* : Exception: -

An unexpected, unwanted event that disturbs normal flow of the program is called exception.

Ex: SleepingException

FileNotFoundException

It is highly recommended to handle exception and the main objective of exception handling is graceful termination of program.

Exception handling doesn’t mean repairing an exception we have to provide alternative way to continue rest of the program normally.

Ex:

try{

read data from remote file

Location as London

}

catch(FileNotFoundException e){

use local file and continue rest of the program normally

}

**Runtime stack mechanism:**

For every thread JVM will create a run time stack each and every method call performed by that thread will be stored in the corresponding stack. Each entry in the stack is called activation record or stack frame. After completing every method call the corresponding entry from the stack will be removed. After completing all method calls the stack will become empty and that empty stack will be destroyed by JVM just before the termination of main thread.

Ex:

class Test{

p s v m(){

doStuff();

}

p s v doStuff(){

doMoreStuff();

}

p s v doMoreStuff(){

sop(“Hello”);

}

}

**Default exception handling in java:**

* Inside a method if any exception occurred the in which it is raised is responsible to create exception object by including the following information.

1. Name of Exception
2. Description of Exception
3. Location at which exception occurs

* After creation of object method hand overs that object to the JVM.
* JVM will check whether the method having any exception handling code or not. If the method doesn’t contain exception handling then JVM terminates that method and remover corresponding method from the stack.
* Then JVM identifies caller method checks caller method contain any handling code or not. If the caller method doesn’t contain handling code then JVM terminates the caller method also abnormally and removes corresponding entry from the stack
* This process will continue until the main method and if the main method also doesn’t contain handling code then JVM terminates main method abnormally and removes corresponding entry from the stack.
* Then JVM hand over responsibility of exception handling to default exception handler, which is the part of JVM.
* Default exception handler prints exception information in the following format and terminates program abnormally.

Exception in the “main” name of Error : Description stack trace

Ex:

class Test{

p s v m(){

doStuff();

}

p s v doStuff(){

doMoreStuff();

}

p s v doMoreStuff(){

sop(10/0);

}

}

Exception in thread “main”:java.lang.ArithmethicException : division by zero

at Test.doMoreStuff()

at Test.doStuff()

at Test.main()

Ex:

class Test{

p s v m(){

doStuff();

}

p s v doStuff(){

doMoreStuff();

Sop(10/0);

}

p s v doMoreStuff(){

sop(“Hello”);

}

}

O/P:-

Hello

Exception in thread “main”:java.lang.ArithmethicException : division by zero

at Test.doStuff()

at Test.main()

**Exception Hierarchy:**

Throwable class acts as root for java exception hierarchy.

Throwable class contains two child classes 1. Exception

2. Error

**Exception:**

Most of the times exception are caused by our program and these are recoverable.

Ex: If our programming requirement is to read data from remote file location at London at runtime if remote file is not available then we will get runtime exception saying File not found exception.

If file not found we can provide the local file and continue rest of the program normally.

**Error:**

Most of the time errors are not caused by our program and these are due to lack of resources. Errors are non-recoverable.

Ex: if OutOfMemoryError Occurs being a programmer we can’t do anything and the program will be terminated abnormally.

System admin or servers admin is responsible to provide heap memory.

There are several child classes of Exception are there.

Exception:

* RuntimeException

Child classes are

1. ArithmeticException
2. NullPointerException
3. ClassCastException
4. IndexOutOfBoundException
5. ArrayIndexOutOfBoundException
6. StringIndexOutOfBoundException
7. IllegalArgumentException
8. NumberFormatException

.

.

* IOException

1. EndOfFileException
2. FileNotFoundException
3. InterruptedIOException

* ServeletException
* RemoteException
* InturreptedException

Error:

* VirtualMachineError

1. StackOverflowError
2. OutOfMemoryError

* AssertionError
* ExceptionInInitializerError

**Checked and Unchecked Exception:**

The exception which are checked by the compiler for smoot execution of program is called checked exception.

