# [Exception Handling In Java](https://javaconceptoftheday.com/exception-handling-java/)

An exception is a problem that arises during the execution of a program. An exception can occur for many different reasons. Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Exception Handling in java is implemented using five keywords.

1) try       2)catch       3)finally       4)throw         5)throws

****try block :****In try block, keep those statements which may throw exceptions during run time.

****catch block :**** This block handles the exceptions thrown by try block. It takes one argument of type java.lang.Exception.

****finally block :**** Whether exception is thrown or not and thrown exception is caught or not, this block will be always executed.

**public** **class** ExceptionHandling

{

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

    {

        String[] s = {"abc", "123", "xyz", "456"};   //String Array containing valid and invalid numeric values

**for** (**int** i = 0; i < s.length; i++)

        {

**try**

            {

**int** intValue = Integer.parseInt(s[i]); //This statement may throw NumberFormatException

            }

**catch**(NumberFormatException ex)

            {

                System.out.println("The thrown NumberFormatException will be caught here");

            }

**finally**

            {

                System.out.println("This block is always executed");

            }

        }

    }

}

* When a statement throws an exception in the try block, the remaining part of the try block will not be executed. Program control comes out of the try block and enters directly into catch block.
* try, catch and finally blocks form one unit. i.e You can’t keep other statements in between try, catch and finally blocks.
* You can display the description of an exception thrown using Exception object in the catch block.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | **public** **class** ExceptionHandling  {  **public** **static** **void** main(String[] args)      {  **try**          {  **int** i = 10/0;           //This statement throws ArithmeticException                System.out.println("This statement will not be executed");          }  //You can't keep statements here  **catch**(Exception ex)          {              System.out.println("This block is executed immediately after an exception is thrown");  ex.printStackTrace();     //This prints stack trace of exception          }  //You can't keep statements here  **finally**          {              System.out.println("This block is always executed");          }      }  } |
|  |  |

**Multiple Catch Blocks In Java:**

**public** **class** ExceptionHandling

{

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

    {

        String[] s = {"abc", "123", **null**, "xyz"};   //String array containing one null object

**for** (**int** i = 0; i < 6; i++)

        {

**try**

            {

**int** a = s[i].length() + Integer.parseInt(s[i]);

                //This statement may throw NumberFormatException, NullPointerException and ArrayIndexOutOfBoundsException

            }

**catch**(NumberFormatException ex)

            {

                System.out.println("NumberFormatException will be caught here");

            }

**catch** (ArrayIndexOutOfBoundsException ex)

            {

                System.out.println("ArrayIndexOutOfBoundsException will be caught here");

            }

**catch** (NullPointerException ex)

            {

                System.out.println("NullPointerException will be caught here");

            }

            System.out.println("After executing respective catch block, this statement will be executed");

        }

    }

}

* From Java 7 onward, there is one more way for handling multiple exceptions. Multiple exceptions thrown by the try block can be handled by a single catch block using ****pipe (|) operator****. By using pipe operator, the above example can be written as,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | **public** **class** ExceptionHandling  {  **public** **static** **void** main(String[] args)      {          String[] s = {"abc", "123", **null**, "xyz"};   //String array containing one null object    **for** (**int** i = 0; i < 6; i++)          {  **try**              {  **int** a = s[i].length() + Integer.parseInt(s[i]);                    //This statement may throw NumberFormatException, NullPointerException and ArrayIndexOutOfBoundsException              }    **catch**(NumberFormatException | NullPointerException | ArrayIndexOutOfBoundsException ex)              {                  System.out.println("Now, this block handles NumberFormatException, NullPointerException and ArrayIndexOutOfBoundsException");              }          }      }  } |

* java.lang.Exception is super class of all types of exception. (Types of exceptions will be discussed later). It handles all types of exceptions. In the above example, all catch blocks can be replaced by one catch block which handles all types of exceptions. This type of exception handling comes very handy when you are not sure about the types of exceptions your code may throw.

**try**

            {

**int** a = s[i].length() + Integer.parseInt(s[i]);

                //This statement may throw NumberFormatException, NullPointerException and ArrayIndexOutOfBoundsException

            }

**catch**(Exception ex)

            {

                System.out.println("This block handles all types of exceptions");

            }

* The order of catch blocks should be from most specific to most general ones. i.e Sub classes of Exception must come first and super classes later. If you keep the super classes first and sub classes later, you will get compile time error : ****Unreachable Catch Block****.

