UNIT 4 EXCEPTIONS HANDLING

Structure **Page Nos.**

4.0 INTRODUCTION

During programming in languages like c, c++ you might have observed that even after successful compilation some errors are detected at runtime. For handling these kinds of errors there is no support from programming languages like c, c++. Some error handling mechanisms like returning special values and setting flags are used to determine that there is some problem at runtime.

In C++ programming language there is a very basic provision for exception handling. Basically exception handlings provide a safe escape route from problem or clean-up of error handling code.

In Java exception handling is the only semantic way to report error .In Java exception is an object, which describes error condition, occurs in a section of code. In this unit we will discuss how exceptions are handled in Java, you will also learn to create your own exception classes in this unit.

4.1 OBJECTIVES

After going through this unit you will be able to:

- describe exception;
- explain causes of exceptions;
- writing programs with exceptions handling;
- use built–in exceptions;
- create your own exception classes.

4.2 EXCEPTION

An exceptional condition is considered as a problem, which stops program execution from continuation from the point of occurrence of it. Exception stops you from continuing because of lack of information to deal with the exception condition. In other words it is not known what to do in specific conditions.

If the system does not provide it you would have to write your own routine to test for possible errors. You need to write a special code to catch exceptions before they cause an error.

If you attempt in a Java program to carry out an illegal operation, it does not necessarily halt processing at that point. In most cases, the **JVM** sees for the possibility of *catching* the problem and recovering from it.

If the problems are such which can be caught and recovery can be provided, then we say the problems are **not fatal**, and for this the term ***exception*** is used rather than **error**.

Now let us see what to do if exceptions occur.

Causes of Exception

Exception arises at runtime due to some abnormal condition in program for example when a method. For division encounters an abnormal condition that it can't handle itself, i.e. “divide by zero,” then this method may *throw* an exception.

Exception is an abnormal condition

If a program written in Java does not follow the rule of Java language or violates the Java execution environment, constraints exception may occur. There may be a manually generated exception to pass on some error reports to some calling certain methods.

If an exception is caught, there are several things that can be done:

i. Fix the problem and try again.
ii. Do something else instead to avoid the problem.
iii. Exit the application with System.exit()
iv. Rethrow the exception to some other method or portion of code.
v. Throw a new exception to replace it.
vi. Return a default value (a non-void method: traditional way of handling exceptions).
vii. Eat the exception and return from the method (in a void method). In other words don’t give importance to the exception .
viii. Eat the exception and continue in the same method (Rare and dangerous. Be very careful if you do this).

You should give due care to exceptions in program. Programmers new to programming almost always try to ignore exceptions. Do not simply avoid dealing with the exceptions. Generally you should only do this if you can logically guarantee that the exception will never be thrown or if the statements inside exception checking block do not need to be executed correctly in order for the following program statements to run).

Now let us see how exceptions are handled in Java.

4.3 HANDLING OF EXCEPTION

Exceptions in Java are handled by the use of these five keywords**: try, catch, throw, throws, and finally**. You have to put those statements of program on which you want to monitor for exceptions, in **try** block. If any exceptions occur that will be catched using catch. Java runtime system automatically throws system-generated exceptions.

The throw keyword is used to throw exceptions manually.

4.3.1 Using try catch

Now let us see how to write programs in Java, which take care of exceptions handling.
See the program given below:

```
//program
public class Excep_Test
```

```
{
public static void main(String[] args)
{
int data[] = {2,3,4,5};
System.out.println("Value at : " + data[4]);
}
}
```

Output:

```
java.lang.ArrayIndexOutOfBoundsException
at Excep_Test.main(Excep_Test.java:6)
Exception in thread "main"
```

At runtime this program has got ArrayIndexOutOfBoundsException.This exception occurs because of the attempt to print beyond the size of array.

Now let us see how we can catch this exception.

To catch an exception in Java, you write a try block with one or more catch clauses. Each catch clause specifies one exception type that it is prepared to handle. The *try* block places a fence around the code that is under the watchful eye of the associated catchers. If the bit of code delimited by the *try* block *throws* an exception, the associated *catch* clauses will be examined by the Java virtual machine. If the virtual machine finds a *catch* clause that is prepared to handle the thrown exception, the program continues execution starting with the first statement of that *catch* clause, and the catch block is used for executing code to handle exception and graceful termination of the program.

```
public class Excep_Test
{
public static void main(String[] args)
{
try
{
int data[] = {2,3,4,5};
System.out.println("Value at : " + data[4]);
}
catch( ArrayIndexOutOfBoundsException e)
{
System.out.println("Sorry you are trying to print beyond the size of data[]");
}
}
}
```

Output:

```
Sorry you are trying to print beyond the size of data[]
```

In this program you can observe that after the occurrence of the exception the program is not terminated. Control is transferred to the *catch* block followed by *try* block.

