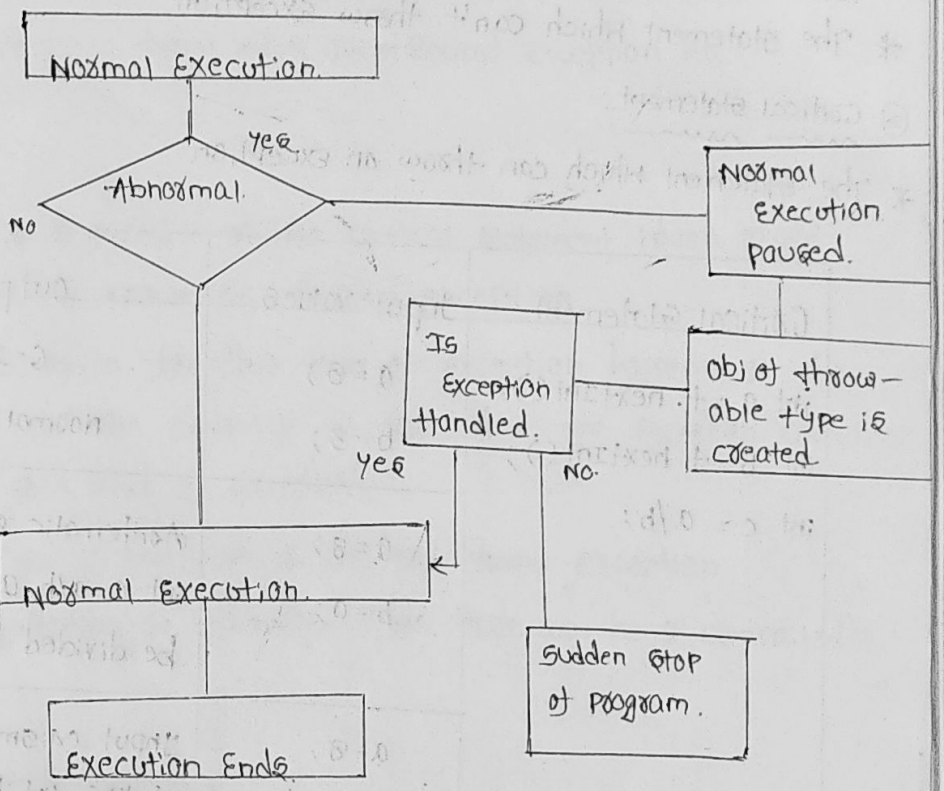
* EXEPTION :- [or] EXCEPTION :-

* Exception Any abnormal situation (or) unexpected situation occured during the run time is known as exception.
* Exceptions are classes in Java. And all the exception classes are child of throwable class.
* When we get an exception object of throwable type is created.
* Execution of simple program during exception :-



* During the normal execution of the program when an Abnormal situation occurs the normal flow of execution is paused & an object of throwable type is created, if the obj of throwable type is handled (the exception is handled). Then the normal flow of execution continues, however if the obj of throwable type. is not handled than the program stops at the same position. Where the object was created.

Ex:-
```
Package trail;
import java.util.Scanner;
Public class Trial Arith{
Psvm(S[]a){
Scanner t = new Scanner(S.in){
int a = t.nextInt();
int b = t.nextInt();
int c = a/b;
S.o.println(c);
}
}.
```

* Types of exception:

* on the basis of the warning there two types of exceptions

① checked exception

② unchecked exception

* unchecked exception :-

* the exception i.e, occured at the run time is known as U.E. since the exception occured at the run time compailer is unaware of exception hence compailer doesn't forces us to deal with the exception.

eg:- Null Pointer exception, Array index out of Bound exception etc.

* checked exception :-

* When the compailer is aware - of the critical statement which might throw an exception is known as checked exception.

* the checking of exception for this type of exception happens at the compile time. And as the compiler is aware of the exception compiler forces to handled this kind of exception.

eg:- file i/p stream as it will give us file not found exception.

Note :- If checked exception or unhandled then there will be a compailed time error.

eg:-
```
import java.io.FileInputStream;
public class Checked{.
psvm (S[]a){
FileInputStream file = new FileInputStream("fileLocation");
}
```

↑
By this error line compailer is forcing to handle the exception.

Hierarchy exception :-

* In the given program when the values for a = 10. and b = 5, we get output as 2, however when we try to divide a number by "0" the same program will give an exception.
* The exception in a program is caused due to statement.
* There are two types of statement.
  1. Normal statement.
  2. Critical statement.

