Exception handling in java with example programs

we will discuss what is an exception and how it can be handled in java programming language.

What is an exception?

An Exception can be anything which interrupts the normal flow of the program. When an exception occurs program processing gets terminated and doesn't continue further. In such cases we get a system generated error message. The good thing about exceptions is that they can be handled. We will cover the handling part later in this same tutorial.

When an exception can occur?

Exception can occur at runtime (known as runtime exceptions) as well as at compile-time (known Compile-time exceptions).

Reasons for Exceptions

There can be several reasons for an exception. For example, following situations can cause an exception –Opening a non-existing file, Network connection problem, Operands being manipulated are out of prescribed ranges, class file missing which was supposed to be loaded and so on.

Difference between error and exception

Errors indicate serious problems and abnormal conditions that most applications should not try to handle. Error defines problems that are not expected to be caught under normal circumstances by our program. For example memory error, hardware error, JVM error etc.

Exceptions are the conditions within code which a developer can handle and take necessary corrective actions. Few examples –

- DivideByZero exception
- NullPointerException
- ArithmeticException
- ArrayIndexOutOfBoundsException

Advantages of Exception Handling

- Exception handling allows us to control the normal flow of the program by using exception handling in program.
- It throws an exception whenever a calling method encounters an error providing that the calling method takes care of that error.
- It also gives us the scope of organizing and differentiating between different error types using a separate block of codes. This is done with the help of try-catch blocks.

Why to handle exception?

If an exception is raised, which has not been handled by programmer then program execution can get terminated and system prints a non user friendly error message.

Ex:-Take a look at the below system generated exception

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
For a novice user the above message won't be easy to understand. In order to let them know that what went wrong we use exception handling in java program. 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.

Types of exceptions
There are two types of exceptions
1)Checked exceptions
2)Unchecked exceptions

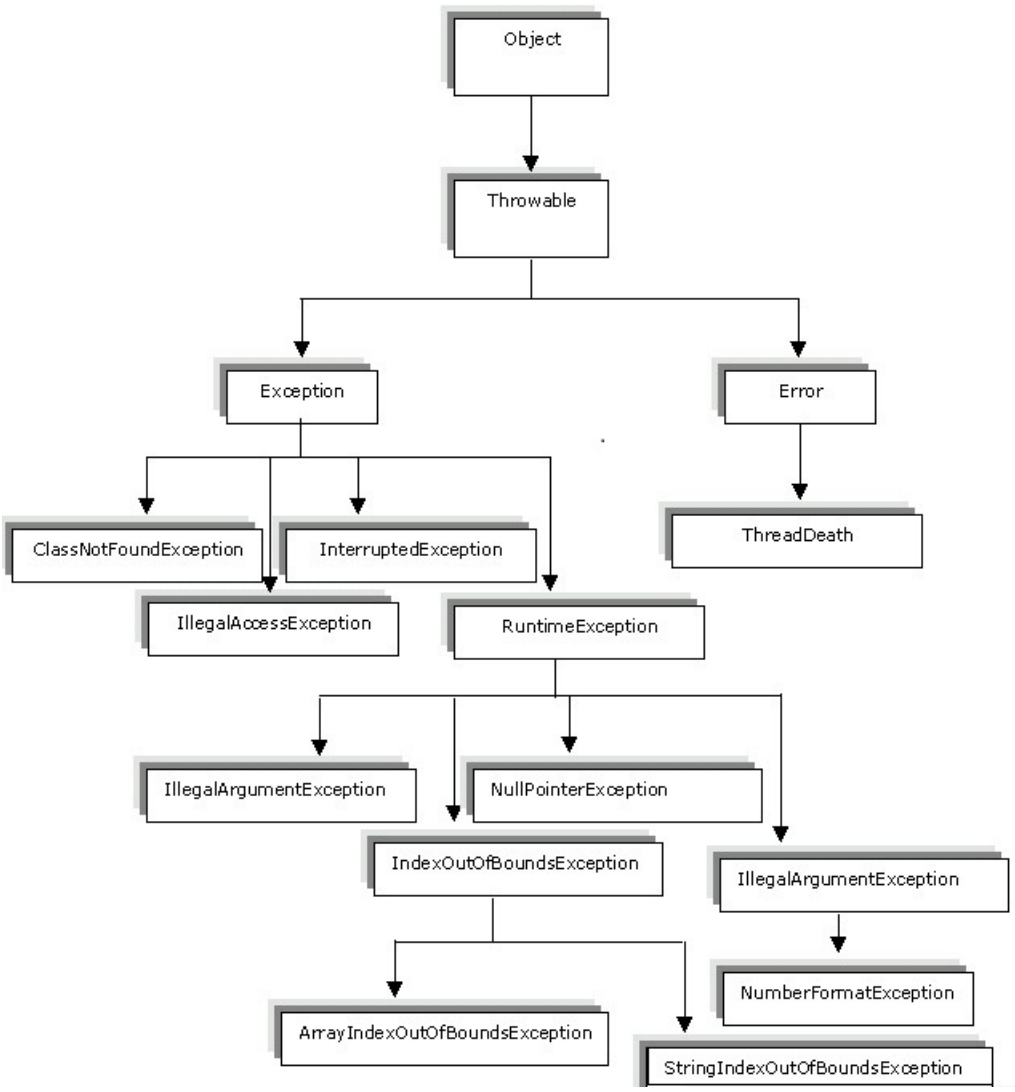
Checked exceptions
All exceptions other than Runtime Exceptions are known as Checked exceptions as the compiler checks them during compilation to see whether the programmer has handled them or not. If these exceptions are not handled/declared in the program, it will give compilation error.
Examples of Checked Exceptions :-

