UNIT 4 EXCEPTION HANDLING

Structure **Page Nos.**

4.0 INTRODUCTION

Exceptions are conditions that, because of their nature, should be explicitly handled by any robust and well-thought-out code. Since Java was originally intended for programming high-reliability electronic devices, it comes with a built-in-exception-handling facility that allows the programmer to detect, and recover from unexpected errors. These are also termed as unseen logical errors, e.g., writing to a file that has not been opened properly or trying to open a file that does not exist, etc.

Exceptions are a subset of **Errors** that can be handled by the programmer. When an error occurs in the JVM, it is unrecoverable and it causes the interpreter to display the message and abruptly terminate the program, whereas an exception is an abnormal condition that can be handled to prevent the program from terminating.

In Java, errors and exceptions are 'thrown' by the JVM which can be 'caught', or processed by special blocks of code written for this purpose. When a critical condition occurs, JVM creates an appropriate error, or exception object.

4.1 OBJECTIVES

After going through this Unit, you will be able to:

identify different exception classes;
use built in exceptions, and
create user defined exceptions.

4.2 EXCEPTION CLASSES

Exception is represented by objects, known appropriately as ***Exceptions***, which have properties similar to all other Java objects. There are two basic classes of Exceptions in Java derived from the parent class **java.lang.Throwable**. These two classes are:

1) java.lang.Error

This represents usually serious and unrecoverable exceptions, such as running out of memory or being unable to locate or load a needed class. The Error objects are created from the **java.lang.Error** class and its subclasses.

2) jav.lang.Exception

Exception objects are created from the **java.lang.Exception** class. It represents an unusual condition that arise in the course of program execution, such as reaching the end of a file, or attempting to reference an array element outside the actual bounds of the array:

Exceptions are always thrown, i.e., after an exception has been created, Java sends the exception object to the program. This process is known as "**throwing an exception".** An exception interrupts the normal flow of program execution. On error, it displays the error message and terminates the program. To override this default functionality, exceptions can be trapped using the **try/catch** construct. Java exception handling is thus built upon four keywords: **try, catch, finally** and **throw**.

4.3 USING TRY AND CATCH

A try statement is used to handle exceptions followed by one or more associated catch blocks. Any code block that has to be monitored for exceptions can be wrapped inside the try block and the associated catch block must immediately follow the try block. The syntax is as follows:

```
Try
{
<statement>
}
catch
```

Any statement that may produce an error can be placed under the try and catch construct. If an exception is thrown from the try block, the exception is captured and processed by the appropriate catch block. The catch block must always specify the class of the exception object it handles. The catch block may, or may not have any code block.

4.4 HANDLING MULTIPLE EXCEPTIONS

Sometimes when using the try and catch construct, the code in the try block may throw multiple types of exceptions. In these cases multiple catch blocks can be coded, and each made to catch a different exception. The exception type is checked against the arguments of the catch block in sequential order. If a match is found, then the catch block is executed. This gives the ability to catch specific errors and to write code to handle each exception independently.

Example:

```
try
{
int x;
x = 1/(x[0};
}
catch (ArithmeticException e)
{
<statement>
}
catch(ArrayIndexOutOfBoundsException e)
{
<statement>
}
```

After a catch block has started execution, the program flow will continue in one of three ways:

1) A return statement returns the control to the function that had called the current function.
2) The catch block throws a new exception, or re-throws the same type of exception, which is then propagated to the next level of exception handling. (try/catch construct can be nested).
3) The program flow reaches the end of the each block, from where it passes to the code that follows the try/catch construct. Flow encounters the method's ending brace, where it returns to the calling method.

4.5 SEQUENCING CATCH BLOCKS

In case of multiple exceptions, the ordering of the corresponding catch block must be properly taken care of as a catch block catches an exception that matches its arguments type, or any subclass of that argument type. See the example in the following code segment:

```
try
{
        <statement>
}
catch(Exception e) {…<statements>…}
catch(ArithmeticException e ) {…<statements>…}
catch (ArrayOutofBoundsException e) {…<statements>…}
```

The first catch block will always get executed because all other exceptions are derived from the class Exception.

☞ **Check Your Progress 1**

1) What do you understand by the process known as **"throwing an exception"?**

……...

……...

……...

……

2) What is the basic construct in Java to support the exception handling feature?

……...

……...

……...

……

4.6 USING FINALLY

The **finally** block is an optional portion of the **try** statement. There can be only one finally block for one try statement, and it must immediately follow the last catch

block. Typically, it is used to clean up after executing exception handling code. It gets executed in the following situations:

1) If no exception is thrown from the try block, then control passes to the **finally** block after the closing brace of the try block.
2) If no catch block is specified, or none of the catch blocks match the exception.
3) If a matching catch block exists, it gets executed, after which the catch block control comes to the finally block.

For example:

```
try{
<statements>
}
catch(Exception e) {;}
finally
{
<statement>
}
```

4.7 BUILT-IN-EXCEPTION

These are logical errors that are identified and thrown by the system but trapping and handling them is the responsibility of the programmer. Some of the built in exceptions are as follows:

NullPointerException: raised when a variable is used without initializing.

ArithmeticException: raised when a number is divided by zero.

ArrayIndexOutofBoundsException: raised when an array element which does not exist is accessed.

ClassNotFoundException raised when a specified class file is not found.

StringIndexOutofBounds Exception raised when attempting to access a string element which does not exist is attempted to be accessed.

IOException deals with file related errors.