4.3.2 Catching Multiple Exceptions

Sometimes there may be a chance to have multiple exceptions in a program. You can use multiple catch clauses to catch the different kinds of exceptions that code can throw. If more than one exception is raised by a block of code, then to handle these exceptions more than one catch clauses are used. When an exception is thrown, different catch blocks associated with try block inspect in order and the first one whose type (the exception type passed as argument in catch clause) matches with the exception type is executed This code snippet will give you an idea how to catch multiple exceptions.

```
//code snippet
```

```
try
{
// some code
}
catch (NumberFormatException e)
{
//Code to handle NumberFormatException
}
catch (IOException e)
{
// Code to handle IOException
}
catch (Exception e)
{
//  Code to handle exceptions than NumberFormatException and IOException
}
finally // optional
{
//Code in this block always executed even if  no exceptions
}
```

Now let us see the program given below :

```
//program
public class MultiCatch
{
public static void main(String[] args)
{
int repeat ;
try
{
repeat = Integer.parseInt(args[0]);
}
catch (ArrayIndexOutOfBoundsException e)
{
// pick a default value  for repeat
repeat = 1;
}
catch (NumberFormatException e)
{
// print an error message
System.err.println("Usage:  repeat as count" );
System.err.println("where repeat is the number of times to say Hello Java" );
System.err.println("and given as an integer like 2 or 5" );
return;
}
for (int i = 0; i < repeat; i++)
{
System.out.println("Hello");
}
}
}
```

Output:

```
Hello
```

Output of the above program is “Hello”. This output is because of no argument is passed to program and exception “ArrayIndexOutOfBoundsException” occurred. If pass some non-numeric value is as argument to this program you will get some other output. Check what output you are getting after passing “Java” as argument.

It is important to ensure that something happens upon exiting a block, no matter how the block is exited. It is the programmer's responsibility to ensure what should happen. For this finally clause is used.

Check Your Progress 1

1) What is an exception? Write any three actions that can be taken after an exception occurs in a program.

   …………………………………………………………………………………

   …………………………………………………………………………………

   …………………………………………………………………………………

   …………………………………………………………………………………

   …………………………………………………………………………………

   .

2) Is the following code block legal?

```
try
{
...
}
finally
{
...
}
```

   …………………………………………………………………………………

   …………………………………………………………………………………

   …………………………………………………………………………………

   …………………………………………………………………………………

   …………………………………………………………………………………

3) Write a program to catch more than two exceptions.

   …………………………………………………………………………………

   …………………………………………………………………………………

   …………………………………………………………………………………

   …………………………………………………………………………………

4.3.3 Using finally clause

There are several ways of exiting from a block of code (the statements between two matching curly braces). Once a **JVM** has begun to execute a block, it can exit that block in any of several ways.

It could, for example simply exit after reaching the closing curly brace.
It could encounter a break, continue, or return statement that causes it to jump out of the block from somewhere in the middle.

If an exception is thrown that isn't caught inside the block, it could exit the block while searching for a catch clause.
Let us take an example, of opening a file in a method. You open the file perform needed operation on the file and most importantly you want to ensure that the file

gets closed no matter how the method completes. In Java, this kind of desires are fulfilled with the help of finally clause.

A finally clause is included in a program in the last after all the possible code to be executed. Basically finally block should be the last block of execution in the program.

```
//program
public class Finally_Test
{
public static void main(String[] args)
{
try
{
System.out.println("Hello " + args[0]);
}
catch (ArrayIndexOutOfBoundsException e)
{
System.out.println("Hello, You are here after ArrayIndexOutOfBoundsException");
}
finally
{
System.out.println("Finally you have to reach here");
}
}
}
```

Output:

```
Hello, You are here after ArrayIndexOutOfBoundsException
Finally you have to reach here
```

Note: At least one clause, either *catch* or *finally*, must be associated with each *try* block. In case you have both *catch* clauses and a *finally* clause with the same *try* block, you must put the *finally* clause after all the *catch* clauses.

