CHAPTER 4 : EXCEPTION HANDLING

EXCEPTION:- A program get abnormal condition during the time of execution is known as exception. When exception occurs in a program then the control jumps out of the java program back to the jvm without executing rest of the statements.

EXCEPTION HANDLING:- It is a mechanism that allows us to keep the control within the java program during the exception. Java provides 3 blocks called try block, catch block, & finally block along with 2 keywords (throw, throws) to deal with exception.

TRY BLOCK:- It is a block of code that generate an exception i.e. the statements which are most error prone to the program execution are generally kept in a try block. The exception will generally occur within the block.

CATCH BLOCK:- It is a block of code that executes during the exception occurrences. When exception occurs in the try block then the control jumps to catch block & it remains in the program. When exception will not in try block then the control will never execute the catch block. Generally we should provide the code which will become a solution to the exception that occurs in try block.

FINALLY BLOCK:- It is a block of code that executes independent of the exception occurrences. When exception occurs in try block then the control execute the catch and then the control execute finally block. If exception will not occurs in try block then the control execute the finally block directly.

TRY-CATCH-FINALLY:- These three blocks generally forms a unit known as exception handler code or exception handling mechanism. The sequence of these blocks must be try, catch, finally. A try block may have multiple catch, but it must have a finally block. A try or catch or finally can not be stand alone. The variable or object declare in a block are not accessible in another block. We cannot write any executable statement after end of one block & before the beginning of another block.

```
class Excp {
  public static void main(String args[]) {
    System.out.println("size : " +
args.length );
    int x = Integer.parseInt( args[0] );
    System.out.println( "finish 1" );
    int y = Integer.parseInt( args[1] );
    System.out.println( "finish 2" );
    int z = x / y ;
    System.out.println( "div : " + z );
    System.out.println("end of prog");
}};
```

```
class Excp1 {
  public static void main(String args[]) {
  try {
    System.out.println("size : " + args.length );
    int x = Integer.parseInt( args[0] );
    System.out.println( "finish 1" );
    int y = Integer.parseInt( args[1] );
    System.out.println( "finish 2" );
    int z = x / y ;
    System.out.println( "div : " + z );
  } catch( ArithmeticException ae){
    System.out.println("2nd no. should not be
zero");
  } catch( ArrayIndexOutOfBoundsException sei){
    System.out.println("less value at command
line");
  } catch( NumberFormatException hei){
    System.out.println("provide proper value");
} finally {
    System.out.println("thanks");
}
    System.out.println("end of prog");
}};
```

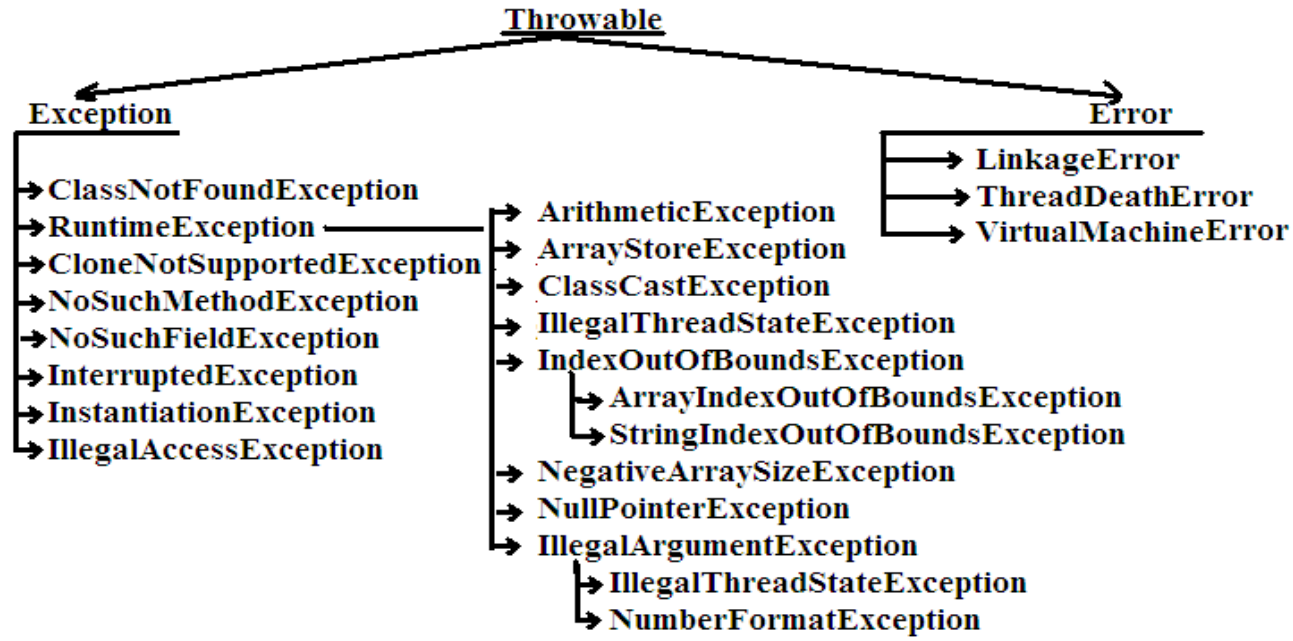
EXCEPTION OBJECT:- It is an object of a predefined or user defined class in java, which is created when interpreter is unable to execute a statement due to incorrect data given by the programmer or user. The

