**Exception Handling**

Exception is an event that can happen during the execution of a program and it disrupts the normal execution flow.

Exceptions can occur from diff situations (scenarios) when user enters wrong data or hardware failure, network connection failure.

When error occurs during execution of java statement one exception object will be created and JRE tries to find the exception handler to handle the exception.

If suitable exception handler is found, then the exception object is passed to that

Handler to process the exception known as catching exception. if there is no exception handler found then application throws the exception and JRE terminates it.

Java exception handling framework used to handle runtime errors only, compile time errors are not handled by exception handling framework.

Throwable is the Parent class of all exception framework

Throwable

Error Exception

Unchecked Exceptions:

Checked Exceptions

Checked Exceptions in java.lang pacakge

ReflectiveOperationException

ClassNotFoundException

InstantiationException

IllegalAccessException

InvocationTargetException

NosuchFieldException

CloneNotSupportedException

InterruptException

Checked Exceptions in Java.IO Pacakge

IO Exception

EofException

FileNotFoundException

InterruptedIOException

UnsupportedEncodingException

UTFDataFormatException

ObjectStreamException

InvalidClassException

NotSerializableException

InvalidObjectException

StreamCorruptException

WriteAbortedException

Checked Exceptions in Java.net package

SocketException

BindException

ConnectException

HttpRetryException

MalFormedURLException

ProtocolException

UnknownHostException

UnknownServiceException

Common checked exception java.sql package

SqlException

BatchUpdateException

SqlClientInfoException

SqlNonTransientException

SqlDataException

SqlFeatureNotSupportedException

SqlIntegrityConstraintVoilationException

SqlSyntaxErrorException

SqlTransientException

SqlTimeOutException

SqlTransacitonRollbackException

SqlTransactionConnectionException

SqlRecoverableException

SqlWarning

Unchecked Exceptions

**4. List of Common Unchecked Exceptions in Java**

Common unchecked exceptions in the java.lang package:

* ArithmeticException
* IndexOutOfBoundsException
  + ArrayIndexOutOfBoundsException
  + StringIndexOutOfBoundsException
* ArrayStoreException
* ClassCastException
* EnumConstantNotPresentException
* IllegalArgumentException
  + IllegalThreadStateException
  + NumberFormatException
* IllegalMonitorStateException
* IllegalStateException
* NegativeArraySizeException
* NullPointerException
* SecurityException
* TypeNotPresentException
* UnsupportedOperationException

Common unchecked exceptions in the java.util package:

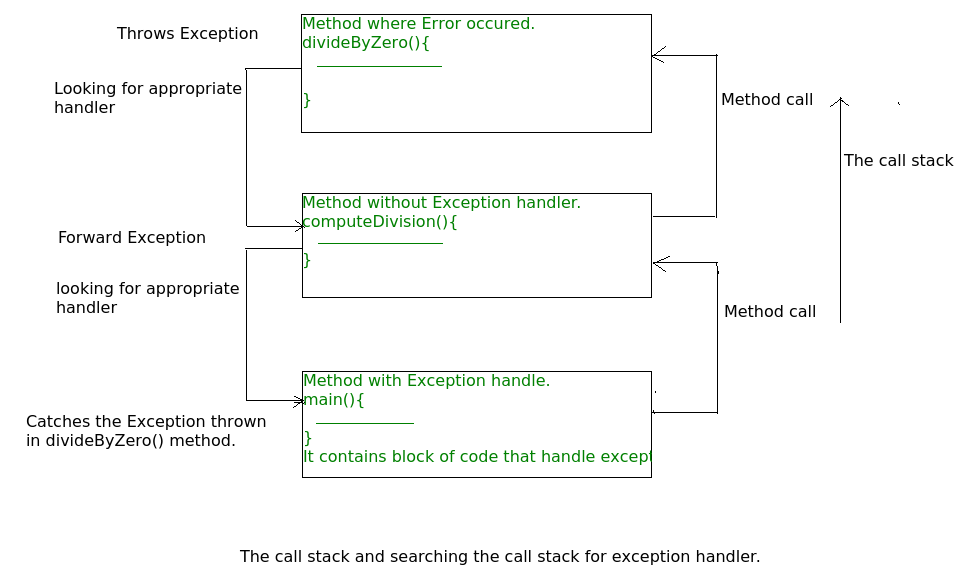
* ConcurrentModificationException
* EmptyStackException
* NoSuchElementException
  + InputMismatchException
* MissingResourceException



