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 exceptions, its type and the difference between checked and unchecked exceptions.

What is Exception in Java

Dictionary Meaning: Exception is an abnormal condition.

In Java, an 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 ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Advantage of Exception Handling

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:

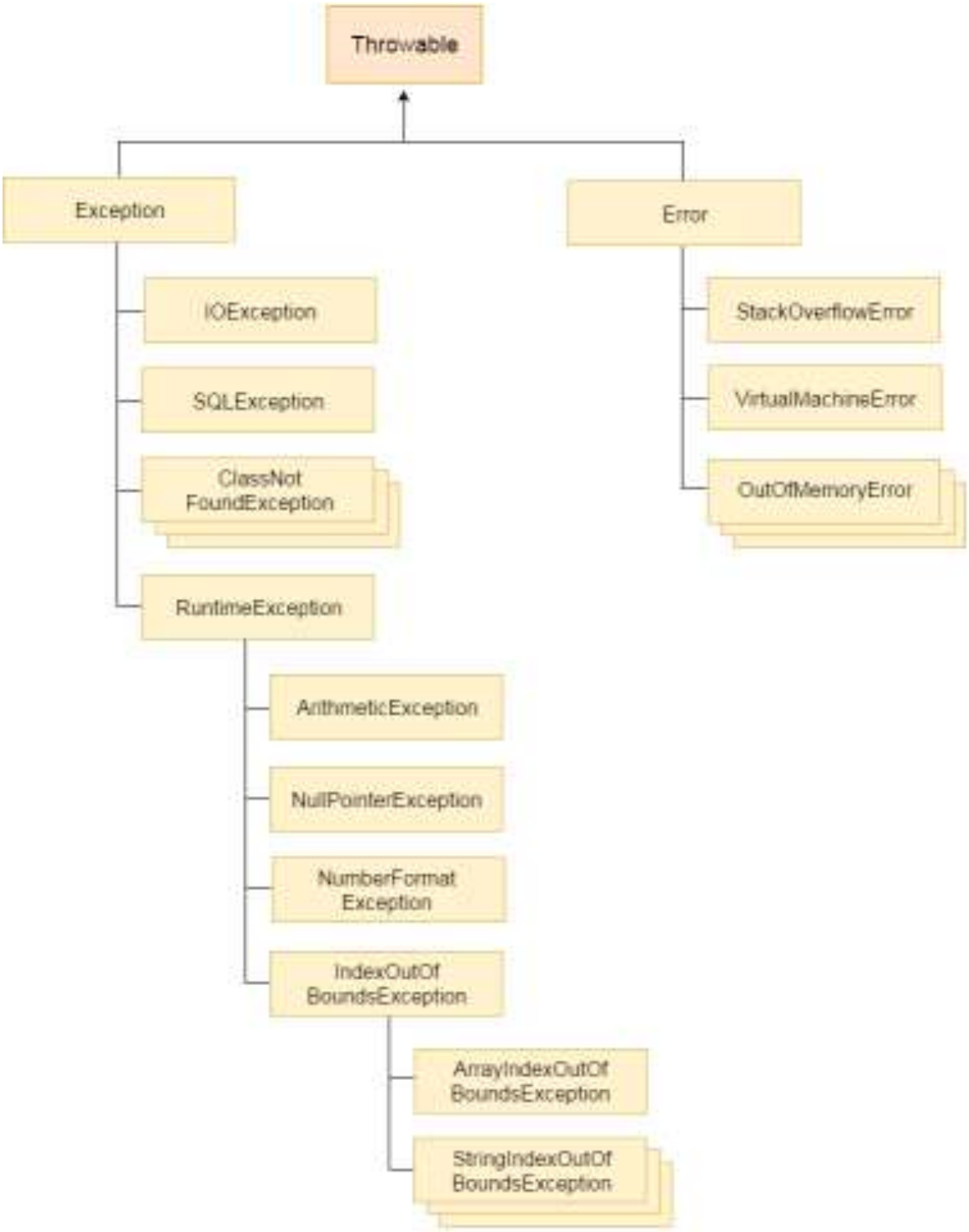
```
1. statement 1;
2. statement 2;
3. statement 3;
4. statement 4;
5. statement 5;//exception occurs
6. statement 6;
7. statement 7;
8. statement 8;
9. statement 9;
10. statement 10;
```

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.

