Exception Handling

Exception handling techniques: try…catch, throw, throws, finally block, user defined Exception.

Exception handling

Exception handling is one of the most important feature of java programming that allows us to handle the runtime errors caused by exceptions.

What is an exception?

An Exception is an unwanted event that interrupts the normal flow of the program. When an exception occurs program execution gets terminated. In such cases we get a system generated error message.

The good thing about exceptions is that they can be handled in Java. By handling the exceptions we can provide a meaningful message to the user about the issue rather than a system generated message, which may not be understandable to a user.

Why an exception occurs?

There can be several reasons that can cause a program to throw exception. For example: Opening a non-existing file in your program, Network connection problem, bad input data provided by user etc.

Exception Handling

If an exception occurs, which has not been handled by programmer then program execution gets terminated and a system generated error message is shown to the user. For example look at the system generated exception below:

An exception generated by the system is given below

```
Exception in thread "main" java.lang.ArithmeticException: / by zero at
ExceptionDemo.main(ExceptionDemo.java:5)
 ExceptionDemo : The class name  main
: The method name
 ExceptionDemo.java : The filename  java:5
: Line number
```

This message is not user friendly so a user will not be able to understand what went wrong. In order to let them know the reason in simple language, we handle exceptions. We handle such conditions and then prints a user friendly warning message to user, which lets them correct the error as most of the time exception occurs due to bad data provided by user.

Advantage of exception handling

Exception handling ensures that the flow of the program doesn't break when an exception occurs. For example, if a program has bunch of statements and an

exception occurs mid way after executing certain statements then the statements after the exception will not execute and the program will terminate abruptly.

By handling we make sure that all the statements execute and the flow of program doesn't break.

The core advantage of exception handling is **to maintain the normal flow of the application**. An 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 are 10 statements in your program and there occurs an exception at statement 5, the rest of the code will not be executed i.e. statement 6 to 10 will not be executed. If we perform exception handling, the rest of the statement will be executed. That is why we use exception handling in Java.

Difference between error and exception

Errors indicate that something severe enough has gone wrong, the application should crash rather than try to handle the error.

Exceptions are events that occurs in the code. A programmer can handle such conditions and take necessary corrective actions.

Few examples:

