04 Exception Handling and I/O operations

IN THIS CHAPTER

Exceptions are certain abnormal conditions or errors that occur at runtime and can cause an abrupt termination of a program. A program should be able to handle exceptions efficiently and can therefore overcome the problem of sudden program termination resulting from the exception. Let's suppose that you created a Java program to perform division operations, but by mistake, used 0 to divide a number. This mistake generates Arithmetic Exception (a built-in Java exception). Java provides various techniques to handle exceptions. Apart from providing techniques to handle exceptions, Java also provides Input/Output (I/O) framework, which provides a simple, standardized Application Programming Interface (API) for reading and writing the data that is in the form of characters or bytes, from various data sources. The I/O framework provides the java.io package, which contains a collection of the classes, such as `PrintWriter`, `FileInputStream`, `FileOutputStream`, which provides useful abstractions and better I/O performance. These classes are wrapper classes of the `InputStream` and `OutputStream` classes. Input streams are streams that receive or read data, whereas output streams are streams that send or write data. Classes in the java.io (input-output) package represent all streams.

The chapter discusses how to handle exceptions in Java with the help of try and catch block. Moreover, the exception hierarchy and the concept of nested try and catch are also discussed. Apart from exception handling, the concept of handling I/O operations is also discussed with the help of the classes within the java.io package. Let's start with handling exceptions.

4.1 Handling Exceptions

When you create a Java program and it compiles successfully, you may feel sure that it will execute successfully as well. However, you can expect program interruptions due to various uncertainties arising from the programmer's ignorance. The mechanism of handling such uncertainties in a Java program is known as exception handling. In a Java program, you can handle such uncertainties by using try and catch block.

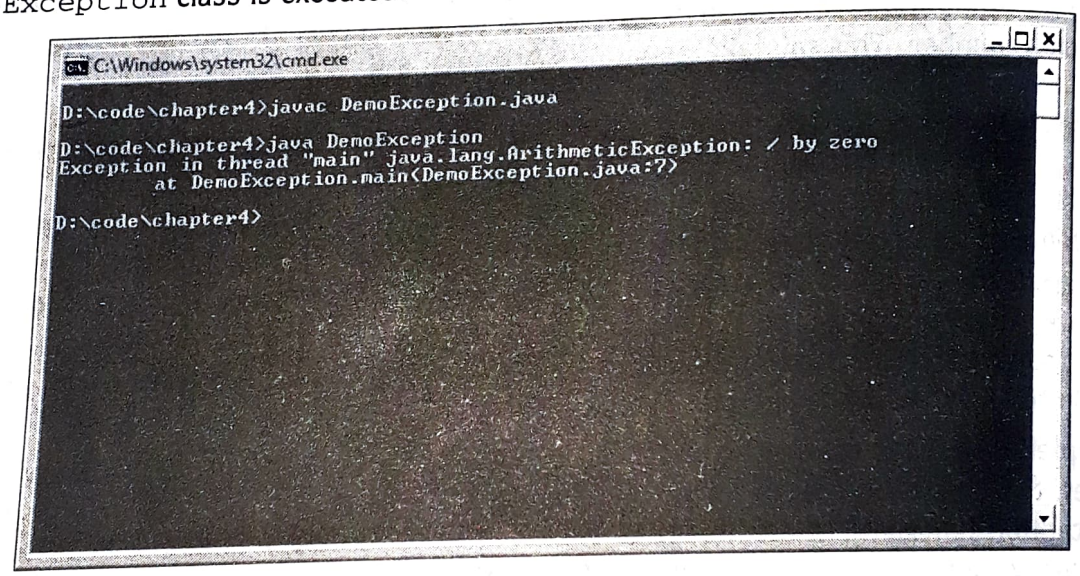
For example, while writing a method to divide two numbers, the programmer may have divided a number by 0, which is not allowed. Dividing a number by 0 will not lead to a compilation error but it will cause program interruption during execution. A program can also be interrupted when a file is not found or there is a memory related problem such as stack overflow. Listing 1 declares the `DemoException` class to explain exceptions that may occur

during the execution of a program (you can find the DemoException.java file on the CD in the code\chapter4 folder):

▶ **Listing 1: The DemoException.java File**

```
class DemoException
{
 public static void main (String args[])
 {
     int x=5;
     int y=0;
     int z=x/y;
     System.out.println("Result of the program is " + z);
 }
}
```

In Listing 1, the value of the `x` variable is divided by the value of the `y` variable (0), which is not allowed and this leads to the program being interrupted. This happens even though the program is syntactically error-free. Figure 1 shows the exception generated when the `DemoException` class is executed:



▲ **Figure 1: Displaying the Exception thrown by the DemoException Class**

Figure 1 shows the cause for the program interruption so that it can be fixed. Here, you can see the `ArithmeticException` exception, which is generated when there are problems related to mathematical calculations in the program's logic. Although, some information about the problem is displayed, but you should have detailed knowledge about exceptions, their types, and the reasons they are raised. Here, we discuss the exceptions in detail under the following topics:

- Errors and Exceptions in a Java Program
- Exception Handling
- Handling Multiple Exceptions
- Types of Exceptions

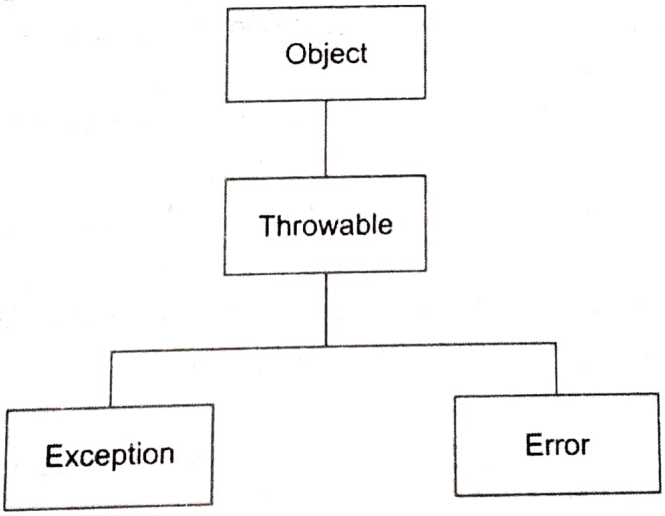
Let's discuss these one by one.

Errors and Exceptions in a Java Program

There are of two types of errors in Java: `compile time` and `run-time errors`. Compile time errors occur due to syntactical problems in a program, whereas run-time errors occur due to certain conditions such as memory failure, or file not found error. Exceptions are events that occur during program execution and interrupt the flow of a program. Whenever an exception occurs, an object of the Exception class is created, which contains information about the exception, such as the type of exception and the state of the program when the exception occurred. After the object is created, it is thrown in the method that caused the exception. The Java runtime system then searches for the exception handling techniques to handle the exception raised in the method that caused the exception. If no exception handling technique is found, the Java runtime system searches the Java class in which the method raising exception is defined for exception handling techniques. In case, no technique is found to handle exceptions, the Java runtime system terminates and the program is interrupted. All types of exceptions come under the `Exception` class. Let's now discuss about the hierarchy of the Exception class in the following section.

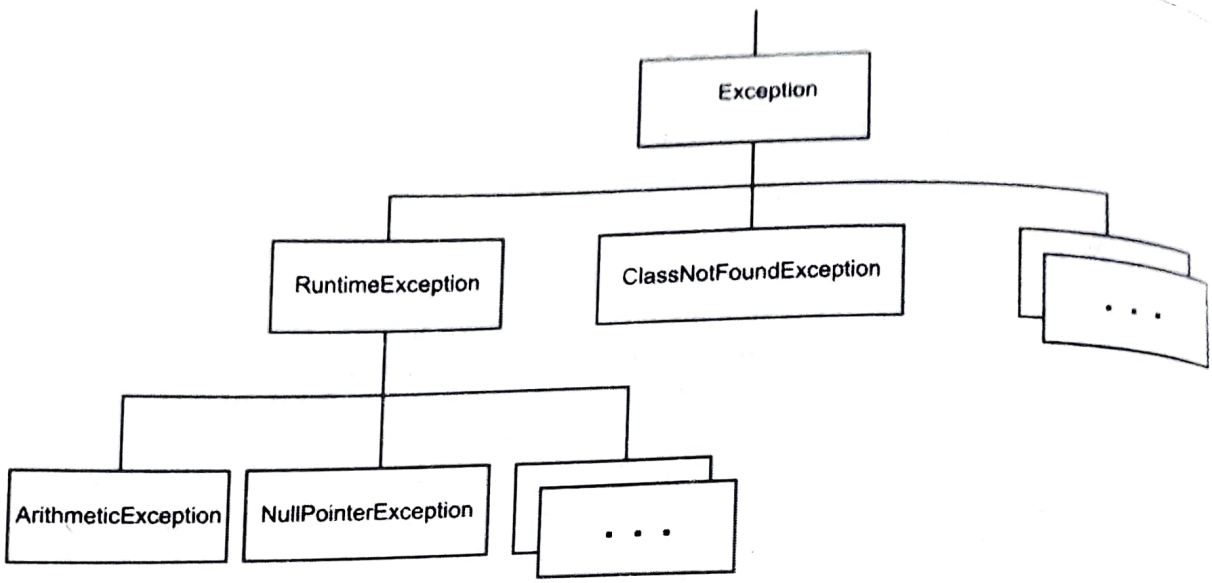
Exception Hierarchy

All the possible exceptions supported by Java are organized as subclasses in a hierarchy under the `Throwable` class. The `Throwable` class is an immediate subclass of the Object class, which is located at the root of the exception hierarchy. Let's take a look at the exception hierarchy, as shown in the Figure 2:



▲ **Figure 2: Displaying the Exception Hierarchy**

Figure 2 shows that the Exception class is the superclass of all exception classes and the Error class is the superclass of all run-time error classes. These error classes represent errors that are not expected to be caught under normal conditions by a Java program. In addition, the Exception and Error classes are subclasses of the `Throwable` class. The `Throwable` class is itself a subclass of the `superclass` of all Java classes, the Object class. The Object class comes under the default Java package `java.lang`, but this does not imply that all exceptions come under this package. For example, I/O exceptions such as `FileNotFoundException` and `EOFException` come under the `IOException` class of the `java.io package`. Figure 3 shows the hierarchy of the `Exception` class:



▲ **Figure 3: Displaying Exception Hierarchy of the Exception Class**

Figure 3 shows the two subclasses of the `Exception` class, that is, `RuntimeException` and `ClassNotFound`. Apart from these classes, there are also other subclasses of the Exception class, which are not shown in the hierarchy structure to avoid complexity. The following is the list of the important subclasses under the Exception class:

- `RuntimeException`: Thrown when the arithmetic operations performed in a program are incorrect and consist of common exceptions such as `ArithmeticException`, `IndexOutOfBoundException`.
- `ClassNotFoundException`: Thrown when a class that needs to be loaded is not found.
- `CloneNotSupportedException`: Thrown when a method does not implement the `Cloneable` interface but uses the `clone ()` method.
- `IllegalAccessException`: Thrown when a method is called in another method/class. However, the calling method/class does not have permission to access that method.
- `InstantiationException`: Thrown when an `abstract class` or an `interface` is instantiated.
- `InterruptedException`: Thrown when a thread in sleeping or waiting state is interrupted by another thread.
- `NoSuchFieldException`: Thrown when an unidentified variable is used in a program.
- `NoSuchMethodException`: Thrown when an undefined method is used in a program.

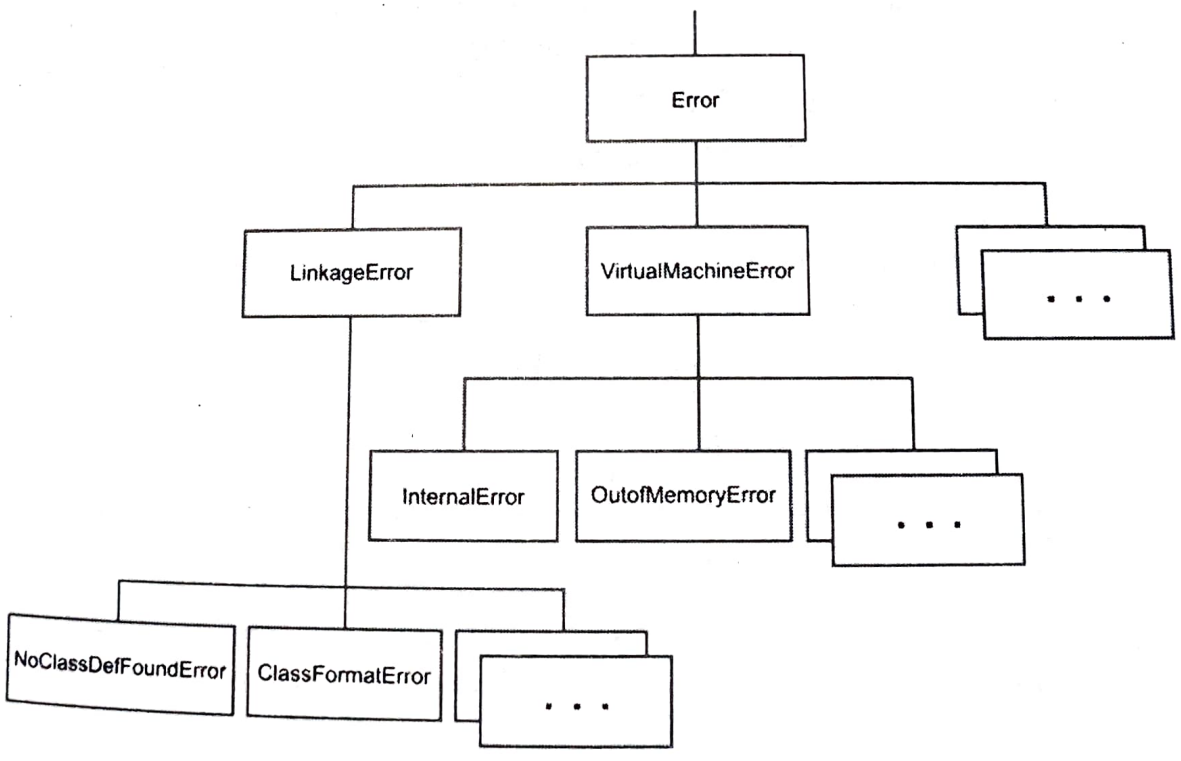
The `RuntimeException` class consists of several other exception subclasses that are used to handle specific types of exceptions. The subclasses of the `RuntimeException` class are:

- `ArithmeticException`: Thrown when an arithmetical problem, such as when a number is divided by 0, is encountered.
- `IndexOutOfBoundsException`: Thrown when an array or String is going out of the specified index. This exception class has two further subclasses: `ArrayIndexOutOfBoundsException` and `StringIndexOutOfBounds Exception`. The `ArrayIndexOutOfBoundsException` exception is thrown when an array element out of an index is accessed. The `StringIndexOutOfBoundsException`

exception is thrown when a String or the StringBuffer element out of the index is accessed.