# [Nested try catch Blocks In Java](https://javaconceptoftheday.com/nested-try-catch-blocks-java/)

try-catch blocks can be nested. i.e one try block can contain another try-catch block. The syntax for nesting try blocks is,

**try** {    //Outer try block

    //Some Statement

**try** {   //Inner try block

        //Some Statements

    }**catch** (Exception ex)    //Inner catch block

    {

    }

}

**catch**(Exception ex)     //Outer catch block

{

}

* If the exception thrown by the inner try block can not be caught by it’s catch block, then this exception is propagated to outer try blocks. Any one of the outer catch block should handle this exception otherwise program will terminate abruptly.

public class ExceptionHandling

{

public static void main(String[] args)

{

String[] s = {"abc", "123", null, "xyz"}; //String array containing one null object

for (int i = 0; i < s.length; i++)

{

//First Level try-catch block

try

{

int a = s[i].length(); //This statement may throw NullPointerException

//second level try-catch block

try

{

System.out.println(s[i+1]); //This statement may throw ArrayIndexOutOfBoundsException

//third level try-catch block

try

{

a = Integer.parseInt(s[i]); //This statement may throw NumberFormatException

}

catch (NullPointerException e)

{

System.out.println("NumberFormatException will not be caught here");

}

}

catch (NumberFormatException ex)

{

System.out.println("NumberFormatException will be caught here");

}

}

catch(Exception ex)

{

System.out.println("This block catches all types of exceptions");

}

}

}

}

**public** **class** ExceptionPropogation {

/\*

\* In the following example also, try-catch blocks are nested. main() method

\* calls nestedTry() method. nestedTry() method has one try-catch block. First

\* statement in try block throws NumberFormatException which is not handled by

\* it’s catch block. So, It propagates to try-catch block of main method which

\* handles this exception.

\*/

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

**try** {

*nestedTry*();

} **catch** (Exception ex) {

System.***out***.println("NumberFormatException will be caught here in main method");

}

}

**static** **void** nestedTry() {

**try** {

**int** i = Integer.*parseInt*("abc"); // This statement throws NumberFormatException

} **catch** (NullPointerException ex) {

System.***out***.println("NumberFormatException will not be caught here");

}

}

}

* finally block overrides any return values from try and catch blocks.

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

    {

        System.out.println(methodReturningValue());    //Output : 50

    }

**static** **int** methodReturningValue()

    {

**try**

        {

**return** 10;

        }

**catch** (Exception e)

        {

**return** 20;

        }

**finally**

        {

**return** 50;    //This method returns 50 not 10 or 20

        }

    }

# [Hierarchy Of Exceptions In Java](https://javaconceptoftheday.com/hierarchy-exceptions-java/)

To understand how exception handling works in Java, you need to understand the categories of exceptions:

## Checked Exceptions

Checked exceptions are known to compiler i.e they are the exceptions that are checked at compile time. Checked exceptions are also called compile time exceptions, because they can be known during compile time.

## Unchecked Exceptions

Unchecked exceptions are not known to compiler.  They are the exceptions that are not checked at compile time, because they occur only at run time.That’s why these exceptions are also called run time exceptions.

