Exception Handling in Java

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

**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**

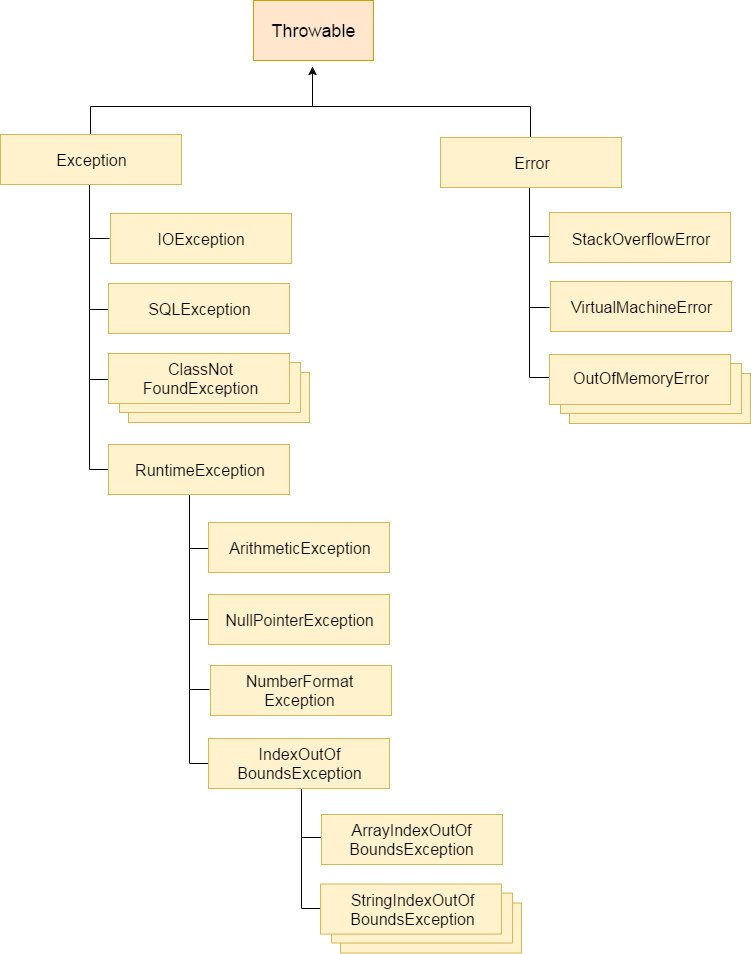
Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

**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 use exception handling.

**Hierarchy of Java Exception classes**

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



**Types of Java Exceptions**

There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the unchecked exception. According to Oracle, there are three types of exceptions:

* Checked Exception
* Unchecked Exception
* Error

**1) Checked Exception**

The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.

**2) Unchecked Exception**

The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

**3) Error**

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

**Java Exception Keywords**

There are 5 keywords which are used in handling exceptions in Java.

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

**1) 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**

The wrong formatting of any value may occur NumberFormatException. Suppose I have a [string](https://www.javatpoint.com/java-string) variable that has characters, converting this variable into digit will occur NumberFormatException.

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

**4) A scenario where ArrayIndexOutOfBoundsException occurs**

If you are inserting any value in the wrong index, it would result in ArrayIndexOutOfBoundsException as shown below:

* **int** a[]=**new** **int**[5];
* a[10]=50; //ArrayIndexOutOfBoundsException

**Java try and catch 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 of try block, the rest of the block code will not execute. So, it is recommended not to keeping 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){ }