- `ArrayStoreException`: Thrown when you try to store any value in an array, which is not of the array type. For example, suppose that an array is of integer type but the value to be stored in the element is of another type.
- `ClassCastException`: Thrown when inappropriate types are typecast.
- `IllegalArgumentException`: Thrown when an illegal argument is passed to a method. This exception class has two further subclasses: `IllegalThreadStateException` and NumberFormatException. The `IllegalThreadStateException` exception is thrown when you try to perform any operation on a thread, which is not compatible with thread's current state. The `NumberFormatException` exception is thrown when a numeric value is parsed into a string but the parsing process fails.
- `IllegalMonitorStateException`: Thrown when a thread that does not have an object's monitor rights tries to access the object's `wait ()`, `notify ()`, and `notifyAll ()` methods.
- `IllegalStateException`: Thrown when the runtime environment is not in an appropriate state to call any method.
- `NegativeArraySizeException`: Thrown when you try to create a negative size array in a program.
- `NullPointerException`: Thrown when an object is accessed through a null object reference.
- `SecurityException`: Thrown when a method violates the security policies defined in Java.

You must remember these exception classes and subclasses, and the reason why a certain exception is thrown. This information will help you to handle exceptions easily. Figure 4 illustrates the hierarchical structure of the Error class:



▲ **Figure 4: Exception Hierarchy of the Error Class**

Figure 4 shows the two subclasses of the Error class, `LinkageError` and `VirtualMachineError`, and these subclasses are further divided into their subclasses. Apart from the `LinkageError` and `VirtualMachineError` subclasses, the following is the complete list of subclasses of the Error class:

- `LinkageError`: Throws an error when there is a problem in linking a class reference to other class. This subclass consists of various other subclasses such as `ClassFormatError`, `NoClassDefFoundError`.
- `VirtualMachineError`: Throws an error when the Java Virtual Machine (JVM) encounters an error. This subclass consists of various other subclasses such as `InternalError`, `OutOfMemoryError`.
- `ThreadDeath`: Throws an error when the `stop()` method of a Thread object is called to kill a Thread.
- `AssertError`: Throws an error when an assertion evaluates to false.

All exceptions are broadly divided into two categories: checked exceptions and unchecked exceptions. Exceptions that are checked at compilation time are called checked exceptions whereas, unchecked exceptions are not checked at compilation time. Checked exceptions are the exceptions that a well-written program can catch and notify a user by providing the reason why the exception occurred. For example, the ArithmeticException exception notifies the user that the exception is due to an error in arithmetic calculation. All the exceptions in the Throwable class, except for those that come under the Error and RuntimeException classes and their subclasses, are considered checked exceptions. Let's now discuss how exceptions are handled.

Exception Handling

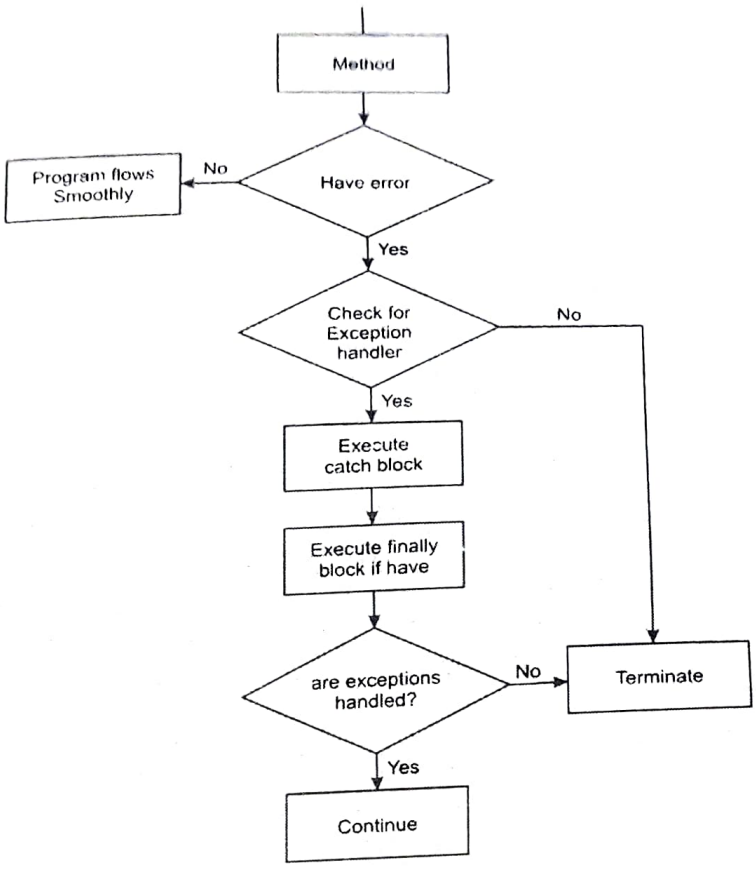
In Java, we have various techniques such as try and catch to handle exceptions. These techniques use various clauses such as try-catch blocks, finally clause, throws clause, and throw clause. You can use these techniques to create programs in which the exceptions are properly handled. This, in turn, helps in creating well designed applications. Exception handling is discussed under following subheading:

- Using the try and catch Block
- Using Nested try and catch Block
- Using the finally Clause
- Using throws Clause
- Using throw Clause

Let's discuss these one by one in detail.

Using the try and catch Block

The try block in a Java program is used to enclose a statement that may lead to an exception. A try block must be followed by a catch block. The catch block acts as an exception handler and contains the object of the specified exception class, which handles the raised exception. The code written in the catch block is executed when the exception handler is invoked by the Java runtime system. Figure 5 shows the flow structure of handling exceptions using the try-catch construct:



▲ **Figure 5: Flow Structure of the try-catch Construct**

Figure 5 shows the process of using the try-catch construct in exception handling. When a program is executed, its methods are checked for errors and if any error is encountered, their corresponding exception handler techniques are searched. If no exception handling technique is found, the program abruptly terminates. However, if the exception handling technique is found then the code written in the catch block is executed. The following code snippet represents the basic working structure of the try-catch construct:

```
try {
  // code
} catch (exception_class var) {
  // code to be executed when exception is thrown
}
```

In the preceding code snippet, `exception_class` can be any exception-related class or subclass and `var` can be any variable name that refers to the exception stack where the exception details are stored. Some rules for using the try-catch construct are:

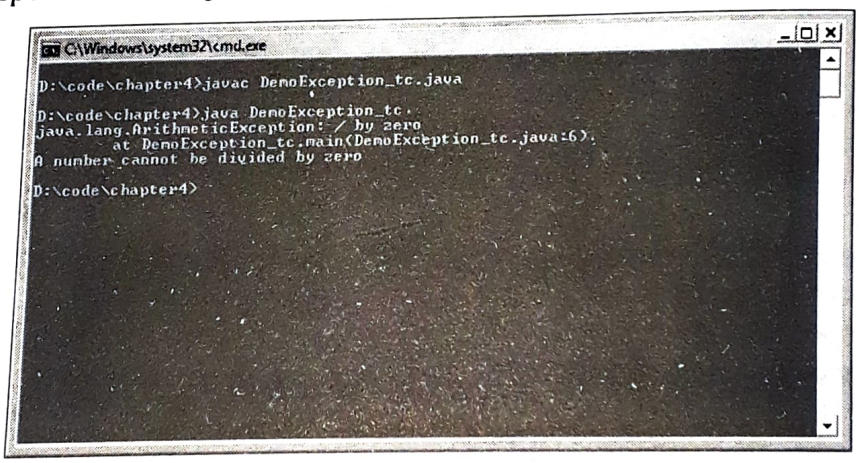
- A try block cannot be used without a catch or finally block.
- A catch block must be followed by a try block, that is, there should be no statement between the end of the try block and the beginning of the catch block.
- A finally block cannot come before the catch block.

Listing 2 declares the DemoException_tc class, which uses the Exception class to handle the raised exception (you can find the DemoException_tc.java file on the CD in the code\chapter4 folder):

▶ **Listing 2: The DemoException_tc.java File**

```
class DemoException_tc {
 public static void main(String args[]) {
     try {
         int x=5;
         int y=0;
         int z=x/y;
         System.out.println("Result of the program is " + z);
     } catch(Exception e) {
     e.printStackTrace ();
     System.out.println("A number cannot be divided by zero " );
     }
 }
}
```

In Listing 2, the `ArithmeticException` exception is generated when the value stored in x is divided by 0, which is illegal. The `ArithmeticException` class is a subclass of the Exception class and hence, it is used to handle all other arithmetical exceptions. The `printStackTrace()` method displays the exception stored in the Exception stack. Figure 6 shows the exception thrown by the `DemoException_tc` class:



▲ **Figure 6: Displaying the Handling of the ArithmeticException**

Figure 6 shows the `ArithmeticException` exception with the reason for the exception, and the line number in Listing 2 that caused the exception. Apart from this, additional information is also displayed, which helps the user to understand the reason for the exception. In Listing 2, we used the Exception class to handle the exception, which is not recommended, especially when you have some idea of the exception that can interrupt the program flow. For example, in Listing 2, we know that the ArithmeticException exception may be thrown. In this case, we can directly use the `ArithmeticException` class instead of the Exception class. This example illustrates the utility of the try and catch blocks in exception handling.

TEST YOUR KNOWLEDGE

Q1. Write a program to handle an **`ArrayIndexOutofBoundsException`** in a Java program.

Ans.

```
class TestYourKnowledge1{
public static void main(String args[]) {
try{
```

```
            int val[]={5,10,15};
            int x;
            for(x=0;x<=3;x++)
            {
                System.out.println(val[x]);
            }
        }
        catch(Exception e) {
        e.printStackTrace ();
        System.out.println("No fourth element in the array " );
        }
    }
    }
```

Execution:

```
D:\code\chapter4>javac TestYourKnowledge1.java
D:\code\chapter4>java TestYourKnowledge1
```

Output:

```
5
10
15
java.lang.ArrayIndexOutOfBoundsException:3
at TestYourKnowledge1.main(TestYourKnowledge1.java: 9)
No fourth element in the array
```

Q2. Write a program, which can raise arithmetic exception.

Ans.

```
class TestYourKnowledge2{
 public static void main(String args[]) {
     try {
         int x=5;
         int y=0;
         int z=x/y;
         System.out.println("Result of the program is " + z);
     } catch(Exception e) {
     e.printStackTrace ();
     System.out.println("A number cannot be divided by zero " );
     }
 }
}
```

Execution:

```
D:\code\chapter4>javac TestYourKnowledge2.java
D:\code\chapter4>java TestYourKnowledge2
```

Output:

```
java.lang.ArithmeticException: / by zero
at TestYourKnowledge2.main(TestYourKnowledge2.java:6)
A number cannot be divided by zero
```

Using Nested try and catch Block

So far, we learned about using a single try-catch block in a Java program. However, you can also nest multiple try-catch blocks inside one try block in a program. The nested try and catch block offers better exception handling since it encapsulates the statements that might generate an exception in a try block. Listing 3 shows how multiple try blocks are used in a Java program (you can find the DemoNestedTry.java file on the CD in the code/chapter4 folder):

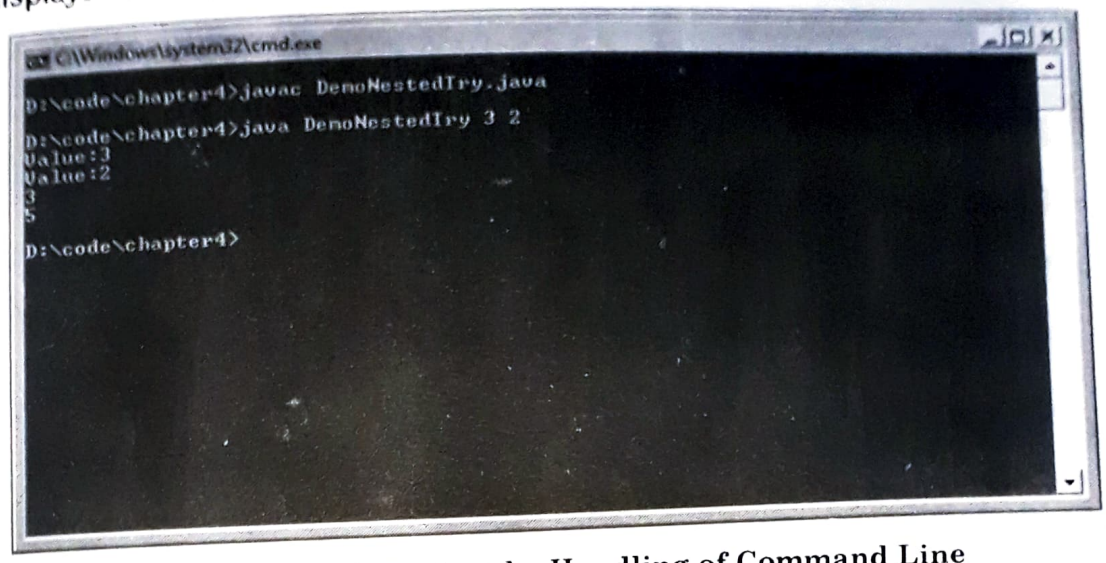
▶ Listing 3: The DemoNestedTry.java File

```
class DemoNestedTry
{
 public static void main (String args[])
 {
     try // first try block
     {
         int j, x=10, y;
         int k[]= new int[10];
         for (int i=0; i<args.length; i++)
         {
             System.out.println("Value:" + args[i]);
         }
         try // second try block inside first try block
         {
             for (j=0; j<args.length; j++)
             {
                 k[j]= Integer.parseInt(args[j]);
                 y=x/k[j];
                 System.out.println(y );
             }
         }
         catch (ArithmeticException ex) // catch block of second try
         {
             System.out.println("Exception is" + ex);
             System.out.println("Value is divided by zero, which is not
             allowed");
         }
 }
 catch(Exception ex1) // catch block of first try
 {
     System.out.println("Exception is" + ex1);
 }
 }
}
```

In Listing 3, we create a class name, `DemoNestedTry`, which receives values through command line arguments. We then display these values. The code to receive and display the values through command line arguments has been enclosed in a try block. Next, we write the code for dividing a variable, x, initialized with a value, 10, from the value retrieved through the command line. We enclose this code in another try block, which is nested inside the first try block. Next, in the catch block of the second try block, we use ArithmeticException to catch the exception that may be raised if a user enters '0' as a command line argument. We then write the catch block of the first try in which the Exception superclass is used to handle all types of exceptions.

