# **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.

## **What is Exception in Java?**

In Java, an exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions. These exceptions can occur for various reasons, such as invalid user input, file not found, or division by zero. When an exception occurs, it is typically represented by an object of a subclass of the java.lang.Exception class.

## **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

In Java, exceptions are categorized into two main types: checked exceptions and unchecked exceptions. Additionally, there is a third category known as errors. Let's delve into each of these types:

1. Checked Exception
2. Unchecked Exception
3. Error



### 1. Checked Exceptions

Checked exceptions are the exceptions that are checked at compile-time. This means that the compiler verifies that the code handles these exceptions either by catching them or declaring them in the method signature using the throws keyword. Examples of checked exceptions include:

**IOException:** An exception is thrown when an input/output operation fails, such as when reading from or writing to a file.

**SQLException:** It is thrown when an error occurs while accessing a database.

**ParseException:** Indicates a problem while parsing a string into another data type, such as parsing a date.

**ClassNotFoundException:** It is thrown when an application tries to load a class through its string name using methods like Class.forName(), but the class with the specified name cannot be found in the classpath.

### 2. Unchecked Exceptions (Runtime Exceptions)

Unchecked exceptions, also known as runtime exceptions, are not checked at compile-time. These exceptions usually occur due to programming errors, such as logic errors or incorrect assumptions in the code. They do not need to be declared in the method signature using the throws keyword, making it optional to handle them. Examples of unchecked exceptions include:

**NullPointerException:** It is thrown when trying to access or call a method on an object reference that is null.

**ArrayIndexOutOfBoundsException:** It occurs when we try to access an array element with an invalid index.

**ArithmeticException:** It is thrown when an arithmetic operation fails, such as division by zero.

**IllegalArgumentException:** It indicates that a method has been passed an illegal or inappropriate argument.

### 3. Errors

Errors represent exceptional conditions that are not expected to be caught under normal circumstances. They are typically caused by issues outside the control of the application, such as system failures or resource exhaustion. Errors are not meant to be caught or handled by application code. Examples of errors include:

**OutOfMemoryError:** It occurs when the Java Virtual Machine (JVM) cannot allocate enough memory for the application.

**StackOverflowError:** It is thrown when the stack memory is exhausted due to excessive recursion.

**NoClassDefFoundError:** It indicates that the JVM cannot find the definition of a class that was available at compile-time.

Understanding the different types of exceptions in Java is crucial for writing robust and reliable code. By handling exceptions appropriately, you can improve the resilience of your applications and provide better user experiences.hierarchy of exception handling

## **Difference between Checked and Unchecked Exceptions**

Here are the key differences between checked exceptions, unchecked exceptions (runtime exceptions), and errors in Java:

### 1. Checked Exceptions:

**Compile-time Check:** Checked exceptions are checked at compile-time by the Java compiler. This means that the compiler ensures that these exceptions are either caught or declared in the method signature using the throws keyword.

**Examples:** Examples of checked exceptions include IOException, SQLException, ParseException, etc.

### 2. Unchecked Exceptions (Runtime Exceptions):

Not Checked at Compile-time: Unlike checked exceptions, unchecked exceptions are not checked at compile-time. This means that the compiler does not enforce handling of unchecked exceptions.

**Examples:** Examples of unchecked exceptions include NullPointerException, ArrayIndexOutOfBoundsException, ArithmeticException, etc.

**Runtime Errors:** Unchecked exceptions often represent programming errors or unexpected conditions during runtime, such as null references or array index out of bounds.

|  |  |
| --- | --- |
| **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. |

Not Meant for Handling: Errors represent exceptional conditions that are typically beyond the control of the application and are not meant to be caught or handled by application code.

**Examples:** Examples of errors include OutOfMemoryError, StackOverflowError, NoClassDefFoundError, etc.

**Critical Conditions:** Errors usually indicate critical conditions, such as JVM failures or system resource exhaustion, where the application cannot recover.

