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:

statement 1;

statement 2;

statement 3;

statement 4;

statement 5;//exception occurs

statement 6;

statement 7;

statement 8;

statement 9;

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).

Do You Know?

|  |
| --- |
| What is the difference between checked and unchecked exceptions?  What happens behind the code int data=50/0;?  Why use multiple catch block?  Is there any possibility when the finally block is not executed?  What is exception propagation?  What is the difference between the throw and throws keyword?  What are the 4 rules for using exception handling with method overriding? |

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:

Checked Exception

Unchecked Exception

Error



AD

Difference between Checked and Unchecked Exceptions

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**

**public** **class** JavaExceptionExample{

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

**try**{

      //code that may raise exception

**int** data=100/0;

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

   //rest code of the program

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

  }

}

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

**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.

AD

Common Scenarios of Java Exceptions

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

A scenario where ArithmeticException occurs

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

**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.

String s=**null**;

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

3) 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.

String s="abc";

**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.

AD

**int** a[]=**new** **int**[5];

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

**try**{

//code that may throw an exception

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

Syntax of try-finally block

**try**{

//code that may throw an exception

}**finally**{}

Java catch block

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

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

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.

AD

Problem without exception handling

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

Example 1

**TryCatchExample1.java**

**public** **class** TryCatchExample1 {

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

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

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

    }

}

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

**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**

**public** **class** TryCatchExample2 {

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

**try**

        {

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

        }

            //handling the exception

**catch**(ArithmeticException e)

        {

            System.out.println(e);

        }

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

    }

}

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

**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.

AD

**TryCatchExample3.java**

**public** **class** TryCatchExample3 {

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

**try**

        {

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

                         // if exception occurs, the remaining statement will not exceute

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

        }

             // handling the exception

**catch**(ArithmeticException e)

        {

            System.out.println(e);

        }

    }

}

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

**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**

AD

**public** **class** TryCatchExample4 {

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

**try**

        {

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

        }

            // handling the exception by using Exception class

**catch**(Exception e)

        {

            System.out.println(e);

        }

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

    }

}

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

**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**

**public** **class** TryCatchExample5 {

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

**try**

        {

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

        }

             // handling the exception

**catch**(Exception e)

        {

                  // displaying the custom message

            System.out.println("Can't divided by zero");

        }

    }

}

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

**Output:**

Can't divided by zero

Example 6

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

**TryCatchExample6.java**

**public** **class** TryCatchExample6 {

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

**int** i=50;

**int** j=0;

**int** data;

**try**

        {

        data=i/j; //may throw exception

        }

            // handling the exception

**catch**(Exception e)

        {

             // resolving the exception in catch block

            System.out.println(i/(j+2));

        }

    }

}

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

**Output:**

25

Example 7

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

**TryCatchExample7.java**

**public** **class** TryCatchExample7 {

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

**try**

        {

**int** data1=50/0; //may throw exception

        }

             // handling the exception

**catch**(Exception e)

        {

            // generating the exception in catch block

**int** data2=50/0; //may throw exception

        }

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

    }

}

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

**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**

**public** **class** TryCatchExample8 {

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

**try**

        {

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

        }

            // try to handle the ArithmeticException using ArrayIndexOutOfBoundsException

**catch**(ArrayIndexOutOfBoundsException e)

        {

            System.out.println(e);

        }

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

    }

}

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

**Output:**

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

Example 9

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

**TryCatchExample9.java**

**public** **class** TryCatchExample9 {

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

**try**

        {

**int** arr[]= {1,3,5,7};

        System.out.println(arr[10]); //may throw exception

        }

            // handling the array exception

**catch**(ArrayIndexOutOfBoundsException e)

        {

            System.out.println(e);

        }

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

    }

}

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

**Output:**

java.lang.ArrayIndexOutOfBoundsException: 10

rest of the code

Example 10

Let's see an example to handle checked exception.