**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**.
* Exception in thread "xxx" Name of Exception : Description
* ... ...... .. // Call Stack

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


**Example** :

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // Java program to demonstrate how exception is thrown.  class ThrowsExecp{        public static void main(String args[]){            String str = null;          System.out.println(str.length());        }  } |

Output :

Exception in thread "main" java.lang.NullPointerException

at ThrowsExecp.main(File.java:8)

Let us see an example that illustrate how run-time system searches appropriate exception handling code on the call stack :

|  |
| --- |
| // Java program to demonstrate exception is thrown  // how the runTime system searches th call stack  // to find appropriate exception handler.  class ExceptionThrown  {      // It throws the Exception(ArithmeticException).      // Appropriate Exception handler is not found within this method.      static int divideByZero(int a, int b){            // this statement will cause ArithmeticException(/ by zero)          int i = a/b;            return i;      }        // The runTime System searches the appropriate Exception handler      // in this method also but couldn't have found. So looking forward      // on the call stack.      static int computeDivision(int a, int b) {            int res =0;            try          {            res = divideByZero(a,b);          }          // doesn't matches with ArithmeticException          catch(NumberFormatException ex)          {             System.out.println("NumberFormatException is occured");          }          return res;      }        // In this method found appropriate Exception handler.      // i.e. matching catch block.      public static void main(String args[]){            int a = 1;          int b = 0;            try          {              int i = computeDivision(a,b);            }            // matching ArithmeticException          catch(ArithmeticException ex)          {              // getMessage will print description of exception(here / by zero)              System.out.println(ex.getMessage());          }      }  } |

Output :

/ by zero.

**How Programmer handles an exception?**

**Customized Exception Handling :**Java exception handling is managed via five keywords: **try**, **catch**, [**throw**](https://www.geeksforgeeks.org/throw-throws-java/), [**throws**](https://www.geeksforgeeks.org/throw-throws-java/), and **finally**. Briefly, here is how they work. Program statements that you think can raise exceptions are contained within a try block. If an exception occurs within the try block, it is thrown. Your code can catch this exception (using catch block) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword [throw](https://www.geeksforgeeks.org/throw-throws-java/). Any exception that is thrown out of a method must be specified as such by a [throws](https://www.geeksforgeeks.org/throw-throws-java/) clause. Any code that absolutely must be executed after a try block completes is put in a finally block.

**Need of try-catch clause(Customized Exception Handling)**

Consider the following java program.

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // java program to demonstrate  // need of try-catch clause    class GFG {      public static void main (String[] args) {            // array of size 4.          int[] arr = new int[4];            // this statement causes an exception          int i = arr[4];            // the following statement will never execute          System.out.println("Hi, I want to execute");      }  } |

Output :

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4

at GFG.main(GFG.java:9)

**Explanation :**In the above example an array is defined with size i.e. you can access elements only from index 0 to 3. But you trying to access the elements at index 4(by mistake) that’s why it is throwing an exception.In this case, JVM terminates the program**abnormally**. The statement *System.out.println(“Hi, I want to execute”);* will never execute. To execute it, we must handled the exception using try-catch. Hence to continue normal flow of the program, we need try-catch clause.

