Exception Handling in Java

The **exception handling in java** is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained.

In this page, we will learn about java exception, its type and the difference between checked and unchecked exceptions.

What is exception

Dictionary Meaning: Exception is an abnormal condition.

In java, exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

What is exception handling

Exception Handling is a mechanism to handle runtime errors such as ClassNotFound, IO, SQL, Remote etc.

Advantage of Exception Handling

The core advantage of exception handling is **to maintain the normal flow of the application**. Exception normally disrupts the normal flow of the application that is why we use exception handling. Let's take a scenario:

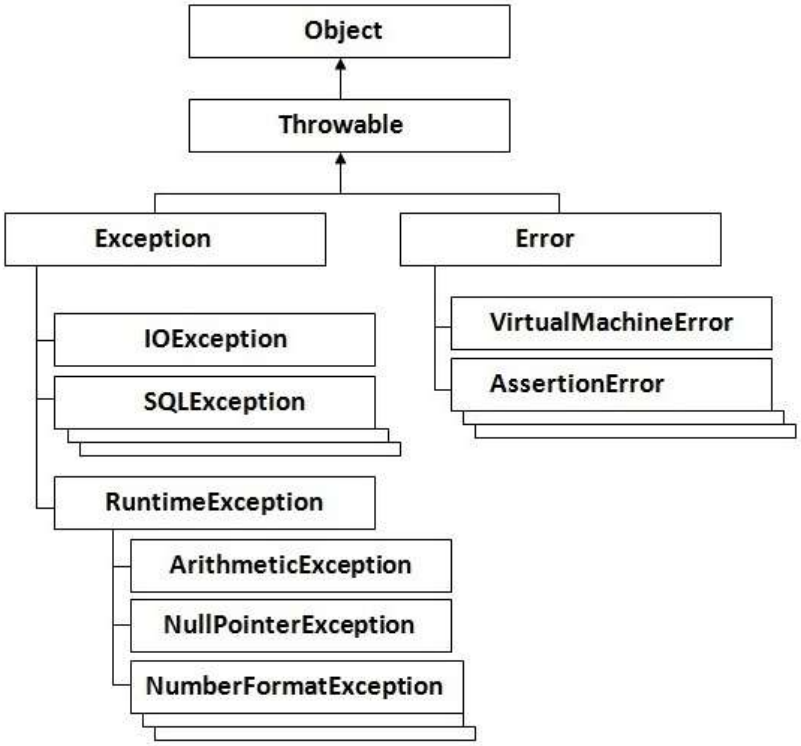
```
1. statement 1;
2. statement 2;
3. statement 3;
4. statement 4;
5. statement 5;//exception occurs
6. statement 6;
7. statement 7;
8. statement 8;
9. statement 9;
10. statement 10;
```

Suppose there is 10 statements in your program and there occurs an exception at statement 5, rest of the code will not be executed i.e. statement 6 to 10 will not run. If we perform exception handling, rest of the statement will be executed. That is why we use exception handling in java.

Questions

- What is the difference between checked and unchecked exceptions ?
- What happens behind the code int data=50/0; ?
- Why use multiple catch block ?
- Is there any possibility when finally block is not executed ?
- What is exception propagation ?
- What is the difference between throw and throws keyword ?
- What are the 4 rules for using exception handling with method overriding ?

Hierarchy of Java Exception classes



Types of Exception

There are mainly two types of exceptions: checked and unchecked where error is considered as unchecked exception. The sun microsystem says there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

Difference between checked and unchecked exceptions

1) Checked Exception

The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions e.g.IOException, SQLException etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes that extend RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time rather they are checked at runtime.