Now let us discuss the types of exceptions that occur in Java.

4.4 TYPES OF EXCEPTIONS

Exceptions in Java are of two kinds, ***checked*** and **unchecked**. Checked exceptions are so called because both the Java compiler and the **JVM** check to make sure that the rules of Java are obeyed. Problems causes to checked exceptions are: Environmental error that cannot necessarily be detected by testing; e.g, disk full, broken socket, database unavailable, etc. Checked exceptions must be handled at compile time. Only checked exceptions need appear in throws clauses. Problems such as Class not found, out of memory, no such method, illegal access to private field, etc, comes under virtual machine error.

Unchecked exceptions

Basically, an unchecked exception is a type of exception for that you option that handle it, or ignore it. If you elect to ignore the possibility of an unchecked exception, then, as a result of that your program will terminate. If you elect to handle an unchecked exception that occur then the result will depend on the code that you have written to handle the exception. Exceptions instantiated from **RuntimeException** and its subclasses are considered as *unchecked* exceptions.

Checked exceptions

Checked exceptions are those that cannot be ignored when you write the code in your methods. “According to Flanagan, the exception classes in this category represent routine abnormal conditions that should be anticipated and caught to prevent program termination.”

All exceptions instantiated from the **Exception** class, or from subclasses, of **Exception** other than **RuntimeException** and its subclasses, must either be:

(i) Handled with a **try** block followed by a **catch** block, or

(ii) Declared in a **throws** clause of any method that can throw them

The conceptual difference between checked and unchecked exceptions is that checked exceptions signal **abnormal conditions** that you have to deal with. When you place an exception in a throws clause, it ***forces*** to invoke your method to deal with the exception, either by catching it or by declaring it in their **own throws** clause. If you don't deal with the exception in one of these two ways, your class will not compile

4.4.1 Throwable class Hierarchy

All exceptions that occur in Java programs are a subclass of built–in class **Throwable**. Throwable class is top of the exception class hierarchy. Two classes **Exception** and **Error** are subclass of Throwable class. Exception class is used to handle exceptional conditions. Error class defines those exceptions which are not expected by the programmer to handle.

Exception class and its Subclasses in Throwable class Hierarchy

```
class java.lang.Object
  |
   + class java.lang.Throwable
                  |
                + class java.lang.Exception
                    |
                   + class java.awt.AWTException
                    |
                   +class java.lang.ClassNotFoundException
                    |
                    + class java.lang.CloneNotSupportedException
                    |
                    + class java.io.IOException
                    |
                    + class java.lang.IllegalAccessException
                    |
                    + class java.lang.InstantiationException
                    |
                    + class java.lang.InterruptedException
                    |
                   + class java.lang.NoSuchMethodException
                    |
                    + class java.lang.NoSuchMethodException
                     |
                    + class java.lang.RuntimeException
                    |
                    + class java.lang.ArithmeticException
                    |
                    + class java.lang.ArrayStoreException
                    |
                    + class java.lang.ClassCastException
                    |
                    + class java.util.EmptyStackException
                    |
                    + class java.lang.IllegalArgumentException
  |
 + class java.lang.Error
```

Figure 1: Throwable Class Partial Hierarchy

Now let us see in Table 1 some exceptions and their meaning.

Table1: Exceptions and Their Meaning

ArithmeticException	Division by zero or some other kind of arithmetic problem
ArrayIndexOutOfBoundsException	An array index is less than zero or greater than or equal to the array's length
FileNotFoundException	Reference to a file that cannot be found
IllegalArgumentException	Calling a method with an improper argument
IndexOutOfBoundsException	An array or string index is out of bounds
NullPointerException	Reference to an object that has not been instantiated
NumberFormatException	Use of an illegal number format, such as when calling a method
StringIndexOutOfBoundsException	A String index is less than zero or greater than or equal to the String's length

4.4.2 Runtime Exceptions

Programming errors that should be detected in testing for example index out of bounds, null pointer, illegal argument, etc. are known as runtime exception. Runtime exceptions do need to be handled (They are of the types that can be handled), but errors often cannot be handled.

Most of the runtime exceptions (members of the RuntimeException family) are thrown by the Java virtual machine itself. These exceptions are usually an indication of software bugs. You know problems with arrays, such as ArrayIndexOutOfBoundsException, or passed parameters, such as IllegalArgumentException, also could happen just about anywhere in a program. When exceptions like these are thrown, you have to fix the bugs that caused them to be thrown.

