**What is an Exception?**

An exception is an unwanted or unexpected event, which occurs during the execution of a program i.e at run time, that disrupts the normal flow of the program’s instructions.

**Error vs Exception**

**Error:**An Error indicates serious problem that a reasonable application should not try to catch.  
**Exception:**Exception indicates conditions that a reasonable application might try to catch.

**Exception Hierarchy**

All exception and errors types are sub classes of class **Throwable**, which is base class of hierarchy.One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. NullPointerException is an example of such an exception.Another branch,**Error** are used by the Java run-time system([JVM](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/)) to indicate errors having to do with the run-time environment itself(JRE). StackOverflowError is an example of such an error.

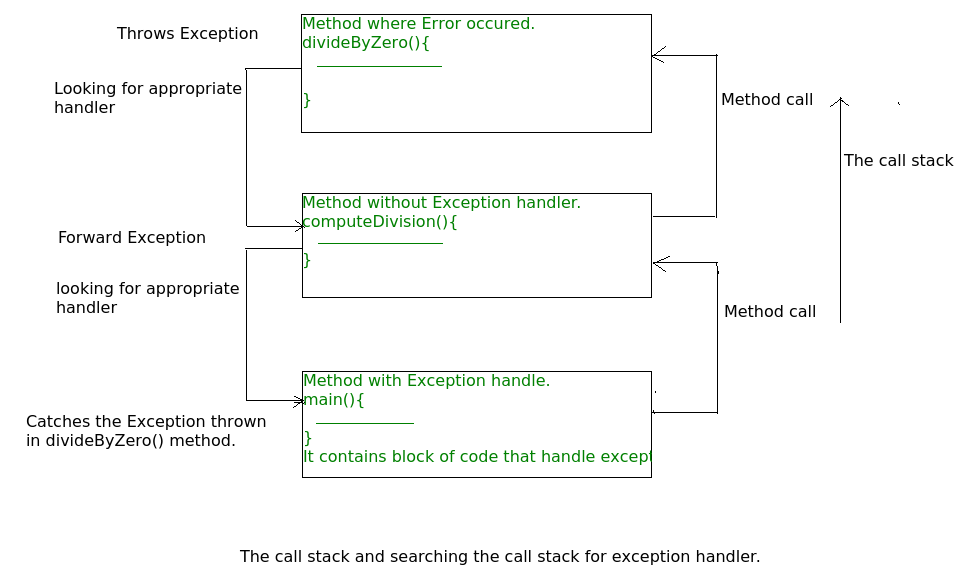
[](https://media.geeksforgeeks.org/wp-content/uploads/Exception-in-java1.png)

**How JVM handle an Exception?**

**Default Exception Handling :**Whenever inside a method, if an exception has occurred, the method creates an Object known as Exception Object and hands it off to the run-time system(JVM). The exception object contains name and description of the exception, and current state of the program where exception has occurred. Creating the Exception Object and handling it to the run-time system is called throwing an Exception.There might be the list of the methods that had been called to get to the method where exception was occurred. This ordered list of the methods is called **Call Stack**

Now the following procedure will happen.

* The run-time system searches the call stack to find the method that contains block of code that can handle the occurred exception. The block of the code is called **Exception handler**.
* The run-time system starts searching from the method in which exception occurred, proceeds through call stack in the reverse order in which methods were called.
* If it finds  appropriate handler then it passes the occurred exception to it. Appropriate handler means the type of the exception object thrown matches the type of the exception object it can handle.
* If run-time system searches all the methods on call stack and couldn’t have found the appropriate handler then run-time system handover the Exception Object to **default exception handler**, which is part of run-time system. This handler prints the exception information in the following format and terminates program **abnormally**.

See the below diagram to understand the flow of the call stack.  


**Java try catch finally blocks**

**Java try, catch and finally** blocks helps in writing the application code which may throw [exceptions](https://howtodoinjava.com/java/exception-handling/checked-vs-unchecked-exceptions-in-java/) in runtime and gives us a chance to either recover from exception by executing alternate application logic or [handle the exception gracefully](https://howtodoinjava.com/best-practices/java-exception-handling-best-practices/) to report back to the user. It helps in preventing the ugly application crashes.

#### 1.1. try block

The try block contains the application code which is expected to work in normal conditions. For example, reading a file, writing to databases or performing complex business operations.