3) Error

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Common scenarios where exceptions may occur

There are given some scenarios where unchecked exceptions can occur. They are as follows:

1) Scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

```
1. int a=50/0;//ArithmeticException
```

2) Scenario where NullPointerException occurs

If we have null value in any variable, performing any operation by the variable occurs an NullPointerException.

```
1. String s=null;
2. System.out.println(s.length());//NullPointerException
```

3) Scenario where NumberFormatException occurs

The wrong formatting of any value, may occur NumberFormatException. Suppose I have a string variable that have characters, converting this variable into digit will occur NumberFormatException.

```
1. String s="abc";
2. int i=Integer.parseInt(s);//NumberFormatException
```

4) Scenario where ArrayIndexOutOfBoundsException occurs

If you are inserting any value in the wrong index, it would result ArrayIndexOutOfBoundsException as shown below:

```
1. int a[]=new int[5];
2. a[10]=50; //ArrayIndexOutOfBoundsException
```

Java Exception Handling Keywords

There are 5 keywords used in java exception handling.

1. try
2. catch
3. finally
4. throw
5. throws

Java try-catch

Java try block

Java try block is used to enclose the code that might throw an exception. It must be used within the method.

Java try block must be followed by either catch or finally block.

Syntax of java try-catch

```
try{
//code that may throw exception
}catch(Exception_class_Name ref){}
```

Syntax of try-finally block

```
try{
//code that may throw exception
} finally{}
```

Java catch block

Java catch block is used to handle the Exception. It must be used after the try block only.

You can use multiple catch block with a single try.

Problem without exception handling

Let's try to understand the problem if we don't use try-catch block.

```
public class Testtrycatch1{
  public static void main(String args[]){
      int data=50/0;//may throw exception
      System.out.println("rest of the code...");
}
}
```

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
```

As displayed in the above example, rest of the code is not executed (in such case, rest of the code... statement is not printed).

There can be 100 lines of code after exception. So all the code after exception will not be executed.

Solution by exception handling

Let's see the solution of above problem by java try-catch block.

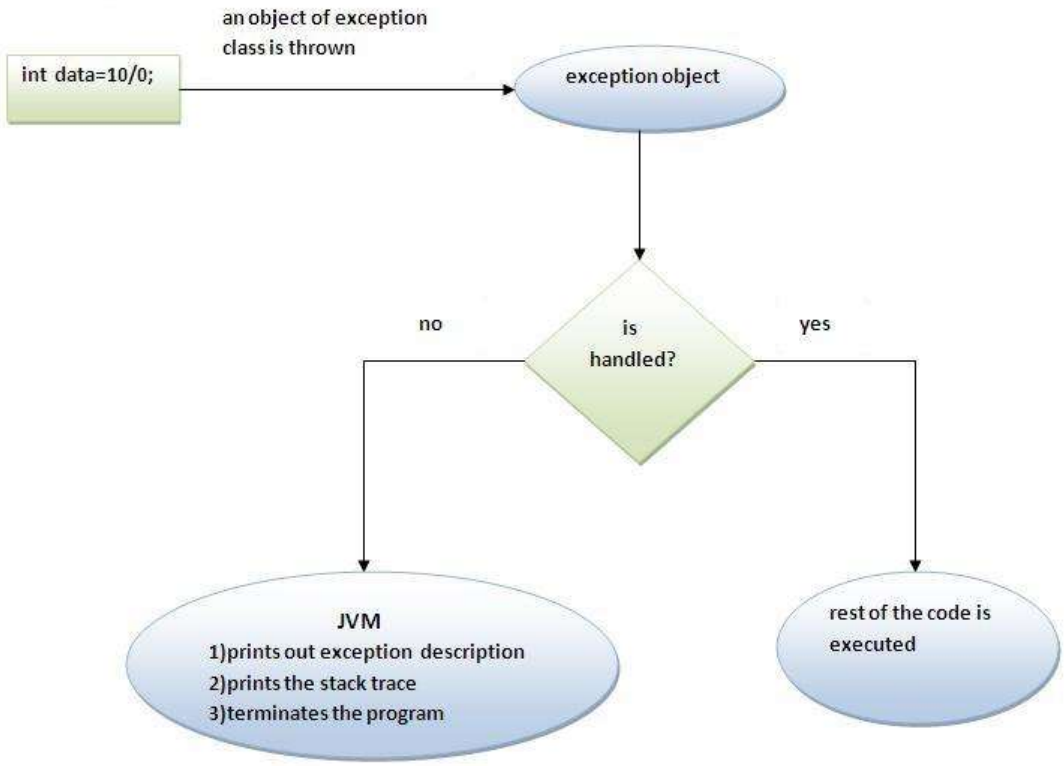
```
1. public class Testtrycatch2{
2.   public static void main(String args[]){
3.    try{
4.       int data=50/0;
5.    }catch(ArithmeticException e){System.out.println(e);}
6.    System.out.println("rest of the code...");
7. }
8. }
```

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...
```

Now, as displayed in the above example, rest of the code is executed i.e. rest of the code... statement is printed.