**How to use try-catch clause**

try {

// block of code to monitor for errors

// the code you think can raise an exception

}

catch (ExceptionType1 exOb) {

// exception handler for ExceptionType1

}

catch (ExceptionType2 exOb) {

// exception handler for ExceptionType2

}

// optional

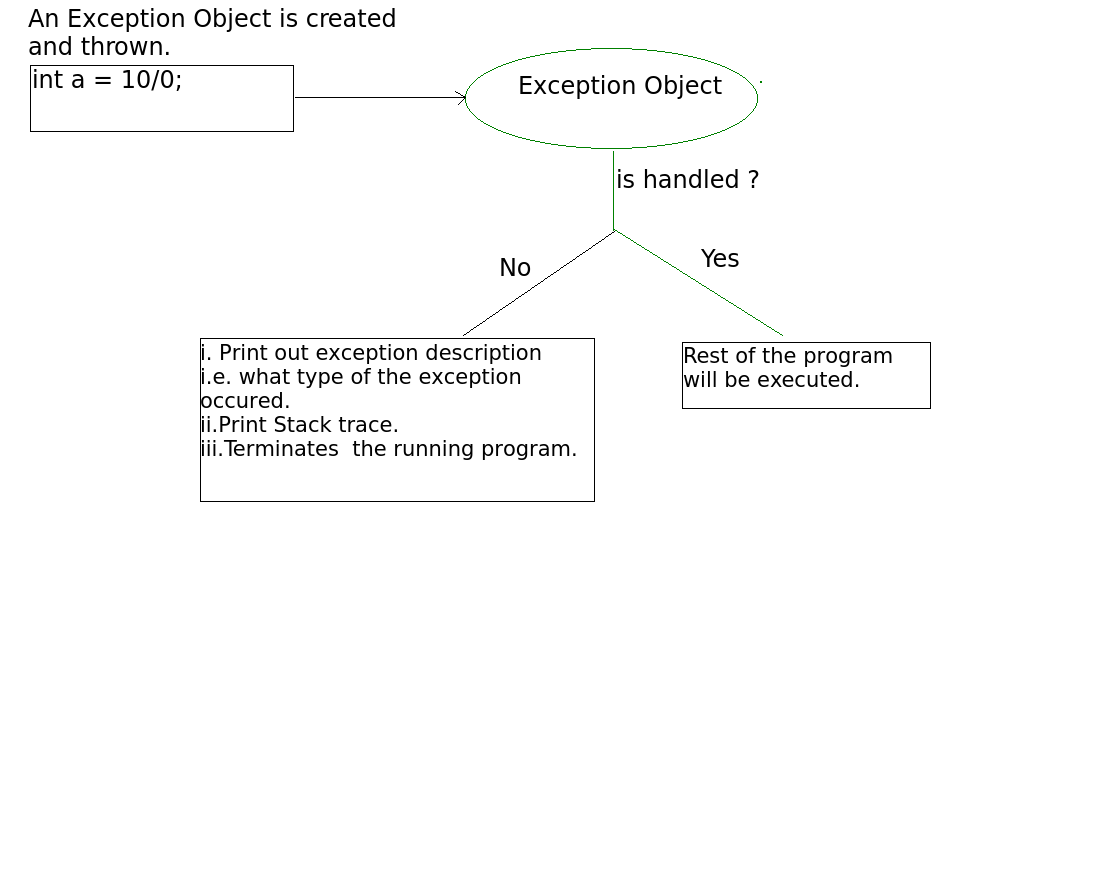
finally {

// block of code to be executed after try block ends

}

**Points to remember :**

* In a method, there can be more than one statements that might throw exception, So put all these statements within its own **try** block and provide separate exception handler within own **catch** block for each of them.
* If an exception occurs within the **try** block, that exception is handled by the exception handler associated with it. To associate exception handler, we must put **catch** block after it. There can be more than one exception handlers. Each **catch** block is a exception handler that handles the exception of the type indicated by its argument. The argument, ExceptionType declares the type of the exception that it can handle and must be the name of the class that inherits from **Throwable** class.
* For each try block there can be zero or more catch blocks, but **only one** finally block.
* The finally block is optional.It always gets executed whether an exception occurred in try block or not . If exception occurs, then it will be executed after **try and catch blocks.** And if exception does not occur then it will be executed after the **try** block. The finally block in java is used to put important codes such as clean up code e.g. closing the file or closing the connection.

**Summary**  


# 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

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| 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.

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| 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.

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

| 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

**Why two types?**  
See [Unchecked Exceptions — The Controversy](http://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html) for details.

**Should we make our exceptions checked or unchecked?**  
Following is the bottom line from [Java documents](http://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html)  
If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception

Chained Exceptions in 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.

**Constructors**Of Throwable class Which support chained exceptions in java :

1. Throwable(Throwable cause) :- Where cause is the exception that causes the current exception.
2. Throwable(String msg, Throwable cause) :- Where msg is the exception message and cause is the exception that causes the current exception.

**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.

Example of using Chained 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

Errors V/s Exceptions In Java

**Errors V/s Exceptions In Java**

* **Error :**An Error “indicates serious problems that a reasonable application should not try to catch.”  
  Both Errors and Exceptions are the subclasses of java.lang.Throwable class. Errors are the conditions which cannot get recovered by any handling techniques. It surely cause termination of the program abnormally. Errors belong to unchecked type and mostly occur at runtime. Some of the examples of errors are Out of memory error or a System crash error.

