# Chapter 05 : Java Package and Exceptions Handling

Part 01 : Packages

A **java package** is a group of similar types of classes, interfaces and sub-packages.

Package in java can be categorized in two form, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Here, we will have the detailed learning of creating and using user-defined packages.

## Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision



## Simple example of java package

The **package keyword** is used to create a package in java.

1. //save as Simple.java
2. **package** mypack;
3. **public** **class** Simple{
4. **public** **static** **void** main(String args[]){
5. System.out.println("Welcome to package");
6. }
7. }

## How to compile java package

If you are not using any IDE, you need to follow the **syntax** given below:

1. javac -d directory javafilename

For **example**

1. javac -d . Simple.java

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

## How to run java package program

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

|  |
| --- |
| ****To Compile:**** javac -d . Simple.java |
| ****To Run:**** java mypack.Simple |

Output:Welcome to package

|  |
| --- |
| The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder. |

## How to access package from another package?

There are three ways to access the package from outside the package.

1. import package.\*;
2. import package.classname;
3. fully qualified name.

#### 1) Using packagename.\*

If you use package.\* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

## Example of package that import the packagename.\*

1. //save by A.java
2. **package** pack;
3. **public** **class** A{
4. **public** **void** msg(){System.out.println("Hello");}
5. }
6. //save by B.java
7. **package** mypack;
8. **import** pack.\*;
10. **class** B{
11. **public** **static** **void** main(String args[]){
12. A obj = **new** A();
13. obj.msg();
14. }
15. }

Output:Hello

#### 2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

## Example of package by import package.classname

1. //save by A.java
3. **package** pack;
4. **public** **class** A{
5. **public** **void** msg(){System.out.println("Hello");}
6. }
7. //save by B.java
8. **package** mypack;
9. **import** pack.A;
11. **class** B{
12. **public** **static** **void** main(String args[]){
13. A obj = **new** A();
14. obj.msg();
15. }
16. }

Output:Hello

#### 3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

## Example of package by import fully qualified name

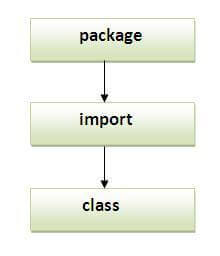
1. //save by A.java
2. **package** pack;
3. **public** **class** A{
4. **public** **void** msg(){System.out.println("Hello");}
5. }
6. //save by B.java
7. **package** mypack;
8. **class** B{
9. **public** **static** **void** main(String args[]){
10. pack.A obj = **new** pack.A();//using fully qualified name
11. obj.msg();
12. }
13. }

Output:Hello

#### Note: If you import a package, subpackages will not be imported.

If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

#### Note: Sequence of the program must be package then import then class.



## Subpackage in java

Package inside the package is called the **subpackage**. It should be created **to categorize the package further**.

Let's take an example, Sun Microsystem has definded a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on. So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on.

#### The standard of defining package is domain.company.package e.g. com.javatpoint.bean or org.sssit.dao.

### Example of Subpackage

1. **package** com.javatpoint.core;
2. **class** Simple{
3. **public** **static** **void** main(String args[]){
4. System.out.println("Hello subpackage");
5. }
6. }

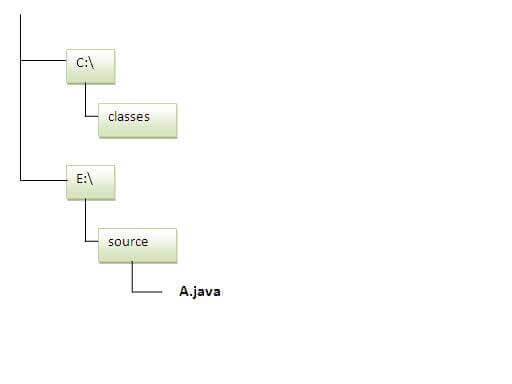
|  |
| --- |
| ****To Compile:**** javac -d . Simple.java |
| ****To Run:**** java com.javatpoint.core.Simple |

Output:Hello subpackage

## How to send the class file to another directory or drive?

There is a scenario, I want to put the class file of A.java source file in classes folder of c: drive. For example:

:



1. //save as Simple.java
2. **package** mypack;
3. **public** **class** Simple{
4. **public** **static** **void** main(String args[]){
5. System.out.println("Welcome to package");
6. }
7. }

### To Compile:

****e:\sources> javac -d c:\classes Simple.java****

### To Run:

|  |
| --- |
| To run this program from e:\source directory, you need to set classpath of the directory where the class file resides. |
| ****e:\sources> set classpath=c:\classes;.;**** |
| ****e:\sources> java mypack.Simple**** |

### Another way to run this program by -classpath switch of java:

The -classpath switch can be used with javac and java tool.

To run this program from e:\source directory, you can use -classpath switch of java that tells where to look for class file. For example:

****e:\sources> java -classpath c:\classes mypack.Simple****

Output:Welcome to package

### Ways to load the class files or jar files

|  |
| --- |
| There are two ways to load the class files temporary and permanent. |

* Temporary
  + By setting the classpath in the command prompt
  + By -classpath switch
* Permanent
  + By setting the classpath in the environment variables
  + By creating the jar file, that contains all the class files, and copying the jar file in the jre/lib/ext folder.

#### Rule: There can be only one public class in a java source file and it must be saved by the public class name.

1. //save as C.java otherwise Compilte Time Error
3. **class** A{}
4. **class** B{}
5. **public** **class** C{}

### How to put two public classes in a package?

|  |
| --- |
| If you want to put two public classes in a package, have two java source files containing one public class, but keep the package name same. For example: |

1. //save as A.java
3. **package** javatpoint;
4. **public** **class** A{}
5. //save as B.java
7. **package** javatpoint;
8. **public** **class** B{}

# Java Static Import

The static import feature of Java 5 facilitate the java programmer to access any static member of a class directly. There is no need to qualify it by the class name.

