# Exception Handling in Java

1. [Exception Handling](http://www.javatpoint.com/exception-handling-in-java)
2. [Advantage of Exception Handling](http://www.javatpoint.com/exception-handling-in-java#exceptionad)
3. [Hierarchy of Exception classes](http://www.javatpoint.com/exception-handling-in-java#exceptionhierarchy)
4. [Types of Exception](http://www.javatpoint.com/exception-handling-in-java#exceptiontypes)
5. [Scenarios where exception may occur](http://www.javatpoint.com/exception-handling-in-java#exceptionscenarios)

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

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

### What is exception

**Dictionary Meaning:** Exception is an abnormal condition.

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

### What is exception handling

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

### Advantage of Exception Handling

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

statement 1;

statement 2;

statement 3;

statement 4;

statement 5;//exception occurs

statement 6;

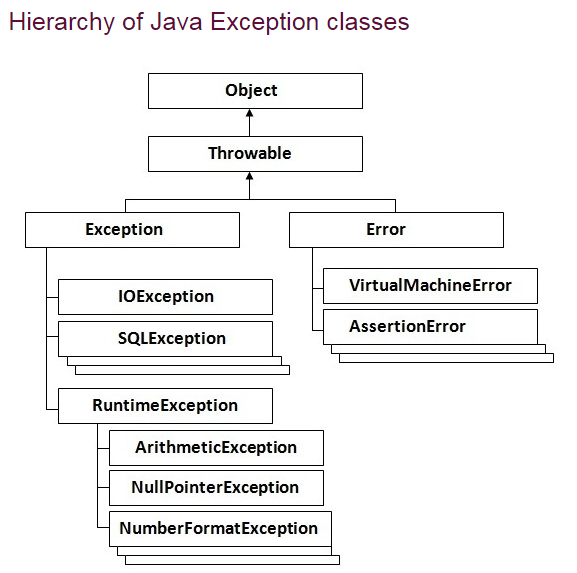
statement 7;

statement 8;

statement 9;

statement 10;

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



Types of Exception

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

1. Checked Exception
2. Unchecked Exception
3. Error

Difference between checked and unchecked exceptions

1) Checked Exception

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

2) Unchecked Exception

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

3) Error

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

Common scenarios where exceptions may occur

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

1) Scenario where ArithmeticException occurs

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

**int** a=50/0;//ArithmeticException

2) Scenario where NullPointerException occurs

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

String s=**null**;

System.out.println(s.length());//NullPointerException

3) Scenario where NumberFormatException occurs

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

String s="abc";

**int** i=Integer.parseInt(s);//NumberFormatException

4) Scenario where ArrayIndexOutOfBoundsException occurs

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

**int** a[]=**new** **int**[5];

a[10]=50; //ArrayIndexOutOfBoundsException

Java Exception Handling Keywords

There are 5 keywords used in java exception handling.

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

# Java try-catch

## Java try block

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

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

#### Syntax of java try-catch

**try**{

//code that may throw exception

}**catch**(Exception\_class\_Name ref){}

#### Syntax of try-finally block

**try**{

//code that may throw exception

}**finally**{}

**Java catch block**

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

You can use multiple catch block with a single try.

## Problem without exception handling

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

**public** **class** Testtrycatch1{

**public** **static** **void** main(String args[]){

**int** data=50/0;//may throw exception

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

}

}

Solution by exception handling

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

**public** **class** Testtrycatch2{

**public** **static** **void** main(String args[]){

**try**{

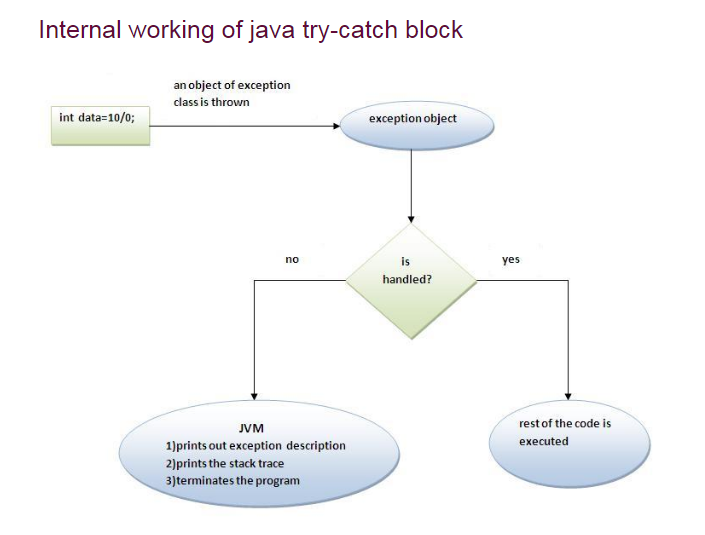
**int** data=50/0;