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // Java program illustrating stack overflow error  // by doing infinite recursion    class StackOverflow {      public static void test(int i)      {          // No correct as base condition leads to          // non-stop recursion.          if (i == 0)              return;          else {              test(i++);          }      }  }  public class ErrorEg {        public static void main(String[] args)      {            // eg of StackOverflowError          StackOverflow.test(5);      }  } |

**Output:**

Exception in thread "main" java.lang.StackOverflowError

at StackOverflow.test(ErrorEg.java:7)

at StackOverflow.test(ErrorEg.java:7)

at StackOverflow.test(ErrorEg.java:7)

at StackOverflow.test(ErrorEg.java:7)

at StackOverflow.test(ErrorEg.java:7)

...

* **Exceptions :** An Exception “indicates conditions that a reasonable application might want to catch.”  
  Exceptions are the conditions that occur at runtime and may cause the termination of program. But they are recoverable using try, catch and throw keywords. Exceptions are divided into two catagories : [checked and unchecked exceptions](https://www.geeksforgeeks.org/checked-vs-unchecked-exceptions-in-java/). Checked exceptions like IOException known to the compiler at compile time while unchecked exceptions like ArrayIndexOutOfBoundException known to the compiler at runtime. It is mostly caused by the program written by the programmer.

|  |
| --- |
| // Java program illustrating exception thrown  // by AritmeticExcpetion class    public class ExceptionEg {        public static void main(String[] args)      {          int a = 5, b = 0;            // Attempting to divide by zero          try {              int c = a / b;          }          catch (ArithmeticException e) {              e.printStackTrace();          }      }  } |

**Output:**

java.lang.ArithmeticException: / by zero

at ExceptionEg.main(ExceptionEg.java:8)

**Summary of Differences**

|  |  |
| --- | --- |
| **ERRORS** | **EXCEPTIONS** |
| Recovering from Error is not possible. | We can recover from exceptions by either using try-catch block or throwing exceptions back to caller. |
| All errors in java are unchecked type. | Exceptions include both checked as well as unchecked type. |
| Errors are mostly caused by the environment in which program is running. | Program itself is responsible for causing exceptions. |
| Errors occur at runtime and not known to the compiler. | All exceptions occurs at runtime but checked exceptions are known to compiler while unchecked are not. |
| They are defined in java.lang.Error package. | They are defined in java.lang.Exception package |
| Examples : java.lang.StackOverflowError, java.lang.OutOfMemoryError | Examples : Checked Exceptions : SQLException, IOException Unchecked Exceptions : ArrayIndexOutOfBoundException, NullPointerException, ArithmeticException. |

Rounding off errors in Java

Compacting many infinite real numbers into a finite number of bits requires an approximate representation. Most programs store the result of integer computations 32 or 64 bits max. Given any fixed number of bits, most calculations with real numbers will produce quantities that cannot be exactly represented using that many bits. Therefore the result of a floating-point calculation must often be rounded in order to fit back into its finite representation. This rounding error is a characteristic feature of floating-point computation.

Therefore, while handling calculations in floating point numbers, (especially if calculations are in term of money), we need to take care of round off errors in a programming language.  
Let’s see an example:

|  |
| --- |
| public class Main {      public static void main(String[] args)      {          double a = 0.7;          double b = 0.9;          double x = a + 0.1;          double y = b - 0.1;            System.out.println("x = " + x);          System.out.println("y = " + y );          System.out.println(x == y);      }  } |

Output:

x = 0.7999999999999999

y = 0.8

false

Here, the answer is not what we expected reason being the rounding off done by java compiler.

**Reason behind round off error**

float and double data types implement IEEE floating point 754 specification. This means that numbers are represented in a form like:

SIGN FRACTION \* 2 ^ EXP

0.15625 = (0.00101)2, which in floating-point format is represented as: 1.01 \* 2^-3  
Not all fractions can be represented exactly as a fraction of a power of two. As a simple example, 0.1 = (0.000110011001100110011001100110011001100110011001100110011001… )2 and thus cannot be stored inside a floating-point variable.

Another Example:

|  |
| --- |
| public class Main {      public static void main(String[] args)      {          double a = 0.7;          double b = 0.9;          double x = a + 0.1;          double y = b - 0.1;            System.out.println("x = " + x);          System.out.println("y = " + y );          System.out.println(x == y);      }  } |

Output:

x = 0.7999999999999999

y = 0.8

false

Another example:

|  |
| --- |
| public class Main {      public static void main(String args[])      {          double a = 1.0;          double b = 0.10;          double x = 9 \* b;          a = a - (x);            // Value of a is expected as 0.1          System.out.println("a = " + a);      }  } |

Output:

a = 0.09999999999999998

**How to rectify round off errors?**

* **Round the result:**The Round() function can be used to minimize any effects of floating point arithmetic storage inaccuracy. The user can round numbers to the number of decimal places that is required by the calculation. For example, while working with currency, you would likely round to 2 decimal places.
* **Algorithms and Functions:**Use numerically stable algorithms or design your own functions to handle such cases. You can truncate/round digits of which you are not sure they are correct (you can calculate numeric precision of operations too)
* **BigDecimal Class:**You may use the [java.math.BigDecimal](http://docs.oracle.com/javase/1.5.0/docs/api/java/math/BigDecimal.html) class, which is designed to give us accuracy especially in case of big fractional numbers.