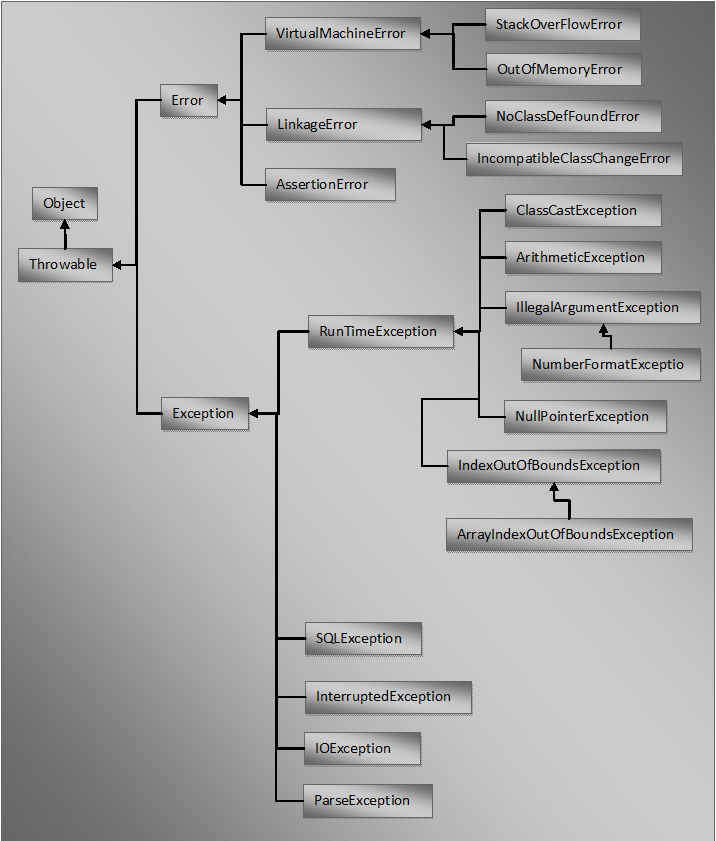
****Errors****:  
These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored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

## java.lang.Throwable :

java.lang.Throwable is the super class of all errors and exceptions in java. Throwable class extends java.lang.Object class. The only argument of catch block must be it’s type or it’s sub class type. You can check the documentation of Throwable class [here](http://docs.oracle.com/javase/7/docs/api/java/lang/Throwable.html" \t "https://javaconceptoftheday.com/hierarchy-exceptions-java/_blank). It has two sub classes.

1)java.lang.Error

2)java.lang.Exception



## Throwing An Exception :

We all know that Throwable class is super class for all types of errors and exceptions. An object to this Throwable class or it’s sub classes can be created in two ways.

* First one is using an argument of catch block. In this way, Throwable object or object to it’s sub classes is implicitly created and thrown by java run time system.
* Second one is using new operator. In this way, Throwable object or object to it’s sub classes is explicitly created and thrown by the code.

****throw InstanceOfThrowableType;****

where, InstanceOfThrowableType must be an object of type Throwable or subclass of Throwable.

Such explicitly thrown exception must be handled some where in the program, otherwise program will be terminated.

It is not compulsory that explicitly thrown exception must be handled by immediately following try-catch block. It can be handled by any one of it’s enclosing try-catch blocks

**public** **class** ExceptionHandling

{

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

    {

**try**

        {

            methodWithThrow();

        }

**catch**(NumberFormatException ex)

        {

            System.out.println("NumberFormatException object thrown in methodWithThrow() method will be handled here");

        }

    }

**static** **void** methodWithThrow()

    {

**try**

        {

            NumberFormatException ex = **new** NumberFormatException("This is an object of NumberFormatException");

**throw** ex;        //throwing NumberFormatException object explicitly using throw keyword

        }

**catch**(ArithmeticException ex)

        {

            System.out.println("Explicitly thrown NumberFormatException object will not be caught here");

        }

    }

}

## Re-throwing An Exception :

We all know that exceptions occurred in the try block are caught in catch block. Thus caught exceptions can be re-thrown using ****throw**** keyword. Re-thrown exception must be handled some where in the program, otherwise program will terminate abruptly.

**static** **void** methodWithThrow()

    {

**try**

        {

            String s = **null**;

            System.out.println(s.length());   //This statement throws NullPointerException

        }

**catch**(NullPointerException ex)

        {

            System.out.println("NullPointerException is caught here");

**throw** ex;     //Re-throwing NullPointerException

        }

    }

# [throws Keyword In Java](https://javaconceptoftheday.com/throws-keyword-java/)

If a method is capable of throwing an exception that it could not handle, then it should specify that exception using throws keyword. It helps the callers of that method in handling that exception. The syntax for using throws keyword is,

|  |  |
| --- | --- |
| 1  2  3  4 | return\_type method\_name(parameter\_list) **throws** exception\_list  {       //some statements  } |

**static** **void** methodWithThrows() **throws** NumberFormatException, NullPointerException

{

**int** i = Integer.parseInt(“abc”);

        String s = **null**;

        System.out.println(s.length());   //This statement throws NullPointerException

    }

* The main use of throws keyword in java is that an exception can be propagated through method calls.
* Even constructor can use throws keyword.For this, object creation statement must be enclosed in try-catch blocks.

