# **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)

.

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:

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](https://www.javatpoint.com/opr/test.jsp?filename=JavaExceptionExample" \t "_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.

## Common Scenarios of Java Exceptions

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

### **1) A scenario where ArithmeticException occurs**

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

1. **int** a=50/0;//ArithmeticException

### **2) A scenario where NullPointerException occurs**

If we have a null value in any [variable](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

### **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.

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

**Output:**

Arithmetic Exception occurs

rest of the code

Arithmetic Exception occurs

# **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. }
22.
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.

Skip Ad

**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. }
15.
16.
17. //inner try block 2
18. **try**{
19. **int** a[]=**new** **int**[5];
20.
21. //assigning the value out of array bounds
22. a[5]=4;
23. }
24.
25. //catch block of inner try block 2
26. **catch**(ArrayIndexOutOfBoundsException e)
27. {
28. System.out.println(e);
29. }
30.
31.
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. }
39.
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.

# **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.

Skip Ad

### **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. }
16.
17. System.out.println("rest of phe code...");
18. }
19. }

**Output:**



## Why use custom exceptions?

## Or creating own exception.

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. }
10.
11. // class that uses custom exception InvalidAgeException
12. **public** **class** TestCustomException1
13. {
14.
15. // method to check the age
16. **static** **void** validate (**int** age) **throws** InvalidAgeException{
17. **if**(age < 18){
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. }
26.
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");
38.
39. // printing the message from InvalidAgeException object
40. System.out.println("Exception occured: " + ex);
41. }
42.
43. System.out.println("rest of the code...");
44. }
45. }

**Output:**



**Difference between Thread Based Multitasking and process based Multitasking**

The java programming language allows us to create a program that contains one or more parts that can run simultaneously at the same time. This type of program is known as a multithreading program. Each part of this program is called a thread. Every thread defines a separate path of execution in java

A thread is a light wieght process.

A thread is a subpart of a process that can run individually.

In java, multiple threads can run at a time, which enables the java to write multitasking programs.

There are two types of multitasking, and they are as follows.

* Process-based multitasking
* Thread-based multitasking

| **Process-based multitasking** | **Thread-based multitasking** |
| --- | --- |
| It allows the computer to run two or more programs concurrently | It allows the computer to run two or more threads concurrently |
| In this process is the smallest unit. | In this thread is the smallest unit. |
| Process is a larger unit. | Thread is a part of process. |
| Process is heavy weight. | Thread is light weight. |
| Process requires seperate address space for each. | Threads share same address space. |
| Process never gain access over idle time of CPU. | Thread gain access over idle time of CPU. |
| Inter process communication is expensive. | Inter thread communication is not expensive. |

**Java Thread Model**

In java, a thread goes through different states throughout its execution. These stages are called thread life cycle states or phases. A thread may in any of the states like new, ready or runnable, running, blocked or wait, and dead or terminated state. The life cycle of a thread in java is shown in the following figure.



Let's look at each phase in detailed.

**New**

When a thread object is created using new, then the thread is said to be in the New state. This state is also known as Born state.

**Example**

Thread t1 = new Thread();

**Runnable/Ready**

When a thread calls start( ) method, then the thread is said to be in the Runnable state. This state is also known as a Ready state.

**Example**

t1.start( );

**Running**

When a thread calls run( ) method, then the thread is said to be Running. The run( ) method of a thread called automatically by the start( ) method.

**Blocked/Waiting**

A thread in the Running state may move into the blocked state due to various reasons like sleep( ) method called, wait( ) method called, suspend( ) method called, and join( ) method called, etc.

When a thread is in the blocked or waiting state, it may move to Runnable state due to reasons like sleep time completed, waiting time completed, notify( ) or notifyAll( ) method called, resume( ) method called, etc.

**Example**

Thread.sleep(1000);

wait(1000);

wait();

suspened();

notify();

notifyAll();

resume();

**Dead/Terminated**

A thread in the Running state may move into the dead state due to either its execution completed or the stop( ) method called. The dead state is also known as the terminated state.

#

 **Creating Threads in Java**

The java programming language provides two methods to create threads, and they are listed below.

* **Using Thread class (by extending Thread class)**
* **Uisng Runnable interface (by implementing Runnable interface)**

**To create a thread using Thread class, follow the step given below**.

* **Step-1**: Create a class as a child of Thread class. That means, create a class that extends Thread class.
* **Step-2**: Override the run( ) method with the code that is to be executed by the thread. The run( ) method must be public while overriding.
* **Step-3**: Create the object of the newly created class in the main( ) method.
* **Step-4**: Call the start( ) method on the object created in the above step.

 class SampleThread extends Thread{

 public void run() {

 System.out.println("Thread is under Running...");

 for(int i= 1; i<=10; i++) {

 System.out.println("i = " + i);

 }

 }

}

public class My\_Thread\_Test {

 public static void main(String[] args) {

 SampleThread t1 = new SampleThread();

 System.out.println("Thread about to start...");

 t1.start();

 }

}

The java contains a built-in interface Runnable inside the java.lang package. The Runnable interface implemented by the Thread class that contains all the methods that are related to the threads.