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 Java Exceptions

There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the unchecked exception. According to Oracle, 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 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.

2) Unchecked Exception

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.

3) Error

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

What are checked exceptions?

Checked exceptions are checked at compile-time. It means if a method is throwing a checked exception then it should handle the exception using try-catch block or it should declare the exception using throws keyword, otherwise the program will give a compilation error.

Lets understand this with the help of an **example**:

Checked Exception Example

In this example we are reading the file myfile.txt and displaying its content on the screen. In this program there are three places where a checked exception is thrown as mentioned in the comments below. FileInputStream which is used for specifying the file path and name, throws FileNotFoundException. The read() method which reads the file content throws IOException and the close() method which closes the file input stream also throws IOException.

```
import java.io.*;
class Example {
   public static void main(String args[])
   {
       FileInputStream fis = null;
       /*This constructor FileInputStream(File filename)
        * throws FileNotFoundException which is a checked
        * exception
         */
        fis = new FileInputStream("B:/myfile.txt");
       int k;

       /* Method read() of FileInputStream class also throws
        * a checked exception: IOException
         */
       while(( k = fis.read() ) != -1)
       {
              System.out.print((char)k);
       }

       /*The method close() closes the file input stream
        * It throws IOException*/
       fis.close();
   }
}
```

Output:

```
Exception in thread "main" java.lang.Error: Unresolved compilation
problems:
Unhandled exception type FileNotFoundException
Unhandled exception type IOException
Unhandled exception type IOException
```

Why this compilation error? As I mentioned in the beginning that checked exceptions gets checked during compile time. Since we didn't handled/declared the exceptions, our program gave the compilation error.
How to resolve the error? There are two ways to avoid this error. We will see both the ways one by one.

Method 1: Declare the exception using throws keyword.
As we know that all three occurrences of checked exceptions are inside main() method so one way to avoid the compilation error is: Declare the exception in the method using throws keyword. You may be thinking that our code is throwing FileNotFoundException and IOException both then why we are declaring the IOException alone. The reason is that IOException is a parent class of FileNotFoundException so it by default covers that. If you want you can declare them like this `public static void main(String args[]) throws IOException, FileNotFoundException`.

```
import java.io.*;
class Example {
   public static void main(String args[]) throws IOException
   {
      FileInputStream fis = null;
      fis = new FileInputStream("B:/myfile.txt");
      int k;

      while(( k = fis.read() ) != -1)
      {
          System.out.print((char)k);
      }
      fis.close();
   }
}
```

Output:
File content is displayed on the screen.

Method 2: Handle them using try-catch blocks.
The approach we have used above is not good at all. It is not the best exception handling practice. You should give meaningful message for each exception type so that it would be easy for someone to understand the error. The code should be like this:

```
import java.io.*;
class Example {
   public static void main(String args[])
   {
       FileInputStream fis = null;
```

```
        try{
            fis = new FileInputStream("B:/myfile.txt");
        }catch(FileNotFoundException fnfe){
             System.out.println("The specified file is not " +
                       "present at the given path");
         }
        int k;
        try{
            while(( k = fis.read() ) != -1)
            {
               System.out.print((char)k);
            }
            fis.close();
        }catch(IOException ioe){
            System.out.println("I/O error occurred: "+ioe);
         }
    }
}
```

This code will run fine and will display the file content.

What are Unchecked exceptions?

Unchecked exceptions are not checked at compile time. It means if your program is throwing an unchecked exception and even if you didn't handle/declare that exception, the program won't give a compilation error. Most of the times these exception occurs due to the bad data provided by user during the user-program interaction. It is up to the programmer to judge the conditions in advance, that can cause such exceptions and handle them appropriately. All Unchecked exceptions are direct sub classes of **RuntimeException** class.