A try bock is written with **try keyword** followed with the curly braces.

|  |
| --- |
| try block syntax |
| try {      //application code  } |

#### 1.2. catch block

The optional catch block(s) follows the try block and MUST handle the checked exceptions thrown by try block as well as any possible unchecked exceptions.

|  |
| --- |
| catch block syntax |
| try {      //code  }  catch(Exception e) {      //handle exception  } |

An application can go wrong in N different ways. That’s why we can associate **multiple catch blocks** with a single try block. In each catch block, we can handle one or more specific exceptions in a unique way.

When one catch block handles the exception, the next catch blocks are not executed. Control shifts directly from the executed catch block to execute the remaining part of the program, including finally block.

|  |
| --- |
| multiple catch block syntax |
| try {      //code  }  catch(NullPointerException e) {      //handle exception  }  catch(NumberFormatException e) {      //handle exception  }  catch(Exception e) {      //handle exception  } |

#### 1.3. finally block

An optional finally block gives us a chance to run the code which we want to execute EVERYTIME a try-catch block is completed – either with errors or without any error.

The finally block statements are guaranteed of execution even if the we fail to handle the exception successfully in catch block.

|  |
| --- |
| finally block syntax |
| try {      //open file      //read file  }  catch(Exception e) {      //handle exception while reading the file  }  finally {      //close the file  } |

#### 1.4. only try block is mandatory

Please note that only try block is mandatory while catch and finally blocks are optional. With a try block, we can use either a catch block or finally block as needed.

It is possible to have below given both combinations in Java. **Both versions are valid**.

|  |
| --- |
| Version 1 - try with catch, without finally block |
| try {    }  catch(Exception e) {    } |
| Version 2 - try with finally, without catch block |
| try {    }  finally {    } |

Example:

try

{

    System.out.println("try block");

    throw new NullPointerException("null occurred");

}

catch (NumberFormatException e)

{

    System.out.println("catch block 1");

}

catch (NullPointerException e)

{

    System.out.println("catch block 2");

}

catch (Exception e)

{

    System.out.println("catch block 3");

}

finally

{

    System.out.println("finally block");

}

## 4. try-with-resources

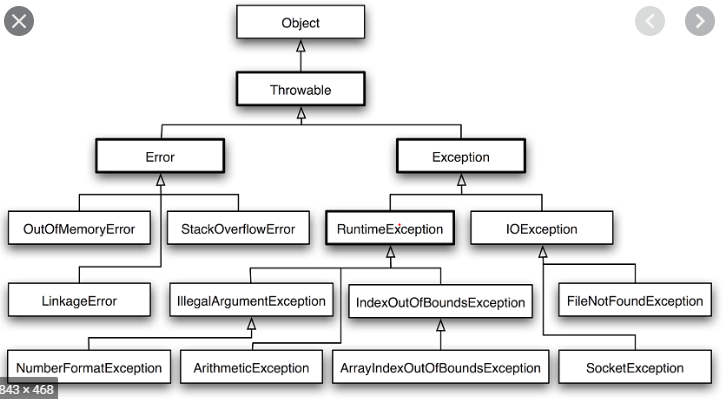
For **Closeable** resources, such as streams, Java SE 7 introduced try-with-resources statements which is recommended ways to handle exceptions in mentioned scenarios. In this approach, we are not required to close the streams and JVM does it for us. It eliminates the need of finally blocks.

In try-with-resources, a resource is opened in try block inside small brackets and finally block is completely gone.

|  |
| --- |
| try (BufferedReader br = new BufferedReader(new FileReader("C:/temp/test.txt")))  {      String sCurrentLine;      while ((sCurrentLine = br.readLine()) != null)      {          System.out.println(sCurrentLine);      }  }  catch (IOException e)  {      e.printStackTrace();  } |

# Checked vs Unchecked Exceptions

The main **difference between checked and unchecked exception** is that the checked exceptions are checked at compile-time while unchecked exceptions are checked at runtime



## What are checked exceptions?

Checked exceptions are checked at compile-time. It means if a method is throwing a checked exception then it should handle the exception using [try-catch block](https://beginnersbook.com/2013/04/try-catch-in-java/) or it should declare the exception using [throws keyword](https://beginnersbook.com/2013/04/java-throws/), otherwise the program will give a compilation error.