The following program shows how the error can be removed:

|  |
| --- |
| public class Main {      public static void main(String args[])      {          double a = 1.0;          double b = 0.10;          double x = 9 \* b;          a = a - (x);        /\* We use Math.round() function to round the answer to           closest long, then we multiply and divide by 1.0 to           to set the decimal places to 1 place (this can be done           according to the requirements.\*/          System.out.println("a = " + Math.round(a\*1.0)/1.0);      }  } |

Output:

0.1

Built-in Exceptions in Java with examples

[Types of Exceptions in Java](https://www.geeksforgeeks.org/types-of-exception-in-java-with-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.  
**Examples of Built-in Exception:**

1. **Arithmetic exception :**It is thrown when an exceptional condition has occurred in an arithmetic operation.

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // Java program to demonstrate  // ArithmeticException  class ArithmeticException\_Demo {  public static void main(String args[])      {          try {              int a = 30, b = 0;              int c = a / b; // cannot divide by zero              System.out.println("Result = " + c);          }          catch (ArithmeticException e) {              System.out.println("Can't divide a number by 0");          }      }  } |

**Output:**

Can't divide a number by 0

1. **ArrayIndexOutOfBounds Exception :**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.

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // Java program to demonstrate  // ArrayIndexOutOfBoundException  class ArrayIndexOutOfBound\_Demo {  public static void main(String args[])      {          try {              int a[] = new int[5];              a[6] = 9; // accessing 7th element in an array of              // size 5          }          catch (ArrayIndexOutOfBoundsException e) {              System.out.println("Array Index is Out Of Bounds");          }      }  } |

**Output:**

Array Index is Out Of Bounds

1. **ClassNotFoundException :** This Exception is raised when we try to access a class whose definition is not found.

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // Java program to illustrate the  // concept of ClassNotFoundException  class Bishal {    } class Geeks {    } class MyClass {  public static void main(String[] args)      {          Object o = class.forName(args[0]).newInstance();          System.out.println("Class created for" + o.getClass().getName());      }  } |

**Output:**

ClassNotFoundException

1. **FileNotFoundException :** This Exception is raised when a file is not accessible or does not open.

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // Java program to demonstrate  // FileNotFoundException  import java.io.File;  import java.io.FileNotFoundException;  import java.io.FileReader;  class File\_notFound\_Demo {    public static void main(String args[])      {          try {                // Following file does not exist              File file = new File("[E://](file:///E:\) file.txt");                FileReader fr = new FileReader(file);          }          catch (FileNotFoundException e) {              System.out.println("File does not exist");          }      }  } |

**Output:**

File does not exist

1. **IOException :** It is thrown when an input-output operation failed or interrupted

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // Java program to illustrate IOException  import java.io.\*;  class Geeks {  public static void main(String args[])      {          FileInputStream f = null;          f = new FileInputStream("abc.txt");          int i;          while ((i = f.read()) != -1) {              System.out.print((char)i);          }          f.close();      }  } |

**Output:**

error: unreported exception IOException; must be caught or declared to be thrown

1. **InterruptedException :**It is thrown when a thread is waiting, sleeping, or doing some processing, and it is interrupted.

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // Java Program to illustrate  // InterruptedException  class Geeks {  public static void main(String args[])      {          Thread t = new Thread();          t.sleep(10000);      }  } |

**Output:**

error: unreported exception InterruptedException; must be caught or declared to be thrown

1. **NoSuchMethodException :**t is thrown when accessing a method which is not found.