Lets understand this with an example:

Unchecked Exception Example

```
class Example {
   public static void main(String args[])
   {
        int num1=10;
        int num2=0;
        /*Since I'm dividing an integer with 0
         * it should throw ArithmeticException
          */
        int res=num1/num2;
        System.out.println(res);
   }
}
```

If you compile this code, it would compile successfully however when you will run it, it would throw ArithmeticException. That clearly shows that unchecked exceptions are not checked at compile-time, they occurs at runtime. Let us see another example.

```
class Example {
   public static void main(String args[])
   {
       int arr[] ={1,2,3,4,5};
       /* My array has only 5 elements but we are trying to
         * display the value of 8th element. It should throw
        * ArrayIndexOutOfBoundsException
         */
       System.out.println(arr[7]);
   }
}
```

This code would also compile successfully since ArrayIndexOutOfBoundsException is also an unchecked exception.
Note: It **doesn't mean** that compiler is not checking these exceptions so we shouldn't handle them. In fact we should handle them more carefully. For e.g. In the above example there should be a exception message to user that they are trying to display a value which doesn't exist in array so that user would be able to correct the issue.

```
class Example {
   public static void main(String args[]) {
       try{
          int arr[] ={1,2,3,4,5};
          System.out.println(arr[7]);
       }
        catch(ArrayIndexOutOfBoundsException e){
          System.out.println("The specified index does not exist " +
              "in array. Please correct the error.");
       }
   }
}
```

Output:

```
The specified index does not exist in array. Please correct the
error.
```

Should we make our exceptions checked or unchecked?
Following is the bottom line from Java documents
If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception

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.

Java Exception Handling Example

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

```
1.  public class JavaExceptionExample{
2.    public static void main(String args[]){
3.     try{
4.        //code that may raise exception
5.        int data=100/0;
6.     }catch(ArithmeticException e){System.out.println(e);}
7.     //rest code of the program
8.     System.out.println("rest of the code...");
9.    }
10. }
```

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.

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 try-catch block

Java try block

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

If an exception occurs at the particular statement of try block, the rest of the block code will not execute. So, it is recommended not to keeping the code in try block that will not throw an exception.

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

Syntax of Java try-catch

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

Syntax of try-finally block

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

Java catch block

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

The catch block must be used after the try block only. You can use multiple catch block with a single try block.

Problem without exception handling

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

Example 1

```
public class TryCatchExample1 {

    public static void main(String[] args) {

```

```
5.        int data=50/0; //may throw exception
6.
7.        System.out.println("rest of the code");
8.
9.    }
10.
11. }
```

Output:

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

As displayed in the above example, the **rest of the code** is not executed (in such case, the **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 the above problem by a java try-catch block.

Example 2

```
1.  public class TryCatchExample2 {
2.
3.     public static void main(String[] args) {
4.        try
5.        {
6.        int data=50/0; //may throw exception
7.        }
8.           //handling the exception
9.        catch(ArithmeticException e)
10.       {
11.          System.out.println(e);
12.       }
13.       System.out.println("rest of the code");
14.   }
15.
16. }
```

Output:

```
java.lang.ArithmeticException: / by zero
rest of the code
```

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

Example 3

In this example, we also kept the code in a try block that will not throw an exception.

```
1.  public class TryCatchExample3 {
2.
3.      public static void main(String[] args) {
4.          try
5.          {
6.          int data=50/0; //may throw exception
7.                          // if exception occurs, the remaining statement will not exceute
8.          System.out.println("rest of the code");
9.          }
10.             // handling the exception
11.         catch(ArithmeticException e)
12.         {
13.             System.out.println(e);
14.         }
15.
16.     }
17.
18. }
```

Output:

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

Here, we can see that if an exception occurs in the try block, the rest of the block code will not execute.

Example 4

Here, we handle the exception using the parent class exception.

```
1.  public class TryCatchExample4 {
2.
3.      public static void main(String[] args) {
4.          try
5.          {
6.          int data=50/0; //may throw exception
7.          }
8.              // handling the exception by using Exception class
```

```
9.        catch(Exception e)
10.       {
11.          System.out.println(e);
12.       }
13.       System.out.println("rest of the code");
14.    }
15.
16. }
```

Output:

```
java.lang.ArithmeticException: / by zero
rest of the code
```

Example 5

Let's see an example to print a custom message on exception.

```
1.  public class TryCatchExample5 {
2.
3.     public static void main(String[] args) {
4.        try
5.        {
6.        int data=50/0; //may throw exception
7.        }
8.            // handling the exception
9.        catch(Exception e)
10.       {
11.               // displaying the custom message
12.          System.out.println("Can't divided by zero");
13.       }
14.    }
15.
16. }
```