When a method is throwing unchecked type of exceptions, then you need not to mention it using throws keyword. But for a method throwing checked type of exceptions, you must declare it with throws keyword or enclose the statement which is throwing an exception in try-catch block.

|  |
| --- |
| **public** **class** ExceptionHandling  {      //method throwing Unchecked Exception declared without throws clause    **static** **void** methodThrowingUncheckedException()      {  **int** i = Integer.parseInt(“abc”);            //Above statement throws NumberFormatException which is unchecked type of exception      }        //method throwing checked Exception declared with throws clause    **static** **void** methodThrowingCheckedException() **throws** ClassNotFoundException      {          Class.forName(“AnyClassName”);            //Above statement throws ClassNotFoundException which is checked type of exception      }    **public** **static** **void** main(String[] args)      {  **try**          {              methodThrowingUncheckedException();          }  **catch**(NumberFormatException ex)          {              System.out.println("NumberFormatException will be caught here");          }    **try**          {              methodThrowingCheckedException();          }  **catch** (ClassNotFoundException e)          {              System.out.println("ClassNotFoundException will be caught here");          }      }  } |

# [Method Overriding With throws Clause](https://javaconceptoftheday.com/method-overriding-with-throws-clause/)

* If super class method is not throwing any exceptions, then it can be overrided with any unchecked type of exceptions, but can not be overrided with checked type of exceptions.
* If a super class method is throwing unchecked exception, then it can be overrided in the sub class with same exception or any other unchecked exceptions but can not be overrided with checked exceptions.
* If super class method is throwing checked type of exception, then it can be overrided with same exception or with it’s sub class exceptions i.e you can decrease the scope of the exception, but can not be overrided with it’s super class exceptions i.e you can not increase the scope of the exception.

# [Checked And Unchecked Exceptions In Java](https://javaconceptoftheday.com/checked-unchecked-exceptions-java/)

## 1) Checked Exceptions :

Checked exceptions are the exceptions which are known during compile time. These are the exceptions that are checked at compile time. They are also called compile time exceptions.

These exceptions must be handled either using try-catch blocks or using throws clause. If not handled properly, they will give compile time error.

All sub classes of java.lang.Exception except sub classes of RunTimeException are checked exceptions.

**public** **class** CheckedException

{

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

    {

        Class.forName("AnyClassName");

        //Compile time error because

        //above statement throws ClassNotFoundException which is a checked exception

        //this statement must be enclosed within try-catch block or declare main method with throws clause

    }

}

## 2) Unchecked Exceptions :

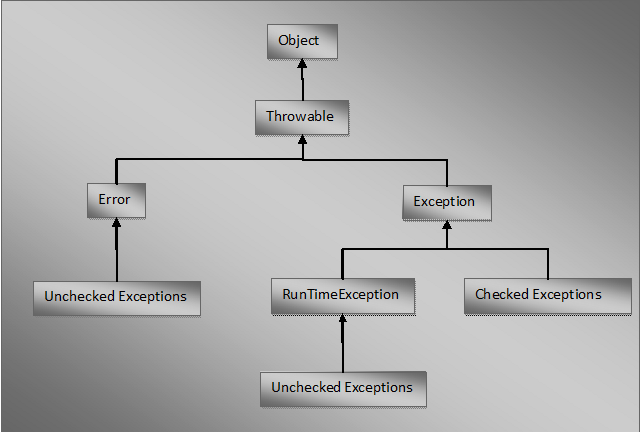
Unchecked exceptions are the exceptions which are known  at run time. They can not be known at compile time because they occur only at run time. That’s why they are also called ****Run Time Exceptions****.

All the sub classes of RunTimeException and all sub classes of Error class are unchecked exceptions.

If any statement in the program throws unchecked exceptions and you are not handling them either using try-catch blocks or throws clause, then it does not give compile time error. Compilation will be successful but program may fail at run time. Therefore, to avoid premature termination of the program, you have to handle them properly.