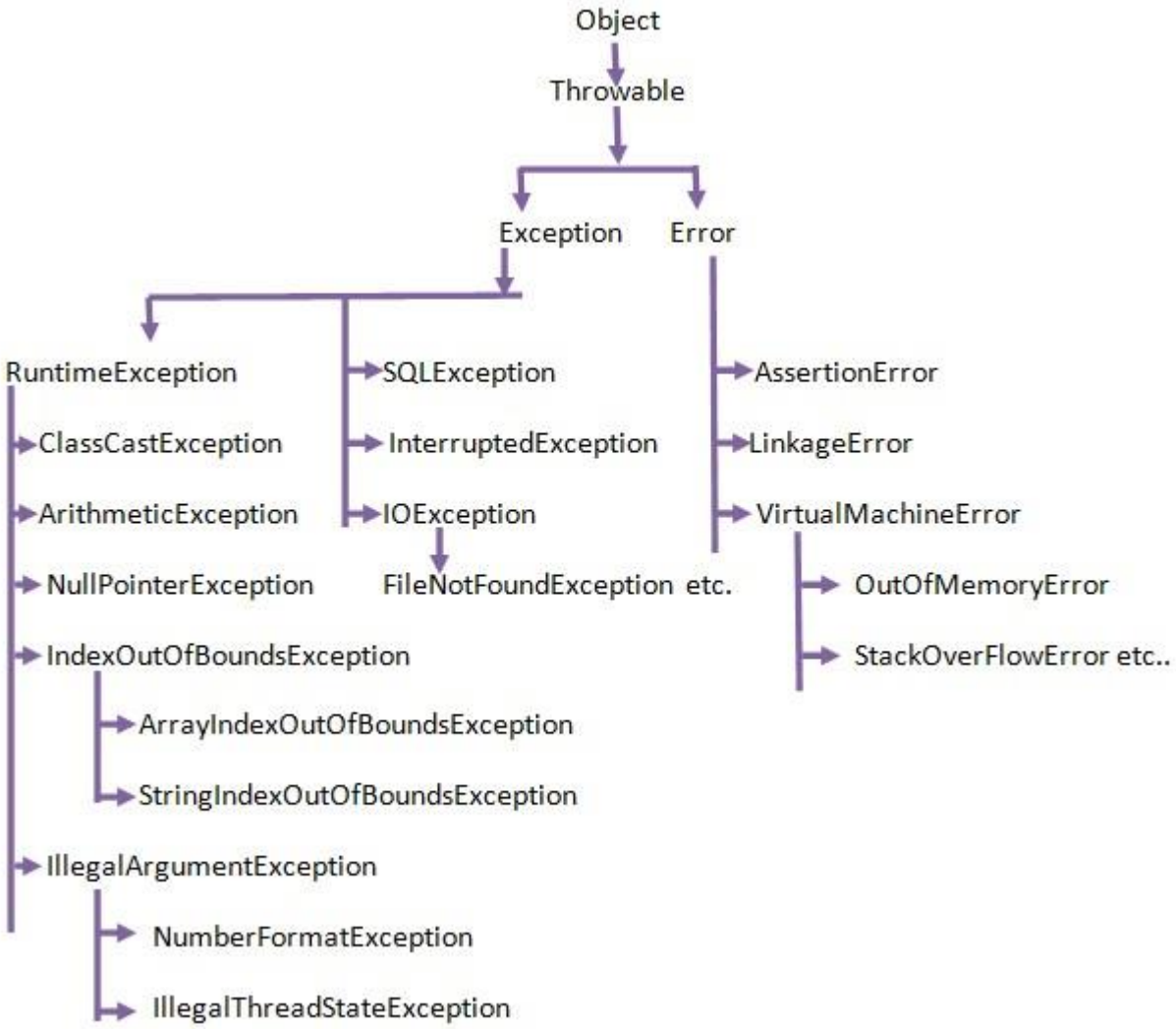
NullPointerException – When you try to use a reference that points to null.

ArithmeticException – When bad data is provided by user, for example, when you try to divide a number by zero this exception occurs because dividing a number by zero is undefined.

ArrayIndexOutOfBoundsException – When you try to access the elements of an array out of its bounds, for example array size is 5 (which means it has five elements) and you are trying to access the 10th element.

Hierarchy of Java Exception classes

The java.lang.Throwable class is the root class of Java Exception hierarchy which is inherited by two subclasses: Exception and Error. A hierarchy of Java Exception classes are given below:



Types of exceptions

There are two types of exceptions in Java:
1)Checked exceptions
2)Unchecked exceptions

Checked exceptions

All exceptions other than Runtime Exceptions are known as Checked exceptions as the compiler checks them during compilation to see whether the programmer has handled them or not. OR

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

If these exceptions are not handled/declared in the program, you will get compilation error.

For example, SQLException, IOException, ClassNotFoundException etc.

Unchecked Exceptions

Runtime Exceptions are also known as Unchecked Exceptions. These exceptions are not checked at compile-time so compiler does not check whether the programmer has handled them or not but it's the responsibility of the programmer to handle these exceptions and provide a safe exit.

OR

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

For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.

Compiler will never force you to catch such exception or force you to declare it in the method using throws keyword.

Error

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

Java Exception Keywords

There are 5 keywords which are used in handling exceptions in Java.

Keyword	**Description**
try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

Try Catch in Java

Try block

The try block contains set of statements where an exception can occur. A try block is always followed by a catch block, which handles the exception that occurs in associated try block. A try block must be followed by catch blocks or finally block or both.

