Exception Handling

- Introduction
- Runtime stack mechanism
- Default Exception Handling
- Exception Hierarchy
- Customized exception handling by try-catch
- Control flow in try catch
- Methods to print exception information
- try with multiple catch blocks
- finally
- Difference between final,finally,finalize()
- Various possible combination of try-catch-finally
- Control-flow in try-catch-finally
- Control-flow in nested try-catch-finally
- Throw
- Throws
- Exception handling keywords summary
- Various possible compile time errors in exception handling
- Customized Exception
- Top Ten Exception

Introduction

Exception

Definition:

When unwanted, unexpected event that disturbs normal flow of program is called 'Exception'.

Example:

Sleeping Exception,FileNotFound Exception

- It is highly recommended to handle exception. The main objective of exception handling is "Graceful termination of the program".
- Exception Handling does not mean repairing an exception, we have to define alternative way to continue rest of the program, normally this is nothing but "Exception Handling".
  Example:
  If our programming requirement is to read data from the file locating at London and at a run-time, if that file is not available our program should not be terminated abnormally. We have to provide a local file to continue rest of the program normally. This is nothing but exception handling.

  Syntax:

```
   try
   {
Read data from London file


    }
  catch (FileNotFoundException e)
 {
Use local file and continue rest of the program normally
 }
```

Runtime stack mechanism

- For every thread JVM create a Runtime Stack.
- ALL the method call performed by the thread will be store in stack.
- Each entry in stack is called “Activation Entry” or “stack frame”.
- After completing every method call JVM deletes corresponding entry from the stack.
- After completing all method calls, just before terminating the thread JVM destroy the stack.

Default Exception Handling

- If any exception raised, the method in which it is raised is responsible to create exception object by including the following information.
  1) Name of exception
  2) Description of exception.
  3) Location of exception.

- After creating exception object method handover that exception object to JVM,
- JVM checks whether the method contain any exception handling code or not.
- If the method contain any handling code, then it will be executed and continue rest of the program normally.
- If it doesn't contain handling code, then JVM terminates that method abnormally and removes corresponding entry from the stack.
- JVM identifies the caller method and checks whether caller method contains code or not. If caller method doesn't contain any handling code, then JVM terminates that caller method also abnormally and removes corresponding entry from stack.
- This process will continue until main() and if the main also doesn't contain handling bode JVM terminates the main() also abnormally and removes corresponding entry from stack.
- Just before terminating the program abnormally JVM handover the responsibility of exception handling to the default exception handler.
- Default Exception handler just print exception information to the console in the following format:
- Name of Exception:description
  Location(StackTrace

Exception Hierarchy

Throwable class acts as a root for entire java exception hierarchy. It has following two c

child classes.

1) Exception

2) Error

1) Exception:

The most of the cases exceptions are cause by our program and these are recoverable.

2) Error:

Most of the cases errors are not cause by the our program these are due to lack of system resources.

Errors are non-recoverable.

- **Checked Vs.Unchecked Exception:**
  - The exceptions which are checked by compiler for smooth execution of the program at run time are called “checked exception”.
    Example:
    1)HallTicketMissingException
    2)PenNotWorkingException
    3)FileNotFoundException
  - The Exception which are not checked by compiler are called “Unchecked Exception”.
    Example:
    1)BombBlastException
    2)Arithmetic Exception

- Whether exception is checked or unchecked compulsory it should run time only. Their is no chance of occurring at compile time.
- Runtime exceptions and its child classes, errors and its child class are unchecked exception and all remaining are checked exceptions.