Internal working of java try-catch block



The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

- Prints out exception description.
- Prints the stack trace (Hierarchy of methods where the exception occurred).
- Causes the program to terminate.

But if exception is handled by the application programmer, normal flow of the application is maintained i.e. rest of the code is executed.

Java catch multiple exceptions

Java Multi catch block

If you have to perform different tasks at the occurrence of different Exceptions, use java multi catch block.

Let's see a simple example of java multi-catch block.

```
1. public class TestMultipleCatchBlock{
2.   public static void main(String args[]){
3.    try{
4.     int a[]=new int[5];
5.     a[5]=30/0;
6.    }
7.    catch(ArithmeticException e){System.out.println("task1 is completed");}
8.    catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed")
   ;}
9.    catch(Exception e){System.out.println("common task completed");}
10.
11.   System.out.println("rest of the code...");
12. }
13. }
```

```
Output:task1 completed
       rest of the code...
```

Rule: At a time only one Exception is occured and at a time only one catch block is executed.

Rule: All catch blocks must be ordered from most specific to most general i.e. catch for ArithmeticException must come before catch for Exception .

```
1. class TestMultipleCatchBlock1{
2.   public static void main(String args[]){
3.    try{
4.     int a[]=new int[5];
5.     a[5]=30/0;
6.    }
7.    catch(Exception e){System.out.println("common task completed");}
8.    catch(ArithmeticException e){System.out.println("task1 is completed");}
9.    catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed")
   ;}
10.   System.out.println("rest of the code...");
11. }
12. }
```

Output:

```
Compile-time error
```

Java Nested try block

The try block within a try block is known as nested try block in java.

Why use nested try block

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

Syntax:

```
....
try
{
    statement 1;
    statement 2;
    try
    {
        statement 1;
        statement 2;
    }
    catch(Exception e)
    {
    }
}
catch(Exception e)
{
}
....
```

Java nested try example

Let's see a simple example of java nested try block.

```
class Excep6{
 public static void main(String args[]){
  try{
    try{
     System.out.println("going to divide");
     int b =39/0;
    }catch(ArithmeticException e){System.out.println(e);}
    try{
    int a[]=new int[5];
    a[5]=4;
    }catch(ArrayIndexOutOfBoundsException e){System.out.println(e);}
    System.out.println("other statement);
  }catch(Exception e){System.out.println("handeled");}
  System.out.println("normal flow..");
 }
```