Command line arguments are used to pass values to the main method. The main method has args[] as a String parameter to store the values. Command line arguments are passed at runtime.

Figure 7 displays the output of the `DemoNestedTry` class:



▲ Figure 7: Displaying the Handling of Command Line Arguments using the Nested Try Block

Figure 7 shows the values passed through the command line arguments in a Java program. In addition to it, the value retrieved through the command line is used to divide a variable x, which has been initialized with a value, 10. Then the console displays the result of this division as values (3, 5). Let's now discuss about the finally clause, which is another exception handling clause.

TEST YOUR KNOWLEDGE

Q3. Write a program to demonstrate the implementation of nested try and catch block.

Ans.

```
class TestYourKnowledge3
{
 public static void main (String args[])
 {
    try // first try block
    {
        int j, x=10, y;
        int k[]= new int[10];
        for (int i=0; i<args.length; i++)
        {
            System.out.println("Value:" + args[i]);
        }
        try // second try block inside first try block
        {
            for (j=0; j<args.length; j++)
            {
                k[j]= Integer.parseInt(args[j]);
                y=x/k[j];
                System.out.println(y );
            }
        }
```

```
            catch (ArithmeticException ex) // catch block of second try
            {
                System.out.println("Exception is" + ex);
                System.out.println("value is divided by zero,
            which is not allowed");
            }
        }
        catch(Exception ex1) // catch block of first try
        {
            System.out.println("Exception is" + ex1);
        }
    }
}
```

Execution:

```
D:\code\chapter4>javac TestYourKnowledge3.java
D:\code\chapter4>java TestYourKnowledge3 4 10
```

Output:

```
value:4
value:10
2
1
```

Using finally Clause

The finally block is used to display information or to execute statements, which needs to be executed irrespective of the catch block. This means that a finally block must be executed in spite of the fact that a catch block is executed or not. Some rules for using the finally block are:

- A finally block must be declared at the end of last catch block. If the finally block is declared before a catch block, then program will not compile successfully.
- Multiple finally blocks cannot be used with a try block. Doing so generates a compile-time error stating that a finally block cannot be used without a try block, even though we have used the try block in the program.

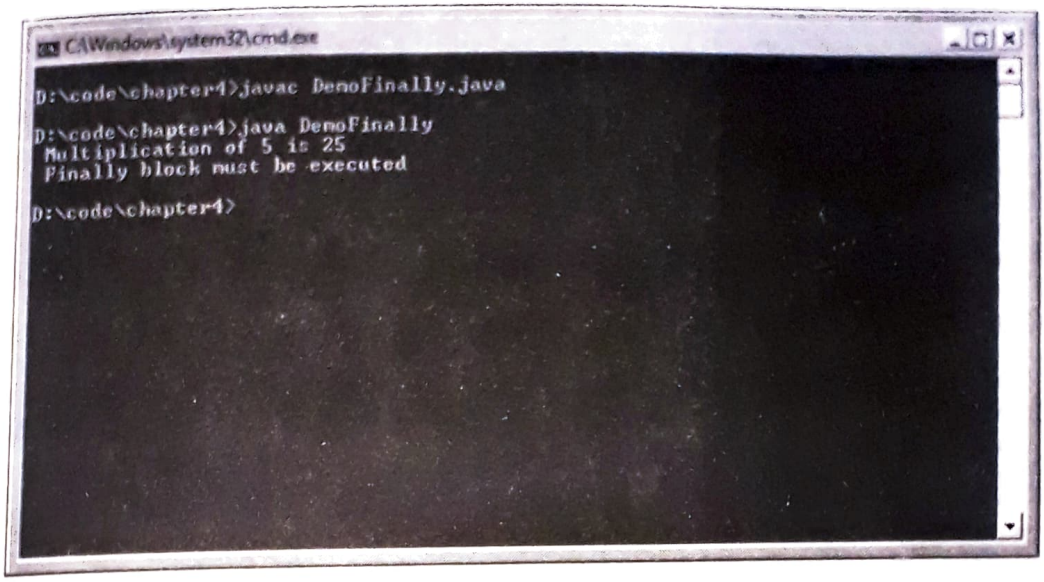
Now, let's discuss the implementation of the finally block. Listing 4 declares the DemoFinally class, which demonstrates the use of the finally block (you can find the DemoFinally.java file on the CD in the code\chapter4 folder):

▶ Listing 4: The DemoFinally.java File

```
public class DemoFinally
{
 public static void main(String args[]){
    int input, result;
    try{
        input= 5;
        result= input * input;
        System.out.println(" Multiplication of " + input + " is " +
        result);
    }catch (ArithmeticException a)
    {
```

```
            a.printStackTrace();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        finally {
            System.out.println(" Finally block must be executed");
        }
    }
}
```

Listing 4 shows multiplication of two numbers and implements exception handling using the try-catch-finally construct. Figure 8 shows the output of the `DemoFinally` class:



▲ **Figure 8: Displaying the Multiplication Result and Statement within the Finally Block**

Figure 8 raises no exception, when Listing 4 executes. This means that a finally block must be executed irrespective of the catch block. However, the statements in the finally block do not execute under the following conditions:

- When the `System.exit()` method is called before executing the finally block.
- When the `return` statement is used in a finally block, then the control is transferred to the calling routine and the statements after the return statement in the finally block are not executed.
- When an exception arises in the code written in the finally block.

TEST YOUR KNOWLEDGE

Q4. Write a program to demonstrate the finally clause.

Ans.

```
public class TestYourKnowledge4 {
  public static void main(String args[]){
      int val, square;
      try{
          val= 9;
          square= val * val;
          System.out.println(" The Square is " + square);
      }catch (ArithmeticException a)
```

```
        {
            a.printStackTrace();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        finally {
            System.out.println(" Finally block must be executed");
        }
    }
}
```

Execution:

```
D:\code\chapter4>javac TestYourKnowledge4.java
D:\code\chapter4>java TestYourKnowledge4
```

Output:

```
The Square is 81
Finally block must be executed
```

Using throws Clause

The throws clause is used when you know that a method may cause exceptions, but the method is incapable of handling exceptions. In such a case, a user has to throw those exceptions to the caller of the method by using the throws clause. A throws clause is used with a method declaration. The syntax of using the throws clause is:

```
return_type method_name (para) throws excep1, excep2, excep_n {
  // method body
}
```

In this snippet, `return_type` is any logical return type according to the values that a method returns. `method_name` is any logical method name. `para` is a list of parameters that needs to be passed in the method. throws is the clause used to throw exceptions. excep1, excep2, and `excep_n` are the exceptions that you want to throw. Multiple exceptions can be thrown. If you want to throw more than one exception, separate them with commas.

The compiler does not check for the exception type that is thrown using the throws clause. Some of the rules that must be kept in your mind when using the throws clause are:

- A subclass method overrides the methods of the superclass, which means that the overriding method also overrides the exceptions thrown in the overridden method.
- An overriding method must throw the exception that is either the same as the overridden method or a subclass of the exception thrown in the overridden method.

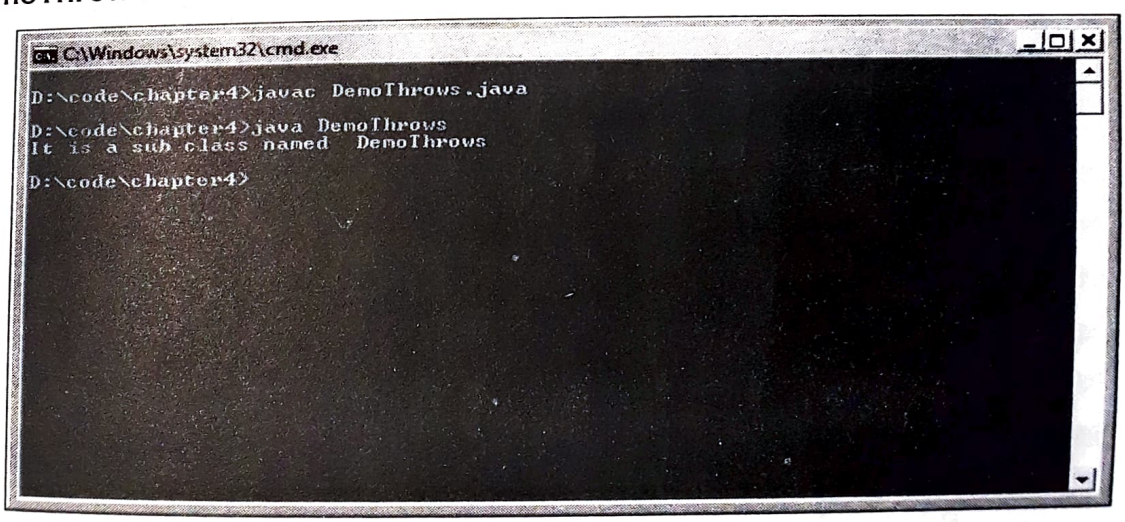
Listing 5 shows the implementation of overriding and overridden methods that are throwing exceptions (you can find the DemoThrows.java file on the CD in the code\chapter4 folder):

▶ Listing 5: The DemoThrows.java File

```
import java.io.*;
class ABC
{
```

```
    public void disp() throws IOException
    {
        System.out.println("It is a super class");
    }
}
class XYZ extends ABC
{
    public void disp() throws IOException
    {
        System.out.println("It is a super class");
    }
}
class DemoThrows extends ABC
{
    public void disp() throws EOFException, FileNotFoundException
    {
        System.out.println("It is a sub class named  DemoThrows.");
    }
    public static void main (String args[]) throws IOException
    {
        DemoThrows obj = new DemoThrows ();
        obj.disp();
    }
}
```

In Listing 5, class ABC is a superclass in which the `disp()` method is created. This method throws the IOException exception (`checked exception`). Class ABC is inherited by class XYZ, which also implements the disp() method and throws the IOException exception, which is legal. Another class called DemoThrows inherits class ABC but its methods throw the EOFException and FileNotFoundException exceptions, which is again legal because these are the subclasses of the IOException class. You can directly use the Exception superclass instead of specifying the Exception subclasses but this is not recommended, because the compiler may not give you the details of the exception raised in a program. Figure 9 shows the output of the DemoThrows class:



▲ **Figure 9: Displaying the Output of the DemoThrows Class**

Figure 9 shows that Listing 5 is compiled and executed successfully, which simply means that the code given in Listing 5 is correct. Let's take a look on the following code snippet from Listing 5 to explain the concept of overriding exceptions:

```
class XYZ extends ABC {
  public void disp() throws IllegalAccessException {
      System.out.println("It is a super class");
  }
}
```

In the preceding code snippet, the disp() method throws the IllegalAccessException exception, whereas in class XYZ in Listing 5 it throws the IOException exception. If you replace the preceding code snippet with the class XYZ given in Listing 5 and then compile the changed code (Listing 5), it will generate a compile-time error. This is because, the IllegalAccessException class is not a subclass of the IOException class.

You need to import java.io. package to use IOException class in a Java program. Importing packages in a Java program is discussed in Chapter 6, Multithreading and Packages in Java.*

TEST YOUR KNOWLEDGE

Q5. Write a program to show the usage of the throws clause.