exception object is said to be tossed or thrown from the try block so that he corresponding catch block will hold or catch the exception object in order to provide the solution for the exception.

EXCEPTION HIERARCHY :- Throwable is the root or based class of all exception in java, from it, all other classes are derived. It is divided into 2 types known as Error class and Exception classes, these are derived from Throwable class.

ERROR :- These are the problems for which a java program can not provide any solution because these problems are related to the computer, OS or to the environment of the language, such as virus in OS corruption in JVM, missing of driver files etc. hence a java program can not provide any solution for the Error. Errors are not caused by our program these are due to lack of system resources, errors are non recoverable.

EXCEPTION :- These are the problem for which the java program can provide the solution of the exception because these problems are related to the java programming language. Exception is divided into two type known as Checked exception & Unchecked exception. Exceptions are caused by our program and this are recoverable.



Checked exception :- It is the type of exception which are checked by the compiler during the time of compilation. Hence, if we do not provide the exception handler code then the compiler will not compile the source code. The Throwable class, Exception class & its derived class are falling under checked exception category, except the RuntimeException class and its derived class.

Unchecked exception:- It is the type of exception which are not checked by the compiler during the time of compilation. Hence, if we do not provide the exception handler code then the compiler will compile the source code. The RuntimeException class & its derived class are known as unchecked exception.

Differences between fully checked and partially checked exceptions?

A Checked Exception is said to be fully checked exception if and only if all its child classes also checked.
ex: IOException, InteruptedException

A Checked Exception is said to be partially Checked exception if and only if some of its child classes are Unchecked.
ex: Exception

Note:
The only possible partially checked exceptions in java are
1. Exception
2. Throwable

NESTED EXCEPTION : - In case of exception hierarchy a reference of parent class can always hold the object of any child class.
class A {
public static void main(String args[]) {
try {
try {
}catch(ArithmeticException e) {
}catch(RuntimeException e) { }
}catch(Exception ae) { }
}}
	In case of multiple catch block the inner most catch block must deal with the exception object of the most derived class of the exception hierarchy & the outer most catch block must deal with the exception object of the most super class of the exception hierarchy. But this explanation will not work in case of nested try-catch blocks.
	When exception occurs in try but if there is no suitable catch, then the control execute finally, but due to the absence of the suitable catch the exception still remains in the program and that makes the control jump back to the jvm after finally block.

Various possible combinations of try- catch- finally?
1. Whenever we are writing try block, it is compulsory we should write catch or finally that is 'try' without catch or finally is invalid syntax.
2. Whenever we are writing catch block, it is compulsory we should write try block that is catch without try is invalid.
3. Whenever we are writing finally block compulsory we should write try block. that is finally without try is invalid.
4. In try catch finally, order is important.
5. 'try' with multiple catch blocks Is valid but the order is important, i.e.we should take from child to parent, if we are try to take from parent to child then we will get compile time error.
6. if we are defining to catch blocks for the same exception we will get compile time error.
7. we can define try-catch-finally within the try, with in the catch and within finally blocks. Hence nesting of try-catch-finally is valid.
8. For try-catch-finally curly braces are mandatory.