## Difference Between Checked And Unchecked Exceptions :

|  |  |
| --- | --- |
| ****Checked Exceptions**** | ****Unchecked Exceptions**** |
| They are known at compile time. | They are known at run time. |
| They are checked at compile time. | They are not checked at compile time. Because they occur only at run time. |
| These are compile time exceptions. | These are run time exceptions. |
| If  these exceptions are not handled properly in the application, they give compile time error. | If these exceptions are not handled properly, they don’t give compile time error. But application will be terminated prematurely at run time. |
| All sub classes of java.lang.Exception Class except sub classes of RunTimeException are checked exceptions. | All sub classes of RunTimeException and sub classes of java.lang.Error are unchecked exceptions. |



# [User Defined Exceptions In Java](https://javaconceptoftheday.com/user-defined-exceptions-java/)

In java, we can define our own exception classes as per our requirements. These exceptions are called ****user defined exceptions in java OR Customized exceptions****. User defined exceptions must extend any one of the classes in the hierarchy of exceptions.

**class** AgeIsNegativeException **extends** Exception

{

    String errorMessage;

**public** AgeIsNegativeException(String errorMessage)

    {

**this**.errorMessage = errorMessage;

    }

    //Modifying toString() method to display customized error message

    @Override

**public** String toString()

    {

**return** errorMessage;

    }

}

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

    {

          Scanner sc = **new** Scanner(System.in);  //Declaring Scanner variable to take input from user

          System.out.println("Enter Your Age");

**int** age = sc.nextInt();         //Taking input from user

**try**

          {

**if**(age < 0)

              {

**throw** **new** AgeIsNegativeException("Age can not be negative");    //throws AgeIsNegativeException if age is negative

              }

          }

**catch**(AgeIsNegativeException ex)

          {

              System.out.println(ex);    //Output : Age can not be negative

          }

    }

# [Chained Exceptions In Java](https://javaconceptoftheday.com/chained-exceptions-java/)

In an application, one exception throws many exceptions. i.e one exception causes another exception and that exception causes another exception thus forming chain of exceptions. It is better to know where the actual cause of the exception lies. This is possible with chained exceptions feature of the Java.

Chained exceptions are introduced from JDK 1.4. To implement chained exceptions in java, two new constructors and two new methods are added in the Throwable class. They are,

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.

**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"));

**throw** ex;

        }

**catch**(NumberFormatException ex)

        {

            System.out.println(ex);     //displaying the exception

            System.out.println(ex.getCause());    //getting the actual cause of the exception

        }

    }

}

## What is the difference between throw, throws and Throwable in java?

## throw In Java :

## **throw** is a keyword in java which is used to throw an exception manually. Using throw keyword, you can throw an exception from any method or block. But, that exception must be of type **java.lang.Throwable** class or it’s sub classes. Below example shows how to throw an exception using throw keyword.

**class** ThrowAndThrowsExample

{

**void** method() **throws** Exception{

        Exception e = **new** Exception();

**throw** e;            //throwing an exception using 'throw'

    }

## }

## throws In Java :

****throws**** is also a keyword in java which is used in the method signature to indicate that this method may throw mentioned exceptions. The caller to such methods must handle the mentioned exceptions either using try-catch blocks or using throws keyword. Below is the syntax for using throws keyword.

|  |  |
| --- | --- |
| 1  2  3  4 | return\_type method\_name(parameter\_list) **throws** exception\_list  {       //some statements  } |

Below is the example which shows how to use throws keyword.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | **class** ThrowsExample  {  **void** methodOne() **throws** SQLException      {          //This method may throw SQLException      }    **void** methodTwo() **throws** IOException      {          //This method may throw IOException      }    **void** methodThree() **throws** ClassNotFoundException      {          //This method may throw ClassNotFoundException      }  } |

## Throwable In Java :

****Throwable**** is a super class for all types of errors and exceptions in java. This class is a member of ****java.lang**** package. Only instances of this class or it’s sub classes are thrown by the java virtual machine or by the throw statement. The only argument of catch block must be of this type or it’s sub classes. If you want to create your own customized exceptions, then your class must extend this class. Click [here](https://javaconceptoftheday.com/hierarchy-exceptions-java/" \o "Hierarchy Of Exceptions In Java) to see the hierarchy of exception classes in java.

