CHAPTER

21

EXCEPTION HANDLING

'To err is human'—as human beings, we commit many errors. A software engineer may also commit several errors while designing the project or developing the code. These errors are also called 'bugs' and the process of removing them is called 'debugging'. Let us take a look at different types of errors that are possible in a program.

Errors in a Java Program

There are basically three types of errors in the Java Program:

- **Compile-time errors:** These are syntactical errors found in the code, due to which a program fails to compile. For example, forgetting a semicolon at the end of a Java statement, or writing a statement without proper syntax will result in compile-time error. Let us see the following program to understand this better.

Program 1: Write a program to demonstrate compile-time error.

```
//Compile-time error
class Err
{
      public static void main(String args[ ])
      {
            System.out.println("Hello")
            System.out.println("Where is error")
      }
}
```

Output:

```
C:\> javac Err.java
Err.java:6: ';' expected
      System.out.println("Hello")
                                 ^
Err.java:7: ';' expected
      System.out.println("Where is error")
                                          ^
2 errors
```

In this program, we are not writing a semicolon at the end of the statements. This is will be detected by the Java compiler.

Detecting and correcting compile-time errors is easy as the Java compiler displays the list of errors with the line numbers along with their description. The programmer can go to the statement and check them word by word and line by line to understand where he has committed the errors.

- **Run-time errors:** These errors represent inefficiency of the computer system to execute a particular statement. For example, insufficient memory to store something or inability of the microprocessor to execute some statement come under run-time errors. The following program explains this further.

Program 2: Write a program to write `main()` method without its parameter: `String args[]`. Hence JVM cannot detect it and cannot execute the code.

```
//Run-time error
class Err
{
      public static void main()
      {
            System.out.println("Hello");
            System.out.println("Where is error");
      }
}
```

Output:

```
C:\> javac Err.java
C:\> java Err
Exception in thread "main" java.lang.NoSuchMethodError: main
```

Run-time errors are not detected by the Java compiler. They are detected by the JVM, only at runtime.

Important Interview Question

What happens if main() method is written without String args[ ] ?

The code compiles but JVM cannot run it, as it cannot see the main() method with String args[ ].

- **Logical errors:** These errors depict flaws in the logic of the program. The programmer might be using a wrong formula or the design of the program itself is wrong. Logical errors are not detected either by Java compiler or JVM. The programmer is solely responsible for them. In the following program, the programmer wants to calculate incremented salary of an employee, but he gets wrong output, since he uses wrong formula.

Program 3: An employee got an increment of 15% in his salary and the programmer wants to calculate his incremented salary. The programmer has used wrong formula where he is calculating the increment only and not the incremented salary. Write a program related to this problem.

```
//Logical error
class Err
{
      public static void main(String args[ ])
      {
            double sal = 5000.00;
            sal = sal * 15/100;   //wrong. Use: sal += sal*15/100;
            System.out.println("Incremented salary= "+ sal);
      }
}
```

Output:

```
C:\> javac Err.java
C:\> java Err
Incremented salary= 750.0
```

By comparing the output of a program with manually calculated results, a programmer can guess the presence of a logical error.

Exceptions

Basically, an exception is a runtime error. Then there arises a doubt: Can't I call a compile time error an exception? The answer is: "No, you can not call compile-time errors also exceptions". They come under errors. All exceptions occur only at runtime but some exceptions are detected at compile time and some others at runtime. The exceptions that are checked at compilation time by the Java compiler are called 'checked exceptions' while the exceptions that are checked by the JVM are called 'unchecked exceptions'.

Important Interview Question

What are checked exceptions?

The exceptions that are checked at compilation-time by the Java compiler are called 'checked exceptions'. The exceptions that are checked by the JVM are called 'unchecked exceptions'.