try{
}catch(X e){
}
	try{
	}catch(X e){
	}catch(X e){
	}
		try{
		}catch(X e){
		}catch(Y e){
		}
			try{
			}catch(Throwable r){
			}catch(RuntimeException e){
			}

try{
}catch(RuntimeException e){
}catch(Throwable r){
	try{
	}
		catch(Exception e){
		}
			finally {
			}

try{
 	
}finally {
 	
}
	try{
	
	}
	System.out.print();
	catch(X e){
	
	}
		try{
		
		}catch(X e){
		
		}
		System.out.print();
		catch(Y e){
		
		}
			try{
			
			}catch(X e){
			
			}
			System.out.print();
			finally{
			
			}

try{
 	
}catch(X e){
 	
}
 	
try{
 	
}finally{
 	
}
	try{
	
	}finally{
	
	}catch(X e){
	
	}
	
	CE: catch without try.
		try{
		
		}catch(X e){
		
		}finally{
		
		}finally{
		
		}
		
		CE: finally without try
			try{
			
			try{
			
			}catch(X e){
			
			}finally{
			
			}
			}catch(Y e){
			
			}
				try{
				
				}catch(X e){
				
				try{
				
				}catch(Y e){
				
				}
				
				}

try {
 	
}catch(X e) {
 	
}finally{
 	
try{
 	
}catch(Y e){
 	
}
}
	try{
	
	try{
	
	}
	}catch(X e){
	
	}
		try
		Sysem.out.print("hi");
		catch(X e){
		}
		try{
		}catch(X e)
		System.out.println("Hello");
		try{
		
		}catch(X e){
		
		}finally
		System.out.printn("bye);

In which situation finally block will not executed ?

There is one situation where the finally block won't be executed if we are using System.exit(0) explicitly then JVM itself will be shutdown and there is no chance of executing finally block.

Note:- If we are not using try-catch it is always abnormal termination if an Exception raised.

Exception in Thread "main" java.lang.ArithmeticException: / by zero

Handled by JVM — type of the Exception — discription

---

throw keyword :- It is used to propagates the exception object. We can use throw keyword at the following situation. We may think of the throw keyword as a special form of the return statement that returns

only exceptions. Once a throw statement executes in a method then the method stops executing, and the exception object is passed back to the previous method in the call stack i.e. the method calling.

1) When we try to propagates the exception from try block to catch block , then we can use it.
2) After handling the exception in a catch block of a method, if we try to propagate the exception object from the catch block of a method to the caller of a method then we can use it. Since it allows us to propagate the exception after handling it, hence it is said that the throw keyword used to rethrow the exception object.
3) It provides information about the exception to the caller during the execution; hence the caller will not generate the incorrect result.

Eg.
```
class A {
 public static void main(String arghs[]) {
    try {
         System.out.println("hi in try");
         ArithmeticException ob = new ArithmeticException();
         throw ob;

     } catch( Exception e) {
         System.out.println("bye in catch");
      }
}};
```

Eg.
```
class Thr2 {
 static void print(String n, int p, int q) {
  try {
     int z = p / q ;
     System.out.println( n + "PRINTED IN LASER " + z);
   }catch(ArithmeticException e){
     System.out.println("error : " + n + " PRINTED IN DMP");
     throw e;
 }}

 public static void main(String args[]) {
   try {
     print("salary ", 49, 7);
     System.out.println("mesg : salary printed in laser");
     print("pension " ,97, 0);
     System.out.println("mesg : pension printed in laser");
   }catch(ArithmeticException ob){
System.out.println("mesg : printed in dmp");
}}};
```

throws keyword :- It is used to propagates the exception. Generally the throws keyword is used after the parameter list of a method along with the Exception class name separated by comma for multiple exception, as given below.

```
public void test(int a, int b, String oper) throws ArithmeticException, InterruptedException {    }
```

When we do not handle the exception by try - catch block in a method (test method), then we can use throws keyword. So that another method calls this method(test method) will handle the exception, hence it propagates the exception from one method to another.

When we mention throws in a method(test), then the other method which will invoke this method(test method) will get an information about the exception that may occur by invoking this method(test method) before the execution and compilation.

```
class  Thrw1  {
  static void test(int p,  int  q) throws ArithmeticException {
     int  z = p/q;
     System.out.println("div : " +z);
  }
  public static void main(String args[]) {
    try {
       test(45,5);
       System.out.println("finish 1");
       test(23,0);
       System.out.println("finish 2");
    }catch(ArithmeticException e){
    System.out.println("catch in main");
   }
}};
```

The throws keyword is used right in the method declaration, whereas the throw method is used within the method body.

Throw	Throws
1. It propagates an exception object from try to catch, or from one method to its caller method. 2. It must be used along with exception object within the method definition. 3. We can only mention one exception object in throw statement. 4. It provides an information about the exception during run time.	1. It propagates exception object only from one method to its caller method. 2. It must be used along with exception class name beside the method declaration. 3. We can mention several exception class name in throws keyword 4. It provides an information about the exception before compilation or run time.