## **Java Exception Keywords**

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

### The try-catch Block

One of the primary mechanisms for handling exceptions in Java is the try-catch block. The try block contains the code that may throw an exception, and the catch block is used to handle the exception if it occurs. Here's a basic example:

1. **try** {
2. // Code that may throw an exception
3. } **catch** (ExceptionType e) {
4. // Exception handling code
5. }

### Handling Multiple Exceptions

You can handle multiple types of exceptions by providing multiple catch blocks, each catching a different type of exception. This allows you to tailor your exception handling logic based on the specific type of exception thrown. Here's an example:

1. **try** {
2. // Code that may throw an exception
3. } **catch** (IOException e) {
4. // Handle IOException
5. } **catch** (NumberFormatException e) {
6. // Handle NumberFormatException
7. } **catch** (Exception e) {
8. // Handle any other exceptions
9. }

### The finally Block

In addition to try and catch, Java also provides a finally block, which allows you to execute cleanup code, such as closing resources, regardless of whether an exception occurs or not. The finally block is typically used to release resources that were acquired in the try block. Here's an example:

1. **try** {
2. // Code that may throw an exception
3. } **catch** (Exception e) {
4. // Exception handling code
5. } **finally** {
6. // Cleanup code
7. }

## **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){
7. System.out.println(e);
8. }
9. //rest code of the program
10. System.out.println("rest of the code...");
11. }
12. }

[Test it Now](javascript:void(0))

**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

Here's a simple Java code example where an ArithmeticException occurs:

**File Name: ArithmeticExceptionExample.java**

1. **public** **class** ArithmeticExceptionExample {
2. **public** **static** **void** main(String[] args) {
3. **int** dividend = 10;
4. **int** divisor = 0;
5. **try** {
6. **int** result = dividend / divisor; // Division by zero
7. System.out.println("Result: " + result);
8. } **catch** (ArithmeticException e) {
9. System.out.println("Error: Division by zero is not allowed.");
10. // Additional error handling code can be added here
11. }
12. }
13. }

**Output:**

*Error: Division by zero is not allowed.*

**Explanation**

We have a main() method where we attempt to perform division by zero that is not allowed in arithmetic.

Inside the try block, we perform the division operation dividend / divisor, where divisor is assigned the value of 0.

When the division by zero occurs, an ArithmeticException is thrown. We catch this exception using a catch block specifically for ArithmeticException.

In the catch block, we handle the exception by printing an error message, indicating that division by zero is not allowed. Additional error handling logic can be added here if needed.

### 2) A scenario where NullPointerException occurs

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

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

Here's a Java code example where a NullPointerException occurs:

**File Name: NullPointerExceptionExample.java**

1. **public** **class** NullPointerExceptionExample {
2. **public** **static** **void** main(String[] args) {
3. String str = **null**; // Initializing a String variable to null
4. **try** {
5. **int** length = str.length(); // Attempting to call a method on a null reference
6. System.out.println("Length of the string: " + length);
7. } **catch** (NullPointerException e) {
8. System.out.println("Error: Null reference encountered.");
9. // Additional error handling code can be added here
10. }
11. }
12. }

**Output:**

*Error: Null reference encountered.*

**Explanation**

We have a main() method where we initialize a String variable str to null.

Inside the try block, we attempt to call the length() method on the str reference, which is null.

When attempting to call a method on a null reference, a NullPointerException is thrown.

We catch this exception using a catch block specifically for NullPointerException.

In the catch block, we handle the exception by printing an error message indicating that a null reference was encountered. Additional error handling logic can be added here if needed.

### 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 variable that has characters; converting this variable into digit will cause NumberFormatException.

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

Here's a Java code example where a NumberFormatException occurs:

**File Name: NumberFormatExceptionExample.java**

1. **public** **class** NumberFormatExceptionExample {
2. **public** **static** **void** main(String[] args) {
3. String str = "abc"; // Initializing a String with non-numeric characters
4. **try** {
5. **int** num = Integer.parseInt(str); // Attempting to parse a non-numeric string to an integer
6. System.out.println("Parsed number: " + num);
7. } **catch** (NumberFormatException e) {
8. System.out.println("Error: Unable to parse the string as an integer.");
9. // Additional error handling code can be added here
10. }
11. }
12. }

**Output:**

*Error: Unable to parse the string as an integer.*

**Explanation:**

We have a main() method where we initialize a String variable str with non-numeric characters.