- **Partially checked Vs. Fully checked**

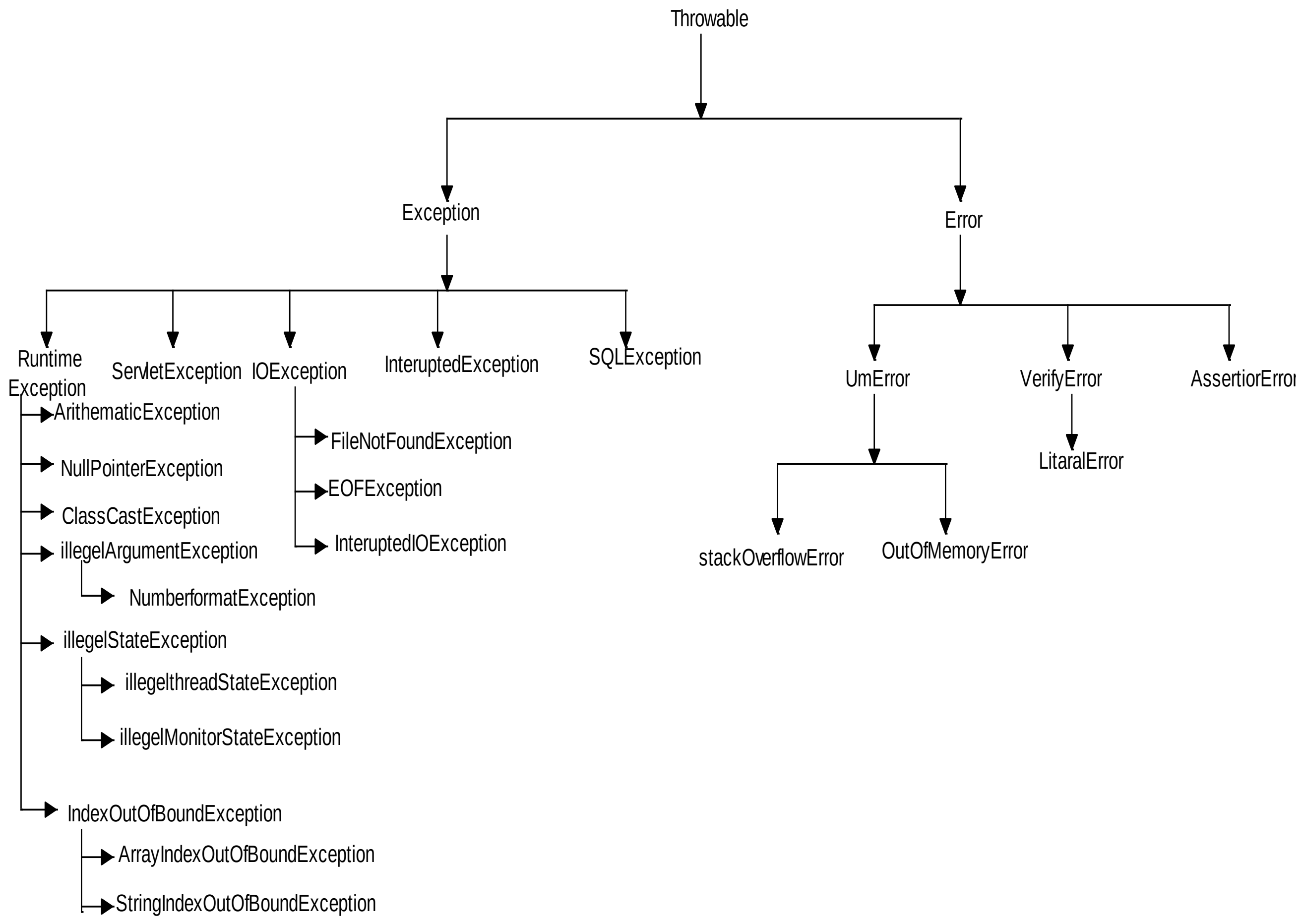
1)A checked exception is said to be fullychecked iff all its child classes also checked.
Example:IOException
2)Achecked exception is said to be partially checked iff some of its child classes are unchecked.
Example:Exception

Qu) Which are the following are checked

1)IOException	: Fullychecked
2)Error	: unchecked
3)Throwable	: partiallychecked
4)NullPointerException	: unchecked
5)InterruptedIOException	: Fullychecked
6)SQLException	: Fullychecked

Note:In java the only partially checked exceptions are 1.Exception
2)Throwable

Exception Hierarchy:



Customized Exception handling by try-catch

- We can maintain risky code with in the try block and corresponding handling code inside catch block.