## Advantage of static import:

* Less coding is required if you have access any static member of a class oftenly.

## Disadvantage of static import:

* If you overuse the static import feature, it makes the program unreadable and unmaintainable.

### Simple Example of static import

1. **import** **static** java.lang.System.\*;
2. **class** StaticImportExample{
3. **public** **static** **void** main(String args[]){
5. out.println("Hello");//Now no need of System.out
6. out.println("Java");
8. }
9. }

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=StaticImportExample" \t "https://www.javatpoint.com/_blank)**

Output:Hello

Java

# Access Modifiers in Java

There are two types of modifiers in Java: ****access modifiers**** and ****non-access modifiers****.

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:

1. ****Private****: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. ****Default****: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. ****Protected****: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. ****Public****: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc. Here, we are going to learn the access modifiers only.

### Understanding Java Access Modifiers

Let's understand the access modifiers in Java by a simple table.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Access Modifier** | **within class** | **within package** | **outside package by subclass only** | **outside package** |
| **Private** | Y | N | N | N |
| **Default** | Y | Y | N | N |
| **Protected** | Y | Y | Y | N |
| **Public** | Y | Y | Y | Y |

### Private

The private access modifier is accessible only within the class.

****Simple example of private access modifier****

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is a compile-time error.

1. **class** A{
2. **private** **int** data=40;
3. **private** **void** msg(){System.out.println("Hello java");}
4. }
6. **public** **class** Simple{
7. **public** **static** **void** main(String args[]){
8. A obj=**new** A();
9. System.out.println(obj.data);//Compile Time Error
10. obj.msg();//Compile Time Error
11. }
12. }

### Role of Private Constructor

If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

1. **class** A{
2. **private** A(){}//private constructor
3. **void** msg(){System.out.println("Hello java");}
4. }
5. **public** **class** Simple{
6. **public** **static** **void** main(String args[]){
7. A obj=**new** A();//Compile Time Error
8. }
9. }

#### Note: A class cannot be private or protected except nested class.

### 2) Default

If you don't use any modifier, it is treated as ****default**** by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

****Example of default access modifier****

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

1. //save by A.java
2. **package** pack;
3. **class** A{
4. **void** msg(){System.out.println("Hello");}
5. }
6. //save by B.java
7. **package** mypack;
8. **import** pack.\*;
9. **class** B{
10. **public** **static** **void** main(String args[]){
11. A obj = **new** A();//Compile Time Error
12. obj.msg();//Compile Time Error
13. }
14. }

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

### 3) Protected

The **protected access modifier** is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

It provides more accessibility than the default modifer.

****Example of protected access modifier****

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

1. //save by A.java
2. **package** pack;
3. **public** **class** A{
4. **protected** **void** msg(){System.out.println("Hello");}
5. }
6. //save by B.java
7. **package** mypack;
8. **import** pack.\*;
10. **class** B **extends** A{
11. **public** **static** **void** main(String args[]){
12. B obj = **new** B();
13. obj.msg();
14. }
15. }

Output:Hello

### 4) Public

The ****public access modifier**** is accessible everywhere. It has the widest scope among all other modifiers.

****Example of public access modifier****

1. //save by A.java
3. **package** pack;
4. **public** **class** A{
5. **public** **void** msg(){System.out.println("Hello");}
6. }
7. //save by B.java
9. **package** mypack;
10. **import** pack.\*;
12. **class** B{
13. **public** **static** **void** main(String args[]){
14. A obj = **new** A();
15. obj.msg();
16. }
17. }

Output:Hello

### Java Access Modifiers with Method Overriding

If you are overriding any method, overridden method (i.e. declared in subclass) must not be more restrictive.

1. **class** A{
2. **protected** **void** msg(){System.out.println("Hello java");}
3. }
5. **public** **class** Simple **extends** A{
6. **void** msg(){System.out.println("Hello java");}//C.T.Error
7. **public** **static** **void** main(String args[]){
8. Simple obj=**new** Simple();
9. obj.msg();
10. }
11. }

The default modifier is more restrictive than protected. That is why, there is a compile-time error.

Part - 2

# Exception Handling in Java

The ****Exception Handling in Java**** is one of the powerful mechanism to handle the runtime errors so that the normal flow of the application can be maintained.

In this tutorial, we will learn about Java exceptions, it's types, 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 need to handle exceptions. Let's consider 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 a Java program and an exception occurs at statement 5; the rest of the code will not be executed, i.e., statements 6 to 10 will not be executed. However, when we perform exception handling, the rest of the statements will be executed. That is why we use exception handling in [Java](https://www.javatpoint.com/java-tutorial).

## Hierarchy of Java Exception classes

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



### Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

1. Checked Exception
2. Unchecked Exception
3. Error

## Difference between Checked and Unchecked Exceptions

### 1) Checked Exception

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

### 2) Unchecked Exception

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

### 3) Error

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

## Java Exception Keywords

Java provides five keywords that are used to handle the exception. The following table describes each.

|  |  |
| --- | --- |
| **Keyword** | **Description** |
| try | The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally. |
| 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 necessary 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 specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature. |

## Java Exception Handling Example

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

****JavaExceptionExample.java****

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. }

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=JavaExceptionExample" \t "https://www.javatpoint.com/_blank)**

****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.

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](https://www.javatpoint.com/java-variables), performing any operation on the variable throws a NullPointerException.

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

### A scenario where NumberFormatException occurs

If the formatting of any variable or number is mismatched, it may result into NumberFormatException. Suppose we have a [string](https://www.javatpoint.com/java-string) variable that has characters; converting this variable into digit will cause NumberFormatException.

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

### 4) A scenario where ArrayIndexOutOfBoundsException occurs

When an array exceeds to it's size, the ArrayIndexOutOfBoundsException occurs. there may be other reasons to occur ArrayIndexOutOfBoundsException. Consider the following statements.