Output:

```
Can't divided by zero
```

Example 6

Let's see an example to resolve the exception in a catch block.

```
1.  public class TryCatchExample6 {
2.
3.     public static void main(String[] args) {
4.        int i=50;
```

```
5.        int j=0;
6.        int data;
7.        try
8.        {
9.        data=i/j; //may throw exception
10.       }
11.          // handling the exception
12.       catch(Exception e)
13.       {
14.           // resolving the exception in catch block
15.          System.out.println(i/(j+2));
16.       }
17.   }
18. }
```

Output:

```
25
```

Example 7

In this example, along with try block, we also enclose exception code in a catch block.

```
1.  public class TryCatchExample7 {
2.
3.     public static void main(String[] args) {
4.
5.        try
6.        {
7.        int data1=50/0; //may throw exception
8.
9.        }
10.           // handling the exception
11.       catch(Exception e)
12.       {
13.          // generating the exception in catch block
14.       int data2=50/0; //may throw exception
15.
16.       }
17.    System.out.println("rest of the code");
18.    }
19. }
```

Output:

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

Here, we can see that the catch block didn't contain the exception code. So, enclose exception code within a try block and use catch block only to handle the exceptions.

Example 8

In this example, we handle the generated exception (Arithmetic Exception) with a different type of exception class (ArrayIndexOutOfBoundsException).

```
1.  public class TryCatchExample8 {
2.
3.     public static void main(String[] args) {
4.        try
5.        {
6.        int data=50/0; //may throw exception
7.
8.        }
9.           // try to handle the ArithmeticException using ArrayIndexOutOfBoundsException

10.       catch(ArrayIndexOutOfBoundsException e)
11.       {
12.          System.out.println(e);
13.       }
14.       System.out.println("rest of the code");
15.    }
16.
17. }
```

Output:

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

Example 9

Let's see an example to handle another unchecked exception.

```
1.  public class TryCatchExample9 {
2.
3.     public static void main(String[] args) {
4.        try
5.        {
6.        int arr[]= {1,3,5,7};
7.        System.out.println(arr[10]); //may throw exception
8.        }
9.           // handling the array exception
10.       catch(ArrayIndexOutOfBoundsException e)
11.       {
```

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

}
```

Output:

```
java.lang.ArrayIndexOutOfBoundsException: 10
rest of the code
```

Example 10

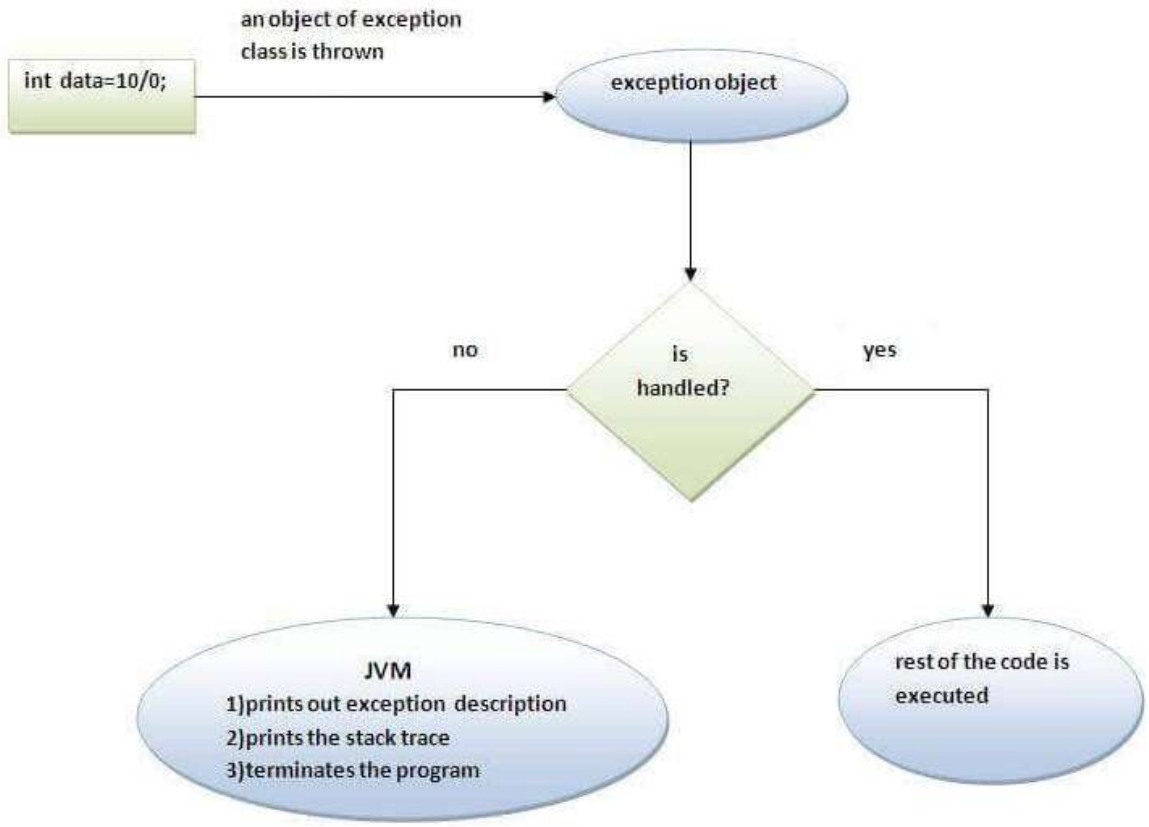
Let's see an example to handle checked exception.

