Exception Chapter

Exception is an abnormal condition.

An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an exception object, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called throwing an exception.

The runtime system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called an exception handler.

The core advantage of exception handling is **to maintain the normal flow of the application**.

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. All userdefined exceptions are checked exceptions.

### 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.

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.

## 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.  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. |
| 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.  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. |
| Finally | The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not. **For each try block there can be zero or more catch blocks, but only one finally block.** |
| 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. |

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

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

Java throw Example

**TestThrow.java**

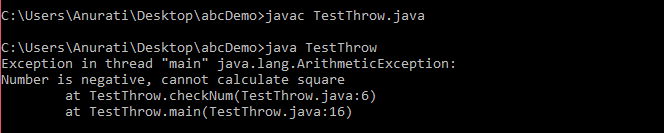
1. **public** **class** TestThrow {
2. //defining a method
3. **public** **static** **void** checkNum(**int** num) {
4. **if** (num < 1) {
5. **throw** **new** ArithmeticException("\nNumber is negative, cannot calculate square");
6. }
7. **else** {
8. System.out.println("Square of " + num + " is " +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

**Output:**

24.8M

339

Triggers in SQL (Hindi)

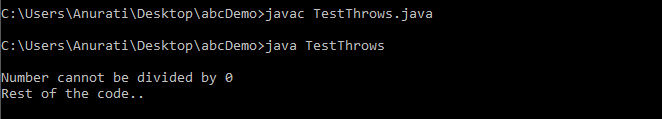


Java throws Example

**TestThrows.java**

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

**Output:**



# **Java 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.

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:**