  Syntax:

  ```
  try
  {
  riskycode
  }
  catch(*** e)
  {
  handling  code
  }
  ```

  Example 1:

  ```
  public class exception {
      public static void main(String ar[])
      {
      System.out.println("state1");
      System.out.println(10/0);
      System.out.println("state3");
      }
  }                                        //Abnormal Termination
  ```

  Output:

  ```
  state1
  Exception in thread "main" java.lang.ArithmeticException: / by zero at
  com.exceptionhandling.exception.main(exception.java:15)
  Java Result: 1
  ```

Example 2:

```
public class exception1 {
    public static void main(String ar[])
    {
    System.out.println("state1");
    try
    {
    System.out.println(10/0);
    }
    catch(ArithmeticException e)
    {
     System.out.println(10/2);
    }
    System.out.println("state3");
    }
}
```

Output:

```
state1
5
state3                              //Normal Termination
```

Control flow in try-catch

Example:

```
public class Exceptionflow {
    try
    {
    state1;
    state2;
    state3;
    }
    catch(*** e)
    {
    state4;
    }
    state5;
}
```

Case1:

If there is no exception 1,2,3,5 statements are normal terminations.

Case2:

If an exception raised at statement2 and corresponding catch block matched,1,4,5 are normal terminations.

Case3:If an exception raised at statement2 and The corresponding catch block not matched,followed by abnormal termination.

Case4:

If an exception at statement4 or statement 5 it always abnormal termination

Note:

1)Within in the try block if any where an exception raised then rest of the try block won't be executed eventhough we handle that exception.Hence,it is recommended to take only risky code with in the try block and length of the try block should be as less as possible.

2)If any exception raised at any statement which is not part of try then it is always abnormal termination.

➢ Various method to print exception information:

- Throwable class defines the following method to print exception information:

  1)printStackTrace():

  This method print exception information in following way.

  Name of exception:Description

  StackTrace

  2)toString:

  It prints exception information in following format.

  Name of Exception:Description

  3)getMessage():

  This method points only description of the exception.

  Description

  Example:

```
public class printingmethod {
    public static void main(String ar[])
    {
    try
    {
     System.out.println(10/0);
    }
    catch(ArithmeticException e)
    {
        e.printStackTrace();
        System.out.println(e.toString());
        System.out.println(e.getMessage());
    }
    }
}
```

Note:

Default exceptionhandler internally use printStackTrace().

try with multiple catch block

The way of handling an exception is varied from exception to exception hence for every exception it is recommended to take seprate catch block.

Example 1:

```
try
{
}
catch(exception e)
{
}
```

Example 2:

```
try
{
}
catch(ArithmeticException e)
{
Perform these arithmetic operation;
}
catch(FileNotFoundException e)
{
Use local file;
}catch(NullPointerException e)
{
Use another resources;
}catch(Exception e)
{
Default Exception Handles;}
```

- Hence,try with multiple catch is possible and highly recommended to use.
- If try with multiple catch block present ,then oreder of catch block is very important and it is should be from child to parent.
- If we are taking fromparent to child we will gwt compile-time error saying “Exception has already been caught”.

Child to parent is follows

Example 1:

```
try
{
---
---
}
catch(Exception e)
{
----
----
}catch(ArithmeticeException e)
{
----
----
}                       //C.E.Exception
                         // java.lang.ArithmeticException has
                         //already been caught
```

Example 2:

```
try
{
---
---
}
catch(ArithmeticException e)
{
----
----
}catch(Exception e)
{
----
----
}
```

finally Block

- It is never recommended to define clean-up code with in the try block because their is no guaranty for the execution of every statement.
- It is never recommended to define clean-up code within the catch-block,because it won't be executed if there is no exception.
- If we required a place to maintain clean-up code which should be executed always irrespective of whether exception is raised or not raised and whether handle or not handle,such type of place is nothing but finally block.
- Hence the main purpose of finally block is to maintain clean-up code which should executed always.