Sometimes you have to decide whether to throw a checked exception or an unchecked runtime exception. In this case you must look at the abnormal condition you are signalling. If you are throwing an exception to indicate an improper use of your class, then here you are signalling a software bug. In this case the class of exception you throw probably should descend from RuntimeException, which will make it unchecked. Otherwise, if you are throwing an exception to indicate not a software bug but an abnormal condition, then you have to deal with it every time your method is used. In other words your exception should be checked.

The *runtime* exceptions are not necessarily to be caught. You don’t have to put a try-catch for runtime exceptions, for example, every integer division operation to catch a divide by zero or around every array variable to watch for whether it is going out of bounds. But it is better if you handle possible runtime exceptions. If you think there is a reasonable chance of such exceptions occurring.

Check Your Progress 2

1) Explain the exception types that can be caught by the following code.

```
try
{
//operations
}
catch (Exception e)
 {
//exception handling.
}
```

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

2) Write a partial program to show the use of finally clause.

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

3) Is there anything wrong with this exception handing code? Will this code compile?

```
try
{
//operation code...
}
catch (Exception e)
{
//exception handling
}
catch (ArithmeticException ae)
{
 // ArithmeticException handling
}
```

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

4) Differentiate between checked and unchecked exceptions.

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

4.5 THROWING EXCEPTIONS

If you don’t want explicitly catching handle an exception and want to declare that your method throws the exception. This passes the responsibility to handle this exception to the method that invokes your method. This is done with the **throws** keyword.

Let us see the code snippet given below:

```
//code snippet
public static void copy (InputStream in, OutputStream out)
throws IOException
{
byte[] Mybuf = new byte[256];
while (true)
{
int bytesRead = in.read(Mybuf);
if (bytesRead == -1) break;
out.write(Mybuf, 0, bytesRead);
}
}
```

In this code snippet copy method throws IOException, and now the method invoke copy method is responsible for handling IOException.

Sometime single method may have to throw more than one type of exception. In this case the exception classes are just separated by commas. For example in the code:

```
public MyDecimal
public divide(MyDecimal value, int roundMode) throws ArithmeticException,
IllegalArgumentException
```

The divide method throws two exceptions ArithmeticException, and IllegalArgumentException.

See the program given below written for showing how IllegalArgumentException exceptions are thrown if arguments are not passed properly.

```
//program
class MyClock
{
int hours ,minutes, seconds;
public MyClock(int hours, int minutes, int seconds)
{
if (hours < 1 || hours > 12)
{
throw new IllegalArgumentException("Hours must be between 1 and 12");
}
if (minutes < 0 || minutes > 59)
{
throw new IllegalArgumentException("Minutes must be between 0 and 59");
}
if (seconds < 0 || seconds > 59)
{
throw new IllegalArgumentException("Seconds must be between 0 and 59");
}
this.hours   = hours;
this.minutes = minutes;
this.seconds = seconds;
}
public MyClock(int hours, int minutes)
{
this(hours, minutes, 0);
}
public MyClock(int hours)
{
this(hours, 0, 0);
}
}
public class ThrowTest
{
public static void main( String args [])
{
try
{
MyClock clock = new MyClock(12, 67,80);
}
catch( IllegalArgumentException e)
{
System.out.println("IllegalArgumentException is caught....");
}
}
}
```

Output:

```
IllegalArgumentException is caught....
```

Now let us see how own exception subclasses can be written.

4.6 WRITING EXCEPTION SUBCLASSES

Most of the exception subclasses inherit all their functionality from the superclass and they serve the purpose of exception handling. Sometimes you will need to create your own exceptions types to handle specific situations that arises in your applications. This you can do quite easily by just defining subclass of **Exception** class. Exception class does not define any method of its own .Of course it inherits methods of **Throwable** class. If you inherit Exception you have the method of Throwable class available for your class. If needed you can override the methods of Throwable class.

Some methods of Throwable class:

Throwable FillInStackTrce(): Fills in the execution stack trace.
String GetMessage() : Returns the detail message string of this throwable.
String ToString() : Returns a short description of this throwable.