```
import java.io.FileNotFoundException;
import java.io.PrintWriter;

public class TryCatchExample10 {

    public static void main(String[] args) {


        PrintWriter pw;
        try {
            pw = new PrintWriter("jtp.txt"); //may throw exception
            pw.println("saved");
        }
// providing the checked exception handler
 catch (FileNotFoundException e) {

            System.out.println(e);
        }
    System.out.println("File saved successfully");
    }
}
```

Output:

```
File saved successfully
```

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

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 1

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

```
1. public class MultipleCatchBlock1 {
2.
3.     public static void main(String[] args) {
4.
5.         try{
6.             int a[]=new int[5];
7.             a[5]=30/0;
8.             }
9.             catch(ArithmeticException e)
10.              {
11.               System.out.println("Arithmetic Exception occurs");
12.              }
13.            catch(ArrayIndexOutOfBoundsException e)
14.              {
15.               System.out.println("ArrayIndexOutOfBounds Exception occurs");
16.              }
17.            catch(Exception e)
18.              {
19.               System.out.println("Parent Exception occurs");
20.              }
21.            System.out.println("rest of the code");
22.    }
23. }
```

Output:

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

Example 2

```
1. public class MultipleCatchBlock2 {
2.
3.     public static void main(String[] args) {
4.
5.         try{
6.             int a[]=new int[5];
```

```
7.
8.              System.out.println(a[10]);
9.             }
10.            catch(ArithmeticException e)
11.              {
12.               System.out.println("Arithmetic Exception occurs");
13.              }
14.            catch(ArrayIndexOutOfBoundsException e)
15.              {
16.               System.out.println("ArrayIndexOutOfBounds Exception occurs");
17.              }
18.            catch(Exception e)
19.              {
20.               System.out.println("Parent Exception occurs");
21.              }
22.            System.out.println("rest of the code");
23.    }
24. }
```

Output:

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

Example 3

In this example, try block contains two exceptions. But at a time only one exception occurs and its corresponding catch block is invoked.

```
1.  public class MultipleCatchBlock3 {
2.
3.     public static void main(String[] args) {
4.
5.          try{
6.              int a[]=new int[5];
7.              a[5]=30/0;
8.              System.out.println(a[10]);
9.             }
10.            catch(ArithmeticException e)
11.              {
12.               System.out.println("Arithmetic Exception occurs");
13.              }
14.            catch(ArrayIndexOutOfBoundsException e)
15.              {
16.               System.out.println("ArrayIndexOutOfBounds Exception occurs");
17.              }
```

```
18.           catch(Exception e)
19.             {
20.              System.out.println("Parent Exception occurs");
21.             }
22.           System.out.println("rest of the code");
23.    }
24. }
```

Output:

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

Example 4

In this example, we generate NullPointerException, but didn't provide the corresponding exception type. In such case, the catch block containing the parent exception class **Exception** will invoked.