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 in the try block, the rest of the block code will not execute. So, it is recommended not to keep 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

1. **try**{
2. //code that may throw an exception
3. }**catch**(Exception\_class\_Name ref){}

### Syntax of try-finally block

1. **try**{
2. //code that may throw an exception
3. }**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.

## 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 the application programmer handles the exception, the normal flow of the application is maintained, i.e., rest of the code is executed.

## Problem without exception handling

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

### Example 1

****TryCatchExample1.java****

1. **public** **class** TryCatchExample1 {
3. **public** **static** **void** main(String[] args) {
5. **int** data=50/0; //may throw exception
7. System.out.println("rest of the code");
9. }
11. }

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample1" \t "https://www.javatpoint.com/_blank)**

****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 might be 100 lines of code after the exception. If the exception is not handled, all the code below the exception won't be executed.

## Solution by exception handling

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

### Example 2

****TryCatchExample2.java****

1. **public** **class** TryCatchExample2 {
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. }
16. }

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample2" \t "https://www.javatpoint.com/_blank)**

****Output:****

java.lang.ArithmeticException: / by zero

rest of the code

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.

****TryCatchExample3.java****

1. **public** **class** TryCatchExample3 {
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. }
16. }
18. }

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample3" \t "https://www.javatpoint.com/_blank)**

****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.

****TryCatchExample4.java****

1. **public** **class** TryCatchExample4 {
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. }
16. }

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample4" \t "https://www.javatpoint.com/_blank)**

****Output:****

java.lang.ArithmeticException: / by zero

rest of the code

### Example 5

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

****TryCatchExample5.java****

1. **public** **class** TryCatchExample5 {
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. }
16. }

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample5" \t "https://www.javatpoint.com/_blank)**

****Output:****

Can't divided by zero

### Example 6

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

****TryCatchExample6.java****

1. **public** **class** TryCatchExample6 {
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. }

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample6" \t "https://www.javatpoint.com/_blank)**

****Output:****

25

### Example 7

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

****TryCatchExample7.java****

1. **public** **class** TryCatchExample7 {
3. **public** **static** **void** main(String[] args) {
5. **try**
6. {
7. **int** data1=50/0; //may throw exception
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
16. }
17. System.out.println("rest of the code");
18. }
19. }

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample7" \t "https://www.javatpoint.com/_blank)**

****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).

****TryCatchExample8.java****

1. **public** **class** TryCatchExample8 {
3. **public** **static** **void** main(String[] args) {
4. **try**
5. {
6. **int** data=50/0; //may throw exception
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. }
17. }

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample8" \t "https://www.javatpoint.com/_blank)**

****Output:****

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

### Example 9

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

****TryCatchExample9.java****

1. **public** **class** TryCatchExample9 {
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. {
12. System.out.println(e);
13. }
14. System.out.println("rest of the code");
15. }
17. }

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample9" \t "https://www.javatpoint.com/_blank)**

****Output:****

java.lang.ArrayIndexOutOfBoundsException: 10

rest of the code

### Example 10

Let's see an example to handle checked exception.

****TryCatchExample10.java****

1. **import** java.io.FileNotFoundException;
2. **import** java.io.PrintWriter;
4. **public** **class** TryCatchExample10 {
6. **public** **static** **void** main(String[] args) {

9. PrintWriter pw;
10. **try** {
11. pw = **new** PrintWriter("jtp.txt"); //may throw exception
12. pw.println("saved");
13. }
14. // providing the checked exception handler
15. **catch** (FileNotFoundException e) {
17. System.out.println(e);
18. }
19. System.out.println("File saved successfully");
20. }
21. }

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample10" \t "https://www.javatpoint.com/_blank)**

****Output:****

File saved successfully

# 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.

### Flowchart of Multi-catch Block



### Example 1

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

****MultipleCatchBlock1.java****

1. **public** **class** MultipleCatchBlock1 {
3. **public** **static** **void** main(String[] args) {
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. }

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=MultipleCatchBlock1" \t "https://www.javatpoint.com/_blank)**

****Output:****

Arithmetic Exception occurs

rest of the code

### Example 2

****MultipleCatchBlock2.java****

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

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=MultipleCatchBlock2" \t "https://www.javatpoint.com/_blank)**

****Output:****

ArrayIndexOutOfBounds Exception occurs

rest of the code

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

****MultipleCatchBlock3.java****

1. **public** **class** MultipleCatchBlock3 {
3. **public** **static** **void** main(String[] args) {
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. }

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=MultipleCatchBlock3" \t "https://www.javatpoint.com/_blank)**

****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.

****MultipleCatchBlock4.java****

1. **public** **class** MultipleCatchBlock4 {
3. **public** **static** **void** main(String[] args) {
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. }

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=MultipleCatchBlock4" \t "https://www.javatpoint.com/_blank)**

****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).

****MultipleCatchBlock5.java****

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. }

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=MultipleCatchBlock5" \t "https://www.javatpoint.com/_blank)**

****Output:****

Compile-time error

# Java Nested try block

In Java, using a try block inside another try block is permitted. It is called as nested try block. Every statement that we enter a statement in try block, context of that exception is pushed onto the stack.

For example, the ****inner try block**** can be used to handle ****ArrayIndexOutOfBoundsException**** while the ****outer try block**** can handle the ****ArithemeticException**** (division by zero).

### 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. //main try block
3. **try**
4. {
5. statement 1;
6. statement 2;
7. //try catch block within another try block
8. **try**
9. {
10. statement 3;
11. statement 4;
12. //try catch block within nested try block
13. **try**
14. {
15. statement 5;
16. statement 6;
17. }
18. **catch**(Exception e2)
19. {
20. //exception message
21. }
23. }
24. **catch**(Exception e1)
25. {
26. //exception message
27. }
28. }
29. //catch block of parent (outer) try block
30. **catch**(Exception e3)
31. {
32. //exception message
33. }
34. ....