Unchecked exceptions and errors are considered as unrecoverable and the programmer cannot do any thing when they occur. The programmer can write a Java program with unchecked exceptions and errors and can compile the program. He can see their effect only when he runs the program. So, Java compiler allows him to write a Java program without handling the unchecked exceptions and errors. In case of other exceptions (checked), the programmer should either handle them or throw them without handling them. He cannot simply ignore them, as Java compiler will remind him of them. Let us now consider a statement:

```
public static void main(String args[ ]) throws IOException
```

Here, `IOException` is an example for checked exception. So, we threw it out of `main()` method without handling it. This is done by `throws` clause written after `main()` method in the above statement. Of course, we can also handle it. Suppose, we are not handling it and not even throwing it, then the Java compiler will give an error.

The point is that an exception occurs at run time but in case of checked exceptions, whether you are handling it or not, it is detected at compilation time. So let us define an exception as a runtime error that can be handled by a programmer. This means a programmer can do something to avoid any harm caused by the rise of an exception. In case of an error, the programmer cannot do any thing and hence if error happens, it causes some damage.

All exceptions are declared as classes in Java. Of course, everything is a class in Java. Even the errors are also represented by classes. All these classes are descended from a super class called `Throwable`, as shown in Figure 21.1.

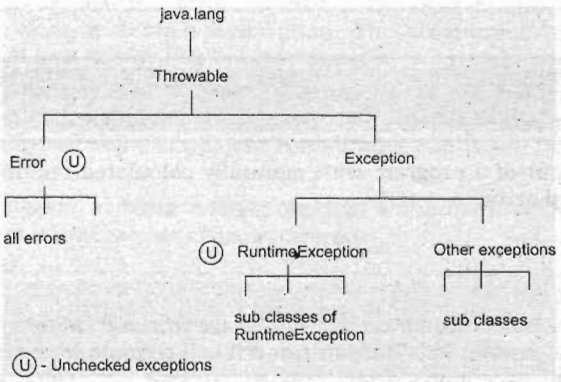

Figure 21.1 The exception hierarchy in Java

Important Interview Question

What is Throwable?

Throwable is a class that represents all errors and exceptions which may occur in Java.

Which is the super class for all exceptions?

Exception is the super class of all exceptions in Java.

What is the difference between an exception and an error?

An exception is an error which can be handled. It means when an exception happens, t programmer can do something to avoid any harm. But an error is an error which cannot be handled happens and the programmer cannot do any thing.

Let's now discuss what happens when there is an exception. Generally, in any program, any fi databases are opened in the beginning of the program. The data from the files is retrieve processed in the middle of the program. At the end of the program, the files are closed prope that the data in the files is not corrupted. The following program shows this situation.

Program 4: Write a program that opens the files in the beginning. Then the number of com line arguments is accepted into n. This n divides a number 45 and the result is stored i Finally the files are closed.

```
//An exception example
class Ex
{
      public static void main(String args[ ])
      {
            //open the files
            System.out.println("Open files");

            //do some processing
            int n = args.length;
            System.out.println("n= "+ n);
            int a = 45/n;
            System.out.println("a= "+ a);

            //close the files
            System.out.println("Close files");
      }
}
```

Output:

```
C:\> javac Ex.java
C:\> java Ex 11 22 44
Open files
n= 3
a= 15
Close files

C:\> java Ex
Open files
n= 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Ex.main(Ex.java:12)
C:\>
```

Please observe the outputs of this program. When we passed 3 command line arguments, then the program executed without any problem. But when we run the program without passing any arguments, then n value became zero. Hence, execution of the following expression fails:

```
int a = 45/n;
```

Here, division by zero happens and this value represents infinity. This value cannot be stored in any variable. So there will be an exception at runtime in the above statement. In this case, JVM displays exception details and then terminates the program abnormally. The subsequent statements in the program are not executed. This means that the files which are open in the program will not be closed and hence the data in the files will be lost. This is the major problem with exceptions. When there is an exception, the files may not be closed or the threads may abnormally terminate or the memory may not be freed properly. These things lead to many other problems in the software. Closing the opened files, stopping any running threads in the program, and releasing the used memory are called 'cleanup operations'. Therefore, it is compulsory to design the program in such a way that even if there is an exception, all the clean up operations are performed and only then the program should be terminated. This is called exception handling.

