An exception (or exceptional event) is a problem that arises during the execution of a program. When an **Exception** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

* A user has entered an invalid data.
* A file that needs to be opened cannot be found.
* A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Based on these, we have three categories of Exceptions. You need to understand them to know how exception handling works in Java.

* **Checked exceptions** − A checked exception is an exception that is checked (notified) by the compiler at compilation-time, these are also called as compile time exceptions. These exceptions cannot simply be ignored, the programmer should take care of (handle) these exceptions.

For example, if you use **FileReader** class in your program to read data from a file, if the file specified in its constructor doesn't exist, then a *FileNotFoundException* occurs, and the compiler prompts the programmer to handle the exception.

Example

[Live Demo](http://tpcg.io/9u4a5O)

import java.io.File;

import java.io.FileReader;

public class FilenotFound\_Demo {

public static void main(String args[]) {

File file = new File("E://file.txt");

FileReader fr = new FileReader(file);

}

}

If you try to compile the above program, you will get the following exceptions.

Output

C:\>javac FilenotFound\_Demo.java

FilenotFound\_Demo.java:8: error: unreported exception FileNotFoundException; must be caught or declared to be thrown

FileReader fr = new FileReader(file);

^

1 error

**Note** − Since the methods **read()** and **close()** of FileReader class throws IOException, you can observe that the compiler notifies to handle IOException, along with FileNotFoundException.

* **Unchecked exceptions** − An unchecked exception is an exception that occurs at the time of execution. These are also called as **Runtime Exceptions**. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

For example, if you have declared an array of size 5 in your program, and trying to call the 6th element of the array then an *ArrayIndexOutOfBoundsExceptionexception* occurs.

Example

[Live Demo](http://tpcg.io/7CUnsL)

public class Unchecked\_Demo {

public static void main(String args[]) {

int num[] = {1, 2, 3, 4};

System.out.println(num[5]);

}

}

If you compile and execute the above program, you will get the following exception.

Output

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5

at Exceptions.Unchecked\_Demo.main(Unchecked\_Demo.java:8)

* **Errors** − These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

Exception Hierarchy

All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class.

Errors are abnormal conditions that happen in case of severe failures, these are not handled by the Java programs. Errors are generated to indicate errors generated by the runtime environment. Example: JVM is out of memory. Normally, programs cannot recover from errors.

The Exception class has two main subclasses: IOException class and RuntimeException Class.

### Exceptions1

### https://cdn.journaldev.com/wp-content/uploads/2013/07/exception-hierarchy.png

### ****Checked Exceptions****

* IOException – This exception is typically a way to say that something on the network, filesystem, or database failed.

### ****7.2. RuntimeExceptions****

* ArrayIndexOutOfBoundsException – this exception means that we tried to access a non-existent array index, like when trying to get index 5 from an array of length 3.
* ClassCastException – this exception means that we tried to perform an illegal cast, like trying to convert a String into a List. We can usually avoid it by performing defensive instanceof checks before casting.
* IllegalArgumentException – this exception is a generic way for us to say that one of the provided method or constructor parameters is invalid.
* IllegalStateException – This exception is a generic way for us to say that our internal state, like the state of our object, is invalid.
* NullPointerException – This exception means we tried to reference a null object. We can usually avoid it by either performing defensive null checks or by using Optional.
* NumberFormatException – This exception means that we tried to convert a String into a number, but the string contained illegal characters, like trying to convert “5f3” into a number.

### ****7.3. Errors****

* StackOverflowError – this exception means that the stack trace is too big. This can sometimes happen in massive applications; however, it usually means that we have some infinite recursion happening in our code.
* NoClassDefFoundError – this exception means that a class failed to load either due to not being on the classpath or due to failure in static initialization.
* OutOfMemoryError –  this exception means that the JVM doesn't have any more memory available to allocate for more objects. Sometimes, this is due to a memory leak.

Java try and catch

The try statement allows you to define a block of code to be tested for errors while it is being executed.

The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

The try and catch keywords come in pairs:

Syntax

try {

// *Block of code to try*

}

catch(Exception *e*) {

// *Block of code to handle errors*

}

Consider the following example:

This will generate an error, because **myNumbers[10]** does not exist.

public class MyClass {

public static void main(String[ ] args) {

int[] myNumbers = {1, 2, 3};

System.out.println(myNumbers[10]); // error!

}

}

The output will be something like this:

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10  
        at MyClass.main(MyClass.java:4)

[Run example »](https://www.w3schools.com/java/showjava.asp?filename=demo_try_error)

If an error occurs, we can use try...catch to catch the error and execute some code to handle it:

**Union *catch* Blocks**

When we know that the way we handle errors is going to be the same, though, Java 7 introduced the ability to catch multiple exceptions in the same block:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public int getPlayerScore(String playerFile) {      try (Scanner contents = new Scanner(new File(playerFile))) {          return Integer.parseInt(contents.nextLine());      } catch (IOException | NumberFormatException e) {          logger.warn("Failed to load score!", e);          return 0;      }  } |

Finally

The finally statement lets you execute code, after try...catch, regardless of the result:

Example

public class MyClass {

public static void main(String[] args) {

try {

int[] myNumbers = {1, 2, 3};

System.out.println(myNumbers[10]);

} catch (Exception e) {

System.out.println("Something went wrong.");

} finally {

System.out.println("The 'try catch' is finished.");

}

}

}

The throw keyword

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");**

The throw statement allows you to create a custom error.