Syntax of try block

```
try{
   //statements that may cause an exception
}
```

While writing a program, if you think that certain statements in a program can throw a exception, enclosed them in try block and handle that exception

Catch block

A catch block is where you handle the exceptions, this block must follow the try block. A single try block can have several catch blocks associated with it.

You can catch different exceptions in different catch blocks. When an exception occurs in try block, the corresponding catch block that handles that particular exception executes. For example if an arithmetic exception occurs in try block then the statements enclosed in catch block for arithmetic exception executes.

Syntax of try catch in java

```
try
{
    //statements that may cause an exception
}
catch (exception(type) e(object))
{
    //error handling code
}
```

Example: try catch block

If an exception occurs in try block then the control of execution is passed to the corresponding catch block.
A single try block can have multiple catch blocks associated with it, you should place the catch blocks in such a way that the generic exception handler catch block is at the last(see in the example below).

The generic exception handler can handle all the exceptions but you should place is at the end, if you place it at the before all the catch blocks then it will display the generic message. You always want to give the user a meaningful message for each type of exception rather then a generic message.

```
class Example1 {    public static void
main(String args[]) {      int num1,
num2;      try {
       /* We suspect that this block of statement can throw
* exception so we handled it by placing these statements
* inside try and handled the exception in catch block
```

```
         */
        num1 = 0;
        num2 = 62 / num1;
        System.out.println(num2);
        System.out.println("Hey I'm at the end of try block");
      }
      catch (ArithmeticException e) {
        /* This block will only execute if any Arithmetic exception
         * occurs in try block
         */
        System.out.println("You should not divide a number by zero");
      }
      catch (Exception e) {
        /* This is a generic Exception handler which means it can handle
         * all the exceptions. This will execute if the exception is not
         * handled by previous catch blocks.
         */
        System.out.println("Exception occurred");
      }
      System.out.println("I'm out of try-catch block in Java.");
    }
}
```

Output:

```
You should not divide a number by zero
I'm out of try-catch block in Java.
```

Java Exception Handling Example

Let's see an example of Java Exception Handling where we using a try-catch statement to handle the exception.

```
public class JavaExceptionExample{
  public static void main(String args[]){
   try{
      //code that may raise exception
      int data=100/0;
   }catch(ArithmeticException e){System.out.println(e);}
   //rest code of the program
   System.out.println("rest of the code...");
  }
}
```

Output:

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

In the above example, 100/0 raises an ArithmeticException which is handled by a try-catch block.

Java catch multiple exceptions::Java Multi-catch block

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

Points to remember

- At a time only one exception occurs and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.

Example

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

```
public class MultipleCatchBlock1 {
    public static void main(String[] args) {

try{
            int a[]=new int[5];
a[5]=30/0;
            }
            catch(ArithmeticException e)
              {
               System.out.println("Arithmetic Exception occurs");
              }
            catch(ArrayIndexOutOfBoundsException e)
              {
               System.out.println("ArrayIndexOutOfBounds Exception occurs");
              }
            catch(Exception e)
           {
               System.out.println("Parent Exception occurs");
              }
            System.out.println("rest of the code");
}
}
```

Output:

```
Arithmetic Exception occurs rest
of the code
```

Example

```
public class MultipleCatchBlock2 {
public static void main(String[] args) {
try{
            int a[]=new int[5];

            System.out.println(a[10]);
            }
            catch(ArithmeticException e)
              {
               System.out.println("Arithmetic Exception occurs");
              }
            catch(ArrayIndexOutOfBoundsException e)
              {
               System.out.println("ArrayIndexOutOfBounds Exception occurs");
              }
            catch(Exception e)
              {
               System.out.println("Parent Exception occurs");
              }
```

```
            System.out.println("rest of the code");
```

```
    }
}
```