Inside the try block, we attempt to parse the string str to an integer using the Integer.parseInt() method.

Since the string contains non-numeric characters, the parsing operation throws a NumberFormatException.

We catch this exception using a catch block specifically for NumberFormatException.

In the catch block, we handle the exception by printing an error message indicating that the string could not be parsed as an integer. Additional error handling logic can be added here if needed.

### 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

Here's a Java code example where an ArrayIndexOutOfBoundsException occurs:

**File Name: ArrayIndexOutOfBoundsExceptionExample.java**

1. **public** **class** ArrayIndexOutOfBoundsExceptionExample {
2. **public** **static** **void** main(String[] args) {
3. **int**[] numbers = {1, 2, 3, 4, 5}; // Initializing an array with 5 elements
4. **try** {
5. **int** index = 10; // Accessing an index that is out of bounds
6. **int** value = numbers[index]; // Attempting to access an element at an invalid index
7. System.out.println("Value at index " + index + ": " + value);
8. } **catch** (ArrayIndexOutOfBoundsException e) {
9. System.out.println("Error: Index is out of bounds.");
10. // Additional error handling code can be added here
11. }
12. }
13. }

**Output:**

*Error: Index is out of bounds.*

**Explanation**

We have a main() method where we initialize an array numbers with 5 elements.

Inside the try block, we attempt to access an element at index 10, which is out of bounds for the array numbers.

Since the index is out of bounds, an ArrayIndexOutOfBoundsException is thrown.

We catch this exception using a catch block specifically for ArrayIndexOutOfBoundsException.

In the catch block, we handle the exception by printing an error message indicating that the index is out of bounds. Additional error handling logic can be added here if needed.

## **Best Practices for Exception Handling**

**Catch Specific Exceptions:** Catch specific exceptions whenever possible rather than catching general Exception objects. It helps in providing more precise error handling and makes your code easier to understand and maintain.

**Keep Exception Handling Simple:** Avoid overly complex exception handling logic. Keep your catch blocks concise and focused on handling the specific exception they are designed for. Complex exception handling logic can make your code difficult to debug and maintain.

**Log Exceptions:** Always log exceptions or error messages when handling them. This helps in troubleshooting issues and diagnosing problems in production environments.

**Throw Exceptions Appropriately:** Throw exceptions when necessary, but avoid excessive use of checked exceptions. Checked exceptions should be used for exceptional conditions that the caller can reasonably be expected to handle.

**Use Custom Exceptions:** Create custom exception classes for specific error conditions in your application. This helps in providing meaningful error messages and makes your code more self-documenting**.**

# **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](javascript:void(0))

**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](javascript:void(0))

**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](javascript:void(0))

**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](javascript:void(0))

**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](javascript:void(0))

**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](javascript:void(0))

**Output:**

*25*

**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](javascript:void(0))

**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](javascript:void(0))

**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](javascript:void(0))

**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](javascript:void(0))

**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](javascript:void(0))

**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](javascript:void(0))

**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](javascript:void(0))

**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](javascript:void(0))

**Output:**

*Compile-time error*

# **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.

Advertisement

### 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

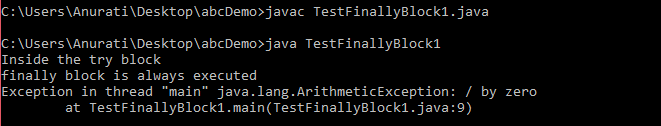
Let's see the the fillowing example. Here, the code throws an exception however the catch block cannot handle it. Despite this, the finally block is executed after the try block and then the program terminates abnormally.