Syntax:

```
try

{

Risky code;

} catch(*** e)

{

Handling code

}

finally

{

Clean-up code}
```

Example 1:

```
public class Test1 {

    public static void main(String ar[])
    {
    try
    {
    System.out.println("try");
    }catch(ArithmeticException e)
     {
     System.out.println("catch");
     }
    finally
    {
    System.out.println("finally");
    }
    }
}
```

Output:

```
try
finally
```

Example 2:

```
public class Test2 {
    public static void main(String ar[])
    {
    try
    {
    System.out.println("try");
     System.out.println(10/0);
    }catch(ArithmeticException e)
     {
     System.out.println("catch");
     }
    finally
    {
    System.out.println("finally");
    }
    }
}
```

Output:

```
try
catch
finally
```

Example 3:

```
public class Test3 {
    public static void main(String ar[])
    {
    try
    {
    System.out.println("try");
     System.out.println(10/0);
    }catch(NullPointerException e)
     {
     System.out.println("catch");
     }
    finally
    {
    System.out.println("finally");
    }
    }


}
```

```
Output:try
finally
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at com.exceptionhandling.Test1.main(Test1.java:17)
Java Result: 1
```

return Vs. finally

- Finally block dominates return statement also.Hence,if there is any return statement present inside try or catch block.First finally wii be executed and then return statement will be considered.
- Example:

```
public class returnexception {
    public static void main(String ar[])
    {
    try
    {
    System.out.println("try");
    return;
    }
    catch(ArithmeticException e)
    {
        System.out.println("catch");
     }
    finally
    {
    System.out.println("finally");
    }
    }
}
```

Output:

try

finally

- There is no situation where the following finally block won't executed is whenever JVM shutdown. i.e. when ever we are using systemexit(0).
  Example:

```
public class RETURNFINALLY {
    public static void main(String ar[])
    {
    try
    {
    System.out.println("try");
      System.exit(0);
    }catch(ArithmeticException e)
    {
      System.out.println("catch");
    }
    finally
    {
      System.out.println("finally");
    }
    }
}
```

  Output:

```
Try
```

  Difference between final,finally and finalize

  1)final:

  It is a modifier apllicable for classes,methods and variables.

  If class declared as final,then child class creation is not possible.

  If method declared as final,then overriding of thay method is not possible.

If variable declared as final,then reassignment is not allowed because,it is constatnt.

2)finally:

It is block always associated with try-catch to maintain clean-up code which should be executed always irrespective of whether exception raised or not raised and eheather handled or not handled.

3)finalize();

It is a method should be executed by garbage collector before destroying any object to perform clean-up activities.

Note:

When compare with finalize(),it is highly recommendedto use finally block to maintain the clan-up code because we can't expect exact behavior of the garbage collection.

Various possible combinations of try-catch-finally:

1)

```
    try
{
   }catch(*** e)
   {
   }                     //Valid
```

2)

```
    try
{
   }catch(*** e)
   {
   }catch(*** e)
   {
      }                              //Valid

3)   try
{
      }
      finally
      {
      }                           //Valid

4)  try
{
      }                          //C.E.try without catch or finally

5)  catch
      {
      }                       //C.E.catch  without try

6)   finally
      {
      }                       ////C.E.finally  without try

7)) try
      {
      }
      System.out.println(“Hello”)
      catch(***  e)
       {
       }                         //C.E.try without catch or finally
                                  //C.E.catch without try

8)      try
{
      }catch(*** e)
      {
      }catch(*** e)
      {
      }

System.out.println("Hello");
                                        //C.E. catch witout try

9)    try
{
      }catch(*** e)
      {
      }
      System.out.println("Hello")
finally
      {
      }                  //C.E. finally  without try

10)   try
{
      }catch(*** e)
      {
     }
finally
      {
      }
      finally
      {
      }                  //C.E. finally without try

11) try
{
      }catch(ArithmeticException e)
     {
     }
    catch(Exception e)
     {
     }                    //Valid

12)try
{
    }catch(Exception e)
    {
     }
  catch(ArithmeticException e)
   {
   }                      //C.E. Exception java.lang AE has
                                 Already been caught
13)try
{
    }catch(ArithmeticException e)
    {
     }
  catch(ArithmeticException e)
   {
   }                      //C.E. Exception java.lang AE has
                                 Already been caught
14)try
{
    }catch(*** e)
    {
try
{
    }catch(*** e)
    {
     }
}                               //Valid

15))try
{
    }catch(*** e)
    {
}
finally
       {
      try
     {
     }
    finally
    {
 }
}                                      //Valid

16)try
   {
   try
  {
  }catch(*** e)
  {
}                                      //C.E.try without catch and
                                           finally
```

Control-flow in try-catch-finally

```
       try
{
        state 1;
        state 2;
        state 3;
      }catch(*** e)
      {
         state 2;
      }
     finally
     {
```

```
     state 5;
      }
      state 6;
```

Case1:

If there is no exception then 1,2,3,5,6 normal termination.

Case2:

If an exception raised at statement 2 and the corresponding catch block matched 1,4,5,6 normal termination.

Case2:

If an exception raised at statement 2 and the corresponding catch block not matched 1,5 abnormal termination.

Case 4:

If an exception raised at statement 4 and then
It is always abnormal termination but before that finally block to be executed.

Case 5:If an exception raised at statement 5,6 it is always abnormal termination.

Control-flow in nested try-catch-finally

```
     try
{
       state 1;
       state 2;
       state 3;
            try
              {
                state 4;
                state 5;
                state 6;
                }
         }catch(*** e)
         {
            state 7;
         }
        finally
        {
        state 8;
         }
         state 9;
      }catch(*** e)
      {
       state 10;
      }
      finally
     {
       state 11
     }
     state12;
```

Case1:

If there is no exception then 1,2,3,4,5,6,8,9,11,12 normal termination.

Case 2:

If there is an exception raised at statement 2 and corresponding catch blocked matched, then 1,10,11,12 normal termination.

Case 3:

If there is an exception raised at statement 2 and corresponding catch blocked not matched ,then 1,11 normal termination.

Case 4:

If there is an exception raised at statement 5 and corresponding inner catch has matched1,2,3,4,7,8,9,11,12 normal termination.

Case 5:

If there is an exception raised at statement 5 and corresponding inner catch has not match but outer catch has matched then 1,2,3,4,8,10,11,12 normal termination.

Case 6:

If there is an exception raised at statement 5 and inner and outer catch block are not matched then 1,2,3,4,8,11 abnormal termination.

Case 7:

If there is an exception raised at statement 7 and corresponding catch block match then 1,2,3……..11,12 normal termination.

Case 8:

If there is an exception raised at statement 7 and the corresponding catch block not matched then 1,2,3…11 abnormal termination.

Case 9:

If there is an exception raised at statement 8and

The corresponding catch block match then 1….12 normal termination.

Case 10:

If there is an exception raised at statement 8 and the corresponding catch has not matched then 1,2,3…11 abnormal termination.

Case 11:

If there is an exception raised at statement 9 and the

Corresponding catch matched then 1,2……12 normal termination.

Case 12:

If there is an exception raised at statement 9 and the

Corresponding catch not matched then 1,2……8,11 abnormal termination.

Case 13:

If there is an exception raised at statement 10 it is always abnormal termination, but before the finally block will be executed.

Case 14:

If there is an exception raised at statement 11 or statement 12 it is always abnormal termination.

