** occurs in the inner try block and is caught and handled. The control then exits the inner try block and the remaining two print statements are executed.

12.2.2 Hierarchy of Multiple Catch Blocks

When multiple catch blocks are set up for a try block, then the subclass exceptions are to be caught first and then the superclass exceptions. In case superclass exception handlers are placed first, followed by handlers for subclass

exceptions, then Java compiler will give a code not reached error. Suppose in a multiple catch block, the first catch block catches an exception of type **Exception** followed by another catch block to catch **RuntimeException**, then code not reached error will occur. The following program 12.5 illustrates this:

Program 12.5

```
// This program illustrates the multiple catch with improper
// hierarchy.
// This program will give compile error.

class Multicatch
    {
    public static void main(String args [])
        {
        int vec [] = {3,5,4,10,2};
        int nmr = 24;
        int quot, sum = 0;
        for (int i = 0; i <= 5; i++)
            {
            try
                {
                quot = nmr / (2 - i);
                sum += vec[i];
                System.out.println("\n Quotient   = " + quot);
                }
            catch (Exception e)
                {
                System.out.println("\n Array index out of
                     bounds at index   = " + i);
                }
            catch (ArithmeticException e)
                {
                System.out.println("\n Divide by zero error
                     at i    = " + i);
                }
              }

        System.out.println("\n Sum of numbers = " + sum);
        }
    }
```

The above program when compiled will give the following error:

```
Multicatch.java:17: catch not reached.
} catch(ArithmeticException e){
1 error
```

In the above program, the first catch block catches exception of a superclass **Exception** while the second catch block catches the Arithmetic-Exception belonging to **RuntimeException,** which is a subclass of **Exception** (see fig. 12.1). Hence, the compile error. If the order of the catch blocks is interchanged, the program will be error-free and can be executed.

In a multiple catch block, the subclass exceptions are to be caught first followed by superclass exceptions.

12.3 Rethrowing Exceptions

When an exception is caught in a method, it is up to the programmer to deal with it. If it is known how to handle the exception, it can be handled, otherwise it can be rethrown. Rethrowing is done through the **throw** clause. Unless the rethrown exception is caught again, the program will abort. After the **throw** statement, no other statement should be placed in that block. If any statement is placed after the **throw** statement in that block, Java will give code not reached, compile error. The following program 12.6 illustrates how to rethrow an exception. The exception is not recaught and hence aborts after rethrow.

Program 12.6

```
// This program demonstrates  rethrowing an exception.
class Rethrow
    {
    public static void main(String args [])
        throws Exception
        {
        int nmr = 24, i, quot;
        for (i = 0; i <= 5; i++)
            try
                {
                quot = nmr / (i - 2);

                System.out.println("\n Quotient = " + quot);
                }
            catch (ArithmeticException e)
                {
                System.out.println("\n  Rethrowing the
                     exception at i =" + i);
               throw e;
                }
        }
    }
```

The above program gives the following result:

```
Quotient = -12
Quotient = -24
Rethrowing the exception at i =2

Exception in thread “main” java.lang.ArithmeticException: / by zero
at Rethrow.main(Compiled Code)
```

The following program 12.7 illustrates recatching a rethrown exception and makes the program to complete its remaining task:

Program 12.7

```
// This program demonstrates  recatching of exception.
class Recatch
    {
    public static void main(String args [])
       {
        int nmr = 24, i, quot;
        for (i = 0; i <= 5; i++)
      try{ try
                {
                quot = nmr / (i - 2);
                 System.out.println(“\n Quotient = “ + quot);
                }
            catch (ArithmeticException e)
                {
                System.out.println(“\n  Rethrowing the
                     exception at i =” + i);
                throw e;
                }
      } catch (ArithmeticException e) {
           System.out.println(“\n Recaught e”);
           }
       }
      }
```

The above program gives the following output:

```
Quotient = -12
Quotient = -24
Rethrowing the exception at i =2
Recaught e
Quotient = 24
Quotient = 12
Quotient = 8
```

12.4 Creating Your Own Exceptions

All the exceptions we have seen so far are defined inside the Java language. We have seen how to catch them and rethrow them. There may be programs that may create errors, which are not covered in Java's predefined exceptions. If any such error occurs, it can be brought under the **Throwable** or its subclasses. This provides a mechanism by which a new exception can be created. The general form of creating an exception is:

```
new ThrowableClass();
        or
new ThrowableClass(String s)
```

The ThrowableClass can be **Throwable** or its subclasses. There are two constructors for **Throwable** and its subclasses. The first constructor needs no argument and the second constructor takes a string as an argument. The string is assigned as the name of the exception. For example, the following statements create a new exception called MyException with an argument and is thrown using **throw** clause.

```
Throwable MyException = new Throwable("Help Me");
throw MyException;
```

The following program 12.8. creates a new exception, throws it and catches it.