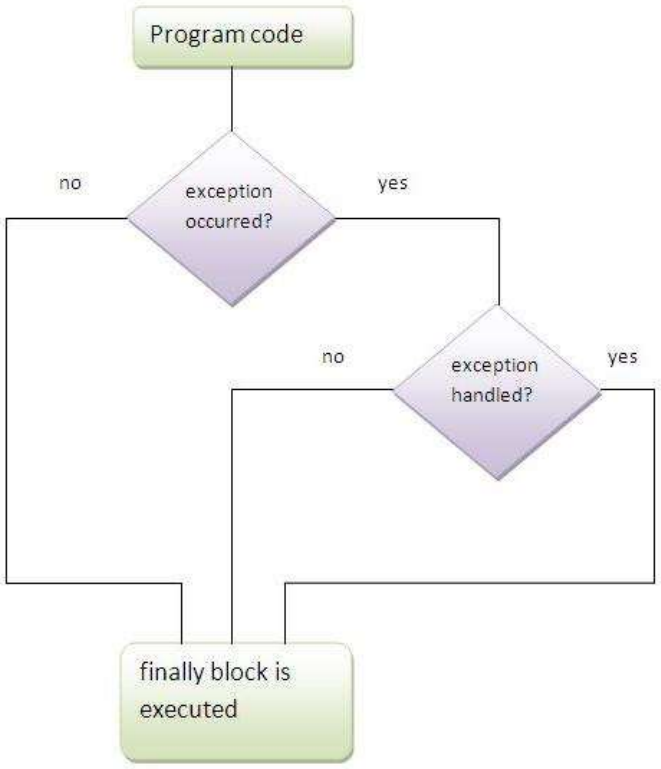
```
16. }
```

Java finally block

Java finally block is a block that is used *to execute important code* such as closing connection, stream etc.

Java finally block is always executed whether exception is handled or not.

Java finally block must be followed by try or catch block.



Note: If you don't handle exception, before terminating the program, JVM executes finally block(if any).

Why use java finally

- Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

Usage of Java finally

Let's see the different cases where java finally block can be used.

Case 1

Let's see the java finally example where **exception doesn't occur**.

```
1. class TestFinallyBlock{
2.   public static void main(String args[]){
3.   try{
4.    int data=25/5;
5.    System.out.println(data);
6.   }
7.   catch(NullPointerException e){System.out.println(e);}
8.   finally{System.out.println("finally block is always executed");}
9.   System.out.println("rest of the code...");
10.  }
11. }
12. Output:5
        finally block is always executed
        rest of the code...
```

Case 2

Let's see the java finally example where **exception occurs and not handled**.

```
1. class TestFinallyBlock1{
2.   public static void main(String args[]){
3.   try{
4.    int data=25/0;
5.    System.out.println(data);
6.   }
7.   catch(NullPointerException e){System.out.println(e);}
8.   finally{System.out.println("finally block is always executed");}
9.   System.out.println("rest of the code...");
10.  }
11. }
Output:finally block is always executed
       Exception in thread main java.lang.ArithmeticException:/ by zero
```

Case 3

Let's see the java finally example where **exception occurs and handled**.

```
1. public class TestFinallyBlock2{
2.   public static void main(String args[]){
3.   try{
4.    int data=25/0;
5.    System.out.println(data);
6.   }
7.   catch(ArithmeticException e){System.out.println(e);}
8.   finally{System.out.println("finally block is always executed");}
9.   System.out.println("rest of the code...");
10.  }
11. }
```

```
Output:Exception in thread main java.lang.ArithmeticException:/ by zero
       finally block is always executed
       rest of the code...
```

Rule: For each try block there can be zero or more catch blocks, but only one finally block.

Note: The finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).

Java throw exception

Java throw keyword

The Java throw keyword is used to explicitly throw an exception.

We can throw either checked or uncheked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception. We will see custom exceptions later.

The syntax of java throw keyword is given below.

```
1. throw exception;
```

Let's see the example of throw IOException.

```
1. throw new IOException("sorry device error);
```

java throw keyword example

In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

```
1.  public class TestThrow1{
2.    static void validate(int age){
3.      if(age<18)
4.       throw new ArithmeticException("not valid");
5.      else
6.       System.out.println("welcome to vote");
7.    }
8.    public static void main(String args[]){
9.       validate(13);
10.      System.out.println("rest of the code...");
11.  }
12. }
```

Output:

```
Exception in thread main java.lang.ArithmeticException:not valid
```

Java Exception propagation

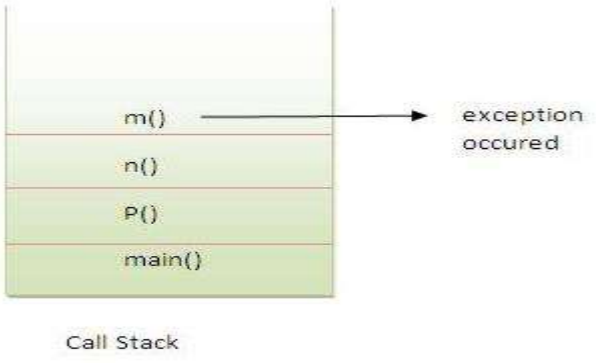
An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method, If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack. This is called exception propagation.