① Normal statement :-

* The statement which can't throw exception.

② Critical statement :-

* The statement which can throw an exception.

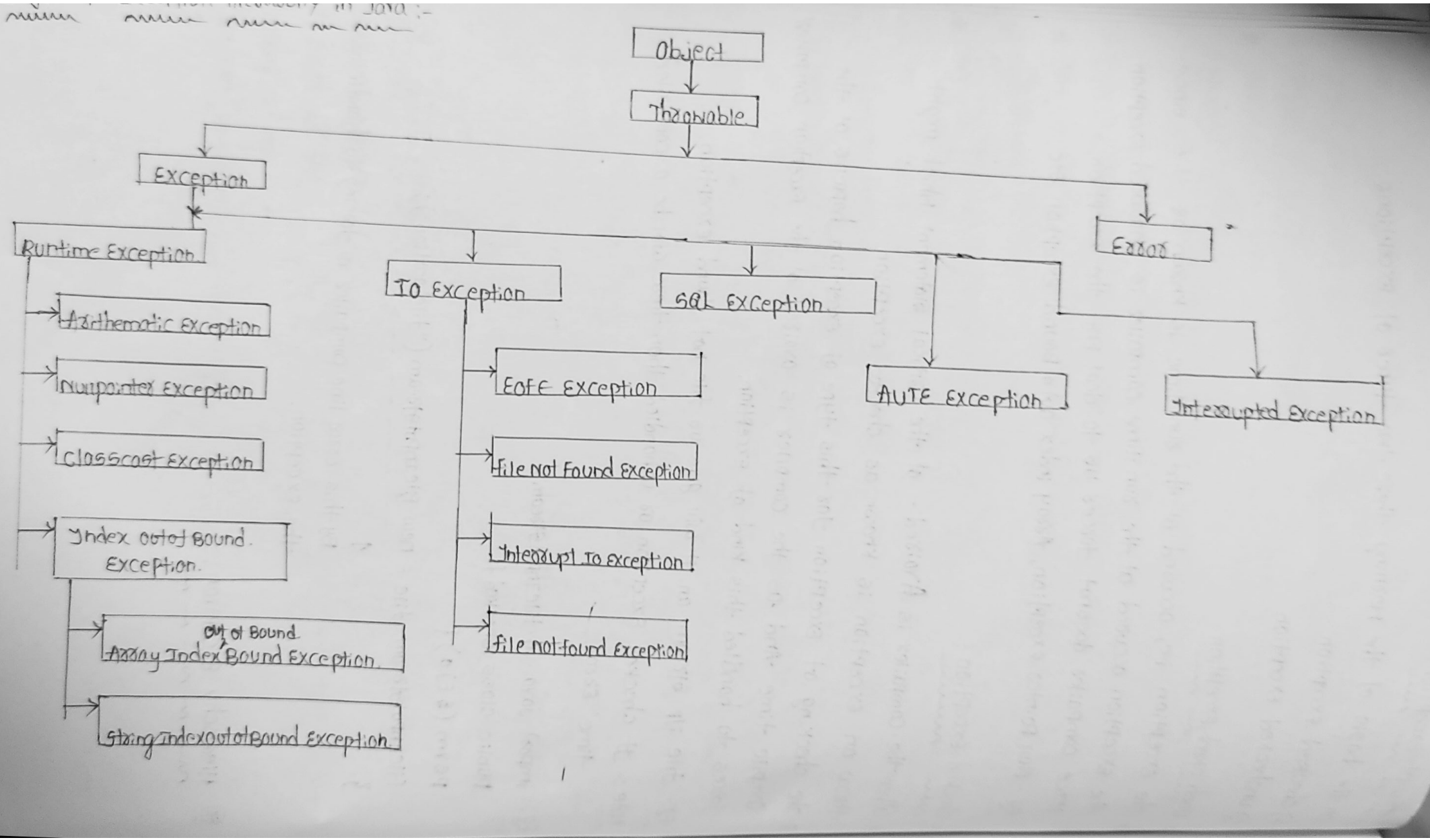
Critical Statement	Input values.	Output

int a = t.nextInt();
int b = t.nextInt();
int c = a/b;
	a = 8;
	b = 3;
		c = 2
		Normal execution.

	a = 8; b = 0;	Arithematic exception at line int c = a/b as number can't be divided by zero.
	a = 8; b = "Hi"	Input Mismatch exception at line int b = t.nextInt(); as datatype of b is int.