Throw

Some times we can create exception object manually and handover that object to the JVM explicitely by using throe keyword.

throw new ArithmeticException("/ by zero")

To hand-over our created Creation of A.E. object explicitly

Exception object to the

JVM manually

Hence, the main purpose4 of throw keyword is to hand-over our ctrated exception object manually to the JVM.

The result of the following two program is exactly same.

Example 1:

```
class test
{
public static void main(String ar[])
{
```

```
System.out.println(10/0);
}
}
```

In this case ArithmeticException object created internally and hand-over that object automatically by the main()

Example 2:

```
class test
{
public static void main(String ar[])
{
throw new ArithmeticException("/by zero");
}
}
```

In this case ArithmeticException object created and we hand-over it to the JVM manually by using throw keyword.

- In general we can use throw keyword for customized exceptions.
  Case 1:
  If we are trying to throw null reference we will get NullPointerException.
  Example 1:

```
  class test
{
static ArithmeticException e;
public static void main(String ar[])

{

throw e;

}

}
```

Runtime Exception:NullPointerException

Example 2

```
    class test

{

static ArithmeticException e=new ArithmeticException()

public static void main(String ar[])

{

throw e;

}

}
```

Runtime Exception:ArithmeticException

Case 2:

After throw statement we are not allow to write any statement directly otherwise we will get compile-time error saying “unreachable statement”.

Example 1:

```
class throwex
```

```
{
public static void main(String ar[])
{
Syatem.out.println(10/0);
Syatem.out.println(“Hello”);
}
}
```

Runtime Exception:ArithmeticException /by zero

Example 2:

```
class throwex1
{
public static void main(String ar[])
{
throw new ArithmeticException(“/ by zer0”)
Syatem.out.println(“Hello”);
}
}
```

C.E. unreachable statement

Case 3:

We can use throw keyword only for throwable type otherwise we will get complile-time errors saying incompatible types.

Example 1)

```
class throwex

{

public static void main(String ar[])

{

throw new throwex();

}

}

                                   // C.E.Incompatible types

Required:java.lang.Throwable

found:Test
```

Example 2:

```
class throwex

{

public static void main(String ar[])

{

throw new throwex()

}

}

Runtime Exception:Exception in thread main:throwex
```

THROWS

In our program if there is any chance of raising checked exception compulsory we should handle it otherwise we will get comple-time error saying “unreported exception *** must be caught or declare to be thrown”.

Example 1:

```
 Class test

{

public static void main(String ar[])

{

Thread.sleep(5000);

}

}
```

C.E. unreported exception java.lang.IE must be caught.

- We can handle this by using the following teo ways.
  1)By using try-catch block.
  2)By using throws keyword.

1)By using try-catch block:

```
class test

{

public static void main(String ar[])

{

try

{

Thread.sleep(5000);
}catch(InterruptedException e)
{
}
}
}
```

2)By using throws keyword

```
class test
{
public static void main(String ar[])throws InterruptedException
{
Thread.sleep(5000);
}
}
```

- Hence,the main purpose of throws keyword is to delegate responsibility of exception handling to yhe caller method in case of checked exception,to convence compiler.
- In case of unchecked exception,it is not required to throws keyword.
  Example 1)

```
class test
{
public static void main(String ar[])throws InterruptedException
{
 dostuff();
{
        public static void dostuff()throws IntrruptedException
        {

       domorestuff()

        }

public static void domorestuff()throws IntrruptedException

        {
        thread.sleep(5000)
        }
        }
```

- In the above program,If we are removing anthrows keyword.The code won’t be compile.Compulsary we should should use 3 throws statements.
- We can use throws keyword only for throwable types otherwise we will get compile-time error saying incompatable types.