Exception Handling

When there is an exception, the user data may be corrupted. This should be tackled by the programmer by carefully designing the program. For this, he should perform the following 3 steps:

Step 1: The programmer should observe the statements in his program where there may be a possibility of exceptions. Such statements should be written inside a `try` block. A `try` block looks like as follows:

```
try{
        statements;
}
```

The greatness of `try` block is that even if some exception arises inside it, the program will not be terminated. When JVM understands that there is an exception, it stores the exception details in an exception stack and then jumps into a catch block.

Step 2: The programmer should write the `catch` block where he should display the exception details to the user. This helps the user to understand that there is some error in the program. The programmer should also display a message regarding what can be done to avoid this error. `Catch` block looks like as follows:

```
catch(Exceptionclass ref)
{
```

```
        Statements;
    }
```

The reference ref above is automatically adjusted to refer to the exception stack where the detai of the exception are available, as shown in Figure 21.2. So, we can display the exception detai using any one of the following ways:

- Using print() or println() methods, such as System.out.println(ref);
- Using printStackTrace() method of Throwable class, which fetches exception details fro the exception stack and displays them.

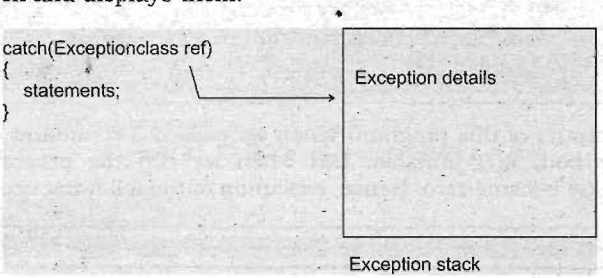

Figure 21.2 Exception class reference referring to exception stack

Step 3: Lastly, the programmer should perform clean up operations like closing the files a terminating the threads. The programmer should write this code in the finally block. Final block looks like as follows:

```
finally{
    Statements;
}
```

The specialty of finally block is that the statements inside the finally block are execu irrespective of whether there is an exception or not. This ensures that all the opened files properly closed and all the running threads are properly terminated. So, the data in the files will be corrupted and the user is at the safe-side.

Performing the above three tasks is called 'exception handling'. Remember, in exception handl the programmer is not preventing the exception, as in many cases it is not possible. But programmer is avoiding any damage that may happen to user data.

Now, let's rewrite Program 4 using try, catch, and finally blocks where we want to handle ArithmeticException.

Program 5: Write a program which tells the use of try, catch and finally block.

```
//Exception handling using try, catch and finally blocks
class Ex
{
    public static void main(String args[ ])
    {
        try
        {
            //open the files
            System.out.println("Open files");

            //do some processing
            int n = args.length;
            System.out.println("n= "+ n);
            int a = 45/n;
            System.out.println("a= "+ a);
        }
```

```
                catch(ArithmeticException ae)
                {
                        //display the exception details
                        System.out.println(ae);

                        //display any message to the user
                        System.out.println("Please pass data while running this
                        program");
                }
                finally
                {
                        //close the files
                        System.out.println("Close files");
                }
        }
}
```

Output:

```
C:\> javac Ex.java
C:\> java Ex 11 22 44
Open files
n= 3
a= 15
Close files

C:\> java Ex
Open files
n= 0
java.lang.ArithmeticException: / by zero
Please pass data while running this program
Close files
```

In this program, the `try` block contains the statements where there is possibility for an exception. When there is an exception, JVM jumps into `catch` block. The `finally` block is executed even if there is exception or not.

Observe the output of the program. When there is no exception, we can see normal execution and the files are closed properly; but when there is an exception, the exception details are displayed along with a message to the user. Then, `finally` block is executed which closes the files. This prevents any loss to user data.

Handling Multiple Exceptions

Most of the times there is possibility of more than one exception present in the program. In this case, the programmer should write multiple `catch` blocks to handle each one of them.