```
1.  public class MultipleCatchBlock4 {
2.
3.     public static void main(String[] args) {
4.
5.          try{
6.              String s=null;
7.              System.out.println(s.length());
8.             }
9.             catch(ArithmeticException e)
10.              {
11.               System.out.println("Arithmetic Exception occurs");
12.              }
13.            catch(ArrayIndexOutOfBoundsException e)
14.              {
15.               System.out.println("ArrayIndexOutOfBounds Exception occurs");
16.              }
17.            catch(Exception e)
18.              {
19.               System.out.println("Parent Exception occurs");
20.              }
21.            System.out.println("rest of the code");
22.    }
23. }
```

Output:

```
Parent Exception occurs
rest of the code
```

Example 5

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

```
1. class MultipleCatchBlock5{
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:

```
1. ....
2. try
3. {
4.     statement 1;
5.     statement 2;
6.     try
7.     {
8.         statement 1;
9.         statement 2;
10.    }
11.    catch(Exception e)
```

```
12.    {
13.    }
14. }
15. catch(Exception e)
16. {
17. }
18. ....
```

Java nested try example

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

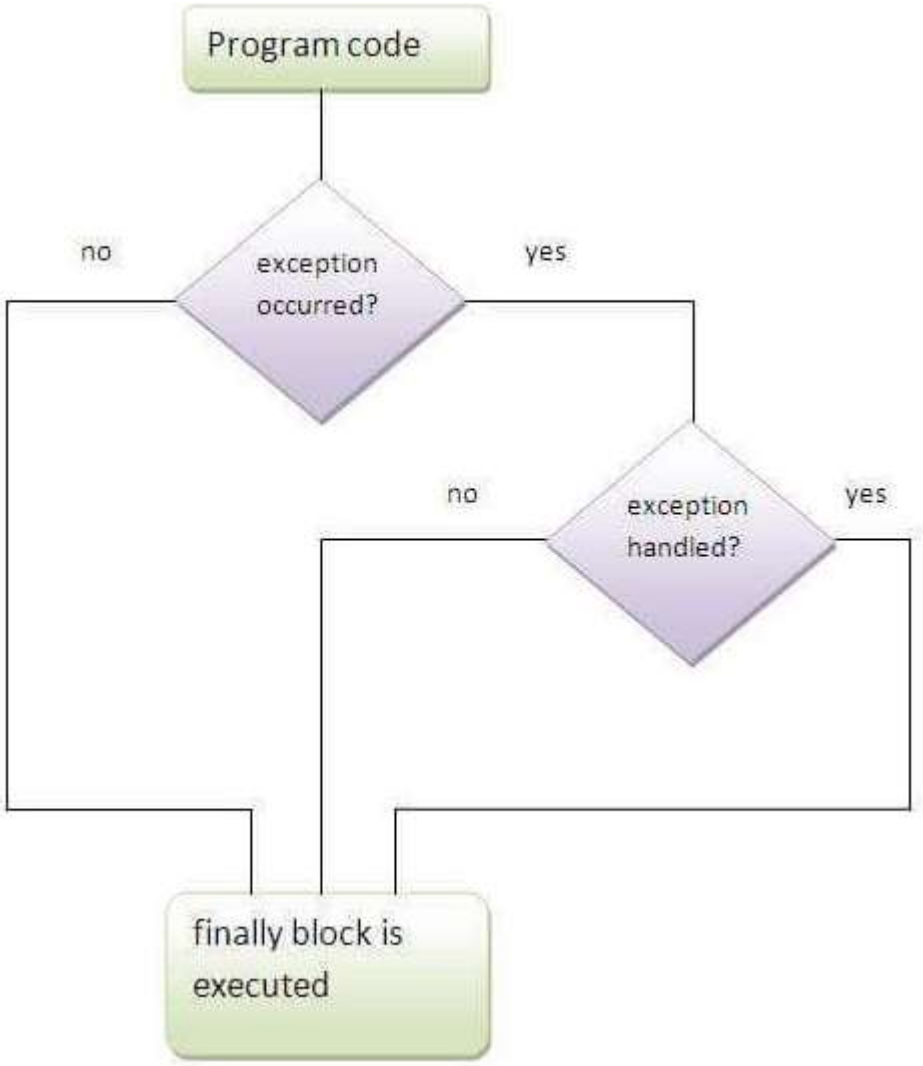
```
1.  class Excep6{
2.   public static void main(String args[]){
3.    try{
4.      try{
5.       System.out.println("going to divide");
6.       int b =39/0;
7.      }catch(ArithmeticException e){System.out.println(e);}
8.
9.      try{
10.     int a[]=new int[5];
11.     a[5]=4;
12.     }catch(ArrayIndexOutOfBoundsException e){System.out.println(e);}
13.
14.     System.out.println("other statement");
15.   }catch(Exception e){System.out.println("handeled");}
16.
17.   System.out.println("normal flow..");
18.  }
19. }
```