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // Java Program to illustrate  // NoSuchMethodException  class Geeks {  public Geeks()      {          Class i;          try {              i = Class.forName("java.lang.String");              try {                  Class[] p = new Class[5];              }              catch (SecurityException e) {                  e.printStackTrace();              }              catch (NoSuchMethodException e) {                  e.printStackTrace();              }          }          catch (ClassNotFoundException e) {              e.printStackTrace();          }      }    public static void main(String[] args)      {          new Geeks();      }  } |

**Output:**

error: exception NoSuchMethodException is never thrown

in body of corresponding try statement

1. **NullPointerException :**This exception is raised when referring to the members of a null object. Null represents nothing

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // Java program to demonstrate NullPointerException  class NullPointer\_Demo {  public static void main(String args[])      {          try {              String a = null; // null value              System.out.println(a.charAt(0));          }          catch (NullPointerException e) {              System.out.println("NullPointerException..");          }      }  } |

**Output:**

NullPointerException..

1. **NumberFormatException :**This exception is raised when a method could not convert a string into a numeric format.

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // Java program to demonstrate  // NumberFormatException  class NumberFormat\_Demo {  public static void main(String args[])      {          try {              // "akki" is not a number              int num = Integer.parseInt("akki");                System.out.println(num);          }          catch (NumberFormatException e) {              System.out.println("Number format exception");          }      }  } |

**Output:**

Number format exception

1. **StringIndexOutOfBoundsException :** It is thrown by String class methods to indicate that an index is either negative than the size of the string.

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // Java program to demonstrate  // StringIndexOutOfBoundsException  class StringIndexOutOfBound\_Demo {  public static void main(String args[])      {          try {              String a = "This is like chipping "; // length is 22              char c = a.charAt(24); // accessing 25th element              System.out.println(c);          }          catch (StringIndexOutOfBoundsException e) {              System.out.println("StringIndexOutOfBoundsException");          }      }  } |

**Output:**

StringIndexOutOfBoundsException

**Some other important Exceptions**

1. **ClassCastException**

|  |
| --- |
| // Java Program to illustrate  // ClassCastException  class Test {  public static void main(String[] args)      {          String s = new String("Geeks");          Object o = (Object)s;          Object o1 = new Object();          String s1 = (String)o1;      }  } |

Output:

Exception in thread "main" java.lang.ClassCastException:

java.lang.Object cannot be cast to java.lang.String

1. **StackOverflowError**

|  |
| --- |
| // Java Program to illustrate  // StackOverflowError  class Test {  public static void main(String[] args)      {          m1();      }  public static void m1()      {          m2();      }  public static void m2()      {          m1();      }  } |

Output:

Exception in thread "main" java.lang.StackOverflowError

1. **NoClassDefFoundError**

|  |
| --- |
| // Java Program to illustrate  // NoClassDefFoundError  class Test //      {  public static void main(String[] args)      {          System.out.println("HELLO GEEKS");      }  } |

Output:

Note: If the corresponding Test.class file is not found

during compilation then we will get Run-time Exception

saying Exception in thread "main" java.lang.NoClassDefFoundError

1. **ExceptionInInitializerError**  
   **Code 1:**

|  |
| --- |
| // Java Program to illustrate  // ExceptionInInitializerError  class Test {      static int x = 10 / 0;  public static void main(String[] args)      {      }  } |

Output:

Exception in thread "main" java.lang.ExceptionInInitializerError

Caused by: java.lang.ArithmeticException: / by zero

**Code 2 :**

|  |
| --- |
| // Java Program to illustrate  // ExceptionInInitializerError  class Test {      static      {          String s = null;          System.out.println(s.length());      }  public static void main(String[] args)      {      }  } |

Output:

Exception in thread "main" java.lang.ExceptionInInitializerError

Caused by: java.lang.NullPointerException

**Explanation :** The above exception occurs whenever while executing static variable assignment and static block if any Exception occurs.

1. **IllegalArgumentException**

|  |
| --- |
| // Java Program to illustrate  // IllegalArgumentException  class Test {  public static void main(String[] args)      {          Thread t = new Thread();          Thread t1 = new Thread();          t.setPriority(7); // Correct          t1.setPriority(17); // Exception      }  } |

Output:

Exception in thread "main" java.lang.IllegalArgumentException

**Explanation:**The Exception occurs explicitly either by the programmer or by API developer to indicate that a method has been invoked with Illegal Argument.

1. **IllegalArgumentException**

|  |
| --- |
| // Java Program to illustrate  // IllegalStateException  class Test {  public static void main(String[] args)      {          Thread t = new Thread();          t.start();          t.start();      }  } |

Output:

Exception in thread "main" java.lang.IllegalThreadStateException

**Explanation :** The above exception rises explicitly either by programmer or by API developer to indicate that a method has been invoked at wrong time.