In the previous program, we can intentionally create another exception by adding the code:

```
int b[ ] = {10, 20, 30};
b[50] = 100;
```

Above, we are creating an array `b[ ]` with 3 elements. Index of the array represents the element position in the array. So, in this case the index is from 0 to 2, since there are 3 elements. There is no index like 50. But we are trying to store 100 at index position 50 in the second statement. This leads to an exception called `ArrayIndexOutOfBoundsException`. To handle this, we should write another `catch` block as shown here:

```
catch(ArrayIndexOutOfBoundsException aie)
{
        //display exception details
        aie.printStackTrace();
```

```
        //display a message to user
        System.out.println("Please see that the array index is within the range"
    }
```

Program 6: Write a program which shows how to handle the ArithmeticException a ArrayIndexOutOfBoundsException.

```
//Handling multiple exceptions using try, catch and finally blocks
class Ex
{
    public static void main(String args[ ])
    {
        try
        {
            //open the files
            System.out.println("Open files");

            //do some processing
            int n = args.length;
            System.out.println("n= "+ n);
            int a = 45/n;
            System.out.println("a= "+ a);

            int b[] = {10, 20, 30};
            b[50] = 100;
        }
        catch(ArithmeticException ae)
        {
            //display the exception details
            System.out.println(ae);

            //display any message to the user
            System.out.println("Please pass data while running this
            program");
        }
        catch(ArrayIndexOutOfBoundsException aie)
        {
            //display exception details
            aie.printStackTrace();

            //display a message to user
            System.out.println("Please see that the array index is
            within the range");
        }
        finally
        {
            //close the files
            System.out.println("Close files");
        }
    }
}
```

Output:

```
C:\> javac Ex.java
C:\> java Ex
Open files
n= 0
java.lang.ArithmeticException: / by zero
Please pass data while running this program
Close files

C:\> java Ex 11 22
Open files
```

```
n= 2
a= 22
java.lang.ArrayIndexOutOfBoundsException: 50
        at Ex.main(Ex.java:18)
Please see that the array index is within the range
Close files
```

In this program, there is provision for two exceptions—`ArithmeticException` and `ArrayIndexOutOfBoundsException`. These exceptions are handled with the help of two `catch` blocks.

Observe the output. When we did not pass any values while running this program, we got `ArithmeticException` and when we pass some values, then there is another exception called `ArrayIndexOutOfBoundsException`. This means, even if there is scope for multiple exceptions, only one exception at a time will occur.

From the preceding discussion, we can conclude that:

- An exception can be handled using `try`, `catch`, and `finally` blocks.
- It is possible to handle multiple exceptions using multiple `catch` blocks.
- Even though there is possibility for several exceptions in `try` block, at a time only one exception will be raised.
- A single `try` block can be followed by several `catch` blocks.
- We cannot write a `catch` without a `try` block, but we can write a `try` without any `catch` block.
- It is not possible to insert some statements between `try` and `catch`.
- It is possible to write a `try` block within another `try`. They are called nested `try` blocks.

throws Clause

Even if the programmer is not handling runtime exceptions, the Java compiler will not give any error related to runtime exceptions. But the rule is that the programmer should handle checked exceptions. In case the programmer does not want to handle the checked exceptions, he should throw them out using `throws` clause. Otherwise, there will be an error flagged by Java compiler. See Program 7 to understand this better, where there is an `IOException` raised by `readLine()` method of `BufferedReader` class. This is checked exception and hence the compiler checks it at the compilation time. If it is not handled, the compiler expects at least to throw it out.