4.8 THROWING EXCEPTIONS

Using Throw

One or more exceptions can be thrown in any method. Follow these steps to use **throw** in methods:

1) In the method declaration, specify the type of exception that can be thrown using the throws modifier.

2) Determine the situation when an error condition, which matches the specified exception type, occurs and then, dynamically create an instance of the exception class, or one of its subclasses for that error condition.

3) **throw** the exception object. When the exception is thrown, pass a string as an argument that can be used by the calling code to display an error message to the user.

The general syntax for throwing an exception is as follows:

```
class Classname
{
        public int MethodName()throws Exception
        {
                <statement>
                if (some condition)
                {
                        throw new Exception ("some description of exception");
                }
        }
}
```

☞ Check Your Progress 2

1) What is the basic purpose of the **finally** block?

……...

……...

……...

2) List three built-in-exceptions with its meaning.

……...

……...

……...

3) What is **throw**?

……...

……...

……...

4.9 CATCHING EXCEPTIONS

Consider the following example, in which the **inrange method** throws an exception. The thrown exception in **inrange method** must be trapped when it is called using the try/catch clause. The exception must be trapped in the **main** method. When the compiler encounters and exception, it displays the exception's string and halts the execution of the main method.

Example:

```
class Validation
{
        public void inrange (int n) throws Exception
        {

                if(n>0)&&(n<10000)
```

```
                System.out.printin ("is a number");
        else
                throw new Exception (" "+n+" is not a number");
         }
}
public class trial
{
        public static void man (String s [ ])
        {
        int num = 0;
        num=Integer.parselnt(s[0]);
        Validation v = new Validation ();
        try
        {
                v.inrange(num);
        }
        catch (Exception e)
        {
                System.out.printIn(e.toString();
                return;
        }

        System.out.printIn(num);
        }
}
```

4.10 USER-DEFINED EXCEPTIONS

Besides using the default exceptions, a programmer can also create his own set of exceptions. The advantage of creating a user-defined exception class is as follows:

1) It can hold more information about the error condition than a standard class. This becomes specially more important when the error condition is more complicated.

2) The user can tailor the catch blocks to the specific exception type, instead of catching a generic exception. This allows the user's code to attain greater control over program execution.

In order to have a user defined exception, create an exception class that is specific to the inrange code error condition. This new class can be created by extending **Exception**.

Example:

```
Class Inrange Exception extends Exception
{
        InrangeException (int wrongnum)
        {
                Super (","+wrongnum+ "is not a valid number");
        }
}

class Validation
{
        //To throw the user defined exception
        public void inrange(int n)throws Exception
        {
```

```
        if (n>0) && (n<10000)
                System.out.printIn ("is a number");
        else
                throw new InrangeException(n);
        }
}
//To catch the user defined exception:
public class Trial
{
        public static void main (String s [ ])
        {
                int num = 0
                num = Integer.parseInt(s[0];
                Validation v = new Validation ();
                try
                {
                        v.inrange(num);
                }
                catch(Exception e)
                {
                        System.out.printIn(e.toString ( );
                        return;
                }
                System.out.printIn ("number:"+num);
        }
}
```

☞ Check Your Progress 3

1) Describe the following classes

   (i) Exception class
   (ii) Runtime Exception
   (iii) I/O Exception

   ……

   ……

   ……

   ……

   ……

2) What is the difference between throws, and keyword throw?

   ……

   ……

   ……

   ……

4.11 SUMMARY

In this Unit, we looked at exception handling features in Java programming which are required for writing a robust program.

A java exception is an instance of a class derived from throwable. The throwable class is contained in the java language package and subclass of throwable is contained

in various packages. For example, errors related to AWT are included in the java awt package, the numeric exceptions are included in the Java language package because they are related to Java language. You can create your own exception classes by extending throwable or a subclass of throwable.

4.12 SOLUTIONS / ANSWERS

Check Your Progress 1

1) Exception represents an unusual condition that arises in the course of program execution, such as reaching the end of a file, or attempting to reference an array element outside the array boundary. **"throwing an exception"** is a process in which Java sends the exception object to the program after an exception has been created.

2) The basic construct in Java to support exception handling feature is:
Try
{
<statement>
}
catch

Check Your Progress 2

1) **"finally block"** is an optional portion of a **try** statement. There can be only one **finally** block for one try statement, and it must immediately follow the last catch block. It is executed in the following situations:

   (i) If no exception is thrown from the try block, then control passes to the **finally** block after the closing brace of the try block.

   (ii) If no catch block is specified, or none of the catch blocks match the exception.

   (iii) If a matching catch block exists, it gets executed, after which the catch block control comes to the finally block.

2) The three built in exceptions are as follows:

   (i) **NullPointerException:** raised when a variable is used without initializing.

   (ii) **ArithmeticException:** raised when a number is divided by zero.

   (iii) **ArrayIndexOutofBoundsException:** raised when an array element which does not exist is accessed.

3) Using **Throw**, one or more exceptions can be thrown in any method.

Check Your Progress 3

1) (i) The Exception class describes the errors caused by your program and external circumstances. These errors can be caught and handled by your program. Exception has many subclasses. Examples are Runtime Exception, and I/O Exception.

(ii) The Runtime Exception class defines programming errors, such as bad casting accessing out of bound array and numeric arrays. Example of subclass of Runtime Exception are arithmetic Exception Null Point Exception etc.

(iii) The I/O Exception class describes errors related to I/O operation, such as invalid input reading past the end of a file, and opening a non existing file.

2) The keyword to claim an exception is throws and the keyword to throw is exception in throw.