**Exception Handling**

The **Exception Handling in Java** is one of the powerful mechanism to handle the runtime errors so that normal flow of the application can be maintained.

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

Exception is an abnormal condition.

In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

If an exception occurs, which has not been handled by programmer then program execution gets terminated and a system generated error message is shown to the user.

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application that is why we use exception handling.

Suppose there are 10 statements in your program and there occurs an exception at statement 5, the rest of the code will not be executed i.e. statement 6 to 10 will not be executed. If we perform exception handling, the rest of the statement will be executed. That is why we use exception handling in [Java](https://www.javatpoint.com/java-tutorial).

There can be several reasons that can cause a program to throw exception. For example: Opening a non-existing file in your program, Network connection problem, bad input data provided by user etc.

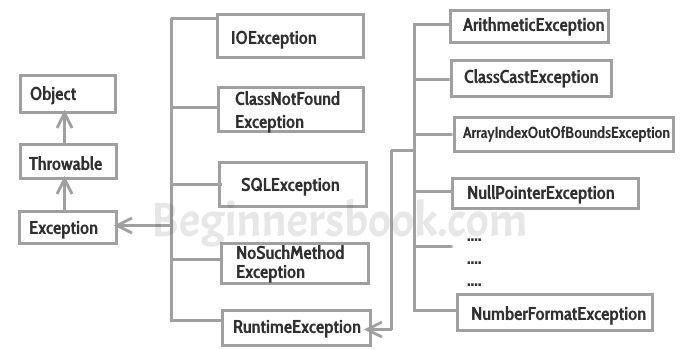
## **Difference between error and exception**

****Errors**** indicate that something severe enough has gone wrong, the application should crash rather than try to handle the error.An Error indicates serious problem that a reasonable application should not try to catch.

****Exceptions**** are events that occurs in the code. A programmer can handle such conditions and take necessary corrective actions.Exception indicates conditions that a reasonable application might try to catch. Few examples:  
**NullPointerException** – When you try to use a reference that points to null.  
**ArithmeticException** – When bad data is provided by user, for example, when you try to divide a number by zero this exception occurs because dividing a number by zero is undefined.  
**ArrayIndexOutOfBoundsException** – When you try to access the elements of an array out of its bounds, for example array size is 5 (which means it has five elements) and you are trying to access the 10th element.

## Hierarchy of Java Exception classes

The java.lang.Throwable class is the root class of Java Exception hierarchy which is inherited by two subclasses: Exception and Error. A hierarchy of Java Exception classes are given below:



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



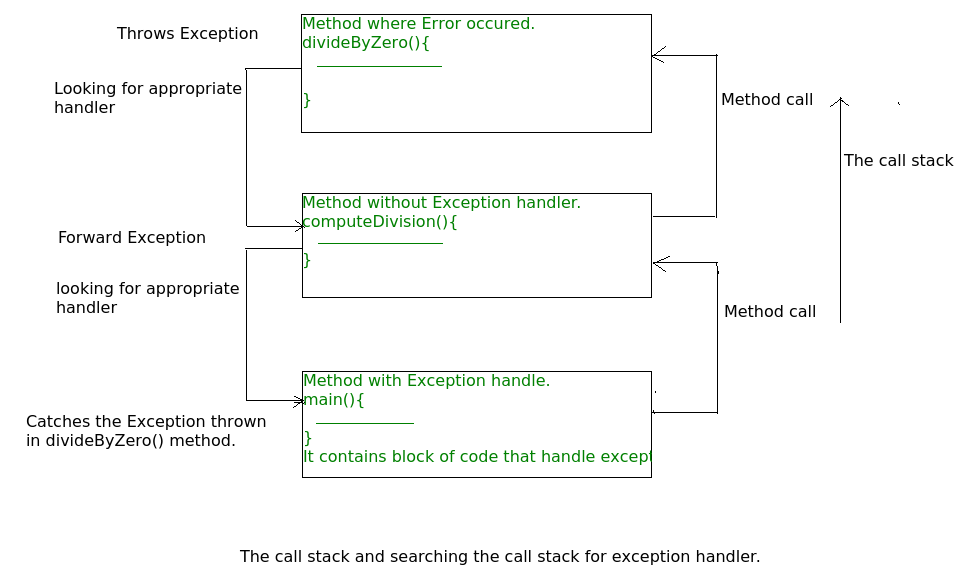
**How JVM handle an Exception?**

**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



****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.**

**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.**

****Explanation** : In the 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

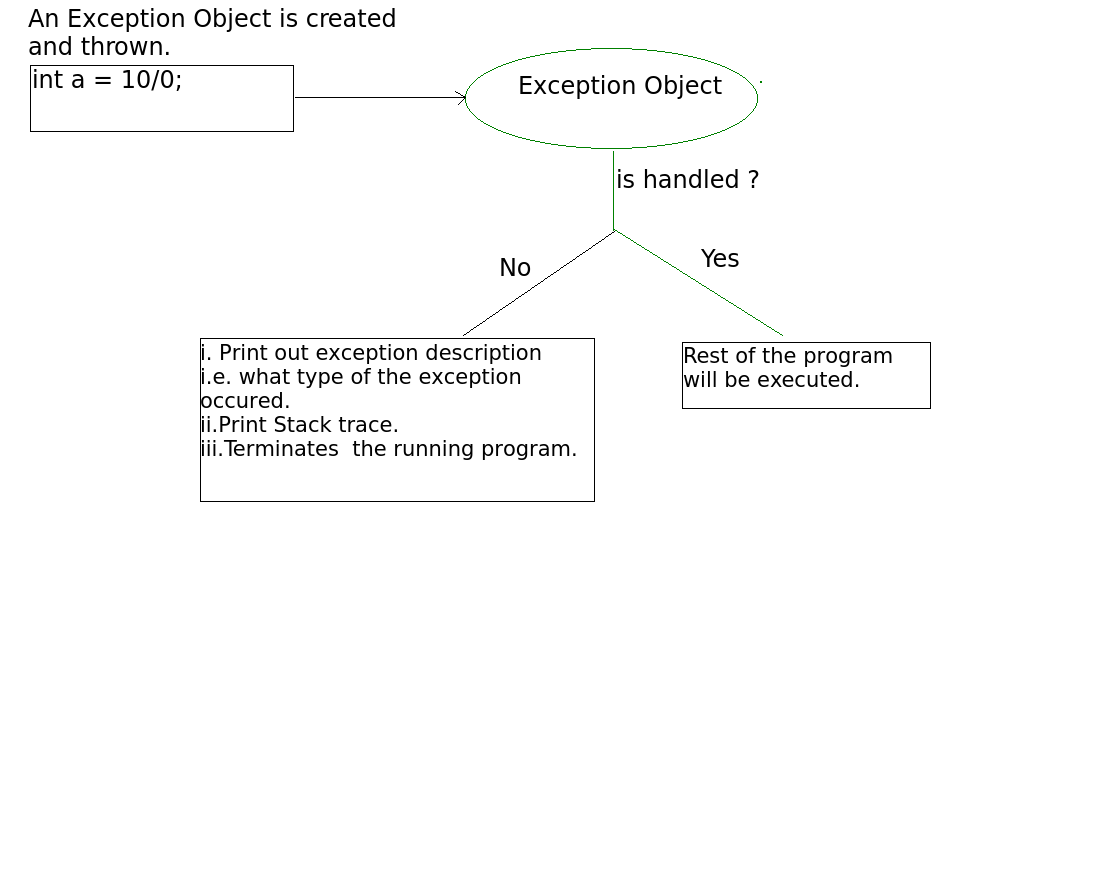
**finally** {

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

}

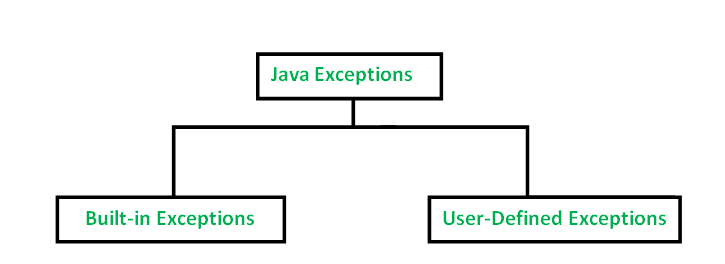
****Points to remember :****

* **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.**



****Types of Exception in Java****

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

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

****User-Defined Exceptions****

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

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

****class MyException extends Exception****

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

****MyException(){}****

* **We can also create a parameterized constructor with a string as a parameter.  
  We can use this to store exception details. We can call super class(Exception) constructor from this and send the string there.**

****MyException(String str)****

****{****

****super(str);****

****}****

* **To raise exception of user-defined type, we need to create an object to his exception class and throw it using throw clause, as:**

****MyException me = new MyException(“Exception details”);****

**throw me;**

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

// 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();

        }

    }

}

****Checked vs Unchecked Exceptions :****

**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**

**To fix the above program, we either need to specify list of exceptions using throws, or we need to use try-catch block.**

**Since FileNotFoundException is a subclass of IOException, we can just specify IOException in the throws list and make the above program compiler-error-free.**

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

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

### ****1) Checked Exception****

**The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.**

### ****2) Unchecked Exception****

**The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.**

### ****3) Error****

**Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.**

****Should we make our exceptions checked or unchecked?****

**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.**

## ****Java Exception Keywords****

There are 5 keywords which are used in handling exceptions in Java.

try : The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.

catch : The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.

finally : The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.

throw : The "throw" keyword is used to throw an exception.

throws : The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

## **Java try block**

Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.

If an exception occurs at the particular statement of try block, the rest of the block code will not execute. So, it is recommended not to keeping the code in try block that will not throw an exception.

Java try block must be followed by either catch or finally block.