## Java Nested try Example

### Example 1

Let's see an example where we place a try block within another try block for two different exceptions.

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

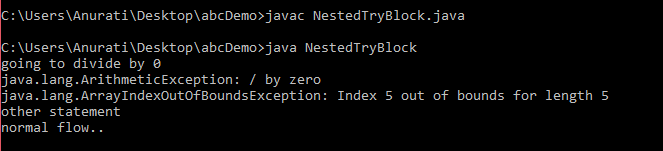
****NestedTryBlock.java****

1. **public** **class** NestedTryBlock{
2. **public** **static** **void** main(String args[]){
3. //outer try block
4. **try**{
5. //inner try block 1
6. **try**{
7. System.out.println("going to divide by 0");
8. **int** b =39/0;
9. }
10. //catch block of inner try block 1
11. **catch**(ArithmeticException e)
12. {
13. System.out.println(e);
14. }

17. //inner try block 2
18. **try**{
19. **int** a[]=**new** **int**[5];
21. //assigning the value out of array bounds
22. a[5]=4;
23. }
25. //catch block of inner try block 2
26. **catch**(ArrayIndexOutOfBoundsException e)
27. {
28. System.out.println(e);
29. }

32. System.out.println("other statement");
33. }
34. //catch block of outer try block
35. **catch**(Exception e)
36. {
37. System.out.println("handled the exception (outer catch)");
38. }
40. System.out.println("normal flow..");
41. }
42. }

****Output:****



When any try block does not have a catch block for a particular exception, then the catch block of the outer (parent) try block are checked for that exception, and if it matches, the catch block of outer try block is executed.

If none of the catch block specified in the code is unable to handle the exception, then the Java runtime system will handle the exception. Then it displays the system generated message for that exception.

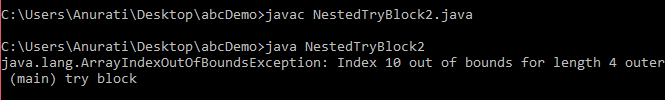
### Example 2

Let's consider the following example. Here the try block within nested try block (inner try block 2) do not handle the exception. The control is then transferred to its parent try block (inner try block 1). If it does not handle the exception, then the control is transferred to the main try block (outer try block) where the appropriate catch block handles the exception. It is termed as nesting.

****NestedTryBlock.java****

1. **public** **class** NestedTryBlock2 {
3. **public** **static** **void** main(String args[])
4. {
5. // outer (main) try block
6. **try** {
8. //inner try block 1
9. **try** {
11. // inner try block 2
12. **try** {
13. **int** arr[] = { 1, 2, 3, 4 };
15. //printing the array element out of its bounds
16. System.out.println(arr[10]);
17. }
19. // to handles ArithmeticException
20. **catch** (ArithmeticException e) {
21. System.out.println("Arithmetic exception");
22. System.out.println(" inner try block 2");
23. }
24. }
26. // to handle ArithmeticException
27. **catch** (ArithmeticException e) {
28. System.out.println("Arithmetic exception");
29. System.out.println("inner try block 1");
30. }
31. }
33. // to handle ArrayIndexOutOfBoundsException
34. **catch** (ArrayIndexOutOfBoundsException e4) {
35. System.out.print(e4);
36. System.out.println(" outer (main) try block");
37. }
38. **catch** (Exception e5) {
39. System.out.print("Exception");
40. System.out.println(" handled in main try-block");
41. }
42. }
43. }

****Output:****



# Java finally block

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

Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.

The finally block follows the try-catch block.

### Flowchart of finally block



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

## Why use Java finally block?

* finally block in Java can be used to put "****cleanup****" code such as closing a file, closing connection, etc.
* The important statements to be printed can be placed in the finally block.

## Usage of Java finally

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

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

### Case 1: When an exception does not occur

Let's see the below example where the Java program does not throw any exception, and the finally block is executed after the try block.

****TestFinallyBlock.java****

1. **class** TestFinallyBlock {
2. **public** **static** **void** main(String args[]){
3. **try**{
4. //below code do not throw any exception
5. **int** data=25/5;
6. System.out.println(data);
7. }
8. //catch won't be executed
9. **catch**(NullPointerException e){
10. System.out.println(e);
11. }
12. //executed regardless of exception occurred or not
13. **finally** {
14. System.out.println("finally block is always executed");
15. }
17. System.out.println("rest of phe code...");
18. }
19. }

****Output:****



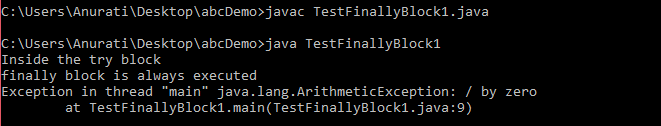
### Case 2: When an exception occurr but not handled by the catch block

Let's see the the fillowing example. Here, the code throws an exception however the catch block cannot handle it. Despite this, the finally block is executed after the try block and then the program terminates abnormally.

****TestFinallyBlock1.java****

1. **public** **class** TestFinallyBlock1{
2. **public** **static** **void** main(String args[]){
4. **try** {
6. System.out.println("Inside the try block");
8. //below code throws divide by zero exception
9. **int** data=25/0;
10. System.out.println(data);
11. }
12. //cannot handle Arithmetic type exception
13. //can only accept Null Pointer type exception
14. **catch**(NullPointerException e){
15. System.out.println(e);
16. }
18. //executes regardless of exception occured or not
19. **finally** {
20. System.out.println("finally block is always executed");
21. }
23. System.out.println("rest of the code...");
24. }
25. }

****Output:****



### Case 3: When an exception occurs and is handled by the catch block

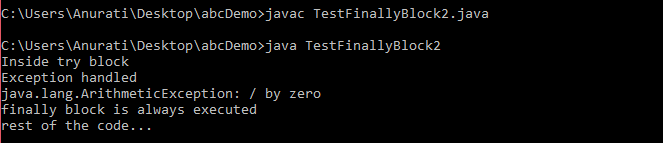
****Example:****

Let's see the following example where the Java code throws an exception and the catch block handles the exception. Later the finally block is executed after the try-catch block. Further, the rest of the code is also executed normally.