Ex: Hall ticketMissingException, PenNotWorkingException, FileNotFoundException etc.

In our program if there is a chance rising checked exception then compulsory we should handle that checked exception (either by try catch or by throws keyword) otherwise we will get compile time error.

Ex:

import java.io.\*;

class Test{

p s v m(){

PrintWrite pw = new PrintWrite(“abc.txt”);

Pw.println(“Hello”);

}

}

The exception which are not checked by compiler whether programmer handling or not such type of exception are called unchecked exception.

Ex: ArithmeticException, BombBlastException etc

**Note:**

1. Whether it is checked or unchecked every exception occurs at runtime only. There is no chance of occurring any exception at compile time.
2. RuntimeException and its child classes, Error and its child classes are unchecked except this remaining are checked.

**Fully checked vs Partially checked:**

A checked exception is said to be fully checked if and only if all it’s child classes are checked.

Ex: IOException , InturreptedException

A checked exception is said to be partially checked if and only if some of its child classes are unchecked.

Ex: Exception, Throwable.

**Note:**

The only possible partially checked exception in java are Exception, Throwable.

Q. Describe the behaviour of following Exception.

IOException //Checked (Fully)

RuntimeException //unchecked

InterruptedException //Checked (Fully)

Error //Unchecked

Throwable //Checked (Partially)

ArithmeticException //Unchecked

NullPointerException // Unchecked

Exception //Checked (Partially)

FileNotFoundException //Checked (Fully)

**Customized Exception Handling by using try catch:**

The code which may rise an exception is called risky code. We have to write that code within try block and corresponding handling code we have to write within catch block.

Syntax:

try{

risky code;

}

catch( Exception e){

Exception handling code

}

Ex: without try catch

class Test{

p s v m (){

Sop(“statement 1”);

Sop(10/0);

Sop(“Statement 3”);

}

}

O/P: statement 1

AE:

Ex: with try catch

class Test{

p s v m (){

Sop(“statement 1”);

try{

Sop(10/0);

}

catch( Exception e){

Sop(10/2)

}

Sop(“Statement 3”);

}

}

O/P: normal termination will occur

**Control flow in try catch:**

try{

statement 1;

statement 2;

statement 3;

}

catch(Exception e){

statement 4;

}

Statement 5;

**Case 1:** if there is no exception statement 1, 2, 3, 5 (normal termination)

**Case 2:** If exception at 2, and corresponding catch block matched order is 1, 4, 5

**Case 3:** If exception at 2, and corresponding catch block not matched then order 1, abnormal termination.

**Case 4:** if exception rise at statement 4 or statement 5 then it is always abnormal termination.

**Note:**

1. Within try block if exception is raised then rest of the code will not be executed even though we handled that exception hence within the try block we have to take risky code only and length of the try block should be as less as possible.
2. In addition to try block there may be a chance of rising an exception inside catch and finally blocks.
3. If any statement which not part of try block and rises an exception then it is always abnormal termination.

**Methods to print Exception information:**

There are three methods are to print the information about the exception. These are present in Throwable class.

1. printStackTrace()

Name of Exception : description

Stack Trace

1. toString()

Name of Exception : Description

1. getMessage()

Description

Ex:

class Test{

p s v m (){

try{

Sop(10/0);

}

catch( Exception e){

e.printStackTrace();

Sop(e.toString());

Sop(e.getMessage());

}

}

}

O/P: java.lang.ArithmeticException : / by Zero

at Test.main()

java.lang. ArithmeticException : / by Zero

/ by Zero

Internally default exception handler will use printStackTrace to print information to the control.

**try with multiple catch block:**

The way of handling an exception is varied from exception to exception. Hence it is highly recommended to take separate catch block i.e. try with multiple catch block is always possible and recommended to use.

try {

Risky code;

}

catch(ArithmeticException e){

program for arithmetic exception

}

catch(SQLException e){

program for sql exception

}

catch(FileNotFoundException e){

program for filer not found exception

}

* If try with multiple catch blocks present then the order of catch block is very important. We have to take child first then parent otherwise we will get compile time error saying : Exception XXX has already been caught.