Ans.

```
import java.io.*;
class LMN {
  public void disp() throws IOException {
      System.out.println("It is a super class");
  }
}
class XYZ extends LMN {
  public void disp() throws IOException {
      System.out.println("It is a super class");
  }
}
class TestYourKnowledge5 extends LMN {
  public void disp() throws EOFException, FileNotFoundException {
      System.out.println("It is a sub class named TestYourKnowledge5");
  }
  public static void main (String args[]) throws IOException {
      TestYourKnowledge5 obj = new TestYourKnowledge5();
      obj.disp();
  }
}
```

Execution:

```
D:\code\chapter4>javac TestYourKnowledge5.java
D:\code\chapter4>java TestYourKnowledge5
```

Output:

```
It is a sub class named TestYourKnowledge5
```

Using the throw Clause

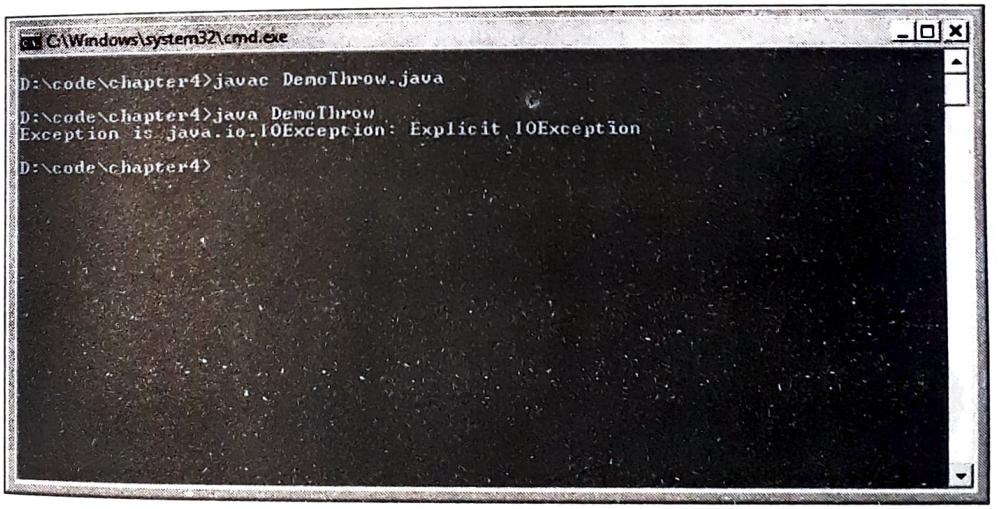
So far, we learned how exceptions are thrown implicitly. However, an exception can also be thrown explicitly. To throw an exception explicitly, an object of that exception type needs to be

created. To create an object of any exception type, the `throw` clause is used. Listing 6 shows how to create an object of an exception and throw the exception (you can find the DemoThrow.java file on the CD in the code\chapter4 folder):

▶ Listing 6: The DemoThrow.java File

```
import java.io.*;
class DemoThrow {
 public static void chk_throw() {
     try {
         // Creating object of IOException
         throw new IOException("Explicit IOException");
     }
     catch (IOException ne) {
         System.out.println("Exception is " + ne);
     }
 }
 public static void main (String args[]) {
     DemoThrow.chk_throw();
 }
}
```

Listing 6 shows the creation of an object of the IOException exception, which is thrown by the chk_thrown() method when the method is called without creating an object of the DemoThrow class. When the Java runtime system encounters such a condition, it throws the exception and interrupts the program flow. The Java runtime system then searches for the required exception handler and if it is not found, Java's default exception handler handles the exception. On the other hand, if the exception handler is found, then the code written in the exception handler is executed. Figure 10 shows the output of the DemoThrow class:



▲ **Figure 10: Displaying the Explicit Throwing of the IOException exception**

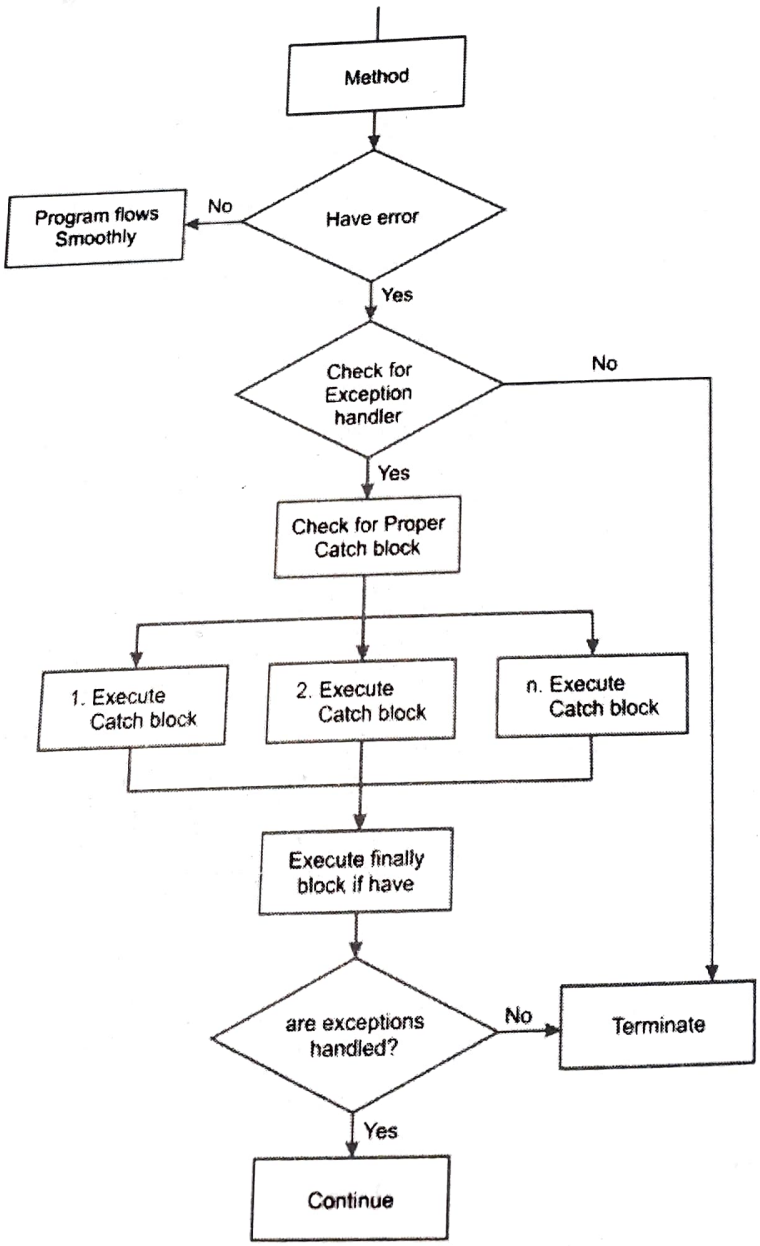
Figure 10 shows that the `IOException` exception is thrown because we tried to call the `chk_throw()` method without creating an object of the `DemoThrow` class. In other words, trying to call a method with a null reference raises the `IOException` exception.

Handling Multiple Exceptions

So far, you learned using only one catch block to handle exceptions and Exception, the super class of exceptions to perform exception handling. However, there are many exception classes

and subclasses, and remembering all of them is not easy. It may happen that you may forget to include some exceptions in the catch clauses that may lead to program interruption or termination. For example, the RuntimeException class has multiple subclasses but you may be familiar with only the ArithmeticException and NullPointerException subclasses. Let's imagine that a program raises many runtime exceptions, but you have used the ArithmeticException and NullPointerException subclasses to handle the raised exceptions. Now, the question arises that what will happen if any other runtime exception is raised. In that case, your program gets interrupted. Therefore, to avoid the interruption, use the Exception super class in the last catch block so that all the other remaining exceptions can be handled. The Exception class must be used in the last catch block; otherwise, an error will occur during compilation because a super class cannot be used before a subclass.

Multiple exception handlers can be used by using multiple catch blocks for a try block. If an exception is raised by using multiple catch blocks, the Java runtime system first looks for the exception handler with the specified exception class and if it is not found then the search continues for a superclass. Figure 11 shows the flow structure of handling exceptions using multiple try-catch blocks:



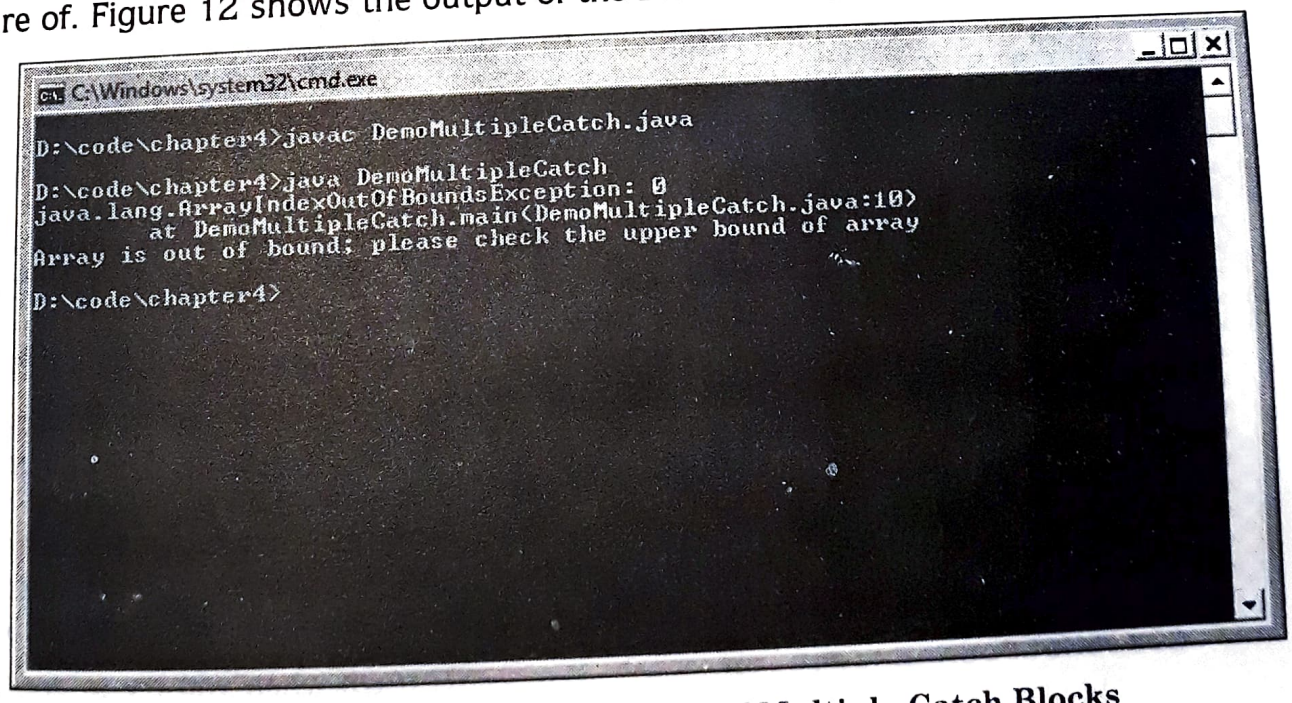
▲ **Figure 11: Flow Structure of Multiple try-catch Blocks**

Figure 11 shows that whenever an exception is raised, the Java runtime system first looks in the first catch block to find an exception handler and if it is not found then the search is continued in other exception blocks. The nth catch block in Figure 11 represents the catch block using the superclass. Listing 7 shows the implementation of multiple try-catch blocks (you can find the DemoMultipleCatch.java file on the CD in the code\chapter4 folder):

▶ **Listing 7: The DemoMultipleCatch.java File**

```
class DemoMultipleCatch {
  public static void main (String args[]) {
      try {
          int arr[] = { };
          for (int i=0; i<=arr.length; i++)
          {
              System.out.println("Value at position " + i + " is " + arr[i]);
          }
      }
      catch (ArrayIndexOutOfBoundsException a) {
          a.printStackTrace();
          System.out.println("Array is out of bound; please check the upper
          bound of array");
      }
      catch (Exception e) {
          e.printStackTrace();
      }
  }
}
```

In Listing 7, the array `arr` is not initialized but the elements of the array are accessed, which leads to an exception. In that case, the Java runtime system starts searching for an exception handler, finds the ArrayIndexOutOfBoundsException class and executes it. In Listing 7, we use the ArrayIndexOutOfBoundsException class instead of its superclass (IndexOutOfBoundsException) as an array is used, not a String. However, in the last catch block, the Exception class is used, because Listing 7 may raise an exception that you may not be aware of. Figure 12 shows the output of the `DemoMultipleCatch` class:



▲ **Figure 12: Displaying the Use of Multiple Catch Blocks**

Figure 12 shows the `ArrayIndexOutOfBoundsException` exception because the array elements are accessed when the array is not initialized. Let's now discuss the types of exceptions available in Java.

■ Types of Exceptions

Java allows users to handle exceptions that are supported by it. These exceptions are known as built-in exceptions. Apart from these Built-in exceptions, Java also allows users to create their own exceptions to be thrown in a Java program, and these are known as user-defined exceptions. In this section, we will explore the types of exceptions in the following sub-heads:

- Built-in Exceptions
- User-defined Exceptions

Let's discuss them one by one.

➡ Built-in Exceptions

Built-in exceptions are exceptions that are already available in Java. These exceptions are thrown by the Java runtime system to explain the exception condition. Built-in exceptions are further categorized into the following types:

- **Checked Exceptions:** The Checked exceptions are classes that are extended from the Exception class except the `RuntimeException` class. The Checked exceptions must be caught using either of the implicit or explicit exception handling techniques. The compiler is able to detect the checked exceptions during compilation-time itself. Some of the examples of Checked exceptions are `ClassNotFoundException`, `CloneNotSupportedException`, and `IllegalAccessException`.
- **Unchecked Exceptions:** The Unchecked exceptions are classes that are extended from the `RuntimeException` class. The compiler is not able to detect the unchecked exceptions at the time of compilation. The Unchecked exceptions are not required to be caught using either of the implicit or explicit exception handling techniques. Some of the examples of Unchecked exceptions are `ArrayIndexOutOfBoundsException`, `ArithmeticException`, and `NullPointerException`.

Table 1 explains some most commonly used built-in exceptions:

Table 1: Built-in Exceptions in Java

Exception	Description
`ArithmeticException`	Thrown when a problem in an arithmetic operation is noticed by the Java runtime system. For example, this exception is thrown when a number is divided by 0.
`ArrayIndexOutOfBoundsException`	Thrown when you try to access an array with an illegal index. Let's suppose you have declared an array to store a maximum of 5 elements, but while traversing the loop, you try to access the 6th element. In such a situation, this exception will be thrown.
`ClassNotFoundException`	Thrown when you try to access a class, which is not defined in your Java program.

Table 1: Built-in Exceptions in Java

Exception	Description
`FileNotFoundException`	Thrown when you try to access a non-existing file or one that fails to open.
`IOException`	Thrown when the input-output operation has failed or interrupted.
`InterruptedException`	Thrown when a thread is interrupted when it is in processing, waiting, or sleeping.
`IlligalAccessException`	Thrown when access to a class is denied.
`NoSuchFieldException`	Thrown when you try to use any field or variable in a class that does not exist.
`NoSuchMethodException`	Thrown when you try to access a non-existing method.
`NullPointerException`	Thrown when you refer the members of a null object.
`NumberFormatException`	Thrown when a method is unable to convert a String into a numeric format.
`StringIndexOutOfBoundsException`	Thrown when you try to access a String array with an illegal index.