Rule: By default Unchecked Exceptions are forwarded in calling chain (propagated).

Program of Exception Propagation

```
1.  class TestExceptionPropagation1{
2.    void m(){
3.      int data=50/0;
4.    }
5.    void n(){
6.      m();
7.    }
8.    void p(){
9.     try{
10.     n();
11.    }catch(Exception e){System.out.println("exception handled");}
12.   }
13.   public static void main(String args[]){
14.    TestExceptionPropagation1 obj=new TestExceptionPropagation1();
15.    obj.p();
16.    System.out.println("normal flow...");
17.   }
18. }
```

```
Output:exception handled
       normal flow...
```



In the above example exception occurs in m() method where it is not handled,so it is propagated to previous n() method where it is not handled, again it is propagated to p() method where exception is handled.

Exception can be handled in any method in call stack either in main() method,p() method,n() method or m() method.

Rule: By default, Checked Exceptions are not forwarded in calling chain (propagated).

Program which describes that checked exceptions are not propagated

```
1.  class TestExceptionPropagation2{
2.    void m(){
3.      throw new java.io.IOException("device error");//checked exception
4.    }
5.    void n(){
6.      m();
7.    }
8.    void p(){
9.     try{
10.     n();
11.    }catch(Exception e){System.out.println("exception handeled");}
12.   }
13.   public static void main(String args[]){
14.    TestExceptionPropagation2 obj=new TestExceptionPropagation2();
15.    obj.p();
16.    System.out.println("normal flow");
17.   }
18. }
```

```
Output:Compile Time Error
```

Java throws keyword

In java if a code written within a method throws an exception, there can be two cases. Either the code is being enclosed by try block and exception handling is implemented in the same method, or the method can throws the exception to the calling method simply.

This is what 'throws' does in exception handling, it throws the exception to immediate calling method in the hierarchy. The throws keyword appears at the end of a method's signature.

Exception Handling is mainly used to handle the checked exceptions.

Syntax of java throws

```
1. return_type method_name() throws exception_class_name{
2. //method code
3. }
```

Advantage of Java throws keyword

Now Checked Exception can be propagated (forwarded in call stack).

It provides information to the caller of the method about the exception.

Java throws example

Let's see the example of java throws clause which describes that checked exceptions can be propagated by throws keyword.

```
1. import java.io.IOException;
2. class Testthrows1{
3.   void m()throws IOException{
4.     throw new IOException("device error");//checked exception
5.   }
6.   void n()throws IOException{
7.     m();
8.   }
9.   void p(){
10.   try{
11.    n();
12.   }catch(Exception e){System.out.println("exception handled");}
13.  }
14.  public static void main(String args[]){
15.   Testthrows1 obj=new Testthrows1();
16.   obj.p();
17.   System.out.println("normal flow...");
18.  }
19. }
```

Output:

```
exception handled
normal flow...
```

The calling method itself can use a throws keyword and forward the exception to handle by its own calling method.. so on . This can be forwarder up in the hierarchy, if all calling methods are using throws and no more calling method is available to handle the exception that the exceptions are ultimately be handled by main method. And if the main method is also using an throws itself the exception will be handled by JVM itself.

Rule: If you are calling a method that declares an exception, you must either caught or declare the exception.

There are two cases:

1. **Case1:**You caught the exception i.e. handle the exception using try/catch.
2. **Case2:**You declare the exception i.e. specifying throws with the method.

Case1: You handle the exception

- In case you handle the exception, the code will be executed fine whether exception occurs during the program or not.

```
1. import java.io.*;
2. class M{
3.  void method()throws IOException{
4.   throw new IOException("device error");
5.  }
6. }
7. public class Testthrows2{
8.    public static void main(String args[]){
9.     try{
10.     M m=new M();
11.     m.method();
12.    }catch(Exception e){System.out.println("exception handled");}
13.
14.    System.out.println("normal flow...");
15.  }
16. }
```

```
Output:exception handled
       normal flow...
```

Case2: You declare the exception

- A)In case you declare the exception, if exception does not occur, the code will be executed fine.
- B)In case you declare the exception if exception occures, an exception will be thrown at runtime because throws does not handle the exception.