Program 7: Write a program that shows the compile time error for IOException.

```
//Not handling the exception
import java.io.*;
class Sample
{
        //instance variable
        private String name;

        //method to accept name
        void accept()
        {
                //to accept data from keyboard
                BufferedReader br = new BufferedReader(new
                InputStreamReader(System.in));

                System.out.print("Enter name: ");
                name = br.readLine();
        }

        //method to display name
        void display()
```

```
        {
                System.out.println("Name: "+ name);
        }
}

class Ex1
{
        public static void main(String args[ ])
        {
                Sample s = new Sample();
                s.accept();
                s.display();
        }
}
```

Output:

```
C:\> javac Ex1.java
Ex1.java:15: unreported exception java.io.IOException; must be caught or
declared to be thrown
        name = br.readLine();
        ^
1 error
```

In this program, the Java compiler expects the programmer to handle the `IOException` using t and `catch` blocks; else, he should throw out the `IOException` without handling it. But t programmer is not performing either. So there is a compile time error displayed.

Here, we are not handling the `IOException` given by `readLine()` method. Since it is a chec exception, we should throw it out of the method using `throws` clause as:

```
throws IOException
```

The `throws` clause is written at the side of `accept()` method since in this method the `readLin` is called. Also, we should write `throws` clause next to `main()` method since in this method, `accept()` is called, as shown here:

```
void accept() throws IOException
public static void main(String args[ ]) throws IOException
```

Now, if the above code is inserted into the program, the preceding program executes without problem. This version of the program is shown in Program 8. Note that since in this program, we not handling the exception, it is not a robust program. This means that if exception happens, program goes for abnormal termination.

Program 8: Write a program which shows the use of throws clause.

```
//Not handling the exception -using throws clause
import java.io.*;
class Sample
{
        //instance variable
        private String name;

        //method to accept name
        void accept() throws IOException
        {
                //to accept data from keyboard
                BufferedReader br = new BufferedReader(new
                InputStreamReader(System.in));

                System.out.print("Enter name: ");
```

```
            name = br.readLine();
        }

        //method to display name
        void display()
        {
            System.out.println("Name: "+ name);
        }
    }

    class Ex1
    {
        public static void main(String args[ ]) throws IOException
        {
            Sample s = new Sample();
            s.accept();
            s.display();
        }
    }
```

Output:

```
C:\> javac Ex1.java
C:\> java Ex1
Enter name: Kumar
Name: Kumar
```

In this program, we are using `throws` clause to throw out an exception without handling it, from a method.

throw Clause

There is also a `throw` statement available in Java to throw an exception explicitly and catch it. Let us see how it can be used.

In the following program, we are creating an object of `NullPointerException` class and throwing it out of `try` block, as shown here:

```
throw new NullPointerException("Exception data");
```

In the above statement, `NullPointerException` class object is created and 'Exception data' is stored into its object. Then it is thrown using `throw` statement. Now, we can `catch` it using `catch` block as:

```
catch(NullPointerException ne)
{
}
```

Program 9: Write a program that shows the use of throw clause for throwing the NullPointerException.

```
//using throw
class Sample
{
    static void demo()
    {
        try{
        System.out.println("Inside demo()");
        throw new NullPointerException("Exception data");
    }
```

```
        catch(NullPointerException ne)
        {
                System.out.println(ne);
        }
        }
}

class ThrowDemo
{
        public static void main(String args[ ])
        {
                Sample.demo();
        }
}
```

Output:

```
C:\> javac ThrowDemo.java
C:\> java ThrowDemo
Inside demo()
java.lang.NullPointerException: Exception data
```

In this program, we are using `throw` clause to throw `NullPointerException` class object. Then is caught and its details are displayed in `catch` block.

Above, we used throw to throw out an exception object and handle it. This activity is useful software development in the following two ways:

1. `throw` clause is used in software testing to test whether a program is handling all t exceptions as claimed by the programmer. Now, let us see how this is done. Suppose programmer has written a program to handle some 5 exceptions properly. Now, the softw tester has to test and certify whether the program is handling all the 5 exceptions as said the programmer or not. For this, the tester should plan the input data such that if he provic that input, the said exceptions will occur. Causing the exceptions intentionally like this will at times very difficult. In this case, the tester can take the help of the `throw` clause. Supp the tester wants to test whether `IOException` is handled by the program or not, he c introduce a statement in the source code as:

```
throw new IOException("My IOException");
```

Then he will compile and run the program. Then the above statement will raise the `IOExcept` and the rest of the program should handle it. How the program handles it will be noted down by tester for a feedback to the programmer. This is the way `throw` will help the tester to test a prog for exceptions.