Now see the program given below to create exception subclass which throws exception if argument passed to MyException class method compute has value greater that 10.

```
//program
MyException extends Exception
{
private int Value;
MyException (int a)
{
Value = a;
}
public String toString()
{
return ("MYException [for Value="+ Value +"]");
}
}
class ExceptionSubClassDemo
{
static void compute(int a) throws MyException
{
System.out.println("Call compute method ("+a+")");
if(a>10)
throw new MyException(a);
System.out.println("Normal Exit of compute method[for Value="+a +"]");
}
public static void main(String args[])
{
try
{
compute(5);
compute(25);
}
catch(MyException e)
{
System.out.println("MyException Caught :"+e);
}
}
}
```

Output:

```
Call compute method (5)
Normal Exit of compute method [for Value=5]
Call compute method (25)
MyException Caught: MYException [for Value=25]
```

Embedding information in an exception object

You can say that when an exception is thrown by you ,it is like performing a kind of structured go-to from the place in your program where an abnormal condition was detected to a place where it can be handled. The Java virtual machine uses the class of the exception object you throw to decide which catch clause, if any, should be allowed to handle the exception.

But you cannot take an exception just as a transfer control from one part of your program to another, it also transmits information. As mentioned earlier the exception is a full-fledged object that you can define yourself. Also you can embed information about the abnormal condition in the object before you throw it. The catch clause can then get the information by querying the exception object directly.

The Exception class allows you to specify a String detail message that can be retrieved by invoking getMessage() of Throwable class on the exception object. In your program at the time of defining it you can give the option of specifying a detail message like this:

```
class NotAcceptableValuException extends Exception
{
NotAcceptableValuException
{
}
NotAcceptableValuException (String msg)
{
super(msg);
}
}
```

Given the above declaration of NotAcceptableValuException, now you can create an object of NotAcceptableValuException in one of two ways:

```
new NotAcceptableValuException ()
new NotAcceptableValuException ("This Value is not Acceptable.")
```

Now a catch clause can query the object for a detail string, like this code snippet:

```
class MyValue
```

```
{
public void serveCustomers()
{
try
{
// operations
}
catch (NotAcceptableValuException e)
{
System.out.println (“NotAcceptableValuException Caught:”+e);
}
}
}
```

If during operations NotAcceptableValuException is generated and throw it will be handled by catch then statement
System.out.println (“NotAcceptableValuException Caught:”+e);

will print:
NotAcceptableValuException Caught:" This Value is not Acceptable." provided NotAcceptableValuException object is created as:
new NotAcceptableValuException ("This Value is not Acceptable.") ;

Check Your Progress 3

1) Explain how you can throw an exception from a method in Java.
………………………………………………………………………………………

………………………………………………………………………………………

………………………………………………………………………………………

……………………………………………………………………………………….

2) Write a program to create your own exception subclass that throws exception if the sum of two integers is greater that 99.

………………………………………………………………………………………

………………………………………………………………………………………

………………………………………………………………………………………

………………………………………………………………………………………
…

3) Dry run the following program and show the output:

```
//program
class MyStack
{
private int MaxSize;
private int size;
private Object[] ob1;
public MyStack(int cap)
{
ob1 = new Object[cap];
MaxSize = cap;
size = 0;
}
public void push(Object o)  throws StackException
{
if (size == MaxSize)
```

```
throw new StackException("overflow");
ob1[size++] = o;
}
public Object pop() throws StackException
{
if (size <= 0)
throw new StackException("underflow");
return ob1[--size];
}
public Object top()  throws StackException
{
if (size <= 0)
throw new StackException("underflow");
return ob1[size-1];
}
public int size()
{
return this.size;
}
}
class StackException extends Exception
{
StackException()  {}
StackException(String msg)
{
super(msg);
}
}
class StackTest
{
public static void main(String[] args)
{
MyStack s = new MyStack(5);
System.out.println("***** Welcome to Stack operations ****");
Test(s);
}
public static void Test(MyStack s)
{
try
{
s.push("Hi");
s.push("Java");
s.push(new Float(1.4));
s.push("Good for all");
s.push("Learn it");
s.push("Now"); // error!
}
catch(StackException se)
{
System.out.println(se);
}
try
{
System.out.println("Top of MyStack : " + s.top());
System.out.println("Popping data ...");
while (s.size() > 0)
System.out.println(s.pop());
}
catch(StackException se)
{
```

```
// This should never happen:
throw new InternalError(se.toString());
}
}
}
```

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

4.7 SUMMARY

This unit discusses how Java goes to great lengths to help you deal with error conditions. In this unit we have discussed how Java's exception mechanisms give a structured way to perform a go-to from the place where an error occurs to the code that knows how to handle the error. In this unit we have discussed different causes of exception, using try, catch, finally, throw and throws clauses in exception handling. This unit deals with ways to handle error conditions in a structured, methodical way. This unit discusses types of exceptions, Throwable class hierarchy, and explains how to write own exception subclasses.