1. **try**{
2. //code that may throw an exception
3. }**catch**(Exception\_class\_Name ref){}

## **Java catch block**

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception ( i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

The catch block must be used after the try block only. You can use multiple catch block with a single try block.

## **Internal working of java try-catch block**

The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

* Prints out exception description.
* Prints the stack trace (Hierarchy of methods where the exception occurred).
* Causes the program to terminate.

But if exception is handled by the application programmer, normal flow of the application is maintained i.e. rest of the code is executed.

## **Java Multi-catch block**

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

## **Points to remember**

* At a time only one exception occurs and at a time only one catch block is executed.
* All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.

try{

int a[]=new int[5];

a[5]=30/0;

}

catch(ArithmeticException e)

{

System.out.println("Arithmetic Exception occurs");

}

catch(ArrayIndexOutOfBoundsException e)

{

System.out.println("ArrayIndexOutOfBounds Exception occurs");

}

catch(Exception e)

{

System.out.println("Parent Exception occurs");

}

# **Java Nested try block**

The try block within a try block is known as nested try block in java.

# **Why use nested try block**

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

# **Java finally block**

**Java finally block** is a block that is used to execute important code such as closing connection, stream etc.

Java finally block is always executed whether exception is handled or not.

Java finally block follows try or catch block.



If you don't handle exception, before terminating the program, JVM executes finally block(if any).

# **Why use java finally**

* Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

For each try block there can be zero or more catch blocks, but only one finally block.

Note: The finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).

**Java throw keyword**

The Java throw keyword is used to explicitly throw an exception.

We can throw either checked or uncheked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception. We will see custom exceptions later.

The syntax of java throw keyword is given below.

**throw exception;**

**throw new IOException("sorry device error);**

In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

1. **public** **class** TestThrow1{
2. **static** **void** validate(**int** age){
3. **if**(age<18)
4. **throw** **new** ArithmeticException("not valid");
5. **else**
6. System.out.println("welcome to vote");
7. }
8. **public** **static** **void** main(String args[]){
9. validate(13);
10. System.out.println("rest of the code...");
11. }
12. }

**Java Exception propagation**

An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method,If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack.This is called exception propagation.

**Rule: By default Unchecked Exceptions are forwarded in calling chain (propagated).**

lass TestExceptionPropagation1{

void m(){

int data=50/0;

}

void n(){

m();

}

void p(){

try{

n();

}catch(Exception e){System.out.println("exception handled");}

}

public static void main(String args[]){

TestExceptionPropagation1 obj=new TestExceptionPropagation1();

obj.p();

System.out.println("normal flow...");

}

}

In the above example exception occurs in m() method where it is not handled,so it is propagated to previous n() method where it is not handled, again it is propagated to p() method where exception is handled.

Exception can be handled in any method in call stack either in main() method,p() method,n() method or m() method.

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

**Java throws keyword**

The Java **throws** keyword is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

**Syntax of java throws**

**return\_type method\_name() throws exception\_class\_name{**

**//method code }**

**Which exception should be declared**

Ans) checked exception only, because:unchecked Exception: under your control so correct your code.

error: beyond your control e.g. you are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

**throw and throws in Java:**

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. 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. 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.

**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:

**By using try catch**

**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.

Example :

public static void main(String[] args)throws InterruptedException

{

Thread.sleep(10000);

System.out.println("Hello Geeks");

}

Explanation : In the above program, by using throws keyword we handled the InterruptedException and we will get the output.

**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.

**Difference between throw and throws in Java :**

**throw:**

> Java throw keyword is used to explicitly throw an exception.

> Checked exception cannot be propagated using throw only.

> Throw is followed by an instance.

> Throw is used within the method.

> You cannot throw multiple exceptions.

**throws:**

> Java throws keyword is used to declare an exception.

> Checked exception can be propagated with throws.

> Throws is followed by class.

> Throws is used with the method signature.

> You can declare multiple exceptions e.g.

public void method()throws IOException,SQLException.

**Difference between final, finally and finalize:**

**final** :

Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed.

Final is a keyword.

**finally** :

Finally is used to place important code, it will be executed whether exception is handled or not.

Finally is a block.

**finalize**:

Finalize is used to perform clean up processing just before object is garbage collected.

Finalize is a method.

**ExceptionHandling with MethodOverriding in Java**

There are many rules if we talk about methodoverriding with exception handling. The Rules are as follows:

> **If the superclass method does not declare an exception**

If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception.

> **If the superclass method declares an exception**

If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

1. Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception.
2. Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but can declare unchecked exception.
3. Rule: If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.
4. Example in case subclass overridden method declares same exception
5. Example in case subclass overridden method declares subclass exception
6. Example in case subclass overridden method declares no exception

**Java Custom Exception**

If you are creating your own Exception that is known as custom exception or user-defined exception. Java custom exceptions are used to customize the exception according to user need.

By the help of custom exception, you can have your own exception and message.

class InvalidAgeException extends Exception{

InvalidAgeException(String s){

super(s); }}