These are some most common built-in exceptions. You can use these exceptions during exception handling to make your program more robust. Let's now discuss about user-defined exceptions next.

User-defined Exceptions

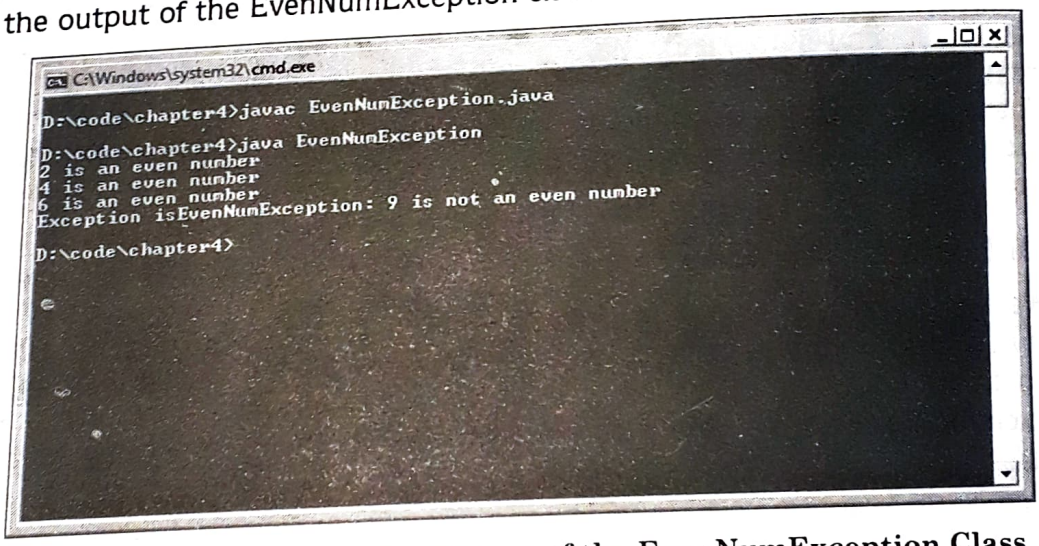
We know that Java provides various built-in exceptions. However, you can also create your own exception types to describe the exceptions related to your applications in your own way. To create your own exception types, you have to create an exception class as a subclass of the Exception class. To create a subclass of the Exception class, you have to inherit the Exception class. Listing 8 shows you how to create your own exception type (you can find the EvenNumException.java file on the CD in the code\chapter4 folder):

▶ Listing 8: The EvenNumException.java File

```
class EvenNumException extends Exception {
 EvenNumException(String str) // constructor of the class
 {
     super(str); // used super to refer the super class constructor
 }
 public static void main(String args[]) {
     try
     { int arr[]= {2,4,6,9};
         int var;
         int i;
         for(i=0; i<arr.length; i++)
         { var = arr[i]%2;
         if(var==0) {
             System.out.println(arr[i]+" "+"is an even number");
         }
```

```
                else
                { EvenNumException exp= new EvenNumException(arr[i]+" "+"is not an
                    even number");
                    throw exp;
                }
                }
            } catch(EvenNumException exp)
            {
                System.out.println("Exception is" + exp);
            }
        }
    }
```

In Listing 8, we create the EvenNumException class, which inherits the `Exception` super class. We then create a constructor of the EvenNumException class in which we use the `super` keyword. In Listing 8, we first declare and initialize an array in the main method. Next, we check whether the array elements are even or not. Then, using the `IF` statement, we check each array element for an even number, in case zero is returned as a value on being divided by 2. We then create an object of the EvenNumException constructor. Next, we use the throw clause, which is used when an integer value other than zero is returned for an array element divided by 2. In the catch block, we handle the EvenNumException exception. Figure 13 shows the output of the EvenNumException class:



▲ **Figure 13: Displaying the Output of the EvenNumException Class**

Figure 13 shows the array elements, which are even numbers. The array elements of the array declared in the Listing 8 are checked for the even numbers. The user-defined exception named `EvenNumException` is thrown when an integer value other than zero is returned and when an array element gets divided by 2. Since, division of 9 with 2 returns a value other than zero, the `EvenNumException` is thrown by the program.

By far, we discussed about handling exceptions in a Java program. Let's now discuss about Input/Output (I/O) operations.

4.2 Handling I/O Operations

Receiving values from a user helps in generating dynamic results as every time user may enter different values. Receiving values/data from a user is known as input and the result that the application generates on the basis of that input is known as output. The process of receiving

input and producing output by an application is known as I/O operations. Let's consider a scenario to clear the concept of I/O operations. Suppose, you want to create a program that calculates the total amount of items purchased by a customer. In this program, you show a menu to customer from which customer selects the items to purchase. When the customer selects an item, customer also needs to specify the quantity of that item which he wants to purchase. Once a customer enters the quantity, a message is shown to customer that if customer wants to purchase more items; if yes then press 'y' else press 'n'. If the customer presses 'y' then customer is again asked to select items to purchase but in case, customer presses 'n', a bill is generated having the total amount of the items that the customer has purchased. In this way, accepting values from a user at run time generates dynamic result. Therefore, the I/O operations are important to get the dynamic result. Java allows user to perform I/O operations with the help of streams. A stream can be defined as a path to communicate between the program and the source or destination of any information. The source of information can be a file or a console and the information can be saved in a file or displayed on console. Streams are considered as the basic of the all I/O operations as all I/O operations in Java are performed through streams. Following are the types of streams:

- **Byte Streams:** Byte streams are used to handle I/O operations on binary data. All the classes of byte stream are derived from InputStream and OutputStream classes. Therefore, InputStream and OutputStream classes are considered as the main classes of byte stream. Some of the examples of byte stream classes are DataInputStream, DataOutputStream, FileInputStream, and FileOutputStream.
- **Character Streams:** Character streams are the streams that represent data in the form of characters. Character streams are used to handle I/O operations on characters. To use the character stream classes, you have to import `java.io` package in your Java program. The two main classes of the character stream are `Reader` and `Writer`. These classes support `Unicode` character set (it is a 16 bit character set designed to support world's major languages such as latin, russian). Some of the examples of character stream classes are Reader, Writer, InputStreamReader, OutputStreamWriter, BufferedReader, BufferedWriter, FileReader, FileWriter, and PrintWriter.

Java also provides users a set of pre-defined streams. These pre-defined stream objects are `in, out, err` and are available in the `System` class. The `System` class is available in the default package, `java.lang` package. These pre-defined stream objects are used to specify the standard output and input streams. The `System.out` represents the standard output stream, which is the console. On the other hand, the `System.in` represents the standard input stream, which is the keyboard. Here we discuss the I/O operations in Java under the following topics:

- Reading Console Input
- Reading Characters
- Reading Strings
- Writing Console Output
- Reading and Writing Files

Let's discuss these in detail in next sections.

Reading Console Input

Java allows users to read data from the standard input device to be displayed on the console. The `System.in` is used to read input from the standard input device. Once the data is being read from the standard input device, the `System.in` should be wrapped in the `DataInputStream` object. This wrapping of the `DataInputStream` object allows users to obtain the byte stream attached with the console. The code to wrap the `System.in` in the `DataInputStream` is shown in the following code snippet:

```
DataInputStream obj=new DataInputStream(System.in);
```

In the preceding code snippet, the object named `obj` is created, which represents a byte stream connected to the console. The `DataInputStream` class allows an application to read binary data of Java primitive data types to an input stream in a portable way. This class, derived from the FilterInputStream class, is used to read only Java primitive data types and not object values. The constructor of the DataInputStream class creates a new data input stream that uses the specified input stream for reading the data. The read() method is used to read the data according to its types.

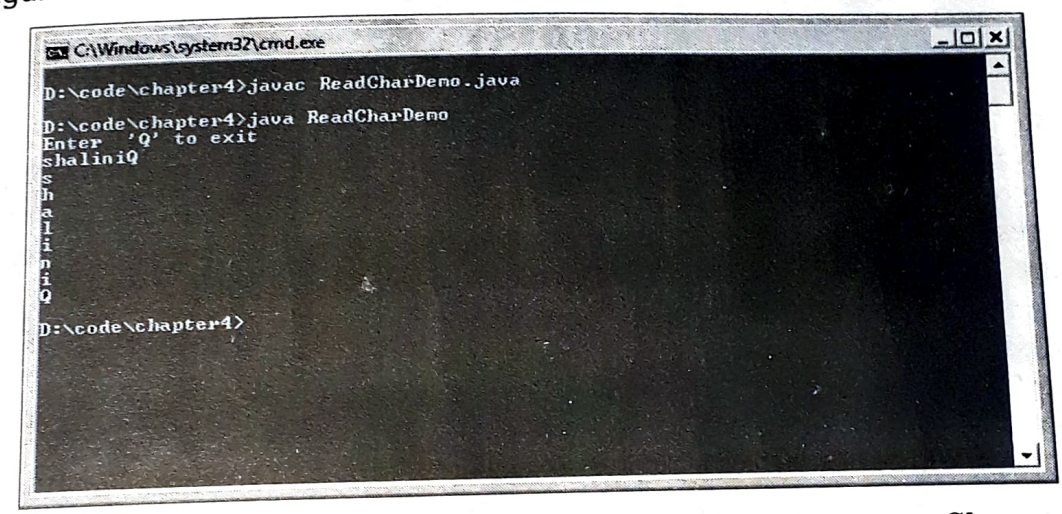
Reading Characters

To read input through console, you have to enter data at console. For example, you have created a program to perform addition on two numbers but you want that the numbers on which addition has to be performed are entered by user at runtime. In that case, you have to add the code to read console input in your program. Listing 9 shows the use of `System.in`, an object of `InputStream` class and `DataInputStream` class to read characters through console (you can find the ReadCharDemo.java file on the CD in the code\chapter4 folder):

▶ **Listing 9: The ReadCharDemo.java File**

```
import java.io.*;
class ReadCharDemo
{
 public static void main(String args[])
 {
    DataInputStream obj=new DataInputStream(System.in);
    char ch=' ';
    try
    {
        System.out.println("Enter  'Q' to exit");
        while(ch!='Q')
        {
            ch=(char)obj.read();
            System.out.println(ch);
        }
    }
    catch(Exception ex)
    {
        System.out.println(ex);
    }
}
}
```

In Listing 9, a class named ReadCharDemo is created in which object 'obj' of DataInputStream class is created. Thereafter, a variable named 'ch' is initialized to ' '. The user is prompted to enter 'Q' to exit from the program. Then, a while loop is used, which displays values entered by the user until 'Q' is entered. The value 'Q' allows the user to exit from the program. Inside the while loop, the read() method is used to read a character entered by the user. We used a pre-defined stream named out in the Listing 9. The println() method of the out object is used to print the text on the console. We will discuss the purpose of out object in detail under the section, *Writing Console Output* later in this chapter. Figure 14 shows the output of the ReadCharDemo class:



▲ Figure 14: Displaying the Output of the ReadCharDemo Class

The output shown in the Figure 14 highlights the limitation of the read() method. The data from console is not passed to the program until you press ENTER. This makes the console input less interactive.

Reading Strings

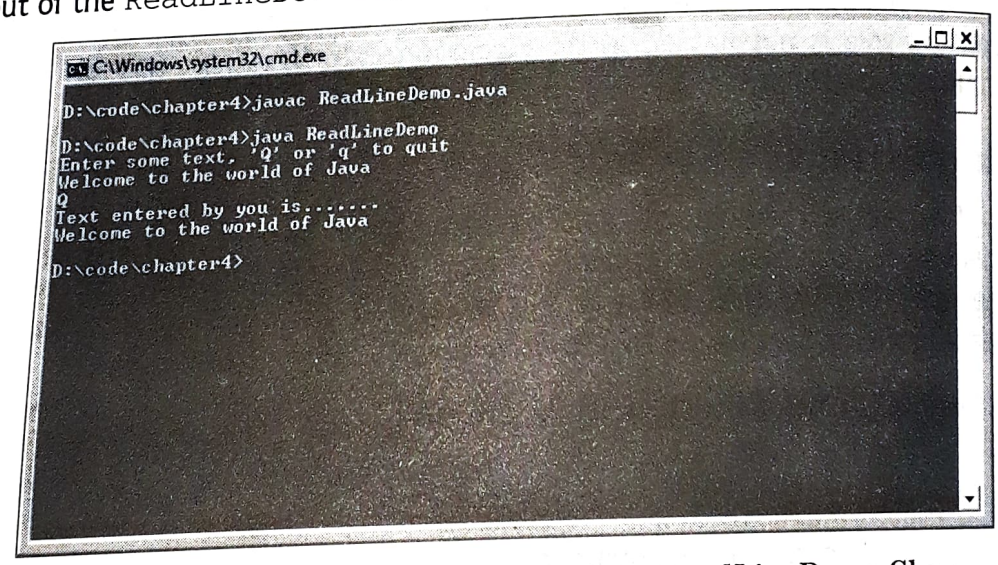
So far, we learned about reading characters from the console. Apart from reading the characters, Java also allows users to read strings of text from the console. Listing 10 shows the use of readLine(), a method of BufferedReader class (you can find the ReadLineDemo.java file on the CD in the code\chapter4 folder):

▶ **Listing 10: The ReadLineDemo.java File**

```
import java.io.*;
import java.io.Reader.*;
public class ReadLineDemo
{
 public static void main (String args[]) throws IOException
 {
    BufferedReader obj=new BufferedReader(new InputStreamReader
    (System.in));
    System.out.println("Enter some text, 'Q' or 'q' to quit");
    String str [] = new String [200];
    for (int i=0; i<10;i++)
    {
       str[i]=obj.readLine(); // reading the entered text
       if (str[i].equals("Q") || str[i].equals ("q"))
       {
```

```
                break;
            }
        }
        System.out.println("Text entered by you is......." );
        for (int i=0; i<10;i++)
        {
            if (str[i].equals("Q") || str[i].equals ("q"))
            {
                break;
            }
            System.out.println(str[i]);
        }
    }
}
```

In the Listing 10, a class named `ReadLineDemo` is created in which object 'obj' of `BufferedReader` class is created. Thereafter, a String array is created, which is used to store the text accepted from the user. Then, for loop is used, which can accept maximum 10 lines including the value 'Q' or 'q' that is used allow the user to exit from the program. Inside the loop, the `readLine()` method is used to read the text entered by the user. The `readLine()` method belongs to the `BufferedReader` class, which extends the Reader class, is used to read single characters, arrays, and lines of data. The `BufferedReader` class performs fast operations when reading text from a character-input stream. The size of the buffer may or may not be specified and in case, the size is not specified, the default size is taken. An instance of `InputStreamReader` is specified in the constructor of the `BufferedReader` class to translate the byte stream to character stream. Figure 15 shows the output of the `ReadLineDemo` class:



▲ **Figure 15: Displaying the Output of the ReadLineDemo Class**

In Figure 15, you can view that a message is displayed to a user that asks to enter some text and 'Q' or 'q' to quit or stop entering the text, according to the program functionality. Then, the user has entered some text and pressed Q. As the user presses the Q key followed by the ENTER key from the keyboard, the text entered by a user is displayed. We have used the `readLine()` method to read the complete text of the line entered by a user.