**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:**

Advertisement



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

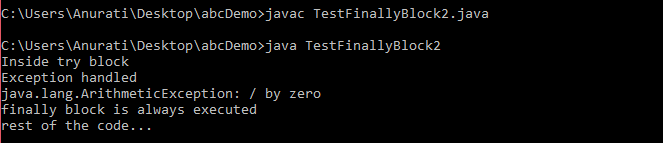
**Example:**

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 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.

Advertisement

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**

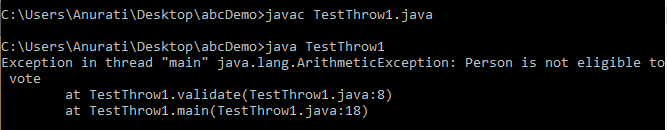
### Example 1: Throwing Unchecked Exception

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.

**File Name: TestThrow1.java**

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 an unchecked exception from a method, it is not required to handle the exception or declare it in throws clause. However, for checked exceptions, handling or declaration in the throws clause is mandatory."**

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

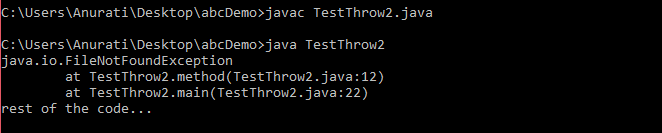
#### **Note: Every subclass of Error and RuntimeException is an unchecked exception in Java. A checked exception is everything else under the Throwable class.**

**File Name: 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:**



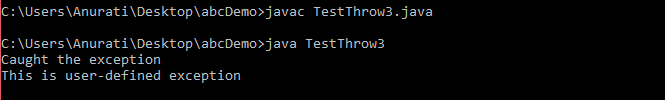
### Example 3: Throwing User-defined Exception

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](javascript:void(0))

**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](javascript:void(0))

**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](javascript:void(0))

**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](javascript:void(0))

**Output:**

Java throw keyword

# **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.

|  |  |  |  |
| --- | --- | --- | --- |
| **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. | Uses | 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 | The keyword throw is used within the method. | The keyword 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. |

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:

# **Exception Handling with Method Overriding in Java**

There are many rules if we talk about method overriding with exception handling.

Some of the rules are listed below:

* **If the superclass method does not declare an exception**
  + If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception.
* **If the superclass method declares an exception**
  + If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

### If the superclass method does not declare an exception

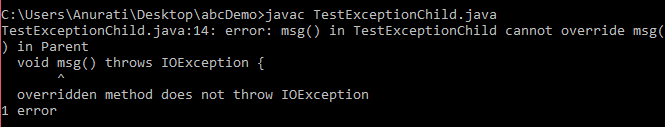
#### **Rule 1: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception.**

Let's consider following example based on the above rule.

**TestExceptionChild.java**

1. **import** java.io.\*;
2. **class** Parent{
4. // defining the method
5. **void** msg() {
6. System.out.println("parent method");
7. }
8. }
10. **public** **class** TestExceptionChild **extends** Parent{
12. // overriding the method in child class
13. // gives compile time error
14. **void** msg() **throws** IOException {
15. System.out.println("TestExceptionChild");
16. }
18. **public** **static** **void** main(String args[]) {
19. Parent p = **new** TestExceptionChild();
20. p.msg();
21. }
22. }

**Output:**

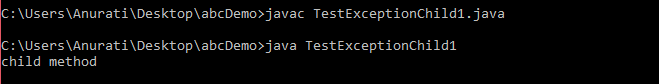


#### **Rule 2: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but can declare unchecked exception.**

**TestExceptionChild1.java**

1. **import** java.io.\*;
2. **class** Parent{
3. **void** msg() {
4. System.out.println("parent method");
5. }
6. }
8. **class** TestExceptionChild1 **extends** Parent{
9. **void** msg()**throws** ArithmeticException {
10. System.out.println("child method");
11. }
13. **public** **static** **void** main(String args[]) {
14. Parent p = **new** TestExceptionChild1();
15. p.msg();
16. }
17. }