1. **AssertionError**

|  |
| --- |
| // Java Program to illustrate  // AssertionError  class Test {  public static void main(String[] args)      {          // If x is not greater than or equal to 10          // then we will get the run-time exception          assert(x >= 10);      }  } |

Output:

Exception in thread "main" java.lang.AssertionError

**Explanation :**The above exception rises explicitly by the programmer or by API developer to indicate that assert statement fails.

throw and throws in 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");**

But this exception i.e, *Instance* must be of type **Throwable** or a subclass of **Throwable**. For example Exception is a sub-class of Throwable and [user defined exceptions typically extend Exception class](https://www.geeksforgeeks.org/g-fact-32-user-defined-custom-exception-in-java/). Unlike C++, data types such as int, char, floats or non-throwable classes cannot be used as exceptions.

The flow of execution of the program stops immediately after the throw statement is executed and the nearest enclosing **try** block is checked to see if it has a **catch** statement that matches the type of exception. If it finds a match, controlled is transferred to that statement otherwise next enclosing **try** block is checked and so on. If no matching**catch**is found then the default exception handler will halt the program.

|  |
| --- |
| // Java program that demonstrates the use of throw  class ThrowExcep  {      static void fun()      {          try          {              throw new NullPointerException("demo");          }          catch(NullPointerException e)          {              System.out.println("Caught inside fun().");              throw e; // rethrowing the exception          }      }        public static void main(String args[])      {          try          {              fun();          }          catch(NullPointerException e)          {              System.out.println("Caught in main.");          }      }  } |

Output:

Caught inside fun().

Caught in main.

**Another Example:**

|  |
| --- |
| // Java program that demonstrates the use of throw  class Test  {      public static void main(String[] args)      {          System.out.println(1/0);      }  } |

Output:

Exception in thread "main" java.lang.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

We can use throws keyword to delegate the responsibility of exception handling to the caller (It may be a method or JVM) then caller method is responsible to handle that exception.

|  |
| --- |
| // Java program to illustrate error in case  // of unhandled exception  class tst  {      public static void main(String[] args)      {          Thread.sleep(10000);          System.out.println("Hello Geeks");      }  } |

Output:

error: unreported exception InterruptedException; must be caught or declared to be thrown

**Explanation :**In the above program, we are getting compile time error because there is a chance of exception if the main thread is going to sleep, other threads get the chance to execute main() method which will cause InterruptedException.

|  |
| --- |
| // Java program to illustrate throws  class tst  {      public static void main(String[] args)throws InterruptedException      {          Thread.sleep(10000);          System.out.println("Hello Geeks");      }  } |

Output:

Hello Geeks

**Explanation :**In the above program, by using throws keyword we handled the InterruptedException and we will get the output as **Hello Geeks**

**Another Example:**

|  |
| --- |
| // Java program to demonstrate working of throws  class ThrowsExecp  {      static void fun() throws IllegalAccessException      {          System.out.println("Inside fun(). ");          throw new IllegalAccessException("demo");      }      public static void main(String args[])      {          try          {              fun();          }          catch(IllegalAccessException e)          {              System.out.println("caught in main.");          }      }  } |

Output:

Inside fun().

caught in main.

**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.
* Try-with-resources was a great feature introduced in Java 7 to automatically manage resources using an AutoCloseable interface. This helps a lot, of course, as we have no need to close the resources explicitly in our code.

## **Java 7 Code**

* public void loadDataFromDB() throws SQLException {
* Connection dbCon = DriverManager.getConnection("url", "user", "password");
* try (ResultSet rs = dbCon.createStatement().executeQuery("select \* from emp")) {
* while (rs.next()) {
* System.out.println("In loadDataFromDB() =====>>>>>>>>>>>> " + rs.getString(1));
* }
* } catch (SQLException e) {
* System.out.println("Exception occurs while reading the data from DB ->" + e.getMessage());
* } finally {
* if (null != dbCon)
* dbCon.close();
* }
* }
* In the above example, we defined the Connection object outside the try-with-resource block, hence we need to close it explicitly in the finally block. We can't put a dbCon object reference in our try-with-resource block, which was an obvious bug in Java 7. Fortunately, it was fixed in Java 9.