**TryCatchExample10.java**

**import** java.io.FileNotFoundException;

**import** java.io.PrintWriter;

**public** **class** TryCatchExample10 {

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

        PrintWriter pw;

**try** {

            pw = **new** PrintWriter("jtp.txt"); //may throw exception

            pw.println("saved");

        }

// providing the checked exception handler

**catch** (FileNotFoundException e) {

            System.out.println(e);

        }

    System.out.println("File saved successfully");

    }

}

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

**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**

**public** **class** MultipleCatchBlock1 {

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

**try**{

**int** a[]=**new** **int**[5];

                a[5]=30/0;

               }

**catch**(ArithmeticException e)

                  {

                   System.out.println("Arithmetic Exception occurs");

                  }

**catch**(ArrayIndexOutOfBoundsException e)

                  {

                   System.out.println("ArrayIndexOutOfBounds Exception occurs");

                  }

**catch**(Exception e)

                  {

                   System.out.println("Parent Exception occurs");

                  }

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

    }

}

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

**Output:**

Arithmetic Exception occurs

rest of the code

Example 2

**MultipleCatchBlock2.java**

**public** **class** MultipleCatchBlock2 {

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

**try**{

**int** a[]=**new** **int**[5];

                System.out.println(a[10]);

               }

**catch**(ArithmeticException e)

                  {

                   System.out.println("Arithmetic Exception occurs");

                  }

**catch**(ArrayIndexOutOfBoundsException e)

                  {

                   System.out.println("ArrayIndexOutOfBounds Exception occurs");

                  }

**catch**(Exception e)

                  {

                   System.out.println("Parent Exception occurs");

                  }

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

    }

}

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

**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**

**public** **class** MultipleCatchBlock3 {

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

**try**{

**int** a[]=**new** **int**[5];

                a[5]=30/0;

                System.out.println(a[10]);

               }

**catch**(ArithmeticException e)

                  {

                   System.out.println("Arithmetic Exception occurs");

                  }

**catch**(ArrayIndexOutOfBoundsException e)

                  {

                   System.out.println("ArrayIndexOutOfBounds Exception occurs");

                  }

**catch**(Exception e)

                  {

                   System.out.println("Parent Exception occurs");

                  }

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

    }

}

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

**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**

**public** **class** MultipleCatchBlock4 {

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

**try**{

                String s=**null**;

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

               }

**catch**(ArithmeticException e)

                  {

                   System.out.println("Arithmetic Exception occurs");

                  }

**catch**(ArrayIndexOutOfBoundsException e)

                  {

                   System.out.println("ArrayIndexOutOfBounds Exception occurs");

                  }

**catch**(Exception e)

                  {

                   System.out.println("Parent Exception occurs");

                  }

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

    }

}

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

**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**

**class** MultipleCatchBlock5{

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

**try**{

**int** a[]=**new** **int**[5];

    a[5]=30/0;

   }

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

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

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

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

 }

}

Compilation 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:

....

//main try block

**try**

{

    statement 1;

    statement 2;

//try catch block within another try block

**try**

    {

        statement 3;

        statement 4;

//try catch block within nested try block

**try**

        {

            statement 5;

            statement 6;

     }

**catch**(Exception e2)

        {

//exception message

        }

    }

**catch**(Exception e1)

    {

//exception message

    }

}

//catch block of parent (outer) try block

**catch**(Exception e3)

{

//exception message

}

....

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.

**NestedTryBlock.java**

**public** **class** NestedTryBlock{

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

 //outer try block

**try**{

  //inner try block 1

**try**{

     System.out.println("going to divide by 0");

**int** b =39/0;

   }

    //catch block of inner try block 1

**catch**(ArithmeticException e)

    {

      System.out.println(e);

    }

    //inner try block 2

**try**{

**int** a[]=**new** **int**[5];

    //assigning the value out of array bounds

     a[5]=4;

     }

    //catch block of inner try block 2

**catch**(ArrayIndexOutOfBoundsException e)

    {

       System.out.println(e);