In the versions prior to JDK 1.5, the readLine() method belongs to the DataInputStream class.

TEST YOUR KNOWLEDGE

Q6. Write a program to read the data from console.

Ans.

```
import java.io.*;
public class TestYourKnowledge6
{
public static void main (String args[]) throws IOException
{
 BufferedReader obj=new BufferedReader(new InputStreamReader
     (System.in));
System.out.println("Enter the names of your five favourite sports");
System.out.println(" Enter 'Q' or 'q' to quit");
String str [] = new String [300];
for (int i=0; i<15;i++)
{
str[i]=obj.readLine(); // reading the entered text
if (str[i].equals("Q") || str[i].equals ("q"))
{
break;
}
}
System.out.println("Text entered:" );
for (int i=0; i<15;i++)
{
if (str[i].equals("Q") || str[i].equals ("q"))
{
break;
}
System.out.println(str[i]);
}
}
}
```

Execution:

```
D:\code\chapter4>javac TestYourKnowledge6.java
D:\code\chapter4>java TestYourKnowledge6
```

Output:

```
Enter the names of your five favourite sports
 Enter 'Q' or 'q' to quit
Cricket
Soccer
BasketBall
Tennis
Hockey
Q
Text entered:
Cricket
Soccer
BasketBall
Tennis
Hockey
```

Q7. Write a program to read a given number and convert into a binary number.

Ans.

```
import java.io.*;
class TestYourKnowledge7 {
 public static void main(String args [])throws IOException
     BufferedReader in= new BufferedReader(new
         InputStreamReader(System.in));
     int NO=1;
     try {
         System.out.println("ENTER THE NUMBER");
         NO=Integer.parseInt(in.readLine());
     }
     catch(NumberFormatException e)
     {
     }
     int R,SUM=0,F=1;
     do {
         R=NO%2;
         SUM+=R*F;
         NO=NO/2;
         F*=10;
     }
     while(NO>0);
     System.out.println("BINARY CONVERSION = "+SUM);
  }
}
```

Execution:

```
D:\code\chapter4>javac TestYourKnowledge7.java
D:\code\chapter4>java TestYourKnowledge7
```

Output:

```
ENTER THE NUMBER
13
BINARY CONVERSION = 1101
```

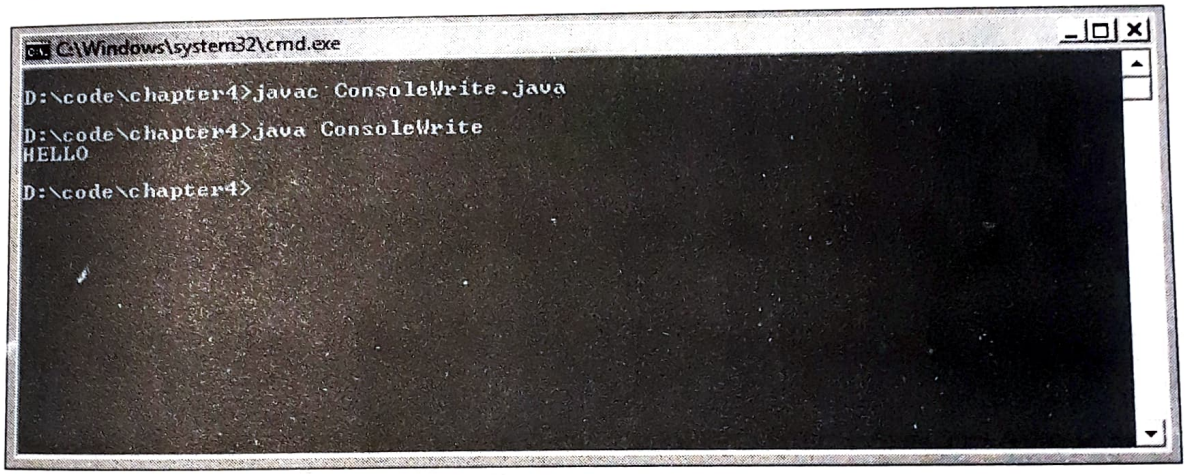
Writing Console Output

So far we learned how to display an output on the console using `println()` method. The `println()` method belongs to the `System.out` object and can be used to accomplish the same task performed by the `print()` method. However, there is a difference between the `println()` and `print()` methods. The `println()` method brings the cursor on the next line after displaying the output. On the other hand, the `print()` method does not bring the cursor on the next line. Both the `print()` and `println()` methods belongs to the `PrintStream` class. The `PrintStream` class is an output stream, which is used to print the output of various data types. You must have been noticed the use of the `println()` with the `System.out` stream, in previous chapters. The `System.out` is a stream that lets you print the output on console. Apart from these two methods, there is another method named `write()`, which is available in the `PrintStream` class to write a single character at a time. Listing 11 shows the use of the write() method to write the data on the console (you can find the ConsoleWrite.java file on the CD in the code\chapter4 folder):

▶ **Listing 11: The ConsoleWrite.java File**

```
class ConsoleWrite {
 public static void main(String args[]) {
     int x,y;
     char c,ch,chr;
     x='H';
     y='E';
     c='L';
     ch='L';
     chr='O';
     System.out.write(x);
     System.out.write(y);
     System.out.write(c);
     System.out.write(ch);
     System.out.write(chr);
     System.out.write('\n');
 }
}
```

In Listing 11, the `write()` method is used with `System.out` stream to print the output on the console. The write() method allows the user to print a single character at a time on the console. A string, "HELLO" is printed in the console because of a single character, with each occurrences of the write() method. Figure 16 shows the output of the `ConsoleWrite` class:



▲ **Figure 16: Displaying a String on the Console using the write() Method**

Figure 16 shows the use of `write()` method of the `PrintStream` class to print a string, HELLO on the console. However, the `write()` method prints a single character at a time but it does not bring the cursor on the next line. The multiple occurrences of the write() method is used until the desired string gets printed. The `System.out.write('\n')` is then used to bring the cursor on the next line.

By far, we discussed about reading and writing streams on the console. Let's now discuss about performing I/O operations on files in the following section.

TEST YOUR KNOWLEDGE

Q8. Write a program to write the data to console.

Ans.

```
class TestYourKnowledge8
{
```

```
    public static void main(string args[])
    {
        int i,j;
        char k,l;
        i='T';
        j='E';
        k='S';
        l='T';
        System.out.write(i);
        System.out.write(j);
        System.out.write(k);
        System.out.write(l);
        System.out.write('\n');
    }
}
```

Execution:

```
D:\code\chapter4>javac TestYourKnowledge8.java
D:\code\chapter4>java TestYourKnowledge8
```

Output:

```
TEST
```

Q9. Write a program to display the sum of digits of given numbers.

Ans.

```
import java.io.*;
class TestYourKnowledge9
{
 public static void main(String args [])throws IOException {
    BufferedReader in= new BufferedReader(new InputStreamReader
        (System.in));
    int no=1;
    try {
        System.out.println("ENTER THE NUMBER");
        no=Integer.parseInt(in.readLine());
    }
    catch(NumberFormatException e)
    {
    }
    int SUM=0,D;
    do {
        D=no%10;
        SUM+=D;
        no=no/10;
    }
    while(no != 0);
    System.out.println("SUM="+SUM);
  }
}
```

Execution:

```
D:\code\chapter4>javac TestYourKnowledge9.java
D:\code\chapter4>java TestYourKnowledge9
```

Output:

```
ENTER THE NUMBER
8493
SUM=24
```

Q10. It is observed that all integers converge to 1 if subjected to the following process:

1. If the number is even divide by 2 else multiply by 3 and add 1to the number.
2. Repeating the above process till it converges to 1.

Write a program to read an integer number from terminal and display all the intermediate numbers till it converges to 1. Also count the number of iterations required for the convergence.

Ans.

```
import java.io.*;
class TestYourKnowledge10
{
  public static void main (String args[])throws IOException
  {
  BufferedReader obj = new BufferedReader ( new InputStreamReader
      (System.in));
      System.out.println("Enter the number");
      int NO = Integer.parseInt(obj.readLine());
      int dv =2;
      int count= 0;
      while (NO>1)
      {
          if (NO%dv != 0)
              NO =NO *3 +1;
          else
              NO= NO/2;
              count++;
              System.out.println(NO);
      }
      System.out.println("Number of iterations required = "+ count);
  }
}
```

Execution:

```
D:\code\chapter4>javac TestYourKnowledge10.java
D:\code\chapter4>java TestYourKnowledge10
```

Output:

```
Enter the number
10
5
16
8
4
2
1
Number of iterations required = 6
```

Q11. Write a program to display the names of companies whoever quoted the lowest price for a specific machine. Input data consists of names of companies and their quoted price.

Ans.

```
import java.io.*;
class TestYourKnowledge11
{
public static void main (String args[])throws IOException
{
String CN[] = new String [10];
int P[] = new int [10];
BufferedReader obj = new BufferedReader (new InputStreamReader
     (System.in));
System.out.print("Enter Number of Companies Quoted   : ");
int N= Integer.parseInt(obj.readLine());
for (int i =1; i<=N;i++)
{
System.out.print(i  + "  Company Name ");
CN[i] = obj.readLine();
System.out.print("       Quoted Price ");
P[i] = Integer.parseInt(obj.readLine());
}
int small = P[1];
for (int i = 1;i<=N;i++)
if (small >P[i])
small = P[i];
System.out.println("Loweset Quoted Price : " + small);
System.out.println(" Company names who quoted the lowest price are
 given below:");
for (int i = 1;i<=N;i++)
if (small== P[i])
     System.out.println(CN[i]);
 }
}
```

Execution:

```
D:\code\chapter4>javac TestYourKnowledge11.java
D:\code\chapter4>java TestYourKnowledge11
```

Output:

```
Enter Number of Companies Quoted   : 2
1  Company Name Kogent
      Quoated Price 20
2  Company Name Wiley
      Quoted Price 30
Loweset Quoted Price : 20
 Company names who quoted the lowest price are given below:
Kogent
```

Q12. Write a program to arrange the names of students in the descending order of marks.

Ans. Given data consists of names and marks of almost 70 students.

```
import java.io.*;
class TestYourKnowledge12
```

```
{
public static void main (String args[])throws IOException
{
String SN[] = new String [70];
int M[] = new int [70];
BufferedReader obj = new BufferedReader (new InputStreamReader
     (System.in));
system.out.print("Enter Number of Student   : ");
int N= Integer.parseInt(obj.readLine());
for (int i =1; i<=N;i++)
{
System.out.print(i  + " Student Name ");
SN[i] = obj.readLine();
System.out.print("       Marks ");
M[i] = Integer.parseInt(obj.readLine());
}

int t= 0;
for (int j = 1 ; j <= N-1; j++)
for (int i = 1;i<=N-j;i++)
if (M[i] >M[i+1])
{
     t = M[i];
     M[i]= M[i+1];
     M[i+1] = t;
     String  st = SN[i];
     SN[i]= SN[i+1];
     SN[i+1] = st;

}

System.out.println("Name of Students As per their Marks");
System.out.println("Student's Name    Marks");
for (int i = 1;i<=N;i++)
{
     System.out.println(SN[i] + "                    " +M[i] );

}

}
}
```

Execution:

```
D:\code\chapter4>javac TestYourKnowledge12.java
D:\code\chapter4>java TestYourKnowledge12
```

Output:

```
Enter Number of Student   : 2
1 Student Name Vikash
      Marks 60
2 Student Name Deepak
      Marks 90
Name of Students As per their Marks
Student's Name    Marks
Vikash              60
Deepak              90
```

Q13. Develop a Java class to encapsulate a matrix and methods operating on it, incorporate methods for matrix addition, multiplication and transpose in addition to read and display matrix use object as parameter and also introduce a constructor.

