```
throw  => it can be used inside the method body.
        it is used to throw the Exception object manually.
        it is used for throwing the "Userdefined" Exception object manually.

throws => it can be used at method signature level.
        it is used to indicate the compiler about the exception handling code.
        it is mainly used for "CheckedExeption" object.


Case4: we can use throws keyword only for constructors and methods but not for
classes.

eg#1.
class Test throws Exception//invalid
{
      Test() throws Exception{//valid

      }
      methodOne() throws Exception{//valid

      }
}

Exception handling keywords summary
===================================
1. try     => Maintain risky code
2. catch   => Maintain handling code
3. finally => Maintain cleanup code
4. throw   => To handover the created exception object to JVM manually
5. throws  => To delegate the Exception object from called method to caller method.

Various compile time errors in ExceptionHandling
================================================
1. Exception XXXX is already caught [try with mulitple catch blocks]
2. Unreported Exception XXX must be caught or declared to be thrown.
[CheckedException]
3. Exception XXXX is never thrown in the body of corresponding try statement.
[FullyCheckedException]

      SyntacticallyErrors
      +++++++++++++++++++
4. try without catch,finally
5. catch without try
6. finally without try

7. incompatible types :found xxx
                   required:Throwable  [using throws/throw Keyword]

8. unreachable code.[normal coding error]


CustomException in java
+++++++++++++++++++++++
eg#1.
//DrivingLicense Generator App :: ageFactor
/*
      age>60 :: TooOldAgeException
      age<18 :: TooYoungAgeException
       otherwise :: issueDL by checking the skill of driving.
```

```
*/

//InbuiltException -> CheckedException(partial,fully),UnCheckedException
//CustomException  -> UncheckedException[RuntimeException]

import java.util.Scanner;
class TooOldAgeException extends RuntimeException
{
	TooOldAgeException(String msg)
	{
		super(msg);
	}
}
class TooYoungAgeException extends RuntimeException
{
	TooYoungAgeException(String msg)
	{
		super(msg);
	}
}

public class Test
{
	public static void main(String[] args)
	{
		Scanner scanner =new Scanner(System.in);

		System.out.print("Enter the age of the candidate:: ");
		int age = scanner.nextInt();

		if (age>60)
		{
			throw new TooOldAgeException("Sorry DL can't be issued for senior
citizen people");
		}
		else if(age<18)
		{
			throw new TooYoungAgeException("Sorry DL can't be issued for
minor candidates");
		}
		else
		{
			System.out.println("You will get DL soon in registered
email...");
		}
	}
}

Output
Enter the age of the candidate:: 65
Exception in thread "main" TooOldAgeException: Sorry DL can't be issued for senior
citizen people
        at Test.main(Test.java:38)

Enter the age of the candidate:: 15
Exception in thread "main" TooYoungAgeException: Sorry DL can't be issued for minor
candidates
        at Test.main(Test.java:42)
```

```
Enter the age of the candidate:: 19
You will get DL soon in registered email...

Exceptions which are normally occured in java coding
====================================================
1. Based on the events occured exceptions are classified into 2 types
    a. JVM Exceptions
    b. Programtic Exceptions

JVM Exceptions
  => The exceptions which are raised automatically by the jvm whenever a particular
event occurs are called JVM Exceptions
     eg:: ArrayIndexOutOfBoundsException
          NullPointerException

ProgramaticExceptions
  => The exceptions which are raised explicitly by the programmer or by API
developers is called as "Programatic Exceptions".
      eg:: IllegalArgumentException,
           NumberFormatException

ArrayIndexOutOfBoundsException
   =>This exception is raised automatically whenever we are trying to access array
elements which is out of the range.


Top10JavaExceptions
===================
1. ArithmeticException    => while working with division operand on int type data.
2. NullPointerException   => While working with data which is actually holding
null.
3. StackOverFlowError     => while working with calling methods in recursive style.
4. IllegalArgumentException
      eg:: Thread t=new Thread();
         t.setPriority(10);
         t.setPriority(100);//invalid
5. NumberFormatException  => while working with Wrapper classes.

6. ExceptionInInitializerError  => if exception occurs during the initialization of
static variables or static blocks.

eg#1.
public class Test
{
      static int i = 10/0;
}

eg#2.
public class Test
{
      static
      {
            String s= null;
            System.out.println(s.length());
      }
}

7.  ArrayIndexOutOfBoundsException
8.  NoClassDefFoundError
```

9. ClassCastException
10. IllegalStateException(learn in servlet programming :: Session)
11. AssertionError(learn in Junit)