  Example 1:

```
  class test
{
 Public static m1()throws test
{
}
}                              //C.E. incompatable type
                                 required:java.lang.throwable
                                  found:test

class test extends exception
{
public static m1()throws test
{
}
}
```

  Case (1):

- Checked:

```
class test
{
public static void main(String ar[])
{
throw new Exception()
}
}
```

C.E. unreported exception java.lang.Exception must be caught at declared to be thrown.

As exception checked compulsory we should handle either by try-catch or by throws keyword.

- Unchecked:

```
class test

{

public static void main(String ar[])

{

throw new Error()

}

}
```

R.E. Exception in thread “main” java.lang.Error

As Error is unchecked .It is not required to handle by try-catch or by throws.

Case 2)

In our program, if there is no chance of raising an exception then, we can’t define catch block for that exception otherwise we will get compile time error. But this rule is applicable for only fullychecked Exception.

Example 1:

```
try
{
System.out.println("Hello");
}catch(ArithmeticException e)
{
}
Output:Hello
```

Example 2:

```
try
{
System.out.println("Hello");
}catch(Exception e)
{
}
Output:Hello
```

Example 3:

```
try
{
System.out.println("Hello");
}catch(IOException e)
{
```

}

C.E. Exception java.lang.IOException is never thrownin body of corresponding try statement.

Example 4)

```
try

{

System.out.println("Hello");

}catch(Error e)

{

}
```

Output:Hello

Various possible compiletime error in Exception Handling:

1) Exception ****has already been caught
2) Unreported exception *** must be caught or declared to be thrown.
3) Exception ***is never thrown in body of corresponding try statement
4) Try without catch or finally
5) Finally without try
6) Catch without try
7) Unreachable statement
8) Incompatabletypes
   required:java.lang.Throwable
   found:Test

Exception Handling Keyword

1)try :To maintain risky code.
2)catch :To maintain handling code.
3)finally:To maintain clean-up code.
4)throw: To handover our created exception object to the JVM manually.
5)Throws:To delegate the responsibility.

Customized Exception:

To meet our programming requirement some times we have to create our own exception such type of exception are called ‘Customized Exceptions’.
Example:TOOYoungException,TooOldException.

Example 1:

```
public class CustomizedException extends RuntimeException{
   public static void main(String ar[])
   {
   int age=80;
   if (age>60)
   {
      throw new TooOldException("age is crossed for marrige");
   }
   else if (age>60)
   {
      throw new TooYoungException("plz wait some more time");
   }
   }
   }
```

```
class TooYoungException extends RuntimeException
{
   TooYoungException(String s)
   {
      super(s);

   }
}
class TooOldException extends RuntimeException
{
    TooOldException(String s)
    {
       super(s);
    }
}
```

Output:

```
Exception in thread "main" com.exceptionhandling.TooOldException: age is
crossed for marrige
        at com.exceptionhandling.CustomizedException.main
```

Note:It is highly recommended to keep our customized exception class as unchecked,i.e. we have to extend runtime exception class but not exception class while defining our customized exceptions.

Top 10 Exception:

Based on the source,who triggers the exception all exception are divided into two types:
1)JVM Exception
2)Programatic Exception

- **JVM Exception:**

  The Exception which are raised by JVM automatically by JVM,whenever particular event occurs are called "JVM Exception".
  Example:

1)ArrayIndexOutOfBoundException
2)NullPointerException

- **Programatic Exception:**

The Exception which are raised explicitely by the programmer or by the API developers are called programmatic Exception.

Example:
1)IllegalArgumentException
2)NumberFormatException

1)ArrayIndexOutOfBoundException:

- It is the child class of runtime exception and hence it is unchecked.
- Raised automatically by the JVM,whenever we are trying to access array element without of range index.