Ans.

```
import java.io.*;
class MATRIX
{
int A[][] = new int [10][10];
int M,N;
public MATRIX()
{
 int M=6,N=5;
 for (int i=1;i<=M; i++)
 for (int j=1;j<=N; j++)
 A[i][j]=0;
}
void ReadMatrix () throws IOException
{
 BufferedReader obj = new BufferedReader (new InputStreamReader
     (System.in));
 System.out.print("Please Enter M: ");
 M=Integer.parseInt(obj.readLine());
 System.out.print("Please Enter N: ");
 N=Integer.parseInt(obj.readLine());
 for (int i=1;i<=M; i++)
 for (int j=1;j<=N; j++)
 A[i][j]=Integer.parseInt(obj.readLine());
}
void MatADD (MATRIX M1, MATRIX M2)// Object as parameter
{
 M= M1.M;
 N= M1.N;
 for (int i=1;i<=M; i++)
 for (int j=1;j<=N; j++)
 A[i][j] = M1.A[i][j] + M2.A[i][j];
}
void MatMult (MATRIX M1, MATRIX M2)// Object as parameter
{
 for (int i=1;i<=M1.M; i++)
 for (int j=1;j<=M2.N; j++)
 {
 A[i][j]  = 0;
 for (int k=1;k<=M1.N; k++)
 A[i][j] += M1.A[i][k] * M2.A[k][j];
 }
}
void MatTranspos()
{
 if (M > N )
 {  int T = M;
    M=N;
    N=T;
 }
 for (int i=1;i<=M; i++)
 for (int j=i;j<=N; j++)
 {
    int T = A[i][j];
```

```
        A[i][j] = A[j][i];
        A[i][j]  = T;
    }
  }
  void MatDisplay ()
  {
   for (int i=1;i<=M; i++)
   {
   System.out.println();
   for (int j=1;j<=N; j++)
   System.out.print("     " + A[i][j] );
   }
  }
  }
  //MAIN
  class TestYourKnowledge13
  {
  public static void main (String args[]) throws IOException
  {
  MATRIX M1 = new MATRIX();
  MATRIX M2 = new MATRIX();
  MATRIX M3 = new MATRIX();
  M1.ReadMatrix();
  M2.ReadMatrix();
  M3.MatADD(M1,M2);
  M3.MatDisplay();
  M3.MatMult(M1,M2);
  M3.MatDisplay();
  M3.MatTranspos();
  M3.MatDisplay();
  }
  }
```

Execution:

```
D:\code\chapter4>javac TestYourKnowledge13.java
D:\code\chapter4>java TestYourKnowledge13
```

Output:

```
Please Enter M: 2
Please Enter N: 2
1
2
3
4
Please Enter M: 2
Please Enter N: 2
1
2
3
4
2     4
6     8
7     10
15    22
7     10
15    22
```

Q14. Write a program to arrange the name of students in the descending order of marks. Given data consists of names and marks of at most 70 students

Ans.

```
import java.io.*;
class TestYourKnowledge14{
 public static void main (String args[])throws IOException  {
     String SN[] = new String [70];
     int M[] = new int [70];
     BufferedReader obj = new BufferedReader (new InputStreamReader
         (System.in));
     System.out.print("Enter Number of Student   : ");
     int N= Integer.parseInt(obj.readLine());
     for (int i =1; i<=N;i++)
     {
         System.out.print(i  + " Student Name ");
         SN[i] = obj.readLine();
         System.out.print("       Marks ");
         M[i] = Integer.parseInt(obj.readLine());
     }
     int t= 0;
     for (int j = 1 ; j <= N-1; j++)
     for (int i = 1;i<=N-j;i++)
     if (M[i] >M[i+1])
     {
         t = M[i];
         M[i]= M[i+1];
         M[i+1] = t;
         String  st = SN[i];
         SN[i]= SN[i+1];
         SN[i+1] = st;
     }
     System.out.println("Name of Students as per their Marks");
     System.out.println(" Student's Name Marks");
     for (int i = 1;i<=N;i++)
     {
         System.out.println(SN[i] + "                " +M[i] );
     }
 }
}
```

Execution:

```
D:\code\chapter4>javac TestYourKnowledge14.java
D:\code\chapter4>java TestYourKnowledge14
```

Output:

```
Enter Number of Student   : 6
1 Student Name Vikram Patil
      Marks 278
2 Student Name Prashant Thakur
      Marks 290
3 Student Name Shoba Pillai
      Marks 269
4 Student Name Rajesh Bishnoi
      Marks 277
```

```
5 Student Name Premela Kapoor
        Marks 299
6 Student Name Preeti Sharma
        Marks 295
Name of Students As per their Marks

Student's Name        Marks

Shoba Pillai          269

Rajesh Bishnoi        277

Vikram Patil          278

Prashant Thakur       290

Preeti Sharma         295

Premela Kapoor        299
```

Reading and Writing Files

In Java, byte streams are used to handle I/O operations on files. Java provides the java.io package that contains various classes such as FileInputStream and FileOutputStream that create byte streams linked to files. Therefore, the java.io package must be imported in your Java program to perform I/O operations. All the classes of byte stream are derived from InputStream and OutputStream classes, therefore, `InputStream` and `OutputStream` classes are considered as the main classes of byte stream.

Text in a file is internally stored in the form of bytes and if you want to know that how many bytes a file has occupied then you can create Java program to read bytes from a file. To read bytes from a file, Java has provided us byte streams that include various classes to read bytes. Byte streams provide us FileInputStream class to read binary data from a file. The most commonly used method of the FileInputStream class is the read() method. The read() method returns -1 when there is no data in the file. Listing 12 shows how to read bytes from a file (you can find the ReadBytesDemo.java file on the CD in the code\chapter4 folder):

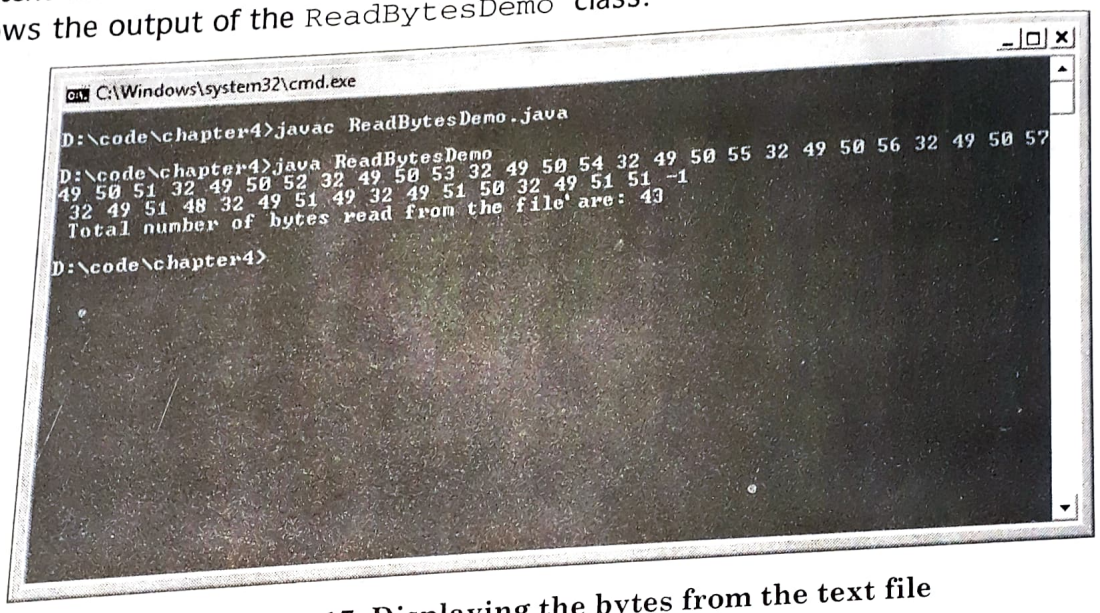
▶ **Listing 12: The ReadBytesDemo.java File**

```
import java.io.*;
public class ReadBytesDemo{
 public static void main(String args[]) {
 try {
     FileInputStream f;
     f = new FileInputStream("study1.txt");
     boolean eof = false;
     int count = 0;
     while (!eof)
     {
         int i = f.read();
         System.out.print(i + " ");
         if (i == -1)
         eof = true;
         else
         count++;
     }
     f.close();
```

```
            System.out.println("\n Total number of bytes read from the file are:
            "+count);
            } catch (Exception e) {
                System.out.println(e);
            }
        }
    }
```

In Listing 12, a class named `ReadBytesDemo` is created to read bytes from a specified file (in our case, file name is study1.txt). Next, an object f of the `FileInputStream` class is created, which is then followed by passing the file name (study1.txt), from which the data has to be read, in the constructor of the `FileInputStream` class. Then, `eof` (end of file) is initialized as false. The `eof` is a `boolean` character that returns true if the file has reached its end else it returns false.

Then, the while loop is used to traverse the file until the status of eof is not changed to true. This program will return the `ASCII` (American Standard Code for Information Interchange) equivalent characters written in the text file and in last, it will return the total number of bytes in the text file that will include all special characters such as space, hyphen, and so on. Figure 17 shows the output of the `ReadBytesDemo` class:



▲ **Figure 17: Displaying the bytes from the text file**

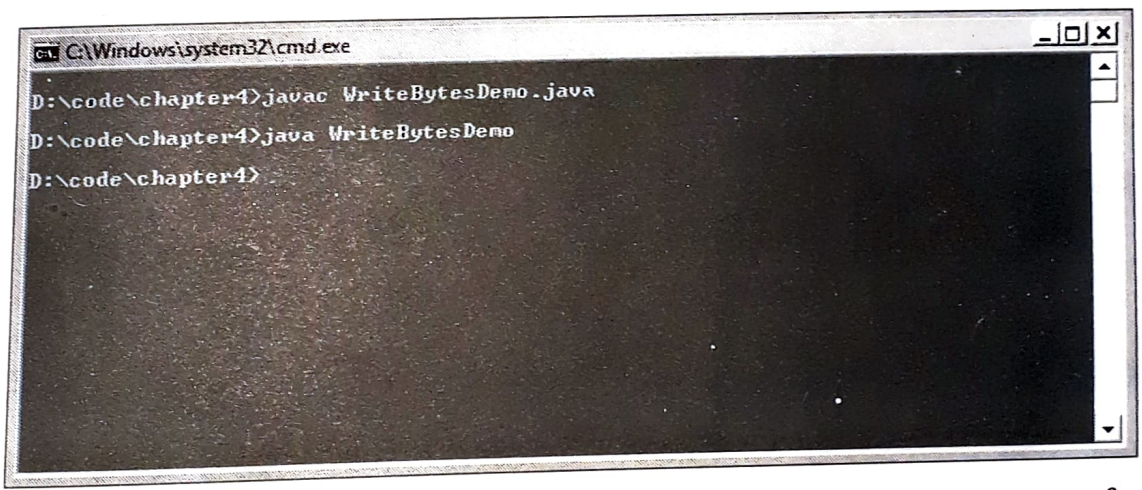
Figure 17 shows the ASCII equivalent characters written in the `study1.txt` file. It also displays the total number of bytes.

Apart from reading text from a text file, Java also allows users to write text into the text file. To write bytes in a text file, Java has provided us bytes streams that include various classes to write bytes. Byte streams provide the `FileOutputStream` class to write binary data to the file. The instance of the `FileOutputStream` class is wrapped inside the object of the `DataOutputStream` class to allow an application to write binary data of Java primitive data types to an output stream in a portable way. The `DataOutputStream` class, derived from the `FilterOutputStream` class, is used to write only Java primitive data types and not object values. The constructor of the `DataOutputStream` class creates a new data output stream to write data to the specified output stream. The `write()` method is used to write data according to its data types. Listing 13 shows how to write bytes to a file (you can find the WriteBytesDemo.java file on the CD in the code\chapter4 folder):

▶ **Listing 13: The WriteBytesDemo.java Class**

```
import java.io.*;
public class WriteBytesDemo
{
  public static void main(String args[]) throws IOException
  {
      DataOutputStream out=new DataOutputStream(new
      FileOutputStream("TestFile.txt"));
      int unit= 10;
      float price=5.16F;
      out.writeInt(unit);
      out.writeFloat(price);
      out.flush();
      out.close();
}
}
```

In Listing 13, a class named `WriteBytesDemo` is created to write bytes to the specified file (in our case, file name is `TestFile.txt`). Next, an object of the `FileOutputStream` class is created, which is then followed by passing the file name (`TestFile.txt`), to which the data has to be written, in the constructor of the `FileOutputStream` class. The `writeInt()` method writes the `int` type of value, whereas the `writeFloat()` method writes the fraction value. The `flush()` method flushes the specified data output stream and the `close()` method closes the specified output stream after flushing it. Figure 18 shows the output of the `WriteBytesDemo` class:



▲ **Figure 18: Displaying the Successful Compilation and Execution of WriteBytesDemo.java File**

Figure 18 shows that the class named `WriteBytesDemo` is compiled and executed successfully. The unit and the price values are appended in the text file (TestFile.txt) specified in the constructor of the `FileOutputStream` class.

TEST YOUR KNOWLEDGE

Q15. Write a program to read the data from a file.

Ans.

```
import java.io.*;
public class TestYourKnowledge15{
  public static void main(String args[]) {
     try {
```

```
            FileInputStream f;
            f = new FileInputStream("Test1.txt");
            boolean eof = false;
            int count = 0;
            while (!eof)
            {
                int i = f.read();
                System.out.print(i + " ");
                if (i == -1)
                eof = true;
                else
                    count++;
            }
            f.close();
            System.out.println("\n Total number of bytes read from the
                file are: " +count);
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

Execution:

```
D:\code\chapter4>javac TestYourKnowledge15.java
D:\code\chapter4>java TestYourKnowledge15
```

Output:

```
49 32 50 32 51 32 52 32 -1
Total number of bytes read from the file is: 8
```

Q16. Write a program to write the data to a file.

Ans.

