# Exceptions in Java

<https://www.geeksforgeeks.org/exceptions-in-java/>

<https://www.geeksforgeeks.org/types-of-exception-in-java-with-examples/>

**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)

For checked vs unchecked exception, see [Checked vs Unchecked Exceptions](https://www.geeksforgeeks.org/checked-vs-unchecked-exceptions-in-java/)

# Checked vs Unchecked Exceptions in Java

<https://www.geeksforgeeks.org/checked-vs-unchecked-exceptions-in-java/>

In Java, there are two types of exceptions:

**1) Checked:** are the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using throws keyword.

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.

**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**.

+-----------+

| Throwable |

+-----------+

/ \

/ \

+-------+ +-----------+

| Error | | Exception |

+-------+ +-----------+

/ | \ / | \

\\_\_\_\_\_\_\_\_/ \\_\_\_\_\_\_/ \

+------------------+

unchecked checked | RuntimeException |

+------------------+

/ | | \

\\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/

unchecked

# **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

# throw and throws in Java

<https://www.geeksforgeeks.org/throw-throws-java/>

**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.

**Important points to remember about throws keyword:**

* throws keyword is required only for checked exception and usage of throws keyword for unchecked exception is meaningless.
* throws keyword is required only to convince compiler and usage of throws keyword does not prevent abnormal termination of program.
* By the help of throws keyword we can provide information to the caller of the method about the exception.

# Chained Exceptions in Java

<https://www.geeksforgeeks.org/chained-exceptions-java/>

Chained Exceptions allows to relate one exception with another exception, i.e one exception describes cause of another exception. For example, consider a situation in which a method throws an ArithmeticException because of an attempt to divide by zero but the actual cause of exception was an I/O error which caused the divisor to be zero. The method will throw only ArithmeticException to the caller. So the caller would not come to know about the actual cause of exception. Chained Exception is used in such type of situations.

# Methods Of Throwable class Which support chained exceptions in java :

1. getCause() method :- This method returns actual cause of an exception.
2. initCause(Throwable cause) method :- This method sets the cause for the calling exception.

// Java program to demonstrate working of chained exceptions

public class ExceptionHandling

{

    public static void main(String[] args)

    {

        try

        {

            // Creating an exception

            NumberFormatException ex =

                       new NumberFormatException("Exception");

            // Setting a cause of the exception

            ex.initCause(new NullPointerException(

                      "This is actual cause of the exception"));

            // Throwing an exception with cause.

            throw ex;

        }

        catch(NumberFormatException ex)

        {

            // displaying the exception

            System.out.println(ex);

            // Getting the actual cause of the exception

            System.out.println(ex.getCause());

        }

    }

}

Output:

java.lang.NumberFormatException: Exception

java.lang.NullPointerException: This is actual cause of the exception

# Infinity or Exception in Java when divide by 0?

<https://www.geeksforgeeks.org/g-fact-33-infinity-or-exception/>

Consider the following code snippets:

public class Geeksforgeeks

{

    public static void main(String[] args)

    {

        double p = 1;

        System.out.println(p/0);

    }

}

**Output**:

Infinity

public class Geeksforgeeks

{

    public static void main(String[] args)

    {

        int p = 1;

        System.out.println(p/0);

    }

}

**Output:**

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

at Geeksforgeeks.main(Geeksforgeeks.java:8)

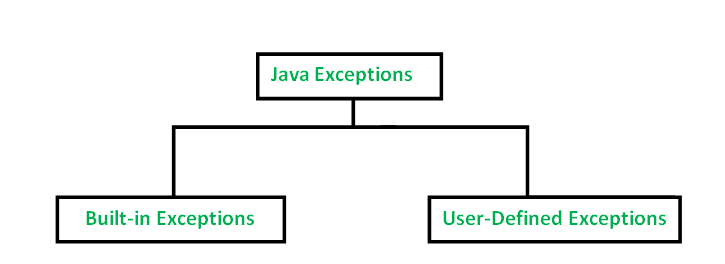
**Explanation**: In the first piece of code, a double value is being divided by 0 while in the other case an integer value is being divide by 0. However the solution for both of them differs.

* In case of double/float division, the output is **Infinity**, the basic reason behind that it implements the floating point arithmetic algorithm which specifies a special values like “Not a number” OR “infinity” for “divided by zero cases” as per IEEE 754 standards.
* In case of integer division, it throws ArithmeticException.

# Types of Exception in Java with Examples

<https://www.geeksforgeeks.org/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.



[**Built-in Exceptions**](https://www.geeksforgeeks.org/built-exceptions-java-examples/)

Built-in exceptions are the exceptions which are available in Java libraries. These exceptions are suitable to explain certain error situations. Below is the list of important built-in exceptions in Java.

1. **ArithmeticException**  
   It is thrown when an exceptional condition has occurred in an arithmetic operation.
2. **ArrayIndexOutOfBoundsException**It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.
3. **ClassNotFoundException**This Exception is raised when we try to access a class whose definition is not found
4. **FileNotFoundException**This Exception is raised when a file is not accessible or does not open.
5. **IOException**It is thrown when an input-output operation failed or interrupted
6. **InterruptedException**It is thrown when a thread is waiting , sleeping , or doing some processing , and it is interrupted.
7. **NoSuchFieldException**It is thrown when a class does not contain the field (or variable) specified
8. **NoSuchMethodException**It is thrown when accessing a method which is not found.
9. **NullPointerException**This exception is raised when referring to the members of a null object. Null represents nothing
10. **NumberFormatException**This exception is raised when a method could not convert a string into a numeric format.
11. **RuntimeException**This represents any exception which occurs during runtime.
12. **StringIndexOutOfBoundsException**It is thrown by String class methods to indicate that an index is either negative than the size of the string

**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;

* The following program illustrates how to create own exception class MyException.
* Details of account numbers, customer names, and balance amounts are taken in the form of three arrays.
* In main() method, the details are displayed using a for-loop. At this time, check is done if in any account the balance amount is less than the minimum balance amount to be ept in the account.
* If it is so, then MyException is raised and a message is displayed “Balance amount is less”.

# Catching base and derived classes as exceptions

<https://www.geeksforgeeks.org/g-fact-60/>

**Exception Handling – catching base and derived classes as exceptions:**

If both base and derived classes are caught as exceptions then catch block of derived class must appear before the base class.

In Java, catching a base class exception before derived is not allowed by the compiler itself.

For example, following Java code fails in compilation with error message “exception Derived has already been caught”

//filename Main.java

class Base extends Exception {}

class Derived extends Base  {}

public class Main {

  public static void main(String args[]) {

    try {

       throw new Derived();

    }

    catch(Base b) {}

    catch(Derived d) {}

  }

}