For example, consider the following Java program that opens file at location “C:\test\a.txt” and prints the first three lines of it. The program doesn’t compile, because the function main() uses FileReader() and FileReader() throws a checked exception *FileNotFoundException*. It also uses readLine() and close() methods, and these methods also throw checked exception *IOException*

|  |
| --- |
| import java.io.\*;    class Main {      public static void main(String[] args) {          FileReader file = new FileReader("C:\\test\\a.txt");          BufferedReader fileInput = new BufferedReader(file);            // Print first 3 lines of file "C:\test\a.txt"          for (int counter = 0; counter < 3; counter++)              System.out.println(fileInput.readLine());            fileInput.close();      }  } |

Output:

Exception in thread "main" java.lang.RuntimeException: Uncompilable source code -

unreported exception java.io.FileNotFoundException; must be caught or declared to be

thrown

at Main.main(Main.java:5)

To fix the above program, we either need to specify list of exceptions using throws, or we need to use try-catch block. We have used throws in the below program. Since *FileNotFoundException* is a subclass of *IOException*, we can just specify *IOException* in the throws list and make the above program compiler-error-free.

|  |
| --- |
| import java.io.\*;    class Main {      public static void main(String[] args) throws IOException {          FileReader file = new FileReader("C:\\test\\a.txt");          BufferedReader fileInput = new BufferedReader(file);            // Print first 3 lines of file "C:\test\a.txt"          for (int counter = 0; counter < 3; counter++)              System.out.println(fileInput.readLine());            fileInput.close();      }  } |

Output: First three lines of file “C:\test\a.txt”

**2) Unchecked** are the exceptions that are not checked at compiled time. In C++, all exceptions are unchecked, so it is not forced by the compiler to either handle or specify the exception. It is up to the programmers to be civilized, and specify or catch the exceptions.  
In Java exceptions under Error and RuntimeException classes are unchecked exceptions, everything else under throwable is checked.

Consider the following Java program. It compiles fine, but it throws *ArithmeticException* when run. The compiler allows it to compile, because *ArithmeticException* is an unchecked exception.

|  |
| --- |
| class Main {     public static void main(String args[]) {        int x = 0;        int y = 10;        int z = y/x;    }  } |

Output:

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

at Main.main(Main.java:5)

Java Result: 1

# Exception Propagation

**Exception Propagation in Checked Exceptions**

Unlike Unchecked Exceptions, the propagation of exception **does not happen** in case of Checked Exception and its mandatory to use [throw keyword](https://www.geeksforgeeks.org/throw-throws-java/) here. Only unchecked exceptions are propagated.**Checked exceptions throw compilation error.**

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

# throw and throws

**throw**

The throw keyword in Java is used to explicitly throw an exception from a method or any block of code. We can throw either [checked or unchecked exception](https://www.geeksforgeeks.org/checked-vs-unchecked-exceptions-in-java/). The throw keyword is mainly used to throw custom exceptions.  
Syntax:

**throw *Instance***

Example:

**throw new ArithmeticException("/ by zero");**

**throws**

throws is a keyword in Java which is used in the signature of method to indicate that this method might throw one of the listed type exceptions. The caller to these methods has to handle the exception using a try-catch block.  
**Syntax:**

**type method\_name(parameters) throws exception\_list**

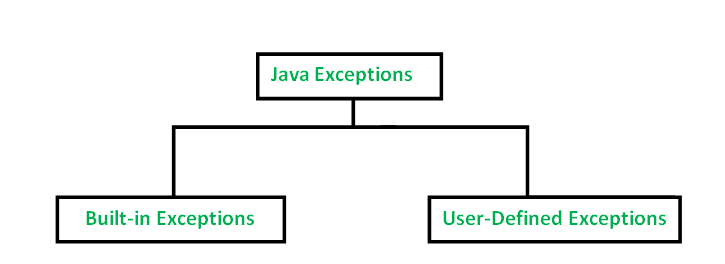
exception\_list is a comma separated list of all the

exceptions which a method might throw.

In a program, if there is a chance of rising an exception then compiler always warn us about it and compulsorily we should handle that checked exception, Otherwise we will get compile time error saying **unreported exception XXX must be caught or declared to be thrown**. To prevent this compile time error we can handle the exception in two ways:

1. By using [try catch](https://www.geeksforgeeks.org/flow-control-in-try-catch-finally-in-java/)
2. By using **throws** keyword

# Types of Exception in Java with Examples

Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.

**User-Defined Exceptions**

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, user can also create exceptions which are called ‘user-defined Exceptions’.  
Following steps are followed for the creation of user-defined Exception.

* The user should create an exception class as a subclass of Exception class. Since all the exceptions are subclasses of Exception class, the user should also make his class a subclass of it. This is done as:

class MyException extends Exception

* We can write a default constructor in his own exception class.

MyException(){}

* We can also create a parameterized constructor with a string as a parameter.  
  We can use this to store exception details. We can call super class(Exception) constructor from this and send the string there.
* MyException(String str)
* {
* super(str);
* }
* To raise exception of user-defined type, we need to create an object to his exception class and throw it using throw clause, as:
* MyException me = new MyException(“Exception details”);

throw me;