//4th rule of overriding :

```
class    A  {
   void  test() {
System.out.println("test in A");
   }
void show() throws Exception{
System.out.println("show in A");
   }
void  disp()  throws Exception ,
RuntimeException {
System.out.println("disp in A");
   }}
```

```
class  B  extends A {
   void  test() throws  RuntimeException {
System.out.println("overriden method in child class can always 'throws' any unchecked excp^, even if method of parent class may or may not 'throws' any unchecked excp^");
   }

void show() throws Exception,  RuntimeException {
System.out.println("overriden method in child class can 'throws'  any checked Excp^, only if method of parent class has 'throws' a checked excp^");
   }

void  disp()  {
System.out.println("overriden method in child class is not bound to 'throws' any excp^, even if the method of parent class has 'throws' any excp^");
 }}
```

USER DEFINED EXCEPTION (UDE) / CUSTOM EXCEPTION :- When the predefined exception calss is not enough to hold information according to our requirement, then we may create UDE. The UDE must inherit from predefined exception class, so that its object can be thrown by the throw keyword.

```
class  MyExp  extends  Exception {
  String   res;  /* reason of excp */
  String   cn;   /* class name in which excp occurs. */
  String   mn;  /* method name in which excp occurs. */
  int    nn;       /*line number in which excp occurs. */

  MyExp(String  r , String  c, String  m,  int  n) {
        res = r;    cn = c;  mn = m;  nn = n;
  }

   public  String  toString(){
       return  "excpetion  in : "+ mn + ", at line no : "+
nn + ", in class : "+cn +", due to "+res ;
  }}
```

```
class  UserExcp {
 static void  test(int  p) throws MyExp {
    if(p>1000) {
      MyExp ob = new MyExp("more than
1000", "UserExp","test()", 3);
          throw  ob;
     } else
          System.out.println("val " +p);
}
public static void main(String  args[]) {
  try {
    test(456);
    System.out.println("success 1");
    test(3456);
     System.out.println("success 2");
  }catch(MyExp  e) {
      System.out.println(e);
}}};
```

Example

```
class  A {
   A  ob;
   void   initCause(A  aa){    ob = aa;  }
   A   getCause(){   return  ob;   }
};
class  B  extends  A {
  int   x;
  B(){    x= 5;   }
  B(int  p){    x= p;   }
  public  String  toString(){
      return  x+"";
}};
```

```
class  C {
 public static void main(String ar[]) {
   B    b1 = new  B();
   B    b2 = new  B(25);
   b2.initCause(b1);
   B  b3 = (B) b2.getCause();
   System.out.println(b3);
   System.out.println(b2);
}};
```

CHAINED EXCEPTION : - This mechanism allows us to keep one exception object as a data member of another exception object, so that we can propagate multiple exception object to the caller.
The Throwable class has defined following two methods :

public Throwable initCause(Throwable ob) :- This method will accept an object of any exception class and keeps within the exception object that invokes it.

public Throwable getCause() :- This method will return an object that available within the exception object that invokes it.

```
/*
public  class Throwable …… {
    Throwable    ob;
    public Throwable initCause(Throwable ee){
              ob=ee; }
    public Throwable getCause() {
              return  ob;
    }
    ----------
};
/*The following class A will get these members of
Throwable class  */
```

```
class A extends RuntimeException {
String  res;
  A(){  res="first exception";  }
  A(String r) {  res=r;  }
  public  String  toString(){
    return  res ;
 }};
```

```
class  ChainExp {
  static void  test(){
    try {
        A  a1= new A();
        throw  a1;
    }catch(A  aa) {  //aa  is  a1
       A  a2  = new  A("second exception");
        a2.initCause(aa);
        throw  a2;
  }}};
```

```
class  Demo {
public static void main(String  args[]) {
    try {
        ChainExp.test();
    }catch(A  e) { /*e  is a2*/
        A  ob = (A) e.getCause();  /*ob is a1*/
        System.out.println(ob);/* toStirng() of a1 */
System.out.println(e); /* a2.toString()  of  a2 */
}}};
```

Exception enhancement from java 1.7

Until java 1.6 it is recommended to use the finally block, so that you can release the resource hold by the program in finally block or any other clean up job in the finally block, but from java 1.7 two new concept has been introduced to reduce the complexity and to enhance the readability of a program.