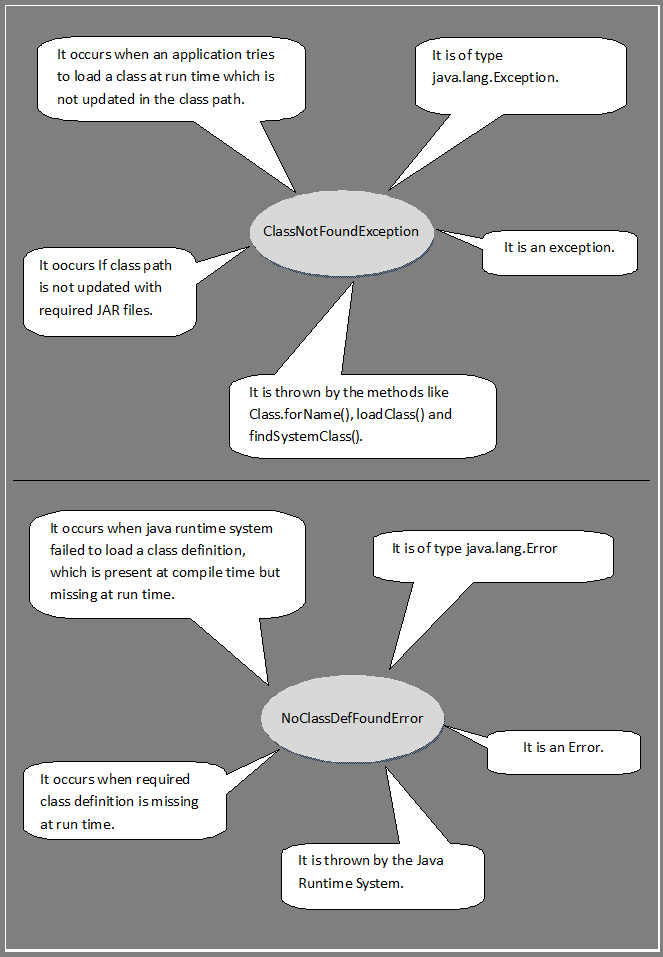
Below example shows how to create customized exceptions by extending ****java.lang.Throwable**** class.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | **class** MyException **extends** Throwable  {             //Customized Exception class  }    **class** ThrowAndThrowsExample  {  **void** method() **throws** MyException      {          MyException e = **new** MyException();    **throw** e;      }  } |

# [Difference Between Error Vs Exception In Java](https://javaconceptoftheday.com/difference-between-error-vs-exception-in-java/)

|  |  |
| --- | --- |
| Errors | Exceptions |
| Errors in java are of type java.lang.Error. | Exceptions in java are of type java.lang.Exception. |
| All errors in java are unchecked type. | Exceptions include both checked as well as unchecked type. |
| Errors happen at run time. They will not be known to compiler. | Checked exceptions are known to compiler where as unchecked exceptions are not known to compiler because they occur at run time. |
| It is impossible to recover from errors. | You can recover from exceptions by handling them through try-catch blocks. |
| Errors are mostly caused by the environment in which application is running. | Exceptions are mainly caused by the application itself. |
| Examples : java.lang.StackOverflowError, java.lang.OutOfMemoryError | Examples : Checked Exceptions : SQLException, IOException Unchecked Exceptions : ArrayIndexOutOfBoundException, ClassCastException, NullPointerException |

|  |  |
| --- | --- |
| ****ClassNotFoundException**** | ****NoClassDefFoundError**** |
| It is an exception. It is of type java.lang.Exception. | It is an error. It is of type java.lang.Error. |
| It occurs when an application tries to load a class at run time which is not updated in the classpath. | It occurs when java runtime system doesn’t find a class definition, which is present at compile time, but missing at run time. |
| It is thrown by the application itself. It is thrown by the methods like Class.forName(), loadClass() and findSystemClass(). | It is thrown by the Java Runtime System. |
| It occurs when classpath is not updated with required JAR files. | It occurs when required class definition is missing at run time. |



### Most Frequently Occurring Exceptions In Java With Examples :

1. ****java.lang.NullPointerException:****

*NullPointerException* is a *RunTimeException* which occurs when your application tries to access null object. It happens if you don’t initialize the reference variable and it is pointing to null instead of actual object. Using such reference variable will cause *NullpointerException*.