****TestFinallyBlock2.java****

1. **public** **class** TestFinallyBlock2{
2. **public** **static** **void** main(String args[]){
4. **try** {
6. System.out.println("Inside try block");
8. //below code throws divide by zero exception
9. **int** data=25/0;
10. System.out.println(data);
11. }
13. //handles the Arithmetic Exception / Divide by zero exception
14. **catch**(ArithmeticException e){
15. System.out.println("Exception handled");
16. System.out.println(e);
17. }
19. //executes regardless of exception occured or not
20. **finally** {
21. System.out.println("finally block is always executed");
22. }
24. System.out.println("rest of the code...");
25. }
26. }

****Output:****



#### 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 the program exits (either by calling System.exit() or by causing a fatal error that causes the process to abort).

# Java finally block

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

Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.

The finally block follows the try-catch block.

### Flowchart of finally block



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

## Why use Java finally block?

* finally block in Java can be used to put "****cleanup****" code such as closing a file, closing connection, etc.
* The important statements to be printed can be placed in the finally block.

## Usage of Java finally

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

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

### Case 1: When an exception does not occur

Let's see the below example where the Java program does not throw any exception, and the finally block is executed after the try block.

****TestFinallyBlock.java****

1. **class** TestFinallyBlock {
2. **public** **static** **void** main(String args[]){
3. **try**{
4. //below code do not throw any exception
5. **int** data=25/5;
6. System.out.println(data);
7. }
8. //catch won't be executed
9. **catch**(NullPointerException e){
10. System.out.println(e);
11. }
12. //executed regardless of exception occurred or not
13. **finally** {
14. System.out.println("finally block is always executed");
15. }
17. System.out.println("rest of phe code...");
18. }
19. }

****Output:****



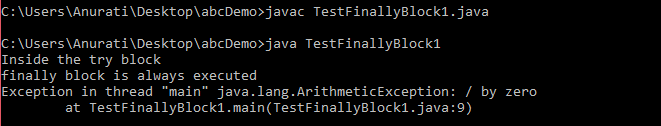
### Case 2: When an exception occurr but not handled by the catch block

Let's see the the fillowing example. Here, the code throws an exception however the catch block cannot handle it. Despite this, the finally block is executed after the try block and then the program terminates abnormally.

****TestFinallyBlock1.java****

1. **public** **class** TestFinallyBlock1{
2. **public** **static** **void** main(String args[]){
4. **try** {
6. System.out.println("Inside the try block");
8. //below code throws divide by zero exception
9. **int** data=25/0;
10. System.out.println(data);
11. }
12. //cannot handle Arithmetic type exception
13. //can only accept Null Pointer type exception
14. **catch**(NullPointerException e){
15. System.out.println(e);
16. }
18. //executes regardless of exception occured or not
19. **finally** {
20. System.out.println("finally block is always executed");
21. }
23. System.out.println("rest of the code...");
24. }
25. }

****Output:****



### Case 3: When an exception occurs and is handled by the catch block

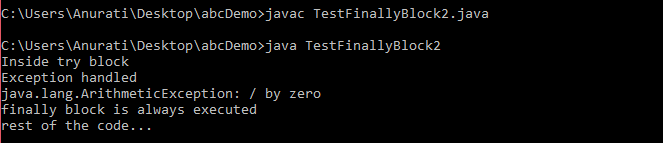
****Example:****

Let's see the following example where the Java code throws an exception and the catch block handles the exception. Later the finally block is executed after the try-catch block. Further, the rest of the code is also executed normally.

****TestFinallyBlock2.java****

1. **public** **class** TestFinallyBlock2{
2. **public** **static** **void** main(String args[]){
4. **try** {
6. System.out.println("Inside try block");
8. //below code throws divide by zero exception
9. **int** data=25/0;
10. System.out.println(data);
11. }
13. //handles the Arithmetic Exception / Divide by zero exception
14. **catch**(ArithmeticException e){
15. System.out.println("Exception handled");
16. System.out.println(e);
17. }
19. //executes regardless of exception occured or not
20. **finally** {
21. System.out.println("finally block is always executed");
22. }
24. System.out.println("rest of the code...");
25. }
26. }

****Output:****



#### 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 the program exits (either by calling System.exit() or by causing a fatal error that causes the process to abort).

# Java throw Exception

In Java, exceptions allows us to write good quality codes where the errors are checked at the compile time instead of runtime and we can create custom exceptions making the code recovery and debugging easier.

## Java throw keyword

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

We specify the ****exception**** object which is to be thrown. The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.

We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception. We will discuss custom exceptions later in this section.

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

We can also define our own set of conditions and throw an exception explicitly using throw keyword. For example, we can throw ArithmeticException if we divide a number by another number. Here, we just need to set the condition and throw exception using throw keyword.

The syntax of the Java throw keyword is given below.

throw Instance i.e.,

1. **throw** **new** exception\_class("error message");

Let's see the example of throw IOException.

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

Where the Instance must be of type Throwable or subclass of Throwable. For example, Exception is the sub class of Throwable and the user-defined exceptions usually extend the Exception class.