```
import java.io.*;
public class TestYourKnowledge16
{
public static void main(String args[]) throws IOException
{
    DataOutputStream out=new DataOutputStream(new FileOutputStream
        ("MyFile.txt"));
    int articles= 10;
    float price=8.16F;
    out.writeInt(articles);
    out.writeFloat(price);
    out.flush();
    out.close();
}
}
```

Execution:

```
D:\code\chapter4>javac TestYourKnowledge16.java
D:\code\chapter4>java TestYourKnowledge16
```

Output:

```
The data is appended in the file specified (MyFile.txt).
```

Summary

The chapter discusses the errors and exceptions that can occur in a Java program. The chapter also discusses the various techniques to handle the exceptions. The mechanism to handle multiple exceptions in a Java program is also discussed. The types of exceptions that can occur in a Java program are also discussed in the chapter. Apart from this, a separate section on handling I/O operations is also discussed in the chapter. The I/O operations such as reading console input, reading characters, reading stings, writing console output, and reading and writing files are discussed with the help of Java programs.

Review Questions

Here let's answer the following types of questions:

- True or False
- Multiple Choice
- Short Answers
- Debugging

True or False

State which of the following is true or false:

1. The Throwable class is an immediate subclass of the Object class. **True**
2. All exceptions are broadly divided into two categories: checked exceptions and unchecked exceptions. **True**
3. A try block can be used without a catch or finally block. **False**
4. You can only use a single try and catch block in a Java program. **False**
5. The finally block is used to display information or to execute statements, which need to be executed irrespective of the catch block. **True**
6. Java provides the java.io package to handle exception in a program. **False**
7. Character streams are the streams that represent data in the form of characters. **True**
8. The ClassNotFoundException exception is thrown when you try to access a class, which is not defined in your Java program. **True**
9. The `println()` method belongs to the `System.in` object. **False**
10. The `PrintStream` class is an output stream, which is used to print the output of various data types. **True**
11. The DataInputStream class is derived from the FilterInputStream class. **True**
12. A finally block can come before the catch block. **False**

Multiple Choice Questions

Choose the correct option for the following multiple choice questions:

Q1. **Which of the following is a method used to write int type values?**

A. writeInt()
B. writeInt(int i)
C. writeint()
D. writeint(int i)

Ans. Option A is correct.

Q2. **Which of the following is a method available in the PrintStream class to write a single character at a time?**

A. print()
B. println()
C. write()
D. read()

Ans. Option C is correct.

Q3. **Which of the following streams is used to handle I/O operations on binary data?**

A. File Stream
B. Character Stream
C. Byte Stream
D. Input Stream

Ans. Option C is correct.

Q4. **The FileNotFoundException class is defined in which of the following packages?**

A. java.lang
B. java.io
C. java.util
D. java.applet

Ans. Option B is correct.

Q5. **Which of the following must be declared at the end of the last catch block?**

A. finally block
B. try block
C. catch block
D. nested try block

Ans. Option A is correct.

Q6. **Which of the following exception is generated when the problems related to mathematical calculations occur?**

A. FileNotFoundException
B. ArrayOutOfBoundsException
C. ArithmeticException
D. IOException

Ans. Option C is correct.

Q7. **Which of the following exception occurs when you try to create a negative size array in a program?**

A. ClassCastException
B. NullPointerExceptionception
C. IllegalStateException
D. NegativeArraySizeException

Ans. Option D is correct.

Q8. **Which of the following class is the immediate subclass of the Object class?**

A. Throwable
B. Exception
C. Error
D. FileNotFoundException

Ans. Option A is correct.

Q9. **Which of the following class is used to read binary data of Java primitive data types in a portable way?**

A. BufferedReader
B. BufferedWriter
C. DataInputStream
D. System

Ans. Option C is correct.

Q10. **Which of the following exception(s) are thrown when you refer the members of a null object?**

A. NoSuchFieldException
B. IOException
C. InterruptedException
D. NullPointerException

Ans. Option D is correct.

Q11. **Which of the following is the full form of eof?**

A. End of File
B. Error of File
C. Error to find
D. Example of file

Ans. Option A is correct.

Q12. **Which of the following is the superclass of the Throwable class?**

A. Object
B. String
C. Exception
D. Error

Ans. Option A is correct.

Short Answer Questions

Answer the following short answer questions.

Q1. **What are exceptions?**

Ans. Exceptions are certain abnormal conditions or errors that occur at runtime and can cause an abrupt termination of a program.

Q2. **Explain two types of errors which can occur in a program.**

Ans. There are two types of errors in Java: `compile-time errors` and `run-time errors`. Compile-time errors occur due to syntactical problems in a program, whereas run-time errors occur due to certain conditions such as memory failure, or internal errors.

Q3. **What is the root class of the exception hierarchy?**

Ans. All the possible exceptions supported by Java are organized as subclasses under a hierarchy in the Throwable class. Therefore, the Throwable class is the root class of the exception hierarchy. The Throwable class is an immediate subclass of the Object class, which is located at the root of the exception hierarchy.

Q4. **List all the important subclasses of the Exception class.**

Ans. Following are subclasses of the Exception class:

- `RuntimeException`: Thrown when the arithmetic operations performed in a program are incorrect and consist of common exceptions such as ArithmeticException, IndexOutOfBoundException.

- `ClassNotFoundException`: Thrown when a class that needs to be loaded is not found.
- `CloneNotSupportedException`: Thrown when a method does not implement the Cloneable interface but uses the clone () method.
- `IllegalAccessException`: Thrown when a method is called in another method/class. However, the calling method/class does not have permission to access that method.
- `InstantiationException`: Thrown when an abstract class or an interface is instantiated.
- `InterruptedException`: Thrown when a thread in sleeping or waiting state is interrupted by another thread.
- `NoSuchFieldException`: Thrown when an unidentified variable is used in a program.
- `NoSuchMethodException`: Thrown when an undefined method is used in a program.

Q5. Explain the subclasses of the Error class.

Ans. Following are the subclasses of the Error class:

- `LinkageError`: Throws an error when there is a problem in linking a class reference to other class. This subclass consists of various other subclasses such as `ClassFormatError`, `NoClassDefFoundError`.
- `VirtualMachineError`: Throws an error when the Java Virtual Machine (JVM) encounters an error. This subclass consists of various other subclasses such as `InternalError`, `OutOfMemoryError`.
- `ThreadDeath`: Throws an error when the `stop()` method of a Thread object is called to kill a Thread.
- `AssertError`: Throws an error when an assertion evaluates to false.

Q6. What do you understand by the try and catch block?

Ans. The try block in a Java program is used to enclose a statement that may lead to an exception. A try block must be followed by a catch block. The catch block acts as an exception handler and contains the object of the specified exception class, which handles the raised exception.

Q7. How can you read data from a file and write data to a file, in Java?

Ans. In Java, byte streams are used to handle I/O operations on files. Java provides the java.io package that contains various classes such as FileInputStream and FileOutputStream. Therefore, the java.io package must be imported in your Java program to perform I/O operations. All the classes of byte stream are derived from InputStream and OutputStream classes therefore InputStream and OutputStream classes are considered as the main classes of the byte stream.

Q8. What are Character streams?

Ans. Character streams are the streams that represent data in the form of characters. Character streams are used to handle I/O operations on characters. To use the character stream classes, you have to import java.io package in your Java program.

The two main classes of the character stream are Reader and Writer. These classes support Unicode character set (it is a 16 bit character set designed to support world's major languages such as latin, russian). Some of the examples of character stream classes are Reader, Writer, InputStreamReader, OutputStreamWriter, BufferedReader, BufferedWriter, FileReader, FileWriter, and PrintWriter.

Q9. What are Byte streams?

Ans. Byte streams are used to handle I/O operations on binary data. All the classes of byte stream are derived from InputStream and OutputStream classes. Therefore, InputStream and OutputStream classes are considered as the main classes of byte stream. Some of the examples of byte stream classes are DataInputStream, DataOutputStream, FileInputStream, and FileOutputStream.

Q10. What are the rules which you must keep in mind while working with try and catch block?

Ans. You must keep in mind the following points, while using the try and catch block:

- A try block cannot be used without a catch or finally block.
- A catch block must be followed by a try block, that is, there should be no statement between the end of the try block and the beginning of the catch block.
- A finally block cannot come before the catch block.

Q11. Explain the finally block.

Ans. The finally block is used to display information or to execute statements, which need to be executed irrespective of the catch block. This means that a finally block must be executed in spite of the fact that a catch block is executed or not. Some rules for using the finally block are as follows:

- A finally block must be declared at the end of the last catch block. If the finally block is declared before a catch block, then program will not compile successfully.
- Multiple finally blocks cannot be used with a try block. Doing so generates a compile-time error stating that a finally block cannot be used without a try block, even though we have used the try block in the program.

Q12. Write the syntax for using the throws clause.

Ans. The syntax of using the throws clause is:

```
return_type method_name (para) throws excep1, excep2, excep_n{
 // method body
}
```

Debugging Exercises

Q1. What will be the output of the following program?

```
class Ques1 {
 public static void main (String args[])  {
    int arr[]={1,2,3,4};
    int i;
    for(i=0;i<=4;i++)  {
```

```
                System.out.println(arr[i]);
            }
        }
    }
```

Ans. Output of the preceding program is as follows:

```
1
2
3
4
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
 at Ques1.main(Ques.java:9).
```

Q2. What will be the output of the following program?

```
class Ques2 {
    public static void main(String args[]) {
        try{
        int arr[]={1,2,3,4};
        int i;
        for(i=0;i<=4;i++) {
            System.out.println(arr[i]);
        }
    }
        catch(Exception e) {
        e.printStackTrace ();
        System.out.println("There is no fifth element available " );
    }
}
}
```

Ans. Output of the preceding program is as follows:

```
1
2
3
4
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
 at Ques2.main(Ques2.java:10)
There is no fifth element available
```

Q3. What will be the output of the following program?

```
class Ques3 {
 public static void main (String args[]) {
    int i=7;
    int j=0;
    int k=i/j;
    System.out.println("Result of the program is " + k);
 }
}
```

Ans. Output of the preceding program is as follows:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero at
    Ques3.main(Ques3.java:7)
```

Q4. What will be the output of the following program?

```
class Ques4 {
 public static void main (String args[])  {
     try {
         int i=7;
         int j=0;
         int k=i/j;
         System.out.println("Result of the program is " + k);
     }
     catch(Exception ex) {
             System.out.println("The exception caught is:"+ex);
         }
     }
}
```

Ans. Output of the preceding program is as follows:

The exception caught is: java.lang.ArithmeticException: / by zero

Q5. What will be the output of the following program?

```
public class Ques5 {
     public void myMethod() throws Exception {
         try {
         throw new Exception();}
         finally {
             System.out.println("Testing Finally");
         }
     }
     public static void main(String args[]) {
         Ques5 obj=new Ques5();
         try {
             obj.myMethod();
         }
         catch(Exception ex) {
             System.out.println("Exception:"+ex);
         }
     }
}
```

Ans. Output of the preceding program is as follows:

```
Testing Finally
Exception:java.lang.Exception
```

Q6. What will be the output of the following program?

```
class Ques6{
     public static void main(String args[]) {
     try{
             int ar[]={4,5,6};
             System.out.println(ar[5]);
         }
         catch(Exception e) {
             System.out.println("exception caught");
             System.exit(0);
         }
```

```
            Finally {
                System.out.println("Finally");
            }
        }
    }
```

Ans. Output of the preceding program is as follows:

```
exception caught
```

Q7. What will be the output of the following program?

```
class Ques7 {
    public static void main(String args[]) {
        String s[]={"a","b","c"};
        try{
            int k=1/0;
            System.out.println(s[2]);
            s[3]="d";
            System.out.println(s[3]);
        }
        catch(ArrayIndexOutOfBoundsException r) {
            System.out.println("caught");
        }
    }
}
```

Ans. Output of the preceding program is as follows:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero at
    Ques7.main(Ques7.java:5)
```

Q8. What will be the output of the following program?

```
class Ques8 {
public static void main(String args[])   {
    DataInputStream obj=new DataInputStream(System.in);
    char ch=' ';
    try {
        System.out.println("Enter  'Q' to quit");
        while(ch!='Q')
            {
            ch=(char)obj.read();
            System.out.println(ch);
        }
    }
    catch(Exception ex) {
        System.out.println(ex);
        }
    }
}
```

Ans. The program gives compilation error.

Q9. What will be the output of the following program?

```
class Ques9 {
public static void main(String args[]) {
```

```
        int i,j;
        char a,b,c;
        i='C';
        j='H';
        a='E';
            b='C';
        c='K';
            System.out.write(i);
            System.out.write(j);
            System.out.write(a);
            System.out.write(b);
            System.out.write(c);
        }
    }
```

Ans. The program executes successfully but prints nothing on the console.

Q10. What will be the output of the following program?

```
import java.io.*;
public class Ques10 {
    public static void main (String args[]) throws IOException {
    BufferedReader obj=new BufferedReader(new InputStreamReader
    (System.in));
    System.out.println("Enter the names of five Asian countries");
    System.out.println(" Enter 'Q' or 'q' to quit");
    String str [] = new String [300];
    for (int i=0; i<15;i++)
    {
        str[i]=obj.readLine(); // reading the entered text
        if (str[i].equals("Q") || str[i].equals ("q"))
        {
            break;
        }
    }
    System.out.println("Text entered by you is......." );
    for (int i=0; i<15;i++)
    {
        if (str[i].equals("Q") || str[i].equals ("q"))
    {
        break;
    }
    System.out.println(str[i]);
    }
    }
}
```

Ans. Enter the names of five Asian countries
Enter 'Q' or 'q' to quit
Japan
China
India
Korea
Thailand
Q

```
Text entered by you is.....
Japan
China
India
Korea
Thailand
```

Q11. What will be the output of the following program?

```
import java.io.*;
public class Ques11{
public static void main(String args[]) {
 try {
     FileInputStream f= new FileInputStream("Test1.txt");
     boolean eof = false;
     int count = 0;
     while (!eof)
         {
         int i = f.read();
         DataOutputStream out=new DataOutputStream(new
             FileOutputStream("Test2.txt"));
         out.writeInt(i);
         if (i == -1)
eof = true;
else
count++;
}
 f.close();
 }
 catch (Exception e) {
System.out.println(e);
 }
}
}
```

Ans. The text contained in the Test1.txt file gets appended in the Test2.txt file.

Q12. What will be the output of the following program?

```
import java.io.*;
public class Ques12 {
 public static void main(String args[]) throws IOException {
     DataOutputStream out=new DataOutputStream(new
         FileOutputStream("Test.txt"));
     int items= 20;
     float val=512.7;
         out.writeInt(items);
         out.writeFloat(val);
         out.flush();
         out.close();
     }
}
```

Ans. A compilation error occurs stating "possible loss of precision".