Output:

```
ArrayIndexOutOfBounds Exception occurs
rest of the code
```

Example

Let's see an example, to handle the exception without maintaining the order of exceptions (i.e. from most specific to most general).

```
class MultipleCatchBlock5{      public
static void main(String args[]){
try{        int a[]=new int[5];
a[5]=30/0;
   }
   catch(Exception e){System.out.println("common task completed");}
catch(ArithmeticException e){System.out.println("task1 is completed");}
catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed
");}
   System.out.println("rest of the code...");
 }
}
```

Output:

```
Compile-time error
```

Common Scenarios of Java Exceptions

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

1) A scenario where ArithmeticException occurs

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

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

2) A scenario where NullPointerException occurs

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

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

3) A scenario where NumberFormatException occurs

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

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

4) A scenario where ArrayIndexOutOfBoundsException occurs

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

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

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.

The syntax of java throw keyword is given below.

```
throw exception;
```

Let's see the example of throw IOException.

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

In Java we have already defined exception classes such as ArithmeticException, NullPointerException, ArrayIndexOutOfBounds exception etc. These exceptions are set to trigger on different-2 conditions.

For example when we divide a number by zero, this triggers ArithmeticException, when we try to access the array element out of its bounds then we get ArrayIndexOutOfBoundsException.

We can define our own set of conditions or rules and throw an exception explicitly using throw keyword.

For example, we can throw ArithmeticException when we divide number by 5, or any other numbers, what we need to do is just set the condition and throw any

exception using throw keyword. Throw keyword can also be used for throwing custom exceptions, **Syntax of throw keyword:**

```
throw new exception_class("error message");
```

For example:

```
throw new ArithmeticException("dividing a number by 5 is not allowed in this program");
```

EXAMPLE:

```
public class TestThrow1{
  static void validate(int age){
    if(age<18)
     throw new ArithmeticException("not valid");
    else
     System.out.println("welcome to vote");
  }
  public static void main(String args[]){
     validate(13);
     System.out.println("rest of the code...");
  }
}
```

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

Example of throw keyword

Lets say we have a requirement where we we need to only register the students when their age is less than 12 and weight is less than 40, if any of the condition is not met then the user should get an ArithmeticException with the warning message "Student is not eligible for registration".

We have implemented the logic by placing the code in the method that checks student eligibility if the entered student age and weight doesn't met the criteria then we throw the exception using throw keyword.

```
/* In this program we are checking the Student age
 * if the student age<12 and weight <40 then our program   *
should return that the student is not eligible for registration.
 */
public class ThrowExample {
   static void checkEligibilty(int stuage, int stuweight){
      if(stuage<12 && stuweight<40) {         throw new
ArithmeticException("Student is not eligible for registration");
      }
else {
        System.out.println("Student Entry is Valid!!");
}
   }

   public static void main(String args[]){
     System.out.println("Welcome to the Registration process!!");
checkEligibilty(10, 39);
     System.out.println("Have a nice day..");
 }
```

```
}
```

Output:

```
Welcome to the Registration process!!Exception in thread "main"
java.lang.ArithmeticException: Student is not eligible for registration at
beginnersbook.com.ThrowExample.checkEligibilty(ThrowExample.java:9) at
beginnersbook.com.ThrowExample.main(ThrowExample.java:18)
```

Throws –Keyword

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there **may occur** an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used. **(OR)**
As we know that there are two types of exception checked and unchecked. Checked exception (compile time) force you to handle them, if you don't handle them then the program will not compile.

On the other hand unchecked exception (Runtime) doesn't get checked during compilation. **Throws keyword** is used for handling checked exceptions . By using throws we can declare multiple exceptions in one go.

Syntax of java throws

```
return_type method_name() throws exception_class_name{
//method code
}
```

What is the need of having throws keyword when you can handle exception using try-catch?
Well, thats a valid question. We already know we can handle exceptions using try-catch block.