ClassNotFoundException
IllegalAccessException
NoSuchFieldException
EOFException etc.

Unchecked Exceptions
Runtime Exceptions are also known as Unchecked Exceptions as the compiler do not check whether the programmer has handled them or not but it's the duty of the programmer to handle these exceptions and provide a safe exit.
These exceptions need not be included in any method's throws list because compiler does not check to see if a method handles or throws these exceptions.

Examples of Unchecked Exceptions:-
ArithmeticException
ArrayIndexOutOfBoundsException
NullPointerException
NegativeArraySizeException etc.

Exception hierarchy



Exception handling in Java

1. Try-catch in Java
2. Nested Try Catch
3. Checked and unchecked exceptions
4. Finally block in Java
5. try-catch-finally
6. finally block & return statement
7. Throw exception in Java
8. Example of throw keyword
9. Example of throws clause
10. Throws in Java
11. throw vs throws
12. Exception handling examples

Try Catch in Java – Exception handling
Chaitanya Singh | Filed Under: Exception Handling | Updated: April 3, 2014

What is Try Block?
The try block contains a block of program statements within which an exception might 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 followed by a Catch block or Finally block or both.

Syntax of try block

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

What is Catch Block?
A catch block must be associated with a try block. The corresponding catch block executes if an exception of a particular type occurs within the try block. 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
}
```

Flow of try catch block

1. If an exception occurs in try block then the control of execution is passed to the catch block from try block. The exception is caught up by the corresponding catch block. A single try block can have multiple catch statements associated with it, but each catch block can be defined for only one exception class. The program can also contain nested try-catch-finally blocks.
2. After the execution of all the try blocks, the code inside the finally block executes. It is not mandatory to include a finally block at all, but if you do, it will run regardless of whether an exception was thrown and handled by the try and catch blocks.

An example of Try catch in Java

```
class Example1 {
  public static void main(String args[]) {
    int num1, num2;
    try {
      // Try block to handle code that may cause exception
      num1 = 0;
      num2 = 62 / num1;
      System.out.println("Try block message");
    } catch (ArithmeticException e) {
          // This block is to catch divide-by-zero error
          System.out.println("Error: Don't divide a number by zero");
      }
    System.out.println("I'm out of try-catch block in Java.");
  }
}
```

Output:

```
Error: Don't divide a number by zero
I'm out of try-catch block in Java.
```

Multiple catch blocks in Java

1. A try block can have any number of catch blocks.
2. A catch block that is written for catching the class Exception can catch all other exceptions

Syntax:

```
catch(Exception e){
//This catch block catches all the exceptions
}
```

3. If multiple catch blocks are present in a program then the above mentioned catch block should be placed at the last as per the exception handling best practices.
4. If the try block is not throwing any exception, the catch block will be completely ignored and the program continues.
5. If the try block throws an exception, the appropriate catch block (if one exists) will catch it

–catch(ArithmeticException e) is a catch block that can catch ArithmeticException

–catch(NullPointerException e) is a catch block that can catch NullPointerException

6. All the statements in the catch block will be executed and then the program continues.

Example of Multiple catch blocks