Program 12.8

```
// This program illustrates the creation of a new exception.
class MyException
    {
    public static void main(String args [])
        {
        Throwable MyExpn = new Throwable("Help Me");
        try
            {
            System.out.println("\n A new exception is
                                        thrown");
            throw MyExpn;
            }
        catch (Throwable e)
            {
            System.out.println("\n The exception is caught
                here. \n \n The exception is " + e);
            }
        }
    }
```

The above program gives the following output:

```
A new exception is thrown
The exception is caught here.
The exception is java.lang.Throwable: Help Me
```

The following program 12.9 creates an **Exception** subclass named NewExp, creates an object of that type, throws it and catches it.

Program 12.9

```
// This program creates subclass of Exception throws it
// and catches it.
class NewExp
    extends Exception
    {
    NewExp(String s)
        {
        System.out.println("\n Untamed exception is thrown
                out:" + s);
        }
    public static void main(String args [])
        {
        NewExp ne = new NewExp("Lion");
        try
            {
            throw ne;
            }
        catch (Exception e)
            {
            System.out.println("\n I caught that exception
                                and tamed it : " + e);
            }
        }
    }
```

The above program gives the following output:

```
Untamed exception is thrown out:Lion
I caught that exception and tamed it : NewExp
```

12.5 Broadcasting that a Method Throws Exception

When a method is developed for carrying out a task, it may generate various types of exceptions. Some of the exceptions generated cannot be handled by the method itself. Therefore, it is important to broadcast that the method is going to throw an exception, so that the caller of that method can handle the exception appropriately. Exceptions of type **Error** that cannot be handled at all and **RuntimeException** that could have been avoided by the

programmer should not be broadcast. Only exceptions other than **Error** or **RuntimeException** can be broadcast. The broadcasting is done using the **throws** keyword. The broadcasting is made while declaring the method. The general form of broadcasting that a method **throws** an exception is:

```
return-type methodname(parameter_list) throws Exception1,
Exception2 {
method_body
}
```

For example, the following method broadcasts two exceptions:

```
static double readDouble() throws IOException,
ParseException {
.
.
.
String s = br.readLine();
}
```

In the above example, the method readDouble() announces that the method is likely to cause two exceptions, **IOException** and **ParseException**. The following program 12.10 illustrates how to broadcast that a method throws an exception.

Program 12.10

```
// This program broadcasts  that its method is going to
// throw an exception.
class Cal
    {
    int x, y;
    int Process(int a, int b)
        throws Exception
        {
        x = a;
        y = b;
        System.out.println("\n I dont know what to do these
                numbers " + a + " and  " + b);
        throw new Exception("AVAJ");
        // return (a+b); This statement cannot be placed
       // after throw. Will give compile error.
        }
    }
class Broadcast
    {
    public static void main(String args [])
        {
```

```
        Cal cl = new Cal();
        try
            {
            cl.Process(10, 25);
            }
        catch (Exception e)
            {
            System.out.println("\n The thrown Excepion is: "
                    + e + " \n\t and is  caught in main");
            System.out.println("\n Sum of the two numbers is
                    " + (cl.x + cl.y));
            }
        }
    }
```

The above program gives the following output:

```
I dont know what to do these numbers 10 and  25
The thrown Exception is: java.lang.Exception: AVAJ
        and is  caught in main
Sum of the two numbers is 35
```

12.6 The finally Block

In a **try...catch** block, all statements will be executed, if no exception occurs. When an exception occurs, the statements placed in between the point of occurring of the exception and the end of the block will be skipped and the control looks for a matching catch block. In certain problems, it is essential to execute certain statements, like closing a file, irrespective of whether an exception occurs or not. Such essential statements are placed in the **finally** block. Program 12.11 illustrates the use of **finally** block.