   }**catch**(ArithmeticException e){System.out.println(e);}

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

}

}



# Java catch multiple exceptions

## Java Multi catch block

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

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

**public** **class** TestMultipleCatchBlock{

**public** **static** **void** main(String args[]){

**try**{

**int** a[]=**new** **int**[5];

    a[5]=30/0;

   }

**catch**(ArithmeticException e){System.out.println("task1 is completed");}

**catch**(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}

**catch**(Exception e){System.out.println("common task completed");}

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

 }

}

**Example 2**

**class** TestMultipleCatchBlock1{

**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...");

 }

}

# Java Nested try block

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

### Why use nested try block

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

### Syntax:

....

**try**

{

    statement 1;

    statement 2;

**try**

    {

        statement 1;

        statement 2;

    }

**catch**(Exception e)

    {

    }

}

**catch**(Exception e)

{

}

....

## Java nested try example

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

**class** Excep6{

**public** **static** **void** main(String args[]){

**try**{

**try**{

     System.out.println("going to divide");

**int** b =39/0;

    }**catch**(ArithmeticException e){System.out.println(e);}

**try**{

**int** a[]=**new** **int**[5];

    a[5]=4;

    }**catch**(ArrayIndexOutOfBoundsException e){System.out.println(e);}

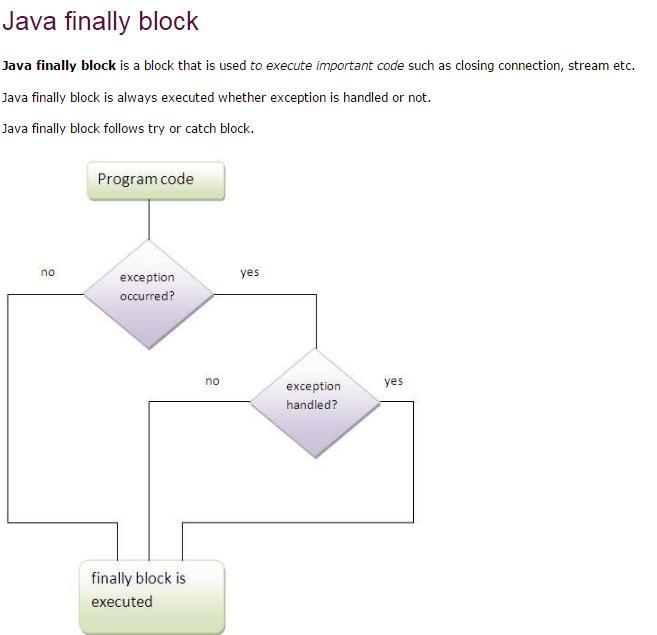
    System.out.println("other statement);

  }**catch**(Exception e){System.out.println("handeled");}

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

 }

}



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**.

**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...");

  }

}

Case 2

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

1. **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...");

  }

}

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...");

  }

}

# 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.

**throw** exception;

Let's see the example of throw IOException.

**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.

**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...");

  }

}

# 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 programmers 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.

**import** java.io.IOException;

**class** Testthrows1{

**void** m()**throws** IOException{

**throw** **new** IOException("device error");//checked exception

  }

**void** n()**throws** IOException{

    m();

  }

**void** p(){

**try**{

    n();

   }**catch**(Exception e){System.out.println("exception handled");}

  }

**public** **static** **void** main(String args[]){

   Testthrows1 obj=**new** Testthrows1();

   obj.p();

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

  }

}

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...");

  }

}

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...");

  }

}

---