**class** A

{

**static** String s;

}

**public** **class** MainClass

{

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

    {

       System.out.println(A.s.length());

    }

}

1. ****java.lang.ArrayIndexOutOfBoundsException:****

*ArrayIndexOutOfBoundsException* is also one of the frequently occurring exception in java. It occurs when you try to access an array element with an invalid index i.e index greater than the array length or with a negative index.

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

    {

       String s[] = **new** String[] {"ZERO", "ONE", "TWO", "THREE", "FOUR"};

       System.out.println(s[5]);    //This will throw ArrayIndexOutOfBoundsException

       System.out.println(s[-1]);   //This will also throw ArrayIndexOutOfBoundsException

    }

1. ****java.lang.NumberFormatException:****

*NumberFormatException* is thrown when you are trying to convert a string to numeric value like integer, float, double etc…, but input string is not a valid number. *NumberFormatException* is also one of *IllegalArgumentException*.

**public** **class** MainClass

{

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

    {

**int** i = Integer.parseInt("abc");

    }

}

1. ****java.lang.ClassNotFoundException:****

*ClassNotFoundException* is a checked type of exception. It is thrown when an application tries to load a class at run time using *Class.forName()* or *loadClass()* or *findSystemClass()* methods, but the class with specified name is not found in the classpath. This frequently occurs when you try to run your application without updating the class path with required JAR files.

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

    {

**try**

       {

           Class.forName("oracle.jdbc.driver.OracleDriver");

       }

**catch** (ClassNotFoundException e)

       {

           e.printStackTrace();

       }

}

The above code will throw *java.lang.ClassNotFoundException* if you don’t update the classpath with Oracle JDBC driver class.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | java.lang.ClassNotFoundException: oracle.jdbc.driver.OracleDriver      at java.net.URLClassLoader.findClass(Unknown Source)      at java.lang.ClassLoader.loadClass(Unknown Source)      at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)      at java.lang.ClassLoader.loadClass(Unknown Source)      at java.lang.Class.forName0(Native Method)      at java.lang.Class.forName(Unknown Source)      at pack1.MainClass.main(MainClass.java:9) |

1. ****java.lang.ArithmeticException:****

*ArithmeticException* is also a *RunTimeException* which is thrown when an abnormal arithmetic condition arises in an application. For example, divide by zero exception

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

    {

**int** i = 10/0;

       System.out.println(i);

    }

1. ****java.sql.SQLException****

*SQLException* is thrown when an application encounters with an error while interacting with the database. For example, passing the wrong username or password, passing the wrong URL of the database, passing invalid column name or column index etc. *SQLException* is also a checked exception.

**public** **class** MainClass

{

**static**

    {

**try**

        {

            Class.forName("oracle.jdbc.driver.OracleDriver");

        }

**catch** (ClassNotFoundException e)

        {

            System.out.println("Unable To Load The Driver class");

        }

    }

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

    {

        Connection con = **null**;

        String URL = "jdbc:oracle:thin:@localhost:1521:XE";

        String username = "username";

        String password = "password";

**try**

        {

            con = DriverManager.getConnection(URL, username, password);

        }

**catch** (SQLException e)

        {

            e.printStackTrace();

        }

    }

}

java.sql.SQLException: ORA-01017: invalid username/password; logon denied

****7) java.lang.ClassCastException****

*ClassCastException* occurs when an object of one type can not be casted to another type. While casting, an object must satisfy “IS-A” relationship. If it doesn’t satisfy then JVM will throw *java.lang.ClassCastException*.

class A{

}

class B extends A{

}

**public** **class** MainClass{

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

       A a = **new** A();

       B b = (B)a;

    }

}

we are trying to cast an object of type *A* to type *B*. But, it doesn’t satisfy “IS-A” relationship. i.e *A* is not of type *B*. This will throw *java.lang.ClassCastException*.

****8) java.io.IOException:****

*OException* occurs when an IO operation fails in your application. IOException is a checked type of exception. This exception is the super class for all type of IO exceptions. Some of the popular IO exceptions are *FileNotFoundException*, *SocketException*, *SSLException* etc.