```
class Example2{
  public static void main(String args[]){
    try{
        int a[]=new int[7];
        a[4]=30/0;
        System.out.println("First print statement in try block");
    }
    catch(ArithmeticException e){
      System.out.println("Warning: ArithmeticException");
    }
    catch(ArrayIndexOutOfBoundsException e){
      System.out.println("Warning: ArrayIndexOutOfBoundsException");
    }
    catch(Exception e){
      System.out.println("Warning: Some Other exception");
    }
  System.out.println("Out of try-catch block...");
 }
}
```

Output:

```
Warning:ArithmeticException
Out of try-catch block...
```

In the above example there are multiple catch blocks and these catch blocks executes sequentially when an exception occurs in try block. Which means if you put the last catch block (catch(Exception e)) at the first place, just after try block then in case of any exception this block will execute as it has the ability to handle all exceptions. This catch block should be placed at the last to avoid such situations.

Nested try catch: Java exception handling

Chaitanya Singh | Filed Under: Exception Handling | Updated: January 9, 2014

The try catch blocks can be nested. One try-catch block can be present in the another try's body. This is called **Nesting of try catch** blocks. Each time a try block does not have a catch handler for a particular exception, the stack is unwound and the next try block's catch (i.e., parent try block's catch) handlers are inspected for a match.

If no catch block matches, then the java run-time system will handle the exception. Lets see the syntax first then we will discuss this with an example.

Syntax of Nested try Catch

```
....
//Main try block
try
{
  statement 1;
  statement 2;
```

```
  //try-catch block inside another try block
  try
  {
    statement 3;
    statement 4;
  }
  catch(Exception e1)
  {
     //Exception Message
  }
  //try-catch block inside another try block
  try
  {
     statement 5;
     statement 6;
  }
  catch(Exception e2)
  {
     //Exception Message
  }
}
catch(Exception e3) //Catch of Main(parent) try block
{
     //Exception Message
}
....
```

Nested try catch example – explanation

```
class Nest{
  public static void main(String args[]){
         //Parent try block
    try{
        //Child try block1
        try{
          System.out.println("Inside block1");
          int b =45/0;
          System.out.println(b);
        }
        catch(ArithmeticException e1){
          System.out.println("Exception: e1");
        }
        //Child try block2
        try{
          System.out.println("Inside block2");
          int b =45/0;
          System.out.println(b);
        }
        catch(ArrayIndexOutOfBoundsException e2){
```

```
            System.out.println("Exception: e2");
          }
         System.out.println("Just other statement");
      }
      catch(ArithmeticException e3){
            System.out.println("Arithmetic Exception");
         System.out.println("Inside parent try catch block");
      }
      catch(ArrayIndexOutOfBoundsException e4){
           System.out.println("ArrayIndexOutOfBoundsException");
         System.out.println("Inside parent try catch block");
      }
      catch(Exception e5){
           System.out.println("Exception");
         System.out.println("Inside parent try catch block");
       }
       System.out.println("Next statement..");
    }
}
```

Output:

```
Inside block1
Exception: e1
Inside block2
Arithmetic Exception
Inside parent try catch block
Next statement..
```

The above example shows Nested try catch use in Java. You can see that there are two try-catch block inside main try block's body. I've marked them as block 1 and block 2 in above example.

Block1: I have divided an integer by zero and it caused an arithmetic exception however the catch of block1 is handling arithmetic exception so "Exception: e1" got printed.

Block2: In block2 also, ArithmeticException occurred but block 2 catch is only handling ArrayIndexOutOfBoundsException so in this case control jump back to Main try-catch(parent) body. Since catch of parent try block is handling this exception that's why "Inside parent try catch block" got printed as output.

Parent try Catch block: Since all the exception handled properly so program control didn't get terminated at any point and at last "Next statement.." came as output.

Note: The main point to note here is that whenever the child try-catch blocks are not handling any exception, the control comes back to the parent try-catch if the exception is not handled there also then the program will terminate abruptly.

Consider this example:

Here we have deep (two level) nesting which means we have a try-catch block inside a child try block. To make you understand better I have given the names to each try block in comments like try-block2 etc.

This is how the structure is: try-block3 is inside try-block2 and try-block2 is inside main try-block, you can say that the main try-block is a grand parent of the try-block3. Refer the explanation which is given at the end of this code.