1. Try with resource
2. Multi catch block

Try with resource: - In this mechanism we can open a resource directly with try keyword and we don't need to close the resource in the finally block as given below.

```
import java.io.*;
class A{
public static void main(String ar[]){

//Traditional try catch finally
FileInputStream fin=null;
try{
fin=new FileInputStream("a.txt");
int d=0;
while((d=fin.read())!=-1)
System.out.print((char)d);
}catch(Exception e){System.out.println("Error while opening a.txt");}

finally{
try{
fin.close();
}catch(Exception e){}
System.out.println("finally block closing a.txt");
}

//try with resource
try(FileInputStream fi=new FileInputStream("a.txt")){
//fi =new FileInputStream("b.txt");
int d=0;
while((d=fi.read())!=-1)
System.out.print((char)d);
}catch(Exception e){System.out.println("Error while opening a.txt");}
}};
```

When we write a program in try-catch-finally, then the programmer has to close/release the resource/connection in the finally block that increases the length of a program & decreases readability.

Java 1.7 allows us to define an object /resource beside try keyword, the resource which has opened as a part of the try block will be closed automatically once the try block completes normally/abnormally. The programmer needn't to close/release the resource or to provide clean up code.

- We can create any number of resource (object) as a part of the try block, but they must be separated by(;).
- The object or resource we are defining as a part of the try block is by default final. Hence we shouldn't recreate he object within the try block.
- The object or resource we are defining as a part of the try block must have inherited/implemented from java.lang.AutoClosable interface.

Multicatch block : -

Java 1.7 allows us to provide multiple exception class name in a single catch block separated by (|) pipe line. It reduces the length of a program and increases readability but we can't provide different solution for different exception in this mechanism.

```
class Excp{
public static void main(String ar[]){
try{
System.out.println("size : "+ar.length);
int x=Integer.parseInt(ar[0]);
System.out.println("Success 1");
int y=Integer.parseInt(ar[1]);
System.out.println("Success 2");
int z=x/y;
System.out.println("Div : "+z);
}catch(ArithmeticException | ArrayIndexOutOfBoundsException | NumberFormatException
e){System.out.println("Please provide proper values");}
finally{System.out.println("Thank you");}
System.out.println("End of the program");
}};
```

The Exception class name we providing in multi catch block shouldn't be any parent-child class relationship in the exception hierarchy otherwise compilation error will occurs.

The Methods to display Exception Information : Throwable class contains the following methods to display error information.

Method Name	Displays error information in the following format
printStackTrace()	Name of Exception : Description StackTace.
toString()	Name of Exception : Description
getMessage()	Description

```
class Err {
public static void main(String args[]) {
  try {
     int z=12/0;
  }catch(Exception e){
    e.printStackTrace();  ---------> java.lang.ArithmeticException: / by zero
                                        at Err.main(Err.java:4)

    System.out.println("\n"+e);  ---------> java.lang.ArithmeticException: / by zero

    System.out.println("\n"+e.getMessage()); ---------> / by zero
  }
}}
```

Note : If a method has a return type and we have try, catch blocks with return statement, then when we call the method if there is no exception we'll get the return value of try block, but when we call the method if there is an exception then we get the return value of catch block.

If we have a throw statement in try or in catch block, then we should not mention return statement and vice versa, otherwise it will generate semantic error as "Unreachable statement".

```
class Err1 {
static  void  test(int p, int q){
 try {
    System.out.println("in try");
    int z=p/q;
     return;
  }catch(Exception e){
    System.out.println("in catch");
    return;
  }
}
public static void main(String args[]) {
  test(12, 4);
  System.out.println("sucess 1");
  test(124,0);
  System.out.println("sucess 2");
}}
```

Output:

```
in try
sucess 1
in try
in catch
sucess 2
```

```
class Err2 {
static int  test(int p, int q){
 try {
    int z=p/q;
    return 100;
  }catch(Exception e){
   return 200;
  }
}
public static void main(String args[]) {
  int p=test(12, 4);
  System.out.println(p);
  p=test(124,0);
  System.out.println(p);
}}
```

Output:

```
100
200
```

Q. What is final, finally & finalize() ?

final:

1. Final is a modifier applicable for classes methods and variables. If a class declared as final then we can't extend that class. i.e we can't create child class for that class.
2. If a method declared as final then we can't override that method in the child class.
3. If a varaiable declared as final then it will become constant and we can't perform re-assignment for that variable.

finally: finally is a block always associated with try catch to maintain cleanup code.

```
try
{
  // risky code…
}catch( X e){
            Handling code
}finally{
//cleanup code
}
```

finalize(): finalize() is a method which is always invoked by garbage collector just before destroying an object to perform cleanup activities.

Note: finally meant for cleanup activities related to try block. whereas finalize() meant for cleanup activities related to object.