④ Null pointer exception :-

```
public class Test12{
PSVM(S[]a){
String S1 = null;
S.O.println(S1.charAt(2)); //Null pointer.
}}
```


Object
Throwable
Exception
Error
Runtime Exception
IO Exception
SQL Exception
AUTE Exception
Interrupted Exception
Arithematic Exception
Nullpointer Exception
Classcast Exception
Index out of Bound Exception
EOFE Exception
File not Found Exception
Interrupt IO exception
file not found exception
out of Bound
Array Index Bound Exception
String Indexoutofbound Exception

* All the exception except run time exception or checked exception.

(#) Exception handling:-

* There are two ways to deal with the exception.
  1. Exception handling by using try and catch block.
  2. Declaring the exception of throwing the exception.

* TRY & catch :-

* We can handled exception using try & catch block & in try block we will write all the critical statements and as a parameter of a catch block we will give the object reference. of the type of exception object that might be thrown from my try block.

```
try {
    ____________  }
    ____________  }  Critical
    ____________  }  Statement.
}
```

If object of exception/throwable type is created.

It will check for the reference type.

```
Catch (            )
{ if reference matches or
  it is of super type then
  catch block extends }.
```

* In this provided example. If the catch block Executes then wen can say the Exception is handled.

* Eg:-
```
Import java.util.Scanner;
public class trail {
P S v m (S [] a){
try { Scanner t = new Scanner (S. in);
int a = t.nextInt();
int b = t.nextInt();
int c = a/b;
S.o.p ln (c);
S.o. pln ("hello there");
} catch (Exception e){
S.o.pln ("Exception occured");
}
```

8) WAP to accept two integers value as a string and return sum of both values. If either one of value is not integer then return 0.

Ans :-

```
import java.util.Scanner;
public class Addition {
public static void main (String[] a){
Scanner t = new Scanner (System.in);
String a = t.next();
String b = t.next();
System.out.println (add(a,b));
}
public static int add(String a, String b){
try {
int a1 = Integer.parseInt(a);
int b1 = Integer.parseInt(b);
return a1+b1;
} catch (Exception e){
return 0;
}
}
```

* TRY-block :-

* TRY-block should be return inside method block or Constructor block or any other block on side try block we will write critical statement which can cause an Exception.

* TRY-block can't exist on it's own so we need to provide a catch block weather try block. (try block can be written with catch block as well as finally block).

* Catch-block :-

* Catch block as a reference of Exception type object and whenever an Exception is caused in the try block. The catch block matching exception type or with the higher exception type will execute.

* Inside catch block we can write alternate program to the excute at the time of Exception we can call inbuild function of exceptionClass.

* TRY-with Multicatch :-

* In this way of handling exception we will use one try block with multiple Catch block.

try {

```
        ---------
        ---------
        ---------
    }
    catch (Type1 e) { ≡ } -> if Exception type matchs then this
                             class will execute.
    catch (Type2 e) { ≡ }
    catch (Exception e)
    {
    }
```

Note:- catch block with exception type reference should always come at the last as this catch block can handle all type of exception.

* print stackTrace method :-

*

eg of try with multicatch :-

```
import java.util.InputMismatchException;
import java.util.Scanner;

public class TrailArith {
psvm (S[]a){
Scanner t = new Scanner (S.in);
S.O.println ("Enter data");
try {
int a = t.nextInt();
int b = t.nextInt();
int c = a/b;
S.O.println(c);
}catch (ArithmeticException e){
  S.err.println("Math problem");
  e.printStackTrace();
} catch (InputMismatchException e){
  S.err.println("Input value issue");
} catch(Exception e){
  S.o.println("All exception handled");
}}
```

Catch block with multiple reference :-

```
* public class Trail {
    PSVM(S[]a){
    Scanner t = new Scanner(S.in);
    S.O.println("Enter data");
    try {
    int a = t.nextInt();
    int b = t.nextInt();
    int c = a/b;
    S.O.println(c);
    } catch (ArithmaticException | InputMismatchException e)
    e.printStackTrace();
  }
}
```

finally-block :- finally block is a block which will execute no

Eg:- finally block with trying catch :-

* fanally block can be written try and catch block and also with only try block

Eg:-
```
finally {
    t.close();
}
```

Eg:-
```
try {
    ==
}
Catch (Exception e)
{
  ==
}
finally {
    ==
}
```

Eg:-
```
try {
    ==
}
finally {
    ==
}
```

*.
```
public class finally {
  PSVM(S[]a){
  Scanner Sc = new Scanner(S.in);
  S.O.P("Enter data");
  try {
    int a = Sc.nextInt();
    int b = Sc.nextInt();
    int c = a/b;
    S.O.P(c);
  }
  finally {
      t.close();
  }
}
```

* WAP to get and handle following exception.