    }

    System.out.println("other statement");

  }

  //catch block of outer try block

**catch**(Exception e)

  {

    System.out.println("handled the exception (outer catch)");

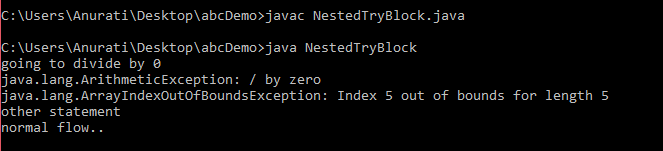
  }

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

 }

}

**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**

**public** **class** NestedTryBlock2 {

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

    {

        // outer (main) try block

**try** {

            //inner try block 1

**try** {

                // inner try block 2

**try** {

**int** arr[] = { 1, 2, 3, 4 };

                    //printing the array element out of its bounds

                    System.out.println(arr[10]);

                }

                // to handles ArithmeticException

**catch** (ArithmeticException e) {

                    System.out.println("Arithmetic exception");

                    System.out.println(" inner try block 2");

                }

            }

            // to handle ArithmeticException

**catch** (ArithmeticException e) {

                System.out.println("Arithmetic exception");

                System.out.println("inner try block 1");

            }

        }

        // to handle ArrayIndexOutOfBoundsException

**catch** (ArrayIndexOutOfBoundsException e4) {

            System.out.print(e4);

            System.out.println(" outer (main) try block");

        }

**catch** (Exception e5) {

            System.out.print("Exception");

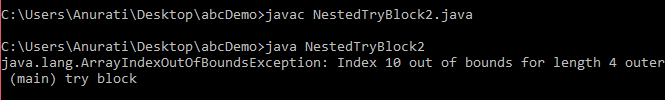
            System.out.println(" handled in main try-block");

        }

    }

}

**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.

### 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**

**class** TestFinallyBlock {

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

**try**{

//below code do not throw any exception

**int** data=25/5;

   System.out.println(data);

  }

//catch won't be executed

**catch**(NullPointerException e){

System.out.println(e);

}

//executed regardless of exception occurred or not

**finally** {

System.out.println("finally block is always executed");

}

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

  }

}

**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**

**public** **class** TestFinallyBlock1{

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

**try** {

        System.out.println("Inside the try block");

        //below code throws divide by zero exception

**int** data=25/0;

       System.out.println(data);

      }

      //cannot handle Arithmetic type exception

      //can only accept Null Pointer type exception

**catch**(NullPointerException e){

        System.out.println(e);

      }

      //executes regardless of exception occured or not

**finally** {

        System.out.println("finally block is always executed");

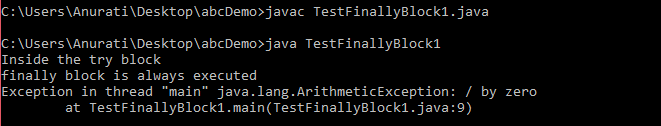
      }

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

      }

    }

**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**

**public** **class** TestFinallyBlock2{

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

**try** {

        System.out.println("Inside try block");

        //below code throws divide by zero exception

**int** data=25/0;

       System.out.println(data);

      }

      //handles the Arithmetic Exception / Divide by zero exception

**catch**(ArithmeticException e){

        System.out.println("Exception handled");

        System.out.println(e);

      }

      //executes regardless of exception occured or not

**finally** {

        System.out.println("finally block is always executed");

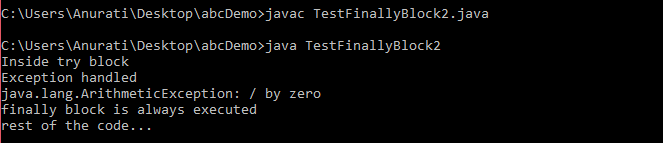
      }

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

      }

    }

**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.

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.,

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

Let's see the example of throw IOException.