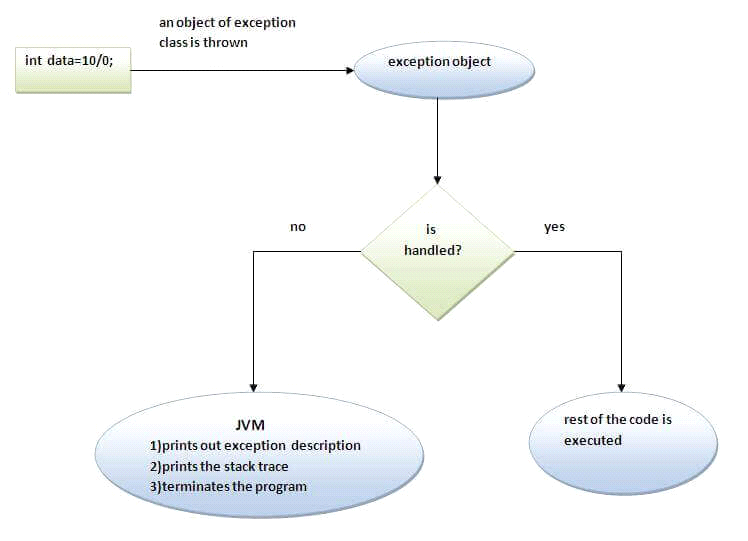
**Syntax of try-finally block**

* **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**



**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.

**Example 1**

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

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

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

**try**{

**int** a[]=**new** **int**[5];

                a[10]=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");

    }

}

Java Nested try block

**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:**

* ....
* **try**
* {
* statement1;
* statement 2;
* **try**
* {
* statement 1;
* statement 2;
* }
* **catch**(Exception e)
* {
* }
* }
* **catch**(Exception e)
* {
* }
* ....

**Java nested try example**

**class** Excep6{

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

**try**{

**try**{

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

**int** b =39/0;

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

**try**{

**int** a[]=**new** **int**[5];

    a[6]=4;

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

    System.out.println("other statement);

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

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

 }

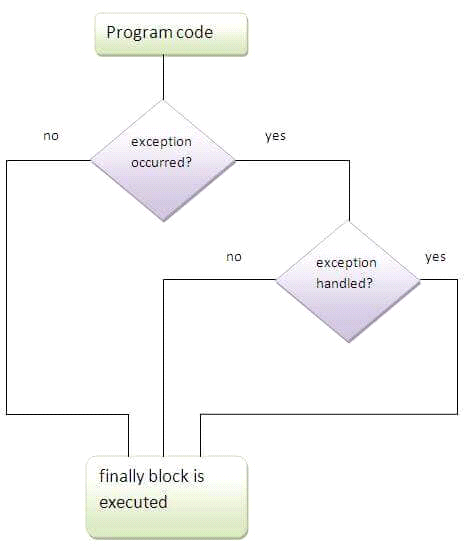
}

Java finally block

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

Java finally block is always executed whether exception is handled or not.

Java finally block follows try or catch block.



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

**Why use java finally**

* Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.Prime Ministers of India | List of Prime Minister of India (1947-2020)
* **class** TestFinallyBlock{
* **public** **static** **void** main(String args[]){
* **try**{
* **int** data=25/0;
* System.out.println(data);
* }
* **catch**(NullPointerException e){System.out.println(e);}
* **finally**{System.out.println("finally block is always executed");}
* System.out.println("rest of the code...");
* }

Output:5

finally block is always executed

rest of the code...

**Java throw keyword**

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

We can throw either checked or uncheked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception. We will see custom exceptions later.

The syntax of java throw keyword is given below.

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

**java throw keyword example**

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{
* **static** **void** validate(**int** age){
* **if**(age<18)
* **throw** **new** ArithmeticException("not valid");
* **else**
* System.out.println("welcome to vote");
* }
* **public** **static** **void** main(String args[]){
* validate(13);
* System.out.println("rest of the code...");
* }
* }

Output:

Exception in thread main java.lang.ArithmeticException:not valid

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 normal flow 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 performing check up before the code being used.

**Syntax of java throws**

* return\_type method\_name(arguments) **throws** exception\_class\_name
* {
* //method code
* }

**Which exception should be declared**

**Ans)** checked exception only, because:

* **unchecked Exception:** under your control so correct your code.
* **error:** beyond your control e.g. you 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 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.

* **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...");
* }
* }

Output:

exception handled

normal flow...

Difference between throw and throws in Java

There are many differences between throw and throws keywords. A list of differences between throw and throws are given below:

|  |  |  |
| --- | --- | --- |
| **No.** | **throw** | **throws** |
| 1) | Java throw keyword is used to explicitly throw an exception. | Java throws keyword is used to declare an exception. |
| 2) | Checked exception cannot be propagated using throw only. | Checked exception can be propagated with throws. |
| 3) | Throw is followed by an instance. | Throws is followed by class. |
| 4) | Throw is used within the method. | Throws is used with the method signature. |
| 5) | You cannot throw multiple exceptions. | You can declare multiple exceptions e.g. public void method()throws IOException,SQLException. |