## Java throw keyword Example

### Example 1: Throwing Unchecked Exception

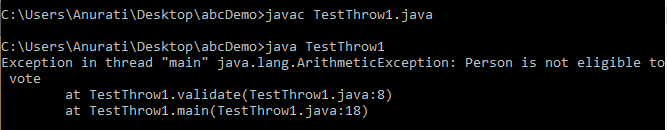
In this example, we have created a method named validate() that accepts an integer as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

****TestThrow1.java****

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. //function to check if person is eligible to vote or not
3. **public** **static** **void** validate(**int** age) {
4. **if**(age<18) {
5. //throw Arithmetic exception if not eligible to vote
6. **throw** **new** ArithmeticException("Person is not eligible to vote");
7. }
8. **else** {
9. System.out.println("Person is eligible to vote!!");
10. }
11. }
12. //main method
13. **public** **static** **void** main(String args[]){
14. //calling the function
15. validate(13);
16. System.out.println("rest of the code...");
17. }
18. }

****Output:****



The above code throw an unchecked exception. Similarly, we can also throw unchecked and user defined exceptions.

#### Note: If we throw unchecked exception from a method, it is must to handle the exception or declare in throws clause.

If we throw a checked exception using throw keyword, it is must to handle the exception using catch block or the method must declare it using throws declaration.

### Example 2: Throwing Checked Exception

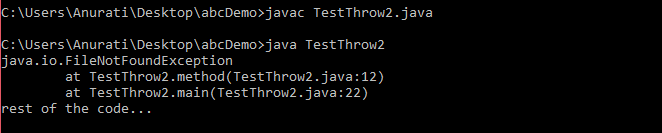
#### Note: Every subclass of Error and RuntimeException is an unchecked exception in Java. A checked exception is everything else under the Throwable class.

****TestThrow2.java****

1. **import** java.io.\*;
3. **public** **class** TestThrow2 {
5. //function to check if person is eligible to vote or not
6. **public** **static** **void** method() **throws** FileNotFoundException {
8. FileReader file = **new** FileReader("C:\\Users\\Anurati\\Desktop\\abc.txt");
9. BufferedReader fileInput = **new** BufferedReader(file);

12. **throw** **new** FileNotFoundException();
14. }
15. //main method
16. **public** **static** **void** main(String args[]){
17. **try**
18. {
19. method();
20. }
21. **catch** (FileNotFoundException e)
22. {
23. e.printStackTrace();
24. }
25. System.out.println("rest of the code...");
26. }
27. }

****Output:****



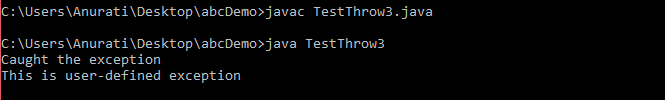
### Example 3: Throwing User-defined Exception

exception is everything else under the Throwable class.

****TestThrow3.java****

1. // class represents user-defined exception
2. **class** UserDefinedException **extends** Exception
3. {
4. **public** UserDefinedException(String str)
5. {
6. // Calling constructor of parent Exception
7. **super**(str);
8. }
9. }
10. // Class that uses above MyException
11. **public** **class** TestThrow3
12. {
13. **public** **static** **void** main(String args[])
14. {
15. **try**
16. {
17. // throw an object of user defined exception
18. **throw** **new** UserDefinedException("This is user-defined exception");
19. }
20. **catch** (UserDefinedException ude)
21. {
22. System.out.println("Caught the exception");
23. // Print the message from MyException object
24. System.out.println(ude.getMessage());
25. }
26. }
27. }

****Output:****



# 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 the normal flow of the program 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 checking the code before it 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 our control so we can correct our code.
* ****error:**** beyond our control. For example, we 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).

Backward Skip 10sPlay VideoForward Skip 10s

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.

****Testthrows1.java****

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. }

**[Test it Now](https://www.javatpoint.com/opr/test.jsp?filename=Testthrows1" \t "https://www.javatpoint.com/_blank)**

****Output:****

exception handled

normal flow...

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

****There are two cases:****

1. ****Case 1:**** We have caught the exception i.e. we have handled the exception using try/catch block.
2. ****Case 2:**** We have declared the exception i.e. specified throws keyword with the method.

### Case 1: Handle Exception Using try-catch block

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

****Testthrows2.java****

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");}
14. System.out.println("normal flow...");
15. }
16. }

**[Test it Now](https://www.javatpoint.com/opr/test.jsp?filename=Testthrows2" \t "https://www.javatpoint.com/_blank)**

****Output:****

exception handled

normal flow...

### Case 2: Declare Exception

* In case we declare the exception, if exception does not occur, the code will be executed fine.
* In case we declare the exception and the exception occurs, it will be thrown at runtime because ****throws**** does not handle the exception.

Let's see examples for both the scenario.

**A) If exception does not occur**

****Testthrows3.java****

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

**[Test it Now](https://www.javatpoint.com/opr/test.jsp?filename=Testthrows3" \t "https://www.javatpoint.com/_blank)**

****Output:****

device operation performed

normal flow...

**B) If exception occurs**

****Testthrows4.java****

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

**[Test it Now](https://www.javatpoint.com/opr/test.jsp?filename=Testthrows4" \t "https://www.javatpoint.com/_blank)**

****Output:****

IMG_256

### Difference between throw and throws

# Difference between throw and throws in Java

The throw and throws is the concept of exception handling where the throw keyword throw the exception explicitly from a method or a block of code whereas the throws keyword is used in signature of the method.