The throw statement is used together with an **exception type**. There are many exception types available in Java: ArithmeticException, FileNotFoundException, ArrayIndexOutOfBoundsException, SecurityException, etc.

The exception type is often used together with a custom **method**. Don't worry if you don't understand the example below, you will learn more about methods in the next chapter:

Example

Throw an exception if **age** is below 18 (print "Access denied"). If age is 18 or older, print "Access granted":

public class MyClass {

static void checkAge(int age) {

if (age < 18) {

throw new ArithmeticException("Access denied - You must be at least 18 years old.");

}

else {

System.out.println("Access granted - You are old enough!");

}

}

public static void main(String[] args) {

checkAge(15); // Set age to 15 (which is below 18...)

}

}

The output will be:

Exception in thread "main" java.lang.ArithmeticException: Access denied - You must be at least 18 years old.  
        at MyClass.checkAge(MyClass.java:4)  
        at MyClass.main(MyClass.java:12)

// 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.");

        }

    }

}

**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().

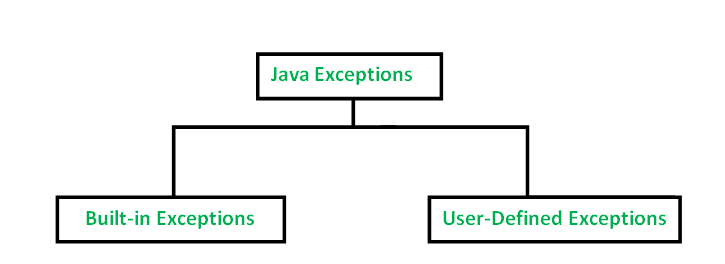
caught in main.

**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.

Actually **Throws** keyword is used to throw the checked exceptions which a programmer doesn’t want to handle it so a programmer throws these exception so that compiler did not give an error but **throw** keyword is used to throw an built-in-exception or user defined exception explicitly so both concepts are different so go through these concepts separately and then read these differences then you will did not get any confusion.

# **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

**Examples of Built-in Exception:**

* **Arithmetic exception**

|  |
| --- |
| // 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

* **NullPointer Exception**

|  |
| --- |
| //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..

* **StringIndexOutOfBound Exception**

|  |
| --- |
| // 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

* **FileNotFound Exception**

|  |
| --- |
| //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.txt](file:///E:\file.txt)");                FileReader fr = new FileReader(file);          } catch (FileNotFoundException e) {             System.out.println("File does not exist");          }      }  } |

**Output:**

File does not exist

* **NumberFormat Exception**

|  |
| --- |
| // 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

* **ArrayIndexOutOfBounds Exception**

|  |
| --- |
| // 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

**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”.

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // Java program to demonstrate user defined exception    // This program throws an exception whenever balance  // amount is below Rs 1000  class MyException extends Exception  {      //store account information      private static int accno[] = {1001, 1002, 1003, 1004};        private static String name[] =                   {"Nish", "Shubh", "Sush", "Abhi", "Akash"};        private static double bal[] =           {10000.00, 12000.00, 5600.0, 999.00, 1100.55};        // default constructor      MyException() {    }        // parametrized constructor      MyException(String str) { super(str); }        // write main()      public static void main(String[] args)      {          try  {              // display the heading for the table              System.out.println("ACCNO" + "\t" + "CUSTOMER" +                                             "\t" + "BALANCE");                // display the actual account information              for (int i = 0; i < 5 ; i++)              {                  System.out.println(accno[i] + "\t" + name[i] +                                                 "\t" + bal[i]);                    // display own exception if balance < 1000                  if (bal[i] < 1000)                  {                      MyException me =                         new MyException("Balance is less than 1000");                      throw me;                  }              }          } //end of try            catch (MyException e) {              e.printStackTrace();          }      }  } |

RunTime Error

MyException: Balance is less than 1000

at MyException.main(fileProperty.java:36)

**Output:**

ACCNO CUSTOMER BALANCE

1001 Nish 10000.0

1002 Shubh 12000.0

1003 Sush 5600.0

1004 Abhi 999.0

Examples copied from below url

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

|  |  |
| --- | --- |
| **THROW** | **THROWS** |
| throw keyword is used to throw an exception explicitly. | throws keyword is used to declare one or more exceptions, separated by commas. |
| Only single exception is thrown by using throw. | Multiple exceptions can be thrown by using throws. |
| throw keyword is used within the method. | throws keyword is used with the method signature. |
| Syntax wise throw keyword is followed by the instance variable. | Syntax wise throws keyword is followed by exception class names. |
| Checked exception cannot be propagated using throw only.Unchecked exception can be propagated using throw. | For the propagation checked exception must use throws keyword followed by specific exception class name. |

Hierarchy of the try catch exception

Lower exceptions to higher exception

Example

try{

throw new FileNotFoundException();

} catch (FileNotFoundException e){

e.printStackTrace();

} catch (IOException e){ // The compiler warns that all the Exceptions possibly

// catched by IOException are already catched even though

// an IOException is not necessarily a FNFException

e.printStackTrace();

} catch (Exception e){

e.printStackTrace();

}