The throws does the same thing that try-catch does but there are some cases where you would prefer throws over try-catch.
For example:
Lets say we have a method myMethod() that has statements that can throw either ArithmeticException or NullPointerException, in this case you can use trycatch as shown below:

```
public void myMethod()
{
  try {
    // Statements that might throw an exception
  }
  catch (ArithmeticException e) {
    // Exception handling statements
  }
  catch (NullPointerException e) {
    // Exception handling statements
  }
}
```

But suppose you have several such methods that can cause exceptions, in that case it would be tedious to write these try-catch for each method. The code will become unnecessary long and will be less-readable.
One way to overcome this problem is by using throws like this: declare the exceptions in the method signature using throws and handle the exceptions where you are calling this method by using try-catch.

Another advantage of using this approach is that you will be forced to handle the exception when you call this method, all the exceptions that are declared using throws, must be handled where you are calling this method else you will get compilation error.

```
public void myMethod() throws ArithmeticException, NullPointerException {
  // Statements that might throw an exception  }

public static void main(String args[]) {
try {
    myMethod();
  }
  catch (ArithmeticException e) {
// Exception handling statements
  }
  catch (NullPointerException e) {
// Exception handling statements
  }
}
```

Example of throws Keyword
In this example the method myMethod() is throwing two **checked exceptions** so we have declared these exceptions in the method signature using **throws** Keyword. If we do not declare these exceptions then the program will throw a compilation error.

```
import java.io.*;
class ThrowExample {
  void myMethod(int num)throws IOException, ClassNotFoundException{
    if(num==1)
      throw new IOException("IOException Occurred");
    else
      throw new ClassNotFoundException("ClassNotFoundException");
  }
}

public class Example1{
  public static void main(String args[]){
   try{
     ThrowExample obj=new ThrowExample();
     obj.myMethod(1);
   }catch(Exception ex){
     System.out.println(ex);
    }
  }}
```

Output: java.io.IOException: IOException Occurred

For more examples on throws refer this tutorial: throws examples.

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.

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.

```
import java.io.*;
class M{
 void method()throws IOException{
  throw new IOException("device error");
 }
}
public class Testthrows2{
   public static void main(String args[]){
    try{
     M m=new M();
     m.method();
    }catch(Exception e){System.out.println("exception handled");}

    System.out.println("normal flow...");
  }
}
```

```
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

```
import java.io.*;   class
M{
 void method()throws IOException{
  System.out.println("device operation performed");
 }
}
class Testthrows3{
   public static void main(String args[])throws IOException{//declare exception
M m=new M();        m.method();

    System.out.println("normal flow...");
}
}
//it will not show error because there is no exception is occurred so that there is no tyrcatch block. It will execute successfully.
```

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

B)Program if exception occurs

```
import java.io.*;
class M{
 void method()throws IOException{
  throw new IOException("device error");
 }
}
class Testthrows4{
   public static void main(String args[])throws IOException{//declare exception
     M m=new M();
     m.method();

    System.out.println("normal flow...");
  }
}
//it will show error because there is no try-catch block to catch the exception
```

Output:Runtime Exception

Difference between throw and throws in Java

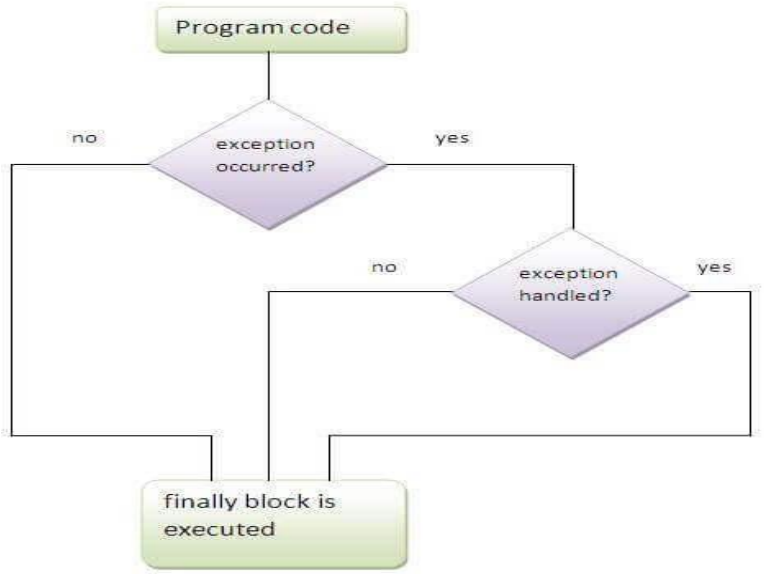
There are many differences between throw and throws keywords. A list of differences between throw and throws are given below:

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 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 follows try or catch block.



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**.