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.



> *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. }
```

```
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

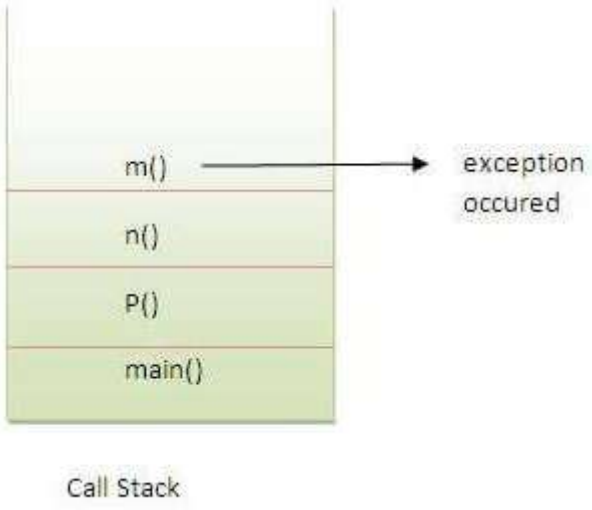
An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method, If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack. This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

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 programmer's fault that he is not performing check up before the code being used.

Syntax of java throws

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

Which exception should be declared

Ans) checked exception only, because:

- **unchecked Exception:** under your control so correct your code.
- **error:** beyond your control e.g. you are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

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...
```

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

There are two cases:

1. **Case1:**You catch 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...");
```

```
 }
}
```

```
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...");
 }
}
```

```
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();

```

```
12.     System.out.println("normal flow...");
13.   }
14. }
```

```
Output:Runtime Exception
```

Que) Can we rethrow an exception?

Yes, by throwing same exception in catch block.

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. }}
```

Exception Handling with Method Overriding in Java

There are many rules if we talk about method overriding 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");}
```

```
}

class TestExceptionChild extends Parent{
  void msg()throws IOException{
    System.out.println("TestExceptionChild");
  }
  public static void main(String args[]){
   Parent p=new TestExceptionChild();
   p.msg();
  }
}
```

```
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.*

```
import java.io.*;
class Parent{
  void msg(){System.out.println("parent");}
}

class TestExceptionChild1 extends Parent{
  void msg()throws ArithmeticException{
    System.out.println("child");
  }
  public static void main(String args[]){
   Parent p=new TestExceptionChild1();
   p.msg();
  }
}
```

```
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

```
import java.io.*;
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");}
```

```
}

class TestExceptionChild4 extends Parent{
  void msg()throws ArithmeticException{System.out.println("child");}

  public static void main(String args[]){
   Parent p=new TestExceptionChild4();
   try{
   p.msg();
   }catch(Exception e){}
  }
}
```

```
Output:child
```

Example in case subclass overridden method declares no exception

```
import java.io.*;
class Parent{
  void msg()throws Exception{System.out.println("parent");}
}

class TestExceptionChild5 extends Parent{
  void msg(){System.out.println("child");}

  public static void main(String args[]){
   Parent p=new TestExceptionChild5();
   try{
   p.msg();
   }catch(Exception e){}
  }
}
```

```
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...
```