A)Program if exception does not occur

```
1. import java.io.*;
2. class M{
3.  void method()throws IOException{
4.   System.out.println("device operation performed");
5.  }
6. }
7. class Testthrows3{
8.    public static void main(String args[])throws IOException{//declare exception
9.      M m=new M();
10.     m.method();
11.
12.    System.out.println("normal flow...");
13.  }
14. }
```

```
Output:device operation performed
       normal flow...
```

B)Program if exception occurs

```
1. import java.io.*;
2. class M{
3.  void method()throws IOException{
4.   throw new IOException("device error");
5.  }
6. }
7. class Testthrows4{
8.    public static void main(String args[])throws IOException{//declare exception
9.      M m=new M();
10.     m.method();
11.
12.    System.out.println("normal flow...");
13.  }
14. }
```

```
Output:Runtime Exception
```

Important notes about Exception Handling in Java

1) A 'try' block must be followed by a catch or a finally block.
2) A 'try' block can contain any number of catch block, but these catch blocks must represent distinct Exception Classe that too in narrow to broad order.
3) A 'catch' or 'finally' block can not be used without a 'try' block.
4) There can not be any type of code in between a try, catch and finally blocks.

Difference between throw and throws in Java

There are many differences between throw and throws keywords. A list of differences between throw and throws are given below:

No.	throw	throws
1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2)	Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
3)	Throw is followed by an instance.	Throws is followed by class.
4)	Throw is used within the method.	Throws is used with the method signature.
5)	You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

Java throw example

```
1. void m(){
2. throw new ArithmeticException("sorry");
3. }
```

Java throws example

```
1. void m()throws ArithmeticException{
2. //method code
3. }
```

Java throw and throws example

```
1. void m()throws ArithmeticException{
2. throw new ArithmeticException("sorry");
3. }
```

Difference between final, finally and finalize

There are many differences between final, finally and finalize. A list of differences between final, finally and finalize are given below:

No.	final	finally	finalize
1)	Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed.	Finally is used to place important code, it will be executed whether exception is handled or not.	Finalize is used to perform clean up processing just before object is garbage collected.
2)	Final is a keyword.	Finally is a block.	Finalize is a method.

Java final example

```
1. class FinalExample{
2. public static void main(String[] args){
3. final int x=100;
4. x=200;//Compile Time Error
5. }}
```

Java finally example

```
1. class FinallyExample{
2. public static void main(String[] args){
3. try{
4. int x=300;
5. }catch(Exception e){System.out.println(e);}
6. finally{System.out.println("finally block is executed");}
7. }}
```

Java finalize example

```
1. class FinalizeExample{
2. public void finalize(){System.out.println("finalize called");}
3. public static void main(String[] args){
4. FinalizeExample f1=new FinalizeExample();
5. FinalizeExample f2=new FinalizeExample();
6. f1=null;
7. f2=null;
8. System.gc();
9. }}
```

ExceptionHandling with MethodOverriding in Java

There are many rules if we talk about methodoverriding with exception handling.

The Rules are as follows:

- **If the superclass method does not declare an exception**
  - If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception.
- **If the superclass method declares an exception**
  - If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

If the superclass method does not declare an exception

1) Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception.

```
1. import java.io.*;
2. class Parent{
3.   void msg(){System.out.println("parent");}
4. }
5.
6. class TestExceptionChild extends Parent{
7.   void msg()throws IOException{
8.     System.out.println("TestExceptionChild");
9.   }
10.  public static void main(String args[]){
11.   Parent p=new TestExceptionChild();
12.   p.msg();
13.  }
14. }
```

```
Output:Compile Time Error
```

2) Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but can declare unchecked exception.

```
1. import java.io.*;
2. class Parent{
3.   void msg(){System.out.println("parent");}
4. }
5.
6. class TestExceptionChild1 extends Parent{
7.   void msg()throws ArithmeticException{
8.     System.out.println("child");
9.   }
10.  public static void main(String args[]){
11.   Parent p=new TestExceptionChild1();
12.   p.msg();
13.  }
14. }
```

```
Output:child
```