2. `throw` clause can be used to throw our own exceptions also. Just like the exceptions avail in Java, we can also create our own exceptions which are called 'user-defined exceptions'. need the `throw` clause to throw these user defined exceptions.

Types of Exceptions

As you have already read about the exceptions in the preceding paragraphs, let us move on fur to understand the types of exceptions. Following are the exceptions available in Java:

- Built-in exceptions
- User-defined exceptions.

Built-in Exceptions

Built-in exceptions are the exceptions which are already available in Java. These exceptions are suitable to explain certain error situations. Table 21.1 lists the important built-in exceptions.

Table 21.1

Exception class	Meaning
`ArithmeticException`	Thrown when an exceptional condition has occurred in an arithmetic operation.
`ArrayIndexOutOfBoundsException`	Thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.
`ClassNotFoundException`	This exception is raised when we try to access a class whose definition is not found.
`FileNotFoundException`	Raised when a file is not accessible or does not open.
`IOException`	Thrown when an input-output operation failed or interrupted.
`InterruptedException`	Thrown when a thread is waiting, sleeping, or doing some processing, and it is interrupted.
`NoSuchFieldException`	Thrown when a class does not contain the field (or variable) specified.
`NoSuchMethodException`	Thrown when accessing a method which is not found.
`NullPointerException`	Raised when referring to the members of a null object. null represents nothing.
`NumberFormatException`	Raised when a method could not convert a string into a numeric format.
`RuntimeException`	This represents any exception which occurs during runtime.
`StringIndexOutOfBoundsException`	Thrown by String class methods to indicate that an index is either negative or greater than the size of the string.

User-defined Exceptions

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, like the built-in exceptions, the user (programmer) can also create his own exceptions which are called 'user-defined exceptions'. The following steps are followed in creation of user-defined exceptions:

- The user should create an exception class as a subclass to `Exception` class. Since all exceptions are subclasses of `Exception` class, the user should also make his class a subclass to it. This is done as:

```
class MyException extends Exception
```

- The user can write a default constructor in his own exception class. He can use it, in case h does not want to store any exception details. If the user does not want to create an empt object to his exception class, he can eliminate writing the default constructor.

```
MyException() {}
```

- The user can create a parameterized constructor with a string as a parameter. He can use thi to store exception details. He can call super class (Exception) constructor from this and sen the string there.

```
MyException(String str)
{
     super(str);  //call Exception class constructor and store str there.
}
```

- When the user wants to raise his own exception, he should create an object to his exceptio class and throw it using throw clause, as:

```
MyException me = new MyException("Exception details");
throw me;
```

To understand how to create user-defined exceptions, let us write a program. In program 10, we a creating our own exception class MyException. In this program, we are taking the details account numbers, customer names, and balance amounts in the form of three arrays. Then main() method, we display these details using a for loop. At this time, we check if in any accou the balance amount is less than the minimum balance amount to be kept in the account. If it is s then MyException is raised and a message is displayed "Balance amount is less".

Program 10: Write a program to throw a user defined exception.

```
//User defined exception
//to throw whenever balance amount is below Rs. 1000
class MyException extends Exception
{
  //store account information
  private static int accno[] = {1001,1002,1003,1004,1005};

  private static String name[] = {"Raja Rao", "Rama Rao", "Subba Rao", "Appa
Rao", "Laxmi Devi"};

  private static double bal[] = {10000.00,12000.00,5600.50,999.00,1100.55};

  //default constructor
  MyException()
 {
 }

  //parameterized constructor
  MyException(String str)
 {
   super(str);
 }

 //write main()
 public static void main(String args[ ])
 {
  try{
  //display the heading for the table
```

```
        System.out.println("ACCNO"+"\t"+"CUSTOMER"+"\t"+ "BALANCE");

        //display actual account information
        for(int i=0; i<5; i++)
        {
          System.out.println(accno[i]+"\t"+name[i]+"\t"+ bal[i]);

          //display own exception if balance < 1000
          if(bal[i]<1000)
          {
           MyException me = new MyException("Balance amount is less");
           throw me;
          }
        } //end of for
      } //end of try

      catch(MyException me){
        me.printStackTrace();
      }
    } //end of main
    }//end of MyException class
```