**To create a thread using Runnable interface, follow the step given below.**

* **Step-1**: Create a class that implements Runnable interface.
* **Step-2**: Override the run( ) method with the code that is to be executed by the thread. The run( ) method must be public while overriding.
* **Step-3**: Create the object of the newly created class in the main( ) method.
* **Step-4**: Create the Thread class object by passing above created object as parameter to the Thread class constructor.
* **Step-5**: Call the start( ) method on the Thread class object created in the above step.

 class SampleThread implements Runnable{

 public void run() {

 System.out.println("Thread is under Running...");

 for(int i= 1; i<=10; i++) {

 System.out.println("i = " + i);

 }

 }

}

public class My\_Thread\_Test {

 public static void main(String[] args) {

 SampleThread threadObject = new SampleThread();

 Thread thread = new Thread(threadObject);

 System.out.println("Thread about to start...");

 thread.start();

 }

}

**Interthread communication**

**class** common

{

 **synchronized** **static** **void** table(**int** x)

 {

 **for**(**int** i=1;i<=20;i++)

 {

 System.***out***.println(x+"\*"+i+"="+x\*i);

 }

 }

}

**class** thread1 **extends** Thread

 {

 String name;

 thread1(String n){name=n;}

 **public** **void** run()

 {

 common.*table*(5);

 }

}

**class** thread2 **extends** Thread

{

 String name;

 thread2(String n){name=n;}

 **public** **void** run()

 {

 common.*table*(10);

 }

}

**public** **class** My\_Thread\_Test {

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

 thread1 t1 = **new** thread1(" First Thread");

 thread2 t2 = **new** thread2(" Second Thread");

 System.***out***.println("Thread about to start...");

 t1.start();

 t2.start();

 }

}

**Thread Synchronization and and interthread communication using wait() and notify() method**

**class** common

{

 **synchronized** **void** table(String str)

 {

 **for**(**int** i=1;i<=10;i++)

 {

 System.***out***.println(str+" "+i);

 notify();

 **try** {

 wait();

 }**catch**(InterruptedException e) {}

 }

 }

}

**class** thread1 **extends** Thread

 {

 String name;

 common o;

 thread1(String n,common obj){name=n;o=obj;}

 **public** **void** run()

 {

 o.table(name);

 }

}

**class** thread2 **extends** Thread

{

 String name;

 common o;

 thread2(String n,common obj){name=n;o=obj;}

 **public** **void** run()

 {

 o.table(name);

 }

}

**public** **class** My\_Thread\_Test {

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

 common obj=**new** common();

 thread1 t1 = **new** thread1(" First Thread",obj);

 thread1 t2 = **new** thread1(" Second Thread",obj);

 System.***out***.println("Thread about to start...");

 t1.start();

 t2.start();

 }

}

**Producer Consumer program in java(A classical interthread communication or Synchronization problem)**

**class material**

**{**

 **boolean avl=false;**

 **public synchronized void consume(String str)**

 **{**

 **if(avl==false)**

 **{**

 **try**

 **{**

 **wait();**

 **}**

 **catch(InterruptedException e)**

 **{**

 **}**

 **}**

 **System.out.println("Material got consumed by the "+str+" Thread");**

 **avl=false;**

 **notify();**

 **}**

 **public synchronized void produce(String str)**

 **{**

 **if(avl==true)**

 **{**

 **try**

 **{**

 **wait();**

 **}**

 **catch(InterruptedException e)**

 **{**

 **}**

 **}**

 **avl=true;**

 **System.out.println("Material got Produced by the "+str+" Thread");**

 **notify();**

 **}**

**}**

**class Consumer\_Thread extends Thread**

**{**

 **material obj;**

 **String c\_string;**

 **Consumer\_Thread(String str,material obj)**

 **{**

 **c\_string=str;**

 **this.obj=obj;**

 **}**

 **public void run()**

 **{**

 **for(int i=1;i<=10;i++)**

 **{**

 **obj.consume(c\_string);**

 **}**

 **}**

**}**

**class Producer\_Thread extends Thread**

**{**

 **material obj;**

 **String p\_string;**

 **Producer\_Thread(String str,material obj)**

 **{**

 **p\_string=str;**

 **this.obj=obj;**

 **}**

 **public void run()**

 **{**

 **for(int i=1;i<=10;i++)**

 **{**

 **obj.produce(p\_string);**

 **}**

 **}**

**}**

**public class PC**

**{**

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

 **{**

 **material m=new material();**

 **Producer\_Thread p=new Producer\_Thread("Producer", m);**

 **Consumer\_Thread c= new Consumer\_Thread("Consumer", m);**

 **p.start();**

 **c.start();**

 **}**

**}**