**import** java.io.BufferedReader;

**import** java.io.FileNotFoundException;

**import** java.io.FileReader;

**import** java.io.IOException;

**public** **class** MainClass

{

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

    {

       String filePath = "C:\Users\Bablad\Desktop\Open.txt";

       BufferedReader reader = **null**;

**try**

       {

           reader = **new** BufferedReader(**new** FileReader(filePath));

       }

**catch** (FileNotFoundException e)

       {

           e.printStackTrace();

       }

**finally**

       {

**try**

           {

**if**(reader != **null**)

               {

                   reader.close();

               }

           }

**catch** (IOException e)

           {

               e.printStackTrace();

           }

       }

    }

}

we are trying to read a file which doesn’t exist in the path specified. It causes *java.io.FileNotFoundException*.

java.io.FileNotFoundException: C:\Users\Bablad\Desktop\Open.txt (The system cannot find the file specified)

    at java.io.FileInputStream.open0(Native Method)

    at java.io.FileInputStream.open(Unknown Source)

    at java.io.FileInputStream.<init>(Unknown Source)

    at java.io.FileInputStream.<init>(Unknown Source)

    at java.io.FileReader.<init>(Unknown Source)

    at pack1.MainClass.main(MainClass.java:18)

1. ****java.lang.InterruptedException:****

You may have come across this exception if you have worked on multithreaded programming. *InterruptedException* is thrown when a sleeping thread or waiting thread is interrupted. The methods like *sleep()*, *wait()* and *join()* methods throw *InterruptedException*. *InterruptedException* is also a checked exception.

**public** **class** MainClass

{

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

    {

        Thread t = **new** Thread()

        {

**public** **void** run()

            {

**try**

                {

                    Thread.sleep(10000);   //Thread sleeps for 10s

                }

**catch** (InterruptedException e)

                {

                    e.printStackTrace();

                }

            }

        };

        t.start();

**try**

        {

            Thread.sleep(3000);   //main thread sleeping for 3s

        }

**catch** (InterruptedException e)

        {

            e.printStackTrace();

        }

        t.interrupt();    //interrupting thread t

    }

}

main thread interrupts thread *****‘t’***** while it is sleeping. This causes *java.lang.InterruptedException* to arise.

java.lang.InterruptedException: sleep interrupted

1. ****java.lang.SecurityException:****

*SecurityException* is thrown by the security manager if an application violates the security rules. For example, changing a thread name or thread priority to which you don’t have access or using a package name which is already used. *SecurityException* indicates that application has violated the security rules and it can not continue the execution.

**package** java.lang;

**public** **class** MainClass {

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

        System.out.println("Java Concept Of The Day");

    }

}

The above code will throw *java.lang.SecurityException*. Because, we are using package name *“java.lang”* which is already used in JDK.

Exception in thread "main" java.lang.SecurityException: Prohibited **package** name: java.lang

1. ****java.lang.StackOverflowError:****

*StackOverflowError* is a run time error which occurs when stack overflows. This happens when you keep calling the methods recursively.

**public** **class** MainClass

{

**static** **void** methodOne()

    {

        methodTwo();

    }

**static** **void** methodTwo()

    {

        methodOne();

    }

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

    {

        methodOne();

    }

}

The above code will throw *java.lang.StackOverflowError*, because *methodOne()* and *methodTwo()* are calling each other recursively.

Exception in thread "main" java.lang.StackOverflowError

1. ****java.lang.NoClassDefFoundError:****

*NoClassDefFoundError* is thrown when Java Runtime System tries to load the definition of a class which is no longer available. The required class definition was present at compile time but it was missing at run time.

**class** A{

}

**public** **class** B{

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

        A a = **new** A();

    }

}

When you compile the above program, two *.class* files will be generated. One is *A.class* and another one is *B.class*. If you run the *B.class* file after deleting the *A.class* file, java.lang.NoClassDefFoundError will be thrown.

|  |
| --- |
| Exception in thread "main" java.lang.NoClassDefFoundError: A          at MainClass.main(MainClass.java:10)  Caused by: java.lang.ClassNotFoundException: A          at java.net.URLClassLoader.findClass(URLClassLoader.java:381)          at java.lang.ClassLoader.loadClass(ClassLoader.java:424)          at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:331)          at java.lang.ClassLoader.loadClass(ClassLoader.java:357) |