There are many differences between [throw](https://www.javatpoint.com/throw-keyword) and [throws](https://www.javatpoint.com/throws-keyword-and-difference-between-throw-and-throws) keywords. A list of differences between throw and throws are given below:

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr. no.** | **Basis of Differences** | **throw** | **throws** |
| 1. | Definition | Java throw keyword is used throw an exception explicitly in the code, inside the function or the block of code. | Java throws keyword is used in the method signature to declare an exception which might be thrown by the function while the execution of the code. |
| 2. | Type of exception Using throw keyword, we can only propagate unchecked exception i.e., the checked exception cannot be propagated using throw only. | Using throws keyword, we can declare both checked and unchecked exceptions. However, the throws keyword can be used to propagate checked exceptions only. |  |
| 3. | Syntax | The throw keyword is followed by an instance of Exception to be thrown. | The throws keyword is followed by class names of Exceptions to be thrown. |
| 4. | Declaration | throw is used within the method. | throws is used with the method signature. |
| 5. | Internal implementation | We are allowed to throw only one exception at a time i.e. we cannot throw multiple exceptions. | We can declare multiple exceptions using throws keyword that can be thrown by the method. For example, main() throws IOException, SQLException. |

## Java throw Example

****TestThrow.java****

1. **public** **class** TestThrow {
2. //defining a method
3. **public** **static** **void** checkNum(**int** num) {
4. **if** (num < 1) {
5. **throw** **new** ArithmeticException("\nNumber is negative, cannot calculate square");
6. }
7. **else** {
8. System.out.println("Square of " + num + " is " + (num\*num));
9. }
10. }
11. //main method
12. **public** **static** **void** main(String[] args) {
13. TestThrow obj = **new** TestThrow();
14. obj.checkNum(-3);
15. System.out.println("Rest of the code..");
16. }
17. }

****Output:****

PlayNext

Unmute

Current TimeÂ 0:00

/

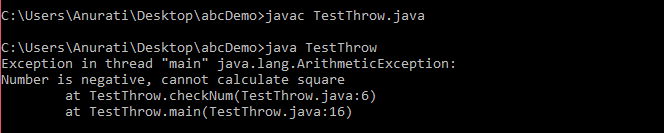
DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

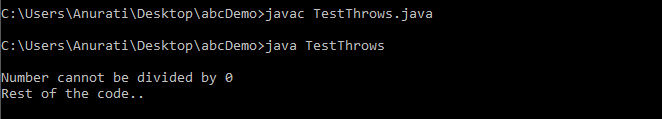


## Java throws Example

****TestThrows.java****

1. **public** **class** TestThrows {
2. //defining a method
3. **public** **static** **int** divideNum(**int** m, **int** n) **throws** ArithmeticException {
4. **int** div = m / n;
5. **return** div;
6. }
7. //main method
8. **public** **static** **void** main(String[] args) {
9. TestThrows obj = **new** TestThrows();
10. **try** {
11. System.out.println(obj.divideNum(45, 0));
12. }
13. **catch** (ArithmeticException e){
14. System.out.println("\nNumber cannot be divided by 0");
15. }
17. System.out.println("Rest of the code..");
18. }
19. }

****Output:****

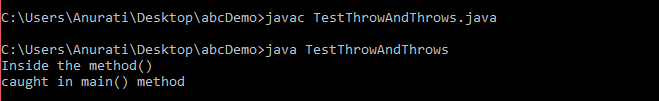


## Java throw and throws Example

****TestThrowAndThrows.java****

1. **public** **class** TestThrowAndThrows
2. {
3. // defining a user-defined method
4. // which throws ArithmeticException
5. **static** **void** method() **throws** ArithmeticException
6. {
7. System.out.println("Inside the method()");
8. **throw** **new** ArithmeticException("throwing ArithmeticException");
9. }
10. //main method
11. **public** **static** **void** main(String args[])
12. {
13. **try**
14. {
15. method();
16. }
17. **catch**(ArithmeticException e)
18. {
19. System.out.println("caught in main() method");
20. }
21. }
22. }

****Output:****



Next Topic

# Difference between final, finally and finalize

The final, finally, and finalize are keywords in Java that are used in exception handling. Each of these keywords has a different functionality. The basic difference between final, finally and finalize is that the **[final](https://www.javatpoint.com/final-keyword)** is an access modifier, **[finally](https://www.javatpoint.com/finally-block-in-exception-handling)** is the block in Exception Handling and **[finalize](https://www.javatpoint.com/java-object-finalize-method)** is the method of object class.

Along with this, there are many differences between final, finally and finalize. A list of differences between final, finally and finalize are given below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sr. no.** | **Key** | **final** | **finally** | **finalize** |
| 1. | Definition | final is the keyword and access modifier which is used to apply restrictions on a class, method or variable. | finally is the block in Java Exception Handling to execute the important code whether the exception occurs or not. | finalize is the method in Java which is used to perform clean up processing just before object is garbage collected. |
| 2. | Applicable to | Final keyword is used with the classes, methods and variables. | Finally block is always related to the try and catch block in exception handling. | finalize() method is used with the objects. |
| 3. | Functionality | (1) Once declared, final variable becomes constant and cannot be modified. (2) final method cannot be overridden by sub class. (3) final class cannot be inherited. | (1) finally block runs the important code even if exception occurs or not. (2) finally block cleans up all the resources used in try block | finalize method performs the cleaning activities with respect to the object before its destruction. |
| 4. | Execution | Final method is executed only when we call it. | Finally block is executed as soon as the try-catch block is executed.  It's execution is not dependant on the exception. | finalize method is executed just before the object is destroyed. |

## Java final Example

Let's consider the following example where we declare final variable age. Once declared it cannot be modified.

****FinalExampleTest.java****

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

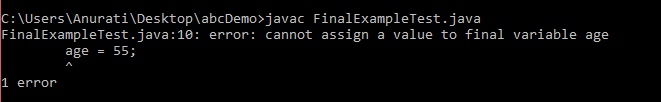
Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

1. **public** **class** FinalExampleTest {
2. //declaring final variable
3. **final** **int** age = 18;
4. **void** display() {
6. // reassigning value to age variable
7. // gives compile time error
8. age = 55;
9. }
11. **public** **static** **void** main(String[] args) {
13. FinalExampleTest obj = **new** FinalExampleTest();
14. // gives compile time error
15. obj.display();
16. }
17. }

****Output:****



In the above example, we have declared a variable final. Similarly, we can declare the methods and classes final using the final keyword.