JVMException
============
 a. ArithmeticException
 b. NullPointerException
 c. ArrayIndexOutOfBoundsException
 d. StackOverFlowError
 e. ClassCastException
 f. ExceptionInInitalizerError

ProgrammaticException[We use API Code given by other developers]
==
 a. IllegalArgumentException
 b. NumberFormatException
 c. IllegalStateException
 d. AssertionError

1.7 version Enhancements
========================
 1. try with resource
 2. try with multicatch block

untill jdk1.6, it is compulsorily required to write finally block to close all the resources which are open as a part of try block.

```
eg:: BufferReader br=null
      try{
       br=new BufferedReader(new FileReader("abc.txt"));
      }catch(IOException ie){
          ie.printStackTrace();
      }finally{
       try{
         if(br!=null){
            br.close();
         }
          }catch(IOException ie){
            ie.printStackTrace();
        }
      }
```

Problems in the apporach
========================
1. Compulsorily the programmer is required to close all opened resources which increases the complexity of the program
2. Compulsorily we should write finally block explicitly, which increases the length of the code and reviews readablity.
   To Overcome this problem SUN MS introduced try with resources in "1.7" version of jdk.

try with resources
==================
In this apporach, the resources which are opened as a part of try block will be closed automatically once the control reaches to the end of
try block normally or abnormally,so it is not required to close explicitly so the complexity of the program would be reduced.

It is not required to write finally block explicitly,so length of the code would be reduced and readability is improved.

```
try(BufferedReader br=new BufferedReader(new FileReader("abc.txt")){
      //use br and perform the necessary operation
        //once the control reaches the end of try automatically br will be closed

}catch(IOException ie){
     //handling code
}
```

Rules of using try with resource
==========
1. we can declare any no of resources, but all these resources should be seperated with ;
 eg#1.
```
   try(R1;R2;R3;){
         //use the resources
   }
```

2. All resources are said to be AutoCloseable resources iff the class implements an interface called "java.lang.AutoCloseable" either
    directly or indirectly
         eg:: java.io package classes, java.sql.package classes

3. All resource reference by default are treated as implicitly final and hence we can't perform reassignment with in try block.
```
      try(BufferedReader br=new BufferedReader(new FileWriter("abc.txt")){
            br=new BufferedReader(new FileWriter("abc.txt"));
      }
```
    output::CE: can't reassign a value

4. untill 1.6 version try should compulsorily be followed by either catch or finally, but from
   1.7 version we can take only take try with resources without cath or finally.
```
         try(R){
            //valid
         }
```

5. Advantage of try with resources concept is finally block will become dummy because we are not required to close resources explicitly.

MultiCatchBlock
==========
Till jdk1.6, eventhough we have multiple exception having same handling code we have to write a seperate catch block for
every exceptions, it increases the length of the code and disturbs readability.

code
++++
```
try{
   ....
   ....
   ....
   ....
}catch(ArithmeticException ae){
      ae.printStackTrace();
}catch(NullPointerExcepion ne){
```

```
      ne.printStackTrace();
}catch(ClassCastException ce){
      System.out.println(ce.getMessage());
}catch(IOException ie){
      System.out.println(ie.getMessage());
}
```

To overcome this problem SUNMS has introduced "Multi catch block" concept in 1.7 version

```
try{
   ....
   ....
   ....
   ....
}catch(ArithmeticException |NullPointerException e){
      e.printStackTrace();
}catch(ClassCastException |IOException e){
      e.printStackTrace();
}
```

In multicatch block,there should not be any relation b/w exception types(either child to parent or parent to child or same type) it would result in compile time error.

eg::

```
try{

}catch( ArithmeticExeption | Exception e){
      e.printStackTrace();
}
```

Output:CompileTime Error

eg::

```
try
{
      int a  = 10/0;
}
catch (ArithmeticException | NullPointerException |ClassCastException e)
{
      //handling logic
      e.printStackTrace();
}
```

Exception Propogation

With in a method, if an exception is raised and if that method does not handle that exception then Exception object will be propogated to the caller method then caller method is responsible to handle that exceptions,

This process is called as "Exception Propogation/Ducking".

ReThrowing an Exception

To convert one exception type to another exception type, we can use rethrowing exception concept.

eg::

```
public class TestApp
```

```
{
	public static void main(String[] args)
	{
		try{
			System.out.println(10/0);
		}catch( ArithmeticException  e){
			throw new NullPointerException();
		}
	}
}
Output::
Exception in thread "main" java.lang.NullPointerException
	at TestApp.main(TestApp.java:10)
```