**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.

**public** **class** TestThrow1 {

    //function to check if person is eligible to vote or not

**public** **static** **void** validate(**int** age) {

**if**(age<18) {

            //throw Arithmetic exception if not eligible to vote

**throw** **new** ArithmeticException("Person is not eligible to vote");

        }

**else** {

            System.out.println("Person is eligible to vote!!");

        }

    }

    //main method

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

        //calling the function

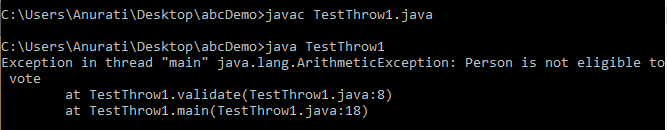
        validate(13);

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

  }

}

**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**

**import** java.io.\*;

**public** **class** TestThrow2 {

    //function to check if person is eligible to vote or not

**public** **static** **void** method() **throws** FileNotFoundException {

        FileReader file = **new** FileReader("C:\\Users\\Anurati\\Desktop\\abc.txt");

        BufferedReader fileInput = **new** BufferedReader(file);

**throw** **new** FileNotFoundException();

    }

    //main method

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

**try**

        {

            method();

        }

**catch** (FileNotFoundException e)

        {

            e.printStackTrace();

        }

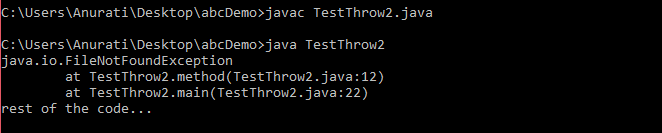
        System.out.println("rest of the code...");

  }

}

**Output:**

AD



### Example 3: Throwing User-defined Exception

exception is everything else under the Throwable class.

**TestThrow3.java**

// class represents user-defined exception

**class** UserDefinedException **extends** Exception

{

**public** UserDefinedException(String str)

    {

        // Calling constructor of parent Exception

**super**(str);

    }

}

// Class that uses above MyException

**public** **class** TestThrow3

{

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

    {

**try**

        {

            // throw an object of user defined exception

**throw** **new** UserDefinedException("This is user-defined exception");

        }

**catch** (UserDefinedException ude)

        {

            System.out.println("Caught the exception");

            // Print the message from MyException object

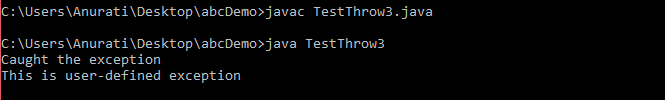
            System.out.println(ude.getMessage());

        }

    }

}

**Output:**



# Java Exception Propagation

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

#### Note: By default Unchecked Exceptions are forwarded in calling chain (propagated).

## Exception Propagation Example

**TestExceptionPropagation1.java**

**class** TestExceptionPropagation1{

**void** m(){

**int** data=50/0;

  }

**void** n(){

    m();

  }

**void** p(){

**try**{

    n();

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

  }

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

   TestExceptionPropagation1 obj=**new** TestExceptionPropagation1();

   obj.p();

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

  }

}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestExceptionPropagation1)

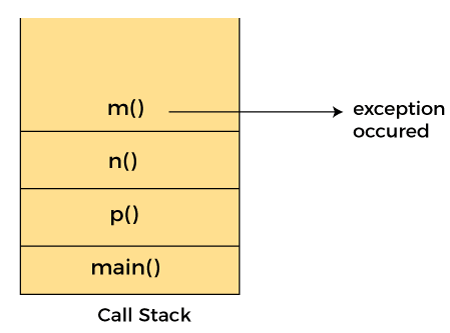
**Output:**

exception handled

normal flow...

In the above example exception occurs in the m() method where it is not handled, so it is propagated to the previous n() method where it is not handled, again it is propagated to the p() method where exception is handled.

Exception can be handled in any method in call stack either in the main() method, p() method, n() method or m() method.