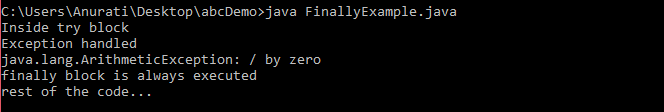
## Java finally Example

Let's see the below example where the Java code throws an exception and the catch block handles that exception. Later the finally block is executed after the try-catch block. Further, the rest of the code is also executed normally.

****FinallyExample.java****

1. **public** **class** FinallyExample {
2. **public** **static** **void** main(String args[]){
3. **try** {
4. System.out.println("Inside try block");
5. // below code throws divide by zero exception
6. **int** data=25/0;
7. System.out.println(data);
8. }
9. // handles the Arithmetic Exception / Divide by zero exception
10. **catch** (ArithmeticException e){
11. System.out.println("Exception handled");
12. System.out.println(e);
13. }
14. // executes regardless of exception occurred or not
15. **finally** {
16. System.out.println("finally block is always executed");
17. }
18. System.out.println("rest of the code...");
19. }
20. }

****Output:****

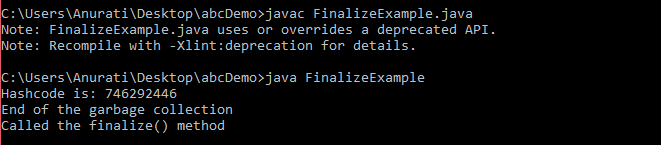


## Java finalize Example

****FinalizeExample.java****

1. **public** **class** FinalizeExample {
2. **public** **static** **void** main(String[] args)
3. {
4. FinalizeExample obj = **new** FinalizeExample();
5. // printing the hashcode
6. System.out.println("Hashcode is: " + obj.hashCode());
7. obj = **null**;
8. // calling the garbage collector using gc()
9. System.gc();
10. System.out.println("End of the garbage collection");
11. }
12. // defining the finalize method
13. **protected** **void** finalize()
14. {
15. System.out.println("Called the finalize() method");
16. }
17. }

****Output:****



# Java Custom Exception

In Java, we can create our own exceptions that are derived classes of the Exception class. Creating our own Exception is known as custom exception or user-defined exception. Basically, Java custom exceptions are used to customize the exception according to user need.

Consider the example 1 in which InvalidAgeException class extends the Exception class.

Using the custom exception, we can have your own exception and message. Here, we have passed a string to the constructor of superclass i.e. Exception class that can be obtained using getMessage() method on the object we have created.

In this section, we will learn how custom exceptions are implemented and used in Java programs.

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

## Why use custom exceptions?

Java exceptions cover almost all the general type of exceptions that may occur in the programming. However, we sometimes need to create custom exceptions.

Following are few of the reasons to use custom exceptions:

* To catch and provide specific treatment to a subset of existing Java exceptions.
* Business logic exceptions: These are the exceptions related to business logic and workflow. It is useful for the application users or the developers to understand the exact problem.

In order to create custom exception, we need to extend Exception class that belongs to java.lang package.

Consider the following example, where we create a custom exception named WrongFileNameException:

1. **public** **class** WrongFileNameException **extends** Exception {
2. **public** WrongFileNameException(String errorMessage) {
3. **super**(errorMessage);
4. }
5. }

#### Note: We need to write the constructor that takes the String as the error message and it is called parent class constructor.

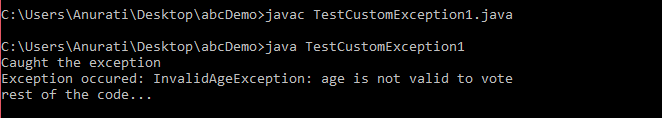
### Example 1:

Let's see a simple example of Java custom exception. In the following code, constructor of InvalidAgeException takes a string as an argument. This string is passed to constructor of parent class Exception using the super() method. Also the constructor of Exception class can be called without using a parameter and calling super() method is not mandatory.

****TestCustomException1.java****

1. // class representing custom exception
2. **class** InvalidAgeException  **extends** Exception
3. {
4. **public** InvalidAgeException (String str)
5. {
6. // calling the constructor of parent Exception
7. **super**(str);
8. }
9. }
11. // class that uses custom exception InvalidAgeException
12. **public** **class** TestCustomException1
13. {
15. // method to check the age
16. **static** **void** validate (**int** age) **throws** InvalidAgeException{
17. **if**(age < 18){
19. // throw an object of user defined exception
20. **throw** **new** InvalidAgeException("age is not valid to vote");
21. }
22. **else** {
23. System.out.println("welcome to vote");
24. }
25. }
27. // main method
28. **public** **static** **void** main(String args[])
29. {
30. **try**
31. {
32. // calling the method
33. validate(13);
34. }
35. **catch** (InvalidAgeException ex)
36. {
37. System.out.println("Caught the exception");
39. // printing the message from InvalidAgeException object
40. System.out.println("Exception occured: " + ex);
41. }
43. System.out.println("rest of the code...");
44. }
45. }

****Output:****



### Example 2:

****TestCustomException2.java****

1. // class representing custom exception
2. **class** MyCustomException **extends** Exception
3. {
5. }
7. // class that uses custom exception MyCustomException
8. **public** **class** TestCustomException2
9. {
10. // main method
11. **public** **static** **void** main(String args[])
12. {
13. **try**
14. {
15. // throw an object of user defined exception
16. **throw** **new** MyCustomException();
17. }
18. **catch** (MyCustomException ex)
19. {
20. System.out.println("Caught the exception");
21. System.out.println(ex.getMessage());
22. }
24. System.out.println("rest of the code...");
25. }
26. }

****Output:****