If the superclass method declares an exception

1) Rule: If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

Example in case subclass overridden method declares parent exception

```
1. import java.io.*;
2. class Parent{
3.   void msg()throws ArithmeticException{System.out.println("parent");}
4. }
5.
6. class TestExceptionChild2 extends Parent{
7.   void msg()throws Exception{System.out.println("child");}
8.
9.   public static void main(String args[]){
10.   Parent p=new TestExceptionChild2();
11.   try{
12.   p.msg();
13.   }catch(Exception e){}
14.  }
15. }
```

```
Output:Compile Time Error
```

Example in case subclass overridden method declares same exception

```
1. import java.io.*;
2. class Parent{
3.   void msg()throws Exception{System.out.println("parent");}
4. }
5.
6. class TestExceptionChild3 extends Parent{
7.   void msg()throws Exception{System.out.println("child");}
8.
9.   public static void main(String args[]){
10.   Parent p=new TestExceptionChild3();
11.   try{
12.   p.msg();
13.   }catch(Exception e){}
14.  }
15. }
```

```
Output:child
```

Example in case subclass overridden method declares subclass exception

```
1. import java.io.*;
2. class Parent{
3.   void msg()throws Exception{System.out.println("parent");}
4. }
5.
6. class TestExceptionChild4 extends Parent{
7.   void msg()throws ArithmeticException{System.out.println("child");}
8.
9.   public static void main(String args[]){
10.   Parent p=new TestExceptionChild4();
11.   try{
12.   p.msg();
13.   }catch(Exception e){}
14.  }
15. }
```

```
Output:child
```

Example in case subclass overridden method declares no exception

```
1. import java.io.*;
2. class Parent{
3.   void msg()throws Exception{System.out.println("parent");}
4. }
5.
6. class TestExceptionChild5 extends Parent{
7.   void msg(){System.out.println("child");}
8.
9.   public static void main(String args[]){
10.   Parent p=new TestExceptionChild5();
11.   try{
12.   p.msg();
13.   }catch(Exception e){}
14.  }
15.}
```

```
Output:child
```

Java Custom Exception

If you are creating your own Exception that is known as custom exception or user-defined exception. Java custom exceptions are used to customize the exception according to user need.

By the help of custom exception, you can have your own exception and message.

Let's see a simple example of java custom exception.

```
1. class InvalidAgeException extends Exception{
2.  InvalidAgeException(String s){
3.   super(s);
4.  }
5. }
```

```
1. class TestCustomException1{
2.
3.    static void validate(int age)throws InvalidAgeException{
4.      if(age<18)
5.       throw new InvalidAgeException("not valid");
6.      else
7.       System.out.println("welcome to vote");
8.    }
9.
10.   public static void main(String args[]){
11.     try{
12.     validate(13);
13.     }catch(Exception m){System.out.println("Exception occured: "+m);}
14.
15.     System.out.println("rest of the code...");
16.  }
17. }
```

```
Output:Exception occured: InvalidAgeException:not valid
       rest of the code...
```

User defined exceptions in java also known as **Custom exceptions**.

Most of the times when we are developing an application in java, we often feel a need to create and throw our own exceptions. These exceptions are known as **User defined or Custom exceptions.**

Example of User defined exception in Java

```
class MyException extends Exception{
   String str1;
   MyException(String str2) {
     str1=str2;
   }
   public String toString(){
     return ("Output String = "+str1) ;
   }
}

class CustomException{
   public static void main(String args[]){
     try{
        throw new MyException("Custom");
        // I'm throwing user defined custom exception above
     }
     catch(MyException exp){
        System.out.println("Hi this is my catch block") ;
        System.out.println(exp) ;
     }
   }
}
```

Output:

```
Hi this is my catch block
Output String = Custom
```

Key-points from above example:
You can see that while throwing custom exception, a string is passed in parenthesis (throw new MyException("Custom");). That's the reason to have a parametric constructor (with a String parameter) in custom exception class.

Notes:

- User defined exception needs to inherit (extends) Exception class in order to act as an exception.
- throw keyword is used to throw such exceptions.