#### Note: By default, Checked Exceptions are not forwarded in calling chain (propagated).

## Exception Propagation Example

**TestExceptionPropagation1.java**

**class** TestExceptionPropagation2{

**void** m(){

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

  }

**void** n(){

    m();

  }

**void** p(){

**try**{

    n();

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

  }

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

   TestExceptionPropagation2 obj=**new** TestExceptionPropagation2();

   obj.p();

   System.out.println("normal flow");

  }

}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestExceptionPropagation2)

**Output:**

Compile Time Error

# Java throws keyword

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

return\_type method\_name() **throws** exception\_class\_name{

//method code

}

### 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).

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**

**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...");

  }

}

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

**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:**

**Case 1:** We have caught the exception i.e. we have handled the exception using try/catch block.

**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**

**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...");

  }

}

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

**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.

AD

Let's see examples for both the scenario.

**If exception does not occur**

**Testthrows3.java**

**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...");

  }

}

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

**Output:**

device operation performed

normal flow...

**B) If exception occurs**

AD

**Testthrows4.java**

**import** java.io.\*;

**class** M{

**void** method()**throws** IOException{

**throw** **new** IOException("device error");

 }

}

**class** Testthrows4{

**public** **static** **void** main(String args[])**throws** IOException{//declare exception

     M m=**new** M();

     m.method();

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

  }

}

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

**Output:**

Java throw keyword

### Difference between throw and throws

[Click me for details](https://www.javatpoint.com/difference-between-throw-and-throws-in-java)

### Que) Can we rethrow an exception?

Yes, by throwing same exception in catch block.

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**

**public** **class** TestThrow {

    //defining a method

**public** **static** **void** checkNum(**int** num) {

**if** (num < 1) {

**throw** **new** ArithmeticException("\nNumber is negative, cannot calculate square");

        }

**else** {

            System.out.println("Square of " + num + " is " + (num\*num));

        }

    }

    //main method

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

            TestThrow obj = **new** TestThrow();

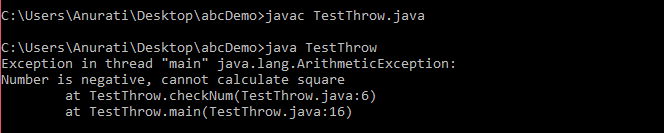
            obj.checkNum(-3);

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

    }

}

**Output:**



Java throws Example

**TestThrows.java**

**public** **class** TestThrows {

    //defining a method

**public** **static** **int** divideNum(**int** m, **int** n) **throws** ArithmeticException {

**int** div = m / n;

**return** div;

    }

    //main method

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

        TestThrows obj = **new** TestThrows();

**try** {

            System.out.println(obj.divideNum(45, 0));

        }

**catch** (ArithmeticException e){

            System.out.println("\nNumber cannot be divided by 0");

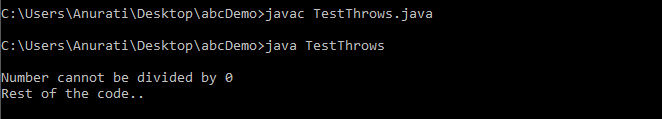
        }

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

    }

}

**Output:**



Java throw and throws Example

**TestThrowAndThrows.java**

**public** **class** TestThrowAndThrows

{

    // defining a user-defined method

    // which throws ArithmeticException

**static** **void** method() **throws** ArithmeticException

    {

        System.out.println("Inside the method()");

**throw** **new** ArithmeticException("throwing ArithmeticException");

    }

    //main method

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

    {

**try**

        {

            method();

        }

**catch**(ArithmeticException e)

        {

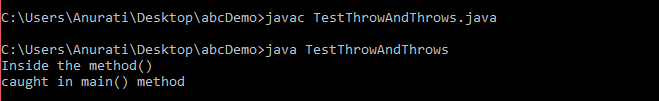
            System.out.println("caught in main() method");

        }

    }

}

**Output:**



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**

**public** **class** FinalExampleTest {

    //declaring final variable

**final** **int** age = 18;

**void** display() {

    // reassigning value to age variable

    // gives compile time error

    age = 55;