4.8 SOLUTIONS/ANSWERS

Check Your Progress 1

1) An exception is a condition or a problem, due to which program stops execution from the point of occurrence of this condition or problem. If an exception has occurred the following things can be done:

   i. Fix the problem and try again.
   ii Exit the application with System.exit ()
   iii. Rethrow the exception to some other method or portion of code.

2) Yes, it is legal. It is not necessary for a try statement to have a catch statement if it has a finally statement. If the code in the try statement has multiple exit points and no catch clauses are associated to the code, then code in the finally statement is executed no matter how the try block is exited.

3)

```
//program
public class TwoCatch_Test
{
public static void main(String[] args)
{
try
{
int i = 1;
int data[] = {2,3,4,5};
System.out.println("Value at : " + data[2]);
i = i/(i-i);
}
catch( ArrayIndexOutOfBoundsException e)
{
System.out.println("Sorry you are trying to print beyond the size of data[]");
}
catch(ArithmeticException e)
{
System.out.println("Divide By 0 :"+e);
}
}
}
```

```
Output:
Value at: 4
Divide By 0: java.lang.ArithmeticException: / by zero
```

Check Your Progress 2

1) This code will handle catch exceptions of type Exception; therefore, it will catch any exception. This can be a poor implementation because you are losing valuable information about the type of exception being thrown and making your code less efficient. It is better to handle different exceptions on the basis of their actual type, which may be ArithmeticException, or IllegalArgumentException or anything else

2)
```
//Code snippet to explain finally
public void MyMethod(File file) throws Exception
{
FileInputStream stream = new FileInputStream(file);
try
{
// process stream object contents
}
finally
{
// No matter what happen in try block this statement will execute.
stream.close();
}
}
```
the code snippet given above give a hint that some file related operations are performed , in this code finally block will make sure in all the situations that object stream get closed.

3) This first handler catches exceptions of type Exception; therefore, it catches any exception, including ArithmeticException. In this situation the second handler could never be reached.

4) An unchecked exception is a type of exception that doesn’t force you to handle it. It is optional to handle it, or ignore it. Exceptions instantiated from **RuntimeException** and its subclasses are considered unchecked exceptions. Checked exceptions are those exceptions that cannot be ignored during you write the code in your methods. The conceptual difference between checked and unchecked exceptions is that checked exceptions signal abnormal conditions that you have to deal with, and it is not the case with unchecked exceptions.

Check Your Progress 3

1) A method in Java can throw exception using throws keyword. In code snippet below MyMethod() throws an ArrayOutOfBoundsExceptio:
```
//code snippet
class TestTrows
{
void MyMethod() throws ArrayOutOfBoundsException
{
 // operation code
throw ArrayOutOfBoundsException(“My ArrayOutOfBoundsException”);
}
```

2)
```
//program
```

```
class MySum extends Exception
{
int Sum;
MySum( int a , int b)
{
Sum = a+ b;
}
public String toString()
{
return ("MYException [for Sum="+ Sum + "]");
}
}
class ExceptionSumDemo
{
static void SumIt(int a, int b ) throws MySum
{
int Sum;
Sum= a+b;
System.out.println("Call SumIt ("+a+","+b+")");
if(Sum>99)
throw new MySum(a,b);
System.out.println("Normal Exit from  SumIt method [for Sum="+Sum +"]");
}
public static void main(String args[])
{
try
{
SumIt(20,50);
SumIt(90,11);
}
catch(MySum e)
{
System.out.println("MyException Caught :"+e);
}
}
}
```

Output:

```
Call SumIt (20,50)
Normal Exit from  SumIt method [for Sum=70]
Call SumIt (90,11)
MyException Caught :MYException [for Sum=101]
```

3) Output:

```
***** Welcome to Stack operations ****
StackException: overflow
Top of MyStack : Learn it
Popping data...
Learn it
Good for all
1.4
Java
Hi
```