① Null pointer
② Giving array out of bound.
③ Class cast exception.

```
* public class Driver{
  p s v m (s[]a){
  try{
  Apple apple = (Apple) new Fruit();
  apple.eat();
  }catch (class exception e){
   s.o.println("class cast ex handled");
  }catch(exception e){
  s.o.println("All ex handled");
  }
}
```

```
* public class finallyTrail{
  p s v m (s[]a){
  Scanner scanner = null;
  try{
  Scanner = new Scanner(s.in);
  s.o.println("Enter a value");
  int a = Scanner.nextInt();
  int b = Scanner.nextInt();
  s.o.println(a+b);
  }catch (InputMismatchException e){
   s.o.println("exception handled");
  }finally{
  Scanner.close();
}}
```

Try with finally block and with "0" exception.

```
* public class finallyTrail{
   psvm(s[]a){
   Scanner Scanner = null;
   try{
   Scanner = new Scanner(s.in);
   s.o.println(Scanner.next());
   }finally{
     Scanned.close();
   }
}
```

* TRY With Resources :-

```
* import java.util.Scanner;
  public class TryWithRes {
  p s v m (s[]a) {
  try (Scanner sc = new Scanner (S.in)) {
  String a = sc.next();
  S.o.println(a);
  Sc.close ();
  } catch (Exception e){
   e.printStackTrace();
  }}
```

T.W.R. Def :- T.W.R was introduced in Java 1.7 update. an in this type of try we have function or an option to create an object of resources as parameter (or) arguments. When there is an exception in the try block (try with resources) than the object of resources. which is local to try block become dereferred object or Abended object which is distroied by garbage collector.

* final	finalize().	finally
* Final is a keyword. * It is applicable for ① Variable. ② Methods & ③ Classes.	* Finalize is a Method. * It is always execute used to deallocate the resources which is allocated by used object.	* finally is a block. * It is always execute weather the exception is handled by user or not.

* Nested try & catch block :-

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
public class checked {
p s v m (s[]a) throws FileNotFoundException {
FileInputStream file = new FileInputStream ("filelocation");
}
}
```

Declaring an exception :-

* Giving information to the color of the function than the called function as the critical statement which might throw or given exception.
* To declare an exception we need to use throws keyword after method signature followed by the type of exception which mightly thrown.

syntax :- accessmodifier modifier returntype identifier (argument) throws Type of Ex1, Ex2, ... n.

```
{
==
}.
```

Note :- If the called function is throwing run time exception than the compailer does not make it mandatory to deal with the exception.

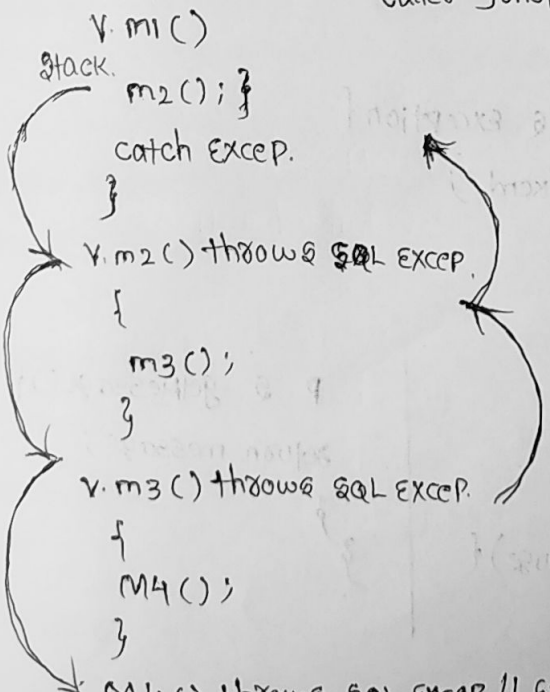
2-classes.

eg :-
```
import java.io.FNFE;
public class umbrella {
public umbrella () throws SQLException{
}
public void raining () throws fnfe {
s.o.println("sunny" + " ");
}
}
```

```
import java.io.fnfe;
public class person {
psvm (s[]a) {
umbrella u = new umbrella;
u.raining;
}
}
```

propogation of exception :-

The flow of exception object from called function to caller function it is known as P.O.E.



Exception :-

Deal :- Handle

```
try {
}
catch(){
}
```

```
try{
}catch(){
}
catch(){
}
```

```
try ( )
{
}
finally {
}
```

* Declare
  ↓
  throws.

Eg :-
```
Import java.util.Scanner;
public class Trail {
P S V M (S[] a) throws WeekendClass {
Scanner scanner = new Scanner (S.in);
S.o.println("Enter day name");
String day = scanner.next();
if (day.equalsIgnoreCase ("saturday") || day.equalsIgnoreCase ("Sunday")){
throw new WeekendClassException();
}else{
S.o.println("enjoy classes");
}
}
}
```

↓

Another class.