Ex:

try{

risky code;

}

catch(Exception e){

}

catch(ArithmeticException e){

}

O/P: CE : Exception java.lang.ArithmeticException has already been caught

Ex:

try{

risky code;

}

catch(ArithmeticException e){

}

catch(Exception e){

}

// Valid

We can’t declare two catch block for same exception otherwise we will get compile time error.

Ex:

try{

risky code;

}

catch(ArithmeticException e){

}

catch(ArithmeticException e){

}

O/P: CE : Exception java.lang.ArithmeticException has already been caught

**final:**

it is a modifier applicable for classes, methods and variable.

If a class declared as final then we can’t extend that class i.e. we can’t create child class i.e. inheritance is not possible for final classes.

If a method is final then we can’t override the method in child class.

If a variable declared as final then we can’t perform reassignment for the variable.

**finally:**

finally is a block always associated with try catch to maintain clean up code.

try{

risky code;

}

catch(Exception e){

Handling code;

}

finally{

clean up code;

}

The speciality of finally block is it will be executed always e-respective of whether exception is raised or not raised or exception is handled or not.

**finalize() :**

finalize() is a method always invoked by garbage collector just before destroying an object to perform clean-up activities. Once finalize method completes garbage collector destroys the object.

**Note:**

finally block is responsible to perform clean up activities related to try block i.e. whatever resources at the try block will be closed inside finally block.

Where as finalize() method is used to perform clean up activities related to object i.e. what ever resources associated with will be deallocated before destroying object.

**Various possible combination of try catch finally:**

* In try catch finally order is important.
* Whenever we are writing try compulsory we have to write catch or finally. Otherwise we will get compile time error i.e. try without catch or finally is invalid.
* Whenever we are writing catch compulsory try block must be required i.e. catch without try is invalid.
* Whenever we are writing finally compulsory try block must be required i.e. finally without try is invalid.
* Inside try catch and finally we can declare try catch and finally blocks i.e. nesting of try, catch finally is allowed.
* For try catch and finally curly braces are mandatory.

try{ try{ try{ }

} } catch(x e){ }

catch(Ex e){ catch(Ex e){ catch(x e){ }

} } //In-valid

// valid catch(Ex2 e){ } //CE:X has already caught

// valid

try{ } try{ } try{ }

catch( Ex e){ } finally{ } //CE: try without catch or finally

finally{ } // valid

//valid

catch(Ex e){ } finally{ }

//CE: catch without try //CE: finally without try

try{ } try{ } try{ }

finally{ } Sop(“Hello”); catch(Ex e){ }

catch(Ex e){ } catch(Ex e){ } Sop(“Hello”);

//CE: catch without try //CE1: try without catch catch(Ex2 e){ }

//CE2: catch without try //catch without try

try{ }

catch(Ex e){ }

Sop(“hello”);

finally{ }

//CE: finally without try

try{ try try{ } try{}

try{ } Sop(“”); catch() finally

catch(){ } catch(){ } Sop(“”); Sop(“”);

} //In valid //Invalid //Invalid

catch( ){ }

//valid

**throws:**

in our if there is a possibility of rising checked exception then compulsory we should handle the checked exception. Otherwise we will get compile time error saying unreported exception XXX ; must be caught or declared to be thrown

Ex:

Import java.io.\*;

class Test{

public static void main(String[ ] args){

PrintWriter pw = new PrintWriter(“abc.txt”);

pw.println(“Hello”);

}

}

CE: unreported exception java.io.FileNotFoundException; must be caught or declared to be thrown

Ex:

class Test{

public static void main(String[ ] args){

Thread.sleep(10000);

}

}

CE: unreported exception java.lang.Inturrepted.Exception; must be caught or declared to be thrown