```
class TestFinallyBlock{
  public static void main(String args[]){
  try{
   int data=25/5;
   System.out.println(data);
  }
  catch(NullPointerException e){System.out.println(e);}
  finally{System.out.println("finally block is always executed");}
  System.out.println("rest of the code...");
  }
}
```

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**.

```
class TestFinallyBlock1{
  public static void main(String args[]){
try{
   int data=25/0;
   System.out.println(data);
  }
  catch(NullPointerException e){System.out.println(e);}
  finally{System.out.println("finally  block  is  always  executed");}
System.out.println("rest of the code...");
  }
}
```

Output:finally block is always executed

Case 3

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

```
public class TestFinallyBlock2{
public static void main(String args[]){
try{
   int data=25/0;
   System.out.println(data);
  }
  catch(ArithmeticException e){System.out.println(e);}
  finally{System.out.println("finally block is always executed");}
System.out.println("rest of the code...");     }
}
```

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).

User defined exception in java

In java we have already defined, exception classes such as ArithmeticException, NullPointerException etc. These exceptions are already set to trigger on predefined conditions such as when you divide a number by zero it triggers ArithmeticException,

In java we can create our own exception class and throw that exception using throw keyword. These exceptions are known as **user-defined** or **custom** exceptions. In this tutorial we will see how to create your own custom exception and throw it on a particular condition.

Example of User defined exception in Java

```
/* This is my Exception class, I have named it MyException
*       you can give any name, just remember that it should
*       extend Exception class
 */
class MyException extends Exception{
  String str1;
  /* Constructor of custom exception class
*       here I am copying the message that we are passing while     * throwing
the exception to a string and then displaying      * that string along with the
message.
   */
  MyException(String str2) {
        str1=str2;
  }
  public String toString(){
        return ("MyException Occurred: "+str1) ;
  }
}

class Example1{    public static void
main(String args[]){
        try{
              System.out.println("Starting of try block");          //
I'm throwing the custom exception using throw              throw new
MyException("This is My error Message");
        }
        catch(MyException exp){
                System.out.println("Catch Block") ;
                System.out.println(exp) ;
        }
  }
}
```

Output:

```
Starting of try block
Catch Block
MyException Occurred: This is My error Message
```

Explanation:

You can see that while throwing custom exception I gave a string in parenthesis (`throw new MyException("This is My error Message");`). That's why we have a parameterized constructor (with a String parameter) in my custom exception class. **Notes:**

1. User-defined exception must extend Exception class.
2. The exception is thrown using throw keyword.

Example:

```
class MyException1 extends Exception{
   String str1;

   MyException1(String str2) {
    str1=str2;
   }
   public String toString(){      return
("MyException Occurred: "+str1) ;
   }
}

class Example1{
   public static void main(String args[]){
    try{
        System.out.println("Starting of try block");
        throw new MyException1("This is error Message");
    }
    catch(MyException1 exp){
        System.out.println("Catch Block") ;
        System.out.println(exp) ;
    }
   }
}
```

Here in the above snippet, I have demonstrated the use of try-catch block and how we can throw exceptions using that. The throw keyword is used to throw the exception by the user. IO Exception is used for this exception handling. The user-defined exception must contain a custom exception class. In the code, we have used a parameterized constructor which displays (This is error Message).

OUTPUT:

```
Starting of try block
Catch Block
MyException1 Occurred: This is error Message
```