```
Public class WeekendClassException extends Exception{
}
```

Custom Exception :-

```
Public class WeekendClassException Extends Exception {
Public String message = " class on Weekend ";
public WeekendClassException () {
}
P WCE (String message){
this.message = message;
}
public WCE (String message, Throwable cause){
super(message, cause);
}
@ override
```

```
P S getMessage(){
return message;
}
}
```

* Another class :-

```
public class Test{
public void m (String[] a){
Scanner sc = new Scanner(System.in);
System.out.println("Enter day name");
String day = scanner.next();
try{
dayname(day);
}catch(WCE e){
e.printStackTrace();
}}
public void dayname(String dayName)throws WeekendException{
if(dayName.equalsIgnoreCase("saturday")||
   dayNam.equalsIgnoreCase("sunday")){
throw new WCE("new message",new RuntimeException());
}else{
System.out.println("enjoy classes");
}}
```

-: Assignment :-

① * WAP to store student type object inside array list an design a function to get student details on the basis of student id, if student data for given id is not present than throw custom no data for given id exception.
* This exception should be runtime exception.

② Difference b/w throws & throw.

② throw keyword :-

* throw keyword is used to manually throw an object of throwable type. throw keyword is usually used to throw custom exception.

③ Difference b/w throw & throws.

S.No	Throw	Throws
①	throw keyword is used to throw an exception object only.	① throws keyword is used to declare an exception as well as by pass the caller.
②	throw keyword always present inside method.	② throws keyword always used with method signature.
③	We can throw only one exception at a time.	③ We can handle multiple exceptions using throws keyword.
④	Throw is followed by an instance.	④. throws is followed by class.

Java 1.8 update:-

- static & default in interface.
- Lamda expression.
- Stream API.
- optional class.

Eg:- public class Implclass implements A, B {
@override.
public void m1(){
S.o.p.ln("hey");
}}

inter A
default void m1(){
}

inter B
default void m1(){
}

Class (implements A, B){
@override // it becomes mandatory to override the common method from parent interface.
public void m1(){
impfrom A or B or new implementation
}}

Eg:- public interface A{
default void m1(){
S.o.p.ln ("m1 method in As")
}
default Boolean equals (obj obj){
return true;
}}

After 1.8 upate (Java) we got many features & the following most important features.

update in interface:-

* After 1.8 upodate now it is possible to create static & default methods in interfaces. Both of this methods are concrete methods.
* When one class will implement more than one interfaces than their might be situation then parent interfaces shared. same method.

[* In this case the child class. it becomes mandatory that to override the common method & choose b/w] -> no point

* In this case it become's mandatory to override the common method in child class & those from parent implementations or provide own implementation.

Note:- We can't create copy of method from object class in interface.

eg: C imp A, B // extends objects.