Another example to modify the error message of Exception Class

Part 1: Created own exception class- **MyException** by inheriting the parent class **Exception** then defined a parametric constructor of the class with a String parameter. In the constructor called super(), super refers to the super class { My class has inherited Exception class so Exception class is my superclass}. In this way the system generated message is modified by own created message.

```
public class MyException extends Exception
{
   public MyException(String mymsg)
   {
      super(mymsg);
   }
}
```

Part2:

```
public class ExceptionSample
{
   public static void main(String args[]) throws Exception
   {
      ExceptionSample es = new ExceptionSample();
      es.displayMymsg();
   }
   public void displayMymsg() throws MyException
   {
      for(int j=8;j>0;j--)
      {
        System.out.println("j= "+j);
        if(j==7)
        {
           throw new MyException("This is my own Custom Message");
        }
      }
   }
}
```

Output:

```
j = 8
j = 7
Exception in thread "main" MyException: This is my own Custom Message
at ExceptionSample.displayMymsg( ExceptionSample.java.19)
```

User defined Exception subclass

You can also create your own exception sub class simply by extending java **Exception** class. You can define a constructor for your Exception sub class (not compulsory) and you can override the **toString()** function to display your customized message on catch.

```
class MyException extends Exception
{
 private int ex;
 MyException(int a)
 {
  ex=a; }
 public String toString()
 {
  return "MyException[" + ex +"] is less than zero";
 }
}

class Test
{
 static void sum(int a,int b) throws MyException
 {
  if(a<0)
  {
   throw new MyException(a);
  }
  else
  {
   System.out.println(a+b);
  }
 }

 public static void main(String[] args)
 {
  try
  {
   sum(-10, 10);  }
  catch(MyException me)
  {
   System.out.println(me);
  }
 }
}
```

Points to Remember

1. Extend the Exception class to create your own ecxception class.
2. You don't have to implement anything inside it, no methods are required.
3. You can have a Constructor if you want.
4. You can override the toString() function, to display customized message.

Chained Exception

Chained Exception was added to Java in JDK 1.4. This feature allow you to relate one exception with another exception, i.e one exception describes cause of another exception. For example, consider a situation in which a method throws an **ArithmeticException** because of an attempt to divide by zero but the actual cause of exception was an I/O error which caused the divisor to be zero. The method will throw only **ArithmeticException** to the caller. So the caller would not come to know about the actual cause of exception. Chained Exception is used in such type of situations.

Two new constructors and two methods were added to **Throwable** class to support chained exception.

1. **Throwable**(*Throwable cause*)
2. **Throwable**(*String str*, *Throwable cause*)

In the first form, the paramter **cause** specifies the actual cause of exception. In the second form, it allows us to add an exception description in string form with the actual cause of exception.

getCause() and **initCause()** are the two methods added to **Throwable** class.

- **getCause()** method returns the actual cause associated with current exception.
- **initCause()** set an underlying cause(exception) with invoking exception.

Example

```
import java.io.IOException;
public class ChainedException
 {
  public static void divide(int a, int b)
  {
   if(b==0)
   {
    ArithmeticException ae = new ArithmeticException("top layer");
    ae.initCause( new IOException("cause") );
    throw ae;
   }
   else
   {
    System.out.println(a/b);
   }
  }

 public static void main(String[] args)
 {
  try {
   divide(5, 0);
  }
  catch(ArithmeticException ae) {
   System.out.println( "caught : " +ae);
   System.out.println("actual cause: "+ae.getCause());
  }
 }
}
```

output

```
caught:java.lang.ArithmeticException: top layer
actual cause: java.io.IOException: cause
```

This example shows how to create user defined exception by extending Exception Class.

```
class WrongInputException extends Exception {
  WrongInputException(String s) {
    super(s);
  }
}
class Input {
  void method() throws WrongInputException {
    throw new WrongInputException("Wrong input");
  }
}
class TestInput {
  public static void main(String[] args){
    try {
      new Input().method();
    }
          catch(WrongInputException wie) {
      System.out.println(wie.getMessage());
    }
  }
}
```

Result:

The above code sample will produce the following result.

Wrong input