Example:

```
    int[] a=new int[10];
System.out.println(a[0]);     //0
System.out.println(a[100]) ;
                   //R.E.ArrayIndexOutOfBoundException
```

2)NullPointerException:

- It is the child class of runtime exception and hence it is unchecked.
- Raised automatically by the JVM,whwnever we are trying to perform any operation an null.
- Example: String s=null;

```
System.out.println(s.length());
//R.E.NullPointerException
```

3)StackOverflowError:

- It is the child class of error and hence it is unchecked.
- Raised automatically by the JVM,whenever we are trying to perform recursive method invocation.
- Example:

```
class test
{
public static void m1()
{
m2();
}
public static void m2()
{
m1();
}
public static void main(String ar[])
{
m1();
}
}
//R.E.StackOverflowError
```

4)StackOverflowError:

- It is the child class of error and hence it is unchecked.
- Raised automatically by the JVM,unable to find required class.
- Example:
  Java CST
- If CST.class file is not available then we will get R.E. NoClassDeFoundError.

5)ClassCastException:

- It is the child class of runtime exception and hence it is unchecked.
- Raised automatically by the JVM,whenever we are trying to typecast parent to the child type.
- Example:

```
String s=new String("CST");
 Object o=(Object)s;                    //Valid

 Object o=new Object();
 String s=(String)o                     //Invalid
```

6)ExceptionInInitializerError::

- It is the child class of error and hence it is unchecked.
- Raised automatically by the JVM,whenever if any exception occurs while performing initialization for static variables and while executing static blocks.

Example 1:

```
public class Test12 {

  static int i=10/0;

 public static void main(String ar[])

 {

  Test12 obj=new Test12();

          }

}

         R.E.java.lang.ExceptionInInitializerError

Caused by: java.lang.ArithmeticException: / by zero
```

Example 2:

```
public class test13 {

 static

{

        String s=null;

        System.out.println(s.length());

 }

 public static void main(String ar[])
{

 }

 }
```

R.E.java.lang.ExceptionInInitializerError

Caused by: java.lang.NullPointerException

7)IllegalArgumentException:

- It is the child class of runtime exception and hence it is unchecked.
- Raised explicitly by the programmer or by API Developer to indicate that a method has been invoked with invalid argument.

Example 1:

```
public class IllegalArgunemt extends Thread{
    public void run()

    {
        for (int i=0;i<10;i++)
        {
        System.out.println("Hello");
        }
    }
    public static void main(String ar[])
    {
       IllegalArgunemt obj=new IllegalArgunemt();
       obj.setPriority(10);
       obj.setPriority(100);
       obj.start();
       for (int i=0;i<10;i++)
        {
        System.out.println("Hiiii");
        }
    }
    }
R.E.Exception in thread "main"
java.lang.IllegalArgumentException
    at java.lang.Thread.setPriority
```

8)NumberFormatException:

- It is the child class of runtime exception and hence it is unchecked.
- Raised explicitly by the programmer or by API Developer to indicate that we are trying to convert String to Number type but the String is not properly formatted.

Example 1:

```
 int i=Integer.parseint("10");                    //Valid
int i=Integer.parseInt("Ten");           //R.E. NumberFormatException.
```

9) IllegalStateException:

- It is child class of runtime exception and hence,it is unchecked.
- Raised explicitly by the programmer or by API developer to indicate that a method has been invoked at in appropriate time.
- Example 1:Once session expires we can't call any method on that object otherwise we will get IllegalStateException.

```
  HttpSession sec=req.getSession();
  System.out.println(session.getId());
 Sesion.invalidate();
 System.out.println(session.getId());
//R.E.IllegalThradStateException.
```

Exmple 2:

```
   Thread t=new Thread();
```

```
    t.start();

    -

    -

    -

  t.start();
//R.E.IllegalThreadStateException.
```

- After starting a thread,we are not allowed to restart the same thread,otherwise we will get R.E. IllegalThreadStateException

10) AssertionError:

- It is child class of error and hence it is unchecked.
- Raised explicitly by the programmer or by API developer to indicate that assert statement fails.
- Example 1:

```
        assert(false).

R.E.AssertionError
```

- Exception Propogation:
  - The process of delegating the responsibility exception handling from one method to another method by using Throws Keyword is called Exception Propogation.