We can handle the compile time error by using two ways

1. By using try catch

Ex:

class Test{

public static void main(String[ ] args){

try{

Thread.sleep(10000); }

catch(InturreptedException e){ }

}

}

1. By using throws Keyword:

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

Ex:

class Test{

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

{

Thread.sleep(10000);

}

}

throws keyword required only for checked exception. Usage of throws for un-checked exception there is no use.

throws keyword is required only to convince the compiler and it doesn’t prevent abnormal termination of the program.

class Test{

p s v m() throws InterruptedException

{

doStuff();

}

p s v doStuff() throws InterruptedException

{

doMoreStuff();

}

p s v doMoreStuff() throws InterruptedException

{

Thread.sleep(10000);

}

}

In the above program if we remove at least only one throws statement then the code won’t compile.

It is recommended to use try catch in place of throws.

**Case 1:**

We can use throws keyword for methods and constructors but not for classes.

Ex:

class Test throws Exception{ // invalid

}

**Case 2:**

We can use throws keyword only for throwable types. If we are trying to use for normal java classes then we will get compile time error saying incompatible type.

Ex:

class Test{

public void m1() throws Test{

}

}

// CE: incompatible type found : Test required : Java.lang.Throwable

Ex:

class Test extends RuntimeException{

public void m1() throws Test{

}

}

// Valid

**Case 3:**

Ex:

class Test{

public static void main(String[ ] args){

throw new Exception();

}

}

//CE: unreported exception java.lang.Exception; must be caught or declared to be thrown

Ex:

class Test{

public static void main(String[ ] args){

throw new Error();

}

}

//RE : Exception in thread “main” java.lang.Error at Test.main()

**Case 4:**

With in the try block if there is no chance of rising an exception then we can’t catch block for that exception otherwise we will get compile time error saying

Exception XXX is never thrown in body of corresponding try statement.

But this rule is applicable only for fully checked exception.

Ex:

class Test{  
 p s v m(){

try{

Sop(“Hello”);

}

catch(AE e){ }

}

}

O/P: Hello

Ex:

class Test{  
 p s v m(){

try{

Sop(“Hello”);

}

catch(Exception e){ }

}

}

O/P: Hello

Ex:

class Test{  
 p s v m(){

try{

Sop(“Hello”);

}

catch(IOE e){ }

}

}

// CE : Exception java.io.IOException is never thrown in body of corresponding try statement.

Ex:

class Test{  
 p s v m(){

try{

Sop(“Hello”);

}

catch(InturreptedE e){ }

}

}

// CE : Exception java.io.InterruptedException is never thrown in body of corresponding try statement.

Ex:

class Test{  
 p s v m(){

try{

Sop(“Hello”);

}

catch(Error e){ }

}

}

O/P: Hello

**Customized or User defined Exception:**

Sometimes to meet programming requirement we can define our own exception such type of exception are customized or user defined Exception.

Ex: TooYoungerException, TooOldException, InsufficientFundException etc

Ex:

class TooYoungerException extends RuntimeException {

TooYoungerException(String s){

super(s); // To make description available to default exception handeler

}

}

class TooOldException extends RuntimeException {

TooOldException(String s){

super(s);

}

}

class CustException {

public static void main(String[ ] args){

int age = Integer.parseInt(args[0]);

if(age>60){

throw new TooYoungerException(“Plz wait some more time … u will get best match soon”);

}

else if(age < 18){

throw new TooOldExceptiopn(“your age already crossed marriage age limit… No chance of getting marriage”);

}

else{

Sop(“You will get match details soon by email.”);

}

}

}

**Note :**

1. throw keyword is best suitable for customized or user defined exception, but not for predefined exception.
2. It is highly recommended to define customized exception as unchecked i.e. we have to extend RuntimeException not Exception.

**Top 10 Exception in java**

Based on the person who is rising exception all exception are divided into two categories.

1. JVM Exception
2. Programmatic Exception

**JVM Exception:**

The exception which are raised automatically by JVM whenever a particular event occurs are called JVM Exception.