Program 12.11

```
// Program to illustrate the finally  block.
class Finally
    {
    public static void main(String args [])
        {
        int i, x, con = 2;
        for (i = 1; i < 4; i++)
            try
                {
                x = 25 / (con - i);
                System.out.println("\n quot =" + x);
                }
            catch (ArithmeticException e)
                {
                System.out.println("\n Divide by zero error
```

```
                    for i =" + i);
                }
            finally
                {
                System.out.println("\n Exception or no
                Exception this will be printed always");
                }
        }
    }
```

The above program gives the following result:

```
quot =25
Exception or no Exception this will be printed always
Divide by zero error for i =2
Exception or no Exception this will be printed always
quot =-25
Exception or no Exception this will be printed always
```

The statements in a **finally** block are always executed, irrespective of whether an exception occurs or not.

12.7 Checked and Unchecked Exceptions

Java developers have identified certain exceptions which must be made known to the programmer at the compile time itself. Such exceptions are listed and the compiler checks whether any of the listed exceptions occur in a program during compilation. Such exceptions are called checked exceptions. Exceptions that are not listed in the list are identified during runtime. Such exceptions are called unchecked exceptions. Only checked exceptions are to be broadcast in a **throws** clause. Table 12.1 gives some checked exceptions and table 12.2 gives some unchecked exceptions.

Table 12.1 Some Checked Exceptions

	Exception	Cause for the Exception
1.	ClassNotFoundException	Class not found
2.	CloneNotSupportedException	Trying to clone an object which has not implemented clonable interface
3.	IllegalAccessException	Access to a class is refused
4.	InterruptedException	One thread is interrupted by another thread.
5.	NoSuchMethodException	The referred method does not exist.

Table 12.2 Some Unchecked Exceptions

	Exception	Cause for the Exception
1.	ArithmeticException	Arithmetic error
2.	ArrayIndexOutOfBoundsException	Array index is beyond the bounds.
3.	ClassCastException	Invalid cast
4.	IllegalArgumentException	Invalid arguments passed to a method
5.	NegativeArraySizeException	Array with negative index
6.	NullPointerException	Invalid use of null
7.	NumberFormatException	Incorrect conversion of string to a number
8.	StringOutOfBoundsException	Trying to access a string beyond its bounds

More exceptions can be found in www.java.sun.com

After reading this chapter, you should have learned the following:

- ➲ What is an exception?
- ➲ The types of exception
- ➲ Establishing try...catch block
- ➲ The use of throw and throws clause
- ➲ Creating your own exception

In the next chapter, you will learn about I/O classes.

Exercise-12

I. Fill in the blanks

12.1. An abnormal condition occurring in a Java program is called _____.

12.2. The programmer avoidable errors are exceptions of type _____.

12.3. Exceptions arising out of memory and memory not enough are of the type _____. They are _____ be caught.

12.4. The codes inside the _____ block of an exception handling structure will be executed irrespective of whether an exception occurs or not.

12.5. When exceptions belonging to a superclass and subclass occur inside a try block, the class exception must be caught first.

12.6. User-defined exceptions can be created using _____ class.

12.7. When a method cannot handle an Exception that is generated in it, the method must declare it using the keyword _____ .

12.8. Exceptions that are identified during compile time are called _____ exception.

II. Write Java programs for the following:

12.9. It is required to compute the following expression:

$$f = \frac{x}{x^2 - y^2}$$

for different values of x = 1.0 to 5.0 in steps of 0.5 and y = 0 to 4.0 in steps of 0.5. Write a Java program to compute the values of f by including an appropriate exception handling block when x = y.

12.10. A mark list containing register number and marks for a subject is given. The marks and register number are to be read. If the marks are < 0, user-defined IllegalMarkException is thrown out and handled with the message "Illegal Mark". For all valid marks, the candidate will be declared as "PASS" if the marks are equal to or greater than 40, otherwise it willl be declared as "FAIL". Write a class called IllegalMarkException by extending the Exception. Write another class MarkProcess to process the mark. In the MarkProcess class write a Validation (int mark) method, which if marks are <0, will throw an Exception of type IllegalMarkException. Write another method Result() which will declare the result. Write a class containing main method that will create an object of type MarkProcess and call the methods in it to declare the result.

12.11. Write a class to sort the given set of n integers in descending order. Include a try block to locate the array index out of bounds exception and catch it.

* * * * * *