Output:

```
C:\> javac MyException.java
C:\> java MyException
ACCNO CUSTOMER        BALANCE
1001          Raja Rao        10000.0
1002          Rama Rao        12000.0
1003          Subba Rao       5600.5
1004          Appa Rao        999.0
MyException: Balance amount is less
        at MyException.main(MyException.java:39)
```

In this program, we are throwing our own exception when the balance amount in a bank account is less than Rs. 1000.

Here, we created our own exception since there is no exception class available to describe situation where the balance amount in a bank account is less than the prescribed minimum. Also note that we are throwing our own exception using `throw` statement.

Important Interview Question

What is the difference between throws and throw ?

throws clause is used when the programmer does not want to handle the exception and throw it out of a method. throw clause is used when the programmer wants to throw an exception explicitly and wants to handle it using catch block. Hence, throws and throw are contradictory.

Re-throwing an Exception

When an exception occurs in a `try` block, it is caught by a `catch` block. This means that the thrown exception is available to the `catch` block. The following code shows how to re-throw the same exception out from the `catch` block:

```
try{
      throw exception;
}
catch(Exception obj)
{
      throw exception;  //re-throw the exception out
}
```

Suppose there are two classes A and B. If an exception occurs in A, we want to display so message to the user and then we want to re-throw it. This re-thrown exception can be caught class B where it can be handled. Hence re-throwing exceptions is useful especially when t programmer wants to propagate the exception details to another class. In this case (Program 1 the exception details are sent from class A to class B where some appropriate action may performed.

Program 11: Write a program to throw the `StringIndexOutOfBoundsException`.

```
//Rethrowing an exception.
class A
{
      void method1()
      {
            try
            {
            //take a string with 5 chars. Their index will be from 0 to 4.
            String str = "Hello";

            //exception is thrown in below statement because there is
            //no index with value 5.
            char ch = str.charAt(5);
            }
            catch(StringIndexOutOfBoundsException sie)
            {
            System.out.println("Please see the index is within the range");
            throw sie; //rethrow the exception
            }
      }
}

class B
{
      public static void main(String args[ ])
      {
            //create an object to A and call method1().
            A a = new A();
            try{
                  a.method1();
            }
            //the rethrown exception is caught by the below catch block
            catch(StringIndexOutOfBoundsException sie){
            System.out.println("I caught rethrown exception");
            }
      }
}
```

Output:

```
C:\> javac B.java
C:\> java B
Please see the index is within the range
I caught rethrown exception
```

In this program, `StringIndexOutOfBoundsException` is thrown in `method1()` of class `A` whi caught by `catch` block in that method. Then the `catch` block is re-throwing it into `main()` me of class `B`.

Important Interview Question

Is it possible to re-throw exceptions?

Yes, we can re-throw an exception from catch block to another class where it can be handled.

Conclusion

Errors in a program are nightmares to the programmer. The errors that can be identified and handled are called exceptions. By handling exceptions, a programmer can make his programs 'robust'. It means the Java program will not abnormally terminate if all the possible exceptions have been handled properly. This is a great boon to the user since there will not be any loss of data. There are two types of exceptions—Built-in and User-defined. Further, the exceptions checked by the compiler are called checked exceptions and the remaining are called unchecked. The programmer should compulsorily handle the checked exceptions or at least write a throws statement to throw them out, if he does not want to handle them. In Unchecked exceptions, he has been given freedom not to handle them. But it is advisable to handle every possible exception in case of a Java program.