```
class NestingDemo{
   public static void main(String args[]){
     //main try-block
     try{
         //try-block2
       try{
          //try-block3
          try{
                int arr[]= {1,2,3,4};
                /* I'm trying to display the value of
                 * an element which doesn't exist. The
                 * code should throw an exception
                 */
                System.out.println(arr[10]);
          }catch(ArithmeticException e){
                System.out.print("Arithmetic Exception");
                System.out.println(" handled in try-block3");
          }
       }
       catch(ArithmeticException e){
          System.out.print("Arithmetic Exception");
          System.out.println(" handled in try-block2");
       }
    }
    catch(ArithmeticException e3){
          System.out.print("Arithmetic Exception");
          System.out.println(" handled in main try-block");
    }
    catch(ArrayIndexOutOfBoundsException e4){
          System.out.print("ArrayIndexOutOfBoundsException");
          System.out.println(" handled in main try-block");
    }
    catch(Exception e5){
          System.out.print("Exception");
          System.out.println(" handled in main try-block");
     }
  }
}
```

Output:

```
ArrayIndexOutOfBoundsException handled in main try-block
```

As you can see that the ArrayIndexOutOfBoundsException has occurred in the grand child try-block3. Since try-block3 is not handling this exception, the control then gets transferred to the parent try-block2 and looked for the catch handlers in try-block2. Since the try-block2 is also not handling that exception, the control

got transferred to the main grand parent try-block where it found the appropriate catch block for exception. This is how the **routing of exception is done in nested structure**.

Checked and unchecked exceptions in java with examples

There are two types of exceptions: checked exceptions and unchecked exceptions. In this tutorial we will learn both of them with the help of examples. The main **difference between checked and unchecked exception** is that the checked exceptions are checked at compile-time while unchecked exceptions are checked at runtime.

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. It is named as ***checked exception*** because these exceptions are ***checked*** at Compile time.

Lets understand this with this **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 an 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. Th reason is that IOException is a parent class of FileNotFoundException so it by default covers that. If you want you can declare that too 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 above approach is not good at all. It is not a 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.

Here are the few other Checked Exceptions –

- SQLException
- IOException
- DataAccessException
- ClassNotFoundException
- InvocationTargetException

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**:

```
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 are being checked at runtime.

Lets 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
         * I'm 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.");
                }
        }
}
```

Here are the few most frequently seen unchecked exceptions –

- NullPointerException
- ArrayIndexOutOfBoundsException
- ArithmeticException
- IllegalArgumentException

Java Finally block – Exception handling

Chaitanya Singh | Filed Under: Exception Handling | Updated: April 3, 2014

What is Finally Block

1. A finally statement must be associated with a try statement. It identifies a block of statements that needs to be executed regardless of whether or not an exception occurs within the try block.
2. After all other try-catch processing is complete, the code inside the finally block executes. It is not mandatory to include a finally block at all, but if you do, it will run regardless of whether an exception was thrown and handled by the try and catch parts of the block.
3. In normal execution the finally block is executed after try block. When any exception occurs first the catch block is executed and then finally block is executed.

4. An exception in the finally block, exactly behaves like any other exception.
5. The code present in the **finally block** executes even if the try or catch block contains control transfer statements like return, break or continue.
To understand above concepts better refer the below examples.

Syntax of Finally block

```
try
{
   //statements that may cause an exception
}
finally
{
  //statements to be executed
}
```

Cases when the finally block doesn't execute

The circumstances that prevent execution of the code in a finally block are:
– The death of a Thread
– Using of the System. exit() method.
– Due to an exception arising in the finally block.

Finally block and Return statement

Finally block executes even if there is a return statement in try-catch block. PFB the example –

```
class JavaFinally
{
  public static void main(String args[])
  {
    System.out.println(JavaFinally.myMethod());
  }
  public static int myMethod()
  {
    try {
      return 112;
    }
    finally {
      System.out.println("This is Finally block");
      System.out.println("Finally block ran even after return statement");
    }
  }
}
```

Output of above program:

```
This is Finally block
Finally block ran even after return statement
112
```

Finally and Close()

Close() is generally used to close all the open streams in one go. Its a good practice to use close() inside finally block. Since finally block executes even if exception occurs so you can be sure that all input and output streams are closed properly regardless of whether the exception occurs or not.

E.g.

```
....
try{
    OutputStream osf = new FileOutputStream( "filename" );
    OutputStream osb = new BufferedOutputStream(opf);
    ObjectOutput op = new ObjectOutputStream(osb);
    try{
      output.writeObject(writableObject);
    }
    finally{
      op.close();
    }
}
catch(IOException e1){
     System.out.println(e1);
}
...
```