**Output:**



### If the superclass method declares an exception

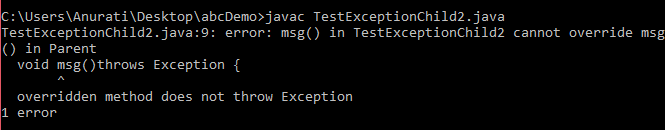
#### **Rule 1: If the superclass method declares an exception, subclass overridden method can declare the same subclass exception or no exception but cannot declare parent exception.**

### Example in case subclass overridden method declares parent exception

**TestExceptionChild2.java**

1. **import** java.io.\*;
2. **class** Parent{
3. **void** msg()**throws** ArithmeticException {
4. System.out.println("parent method");
5. }
6. }
8. **public** **class** TestExceptionChild2 **extends** Parent{
9. **void** msg()**throws** Exception {
10. System.out.println("child method");
11. }
13. **public** **static** **void** main(String args[]) {
14. Parent p = **new** TestExceptionChild2();
16. **try** {
17. p.msg();
18. }
19. **catch** (Exception e){}
21. }
22. }

**Output:**

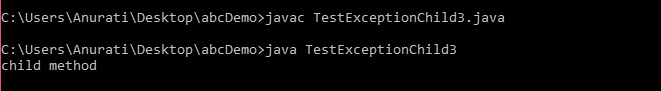


### Example in case subclass overridden method declares same exception

**TestExceptionChild3.java**

1. **import** java.io.\*;
2. **class** Parent{
3. **void** msg() **throws** Exception {
4. System.out.println("parent method");
5. }
6. }
8. **public** **class** TestExceptionChild3 **extends** Parent {
9. **void** msg()**throws** Exception {
10. System.out.println("child method");
11. }
13. **public** **static** **void** main(String args[]){
14. Parent p = **new** TestExceptionChild3();
16. **try** {
17. p.msg();
18. }
19. **catch**(Exception e) {}
20. }
21. }

**Output:**

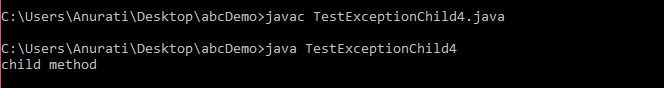


### Example in case subclass overridden method declares subclass exception

**TestExceptionChild4.java**

1. **import** java.io.\*;
2. **class** Parent{
3. **void** msg()**throws** Exception {
4. System.out.println("parent method");
5. }
6. }
8. **class** TestExceptionChild4 **extends** Parent{
9. **void** msg()**throws** ArithmeticException {
10. System.out.println("child method");
11. }
13. **public** **static** **void** main(String args[]){
14. Parent p = **new** TestExceptionChild4();
16. **try** {
17. p.msg();
18. }
19. **catch**(Exception e) {}
20. }
21. }

**Output:**

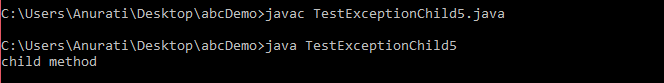


### Example in case subclass overridden method declares no exception

**TestExceptionChild5.java**

1. **import** java.io.\*;
2. **class** Parent {
3. **void** msg()**throws** Exception{
4. System.out.println("parent method");
5. }
6. }
8. **class** TestExceptionChild5 **extends** Parent{
9. **void** msg() {
10. System.out.println("child method");
11. }
13. **public** **static** **void** main(String args[]){
14. Parent p = **new** TestExceptionChild5();
16. **try** {
17. p.msg();
18. }
19. **catch**(Exception e) {}
21. }
22. }

**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.

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.**

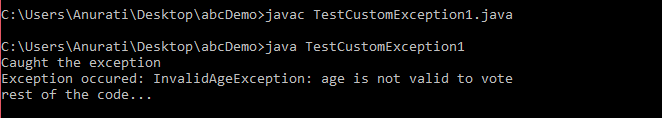
### Example 1:

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:**