    }

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

    FinalExampleTest obj = **new** FinalExampleTest();

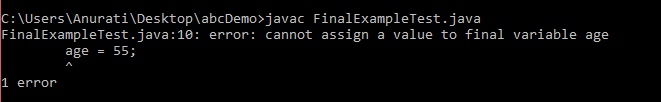
    // gives compile time error

    obj.display();

    }

}

**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**

**public** **class** FinallyExample {

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

**try** {

        System.out.println("Inside try block");

      // below code throws divide by zero exception

**int** data=25/0;

       System.out.println(data);

      }

      // handles the Arithmetic Exception / Divide by zero exception

**catch** (ArithmeticException e){

        System.out.println("Exception handled");

        System.out.println(e);

      }

      // executes regardless of exception occurred or not

**finally** {

        System.out.println("finally block is always executed");

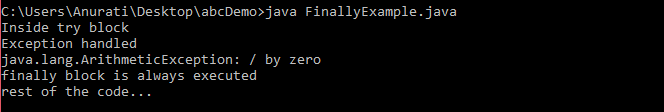
      }

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

      }

    }

**Output:**



Java finalize Example

**FinalizeExample.java**

**public** **class** FinalizeExample {

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

    {

        FinalizeExample obj = **new** FinalizeExample();

        // printing the hashcode

        System.out.println("Hashcode is: " + obj.hashCode());

        obj = **null**;

        // calling the garbage collector using gc()

        System.gc();

        System.out.println("End of the garbage collection");

    }

   // defining the finalize method

**protected** **void** finalize()

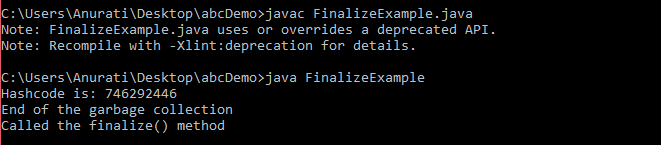
    {

        System.out.println("Called the finalize() method");

    }

}

**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.

## 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:

**public** **class** WrongFileNameException **extends** Exception {

**public** WrongFileNameException(String errorMessage) {

**super**(errorMessage);

    }

}

#### 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**

// class representing custom exception

**class** InvalidAgeException  **extends** Exception

{

**public** InvalidAgeException (String str)

    {

        // calling the constructor of parent Exception

**super**(str);

    }

}

// class that uses custom exception InvalidAgeException

**public** **class** TestCustomException1

{

    // method to check the age

**static** **void** validate (**int** age) **throws** InvalidAgeException{

**if**(age < 18){

        // throw an object of user defined exception

**throw** **new** InvalidAgeException("age is not valid to vote");

    }

**else** {

        System.out.println("welcome to vote");

        }

     }

    // main method

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

    {

**try**

        {

            // calling the method

            validate(13);

        }

**catch** (InvalidAgeException ex)

        {

            System.out.println("Caught the exception");

            // printing the message from InvalidAgeException object

            System.out.println("Exception occured: " + ex);

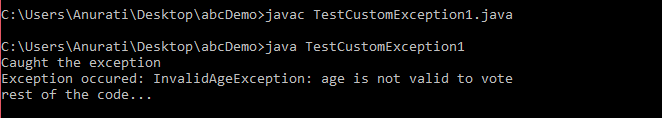
        }

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

    }

}

**Output:**



### Example 2:

**TestCustomException2.java**

// class representing custom exception

**class** MyCustomException **extends** Exception

{

}

// class that uses custom exception MyCustomException

**public** **class** TestCustomException2

{

    // main method

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

    {

**try**

        {

            // throw an object of user defined exception

**throw** **new** MyCustomException();

        }

**catch** (MyCustomException ex)

        {

            System.out.println("Caught the exception");

            System.out.println(ex.getMessage());

        }

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

    }

}

**Output:**