## Java 9's Solution

* Java 9 will give us this feature, available in the java.base module, so the above code could be rewritten as below:
* public void loadDataFromDB() throws SQLException {
* Connection dbCon = DriverManager.getConnection("url", "user", "password");
* try (dbCon; ResultSet rs = dbCon.createStatement().executeQuery("select \* from emp")) {
* while (rs.next()) {
* System.out.println("In loadDataFromDB() =====>>>>>>>>>>>> " + rs.getString(1));
* }
* } catch (SQLException e) {
* System.out.println("Exception occurs while reading the data from DB ->" + e.getMessage());
* }
* }
* You can see that the object reference **dbCon**has been kept inside the try-with-resource block, so any resource as a final or effectively final variable can be placed in try-with-resource blocks and would be eligible for **automatic resource management**.

In Java, it is possible that your program may encounter exceptions, for which the language provides try-catch statements to handle them. However, there is a possibility that the piece of code enclosed inside the ‘try’ block may be vulnerable to more than one exceptions. For example, take a look at the following sample code:

|  |
| --- |
| // A sample Java code with a try catch block  // where the try block has only one catch block  // to handle all possible exceptions    // importing class Random to generate a random number as input  import java.util.Random;  class A {      void func(int n)      {          try {                // this will throw ArithmeticException if n is 0              int x = 10 / n;              int y[] = new int[n];                // this will throw ArrayIndexOutOfBoundsException              // if the value of x surpasses              // the highest index of this array              y[x] = 10;          }          catch (Exception e) {              System.out.println("Exception occurred");          }      }      public static void main(String a[])      {          new A().func(new Random().nextInt(10));      }  } |

**Output in case of either of the exceptions**:

Exception occurred

As you can see from the above code, there is a possibility that either of the two exceptions mentioned above can occur. In order to handle them both, the catch statement is made to accept any exception that can occur by passing a reference to the Exception class, that is the parent of all exception classes. However, this catch statement does the **same thing** for all kinds of exceptions.

**Specify Custom Actions for different Exceptions**

In order to specify custom actions in such cases, programmers usually put multiple catch statements, such as in the following example:

|  |
| --- |
| // A sample Java code with one try block  // having multiple catch blocks to catch  // different exceptions    // importing class Random to generate a random number as input  import java.util.Random;  class A {      void func(int n)      {          try {                // this will throw ArithmeticException if n is 0              int x = 10 / n;              int y[] = new int[n];                // this will throw ArrayIndexOutOfBoundsException              // if the value of x surpasses              // the highest index of this array              y[x] = 10;          }          catch (ArithmeticException e) {              System.out.println("Dividing by 0");          }          catch (ArrayIndexOutOfBoundsException e) {              System.out.println("That index doesn't exist");          }      }      public static void main(String a[])      {          new A().func(new Random().nextInt(10));      }  } |

**Output**:

a) In case of ArithmeticException: Dividing by 0

b) In case of ArrayIndexOutOfBoundsException: That index doesn't exist

**Specify Custom Actions for different Exceptions using instanceof**

However, there is a way to do the same thing by using **only one catch block**. To do so, java provides an operator: [**instanceof**](https://www.geeksforgeeks.org/java-instanceof-and-its-applications/).  
By using this operator, we can specify custom actions for the different exceptions that occur. The following program demonstrates how:

|  |
| --- |
| // Java program to demonstrate the use of  // instanceof to specify different actions for  // different exceptions using only one catch block    // importing class Random to generate a random number as input  import java.util.Random;  class A {      void func(int n)      {          try {                // this will throw ArithmeticException if n is 0              int x = 10 / n;              int y[] = new int[n];              y[x] = 10;                // this will throw ArrayIndexOutOfBoundsException              // if the value of x surpasses              // the highest index of this array              System.out.println("No exception arose");          }          catch (Exception e) {              if (e instanceof ArithmeticException)                  System.out.println("Can't divide by 0");              if (e instanceof ArrayIndexOutOfBoundsException)                  System.out.println("This index doesn't exist in this array");          }      }      public static void main(String a[])      {          new A().func(new Random().nextInt(10));      }  } |

**Output:**

a) In case of ArithmeticException:

Can't divide by 0

b) In case of ArrayIndexOutOfBoundsException:

This index doesn't exist in this array

c) In case of no exception: No exception arose

By using the **instanceof** operator in exception handling, it is easy to achieve the aforementioned objective and at the same time, it makes your code less complicated. One thing that has to be noted here is that as soon as the try block encounters an exception, the control will directly jump to the catch block, to handle, thereby, preventing the rest of the body of the try block from executing. As a result, **even though there may be possibility of more than one exceptions occurring, the try block will only throw one exception, before shifting control to the catch block.**