Ex: ArithmeticException, NullPointerException etc

**Programmatic Exception:**

The exception which are raised explicitly either by programmer or by API developer to indicate that something goes wrong are called Programmatic Exception.

Ex: TooOldException, IllegalArgumentException etc.

1. **ArrayIndexOutOfBoundException :**

It the child class of Runtime\_Exception and hence it is unchecked. Raised automatically by JVM whenever we are trying to access array element with out of range index.

Ex:

int[ ] arr = new int[4];

Sop(arr[0]); // valid O/P: 0

Sop(arr[10]); Sop(arr[-5]); // RE: ArrayIndexOutOfBoundException

1. **NullPointerException :**

It is the child class of Runtime Exception and hence it is unchecked. Raised automatically by JVM whenever we are trying to perform any7 operation on null.

Ex:

String s = null;

Sop(s.length); // Re: NullPointerException

1. **ClassCastException :**

It is the child class of Runtime Exception and hence it is unchecked. Raised automatically by JVM whenever we are trying to typecast parent object to child type.

Ex:

String s = new String(“Durga”);

Object o = (Object)s; // Valid

Ex:

Object o = new Object();

String s = (String )o; //RE: ClassCastException

Ex:

Object o = new String(“Durga”);

String s = (String )o; //Valid

1. **StackOverflowError :**

It is the child class of Error and hence it is unchecked. Raised automatically by JVM when ever we are trying to perform recursive method call.

Ex:

Class Test{

public static void m1 () {

m2();

}

public static void m2 () {

m1();

}

public static void main (String[ ] args) {

m1();

}

Main Thread

|  |
| --- |
| . |
| . |
| m2() |
| m1() |
| m2() |
| m1() |
| main() |

It will be full at a time. // RE : StackOverflowError

1. **NoClassDefFoundError :**

It is the child class of Error and hence it is unchecked. Raised automatically by JVM whenever JVM unable to find required .class file.

Ex:

Java Test

If Test.class file is not available //RE:NoClassDefFoundError

1. **ExceptionInInitializerError :**

It is the child class of error and hence it is unchecked. Raised automatically by JVM if any exception occurs while executing static variable assignment and static block.

Ex:

class Test{

static int x = 10/0; // RE: ExceptionInInitializerError caused by //Java.lang.ArithmeticException / by Zero

}

Ex:

class Test{

static {

String s = null;

Sop(s.length());

}

}

// RE: ExceptionInInitia;izerError caused by : java.lang.NullPointerException

1. **IllegalArgumentException :**

It is the child class of Runtime Exception an hence it is unchecked. Raised explicitly either by programmer or by API developer to indicate that a method has been invoked with illegal argument.

Ex: The valid range of thread priory is 1 to 10. If we are trying to set the priority with any other value then we will get runtime exception saying IllegalArgumentException.

Thread t = new Thread();

t.setPriority(7); //valid

t.setPriority(13); //RE: IllegalArgumentException

1. **NumberFormatException :**

It is the direct child class of IllegalArgumentException which is the child class of Runtime Exception and hence it is unchecked. Raised explicitly either by programmer or by API developer to indicate that we are trying to convert string to number and the string is not properly formatted.

Ex:

int i = Integer.parseInt(“10”); // valid

int i = Integer.parseInt(“ten”); //RE : NumberFormatException

1. **IllegalStateException :**

It is the child class of Runtime Exception and hence it is unchecked. Raised explicitly either by programmer or by API developer to indicate that a method has been invoked at Run time.

Ex: After starting of a thread we are not allowed to restart the thread once again otherwise we will get Run time Exception saying IllegalStateException.

Thread t = new Thread();

t.start();

.

.

t.start(); // RE: IllegalThreadStateException

1. **AssortionError :**

It is the child class of Error and hence it is unchecked. Raised explicitly by programmer or by API developer to indicate that Assert statement fails.

Ex:

Assert(x>10);

If x is not greater than 10 then we will get Runtime Exception saying AssertionError