```
public class Driver {
psvm($[]a) {
Student $ = new Student (1, "Legain");
Student $1 = new Student (2, "Ram");
Student $3 = new Student (3, "Sita");
Student $4 = new Student( 4, "Valmiki");
Student $5 = new Student ( 5, "Mahesh");
List < Student > list = new ArrayList ();
list.add ($4);
list.add ($3);
list.add ($2);
list.add ($1);
list.add ($5);
Comparator < Student > com = (e1, e2) -> {
return e1.getName().CompareToIgnoreCase (e2.getName());
};
Collections.Sort(list,com);
list.forEach (S.o.::pln);
}
}
```

* Lambda Expression :-

() -> {} -> Syntax.

* Lambda Expression was introduced in java 1.8 update. It is used to provide implementation to abstract method of functional interface.
* While using L-E we can provide undeclared argument of the implementation which we want to provide to abstract method of functional interface.
* On this basis of argument an implementation java desides which abstract method we are trying to use.
* Return Value :- By providing an implementation Lambda E. will return an obj of implementing class of specific functional interface.

(①) —②→ {③}.

①. Inside the parantheses we need to provide same num. of argument excepted by Abstract method of functional interface.

② → (Hiphen with greaterthan symbol)

* This symbol is known as Lambda token.

③. Inside the Lambda block we provide the implementation which we need to implement in Abstract method of functional interface.

Note :- If there is one argument an single line implementation than we can ek parantheses & curlybraces.

* Stream API :-

* Stream API was introduced Java 1.8 update this API is used to perform operation bulk object or collection type of object.
* Java.util.Stream was the package added 1.8 update containing important classes. an interface related to Stream API.

* Filter Method :-

* This method is from interfaced called stream. And it is used to filter the elements or data on the basis of implementation provided to abstract method test from functional interface called predicate.

* Predicate :-

* predicate is an functional interface Java.util.Function this interface as a method called test which excepts object type data an return boolean value.

* Collect method :-

* Collect method it is used to except the data & that data is passed to collectors class method which is excepted as parameter of collect method.

* Collectors class :-

* Collectors class has static method like toList(), toSet(). which excepts data and which is used to return the object of collection type having those data.

⑭ Example for filter method() :-

```
* Public class TestFilter {
    P S V m (String [] args) {
  Laptop l1 = new Laptop (1, "Aspire", "Acer", 2);
  Laptop l2 = new Laptop( 2, "omen", "Hp", 16);
  Laptop l3 = new Laptop (3, "Legion", "Lenovo", 8);
  Laptop l4 = new Laptop( 4, "thinkpad", "Lenovo", 1);
  Laptop l5 = new Laptop (5, "predeator", "Acer", 32);
```

```
Laptop L6 = new Laptop (9, "basicx", "HCL", 1);
Laptop L7 =new Laptop (9, "basicx", "HCL", 4);
List <Laptop> list1 = List.of (L1, L2, L3, L4, L5, L6, L7);
List <Laptop > list 2 = list1. stream (). filter (e ->e.getRam () >= 4)
                        . collect (collections.toList ());
list2. forEach (s.o::println);
}
Public class Laptop {
private Integer id;
private string brand;
private string model;
pirte int Lucky;
Laptop (int id, string brand, string model, int Lucky) {
this. id = id;
this. brand = brand;
this. model = model;
this. ram = ram;
}
Getter Setter method for id, brand, model, Lucky;
}
```

* Consumer interface:-

* Consumer interface is from java.util.function package. is a functional interface & has one Abstract method called Except.

* Except method:-

* Except method is from consumer interface it excepts object type data an argument but has return type as void.

① Consumer

① Eg:-

```
package streamAPI;
import java.util.ArrayList;
import java.util.List;
import java.util.function.consumer;
import java.util.*;
Public class Test forEach {
Public static void main (String [] a) {
List <Integer> list1 =new ArrayList<>();
list1.add (20); list1.add (12);
list1.add (15); list1.add (17);
• consumer <Integer> con = new consumer
                <Integer> () {
  @Override
  public void accept (Integer c) {
};
list1. for each (con);
s.o.println (con);
```

⊛ For Each method :-

* It is from iterable method (interface). Inside for each method it have implementation of enhansable for loop. this method accept object refer of implementing class of consumer interface as an argument.

Eg :-
```
public class TestForEach {
    P S V m (S []a){
    Laptop a2 = new Laptop (2, "omen", "HP", 16);
    Laptop a3 = new Laptop (3, "Legion", "Lenovo", 8);
    List<Laptop> list1 = new ArrayList<>();
    List.add (a3) ;
    List.add (a2) ;
    list1.forEach ( e -> {S.o.println (e.getName());
                          S.o.println(e.getBrand());
                         });
    }
}
```

⊛ MAP() :-

Eg :-
```
imp.j.u.Al;
imp.j.u.L;
i.j.u.function.Consumer;
i.j.u.stream.Collectors;
P.C TestForEach { Cos] TestMap1 {
P S V m (S [] a){
List<Integer> list1 = new ArrayList<>();
list1.add(2);
List2.add (3);
List<Integer> list = list1.stream().map (e -> e*e).collect(Collectors.toList());
S.o.println (list);
}}
```

O/P :-
4
25

⇓ Same method

```
Function<Integer, String> fun = new Function<Integer, String>(){
@Override
public String apply (Integer i) {
return i*i+"a";                    S.o.println("Squaring");
};}
List<String> list = list1.stream().map(fun).collect(Collectors.toList());
S.o.println (list);
```

(#) MapMethod():-

* Map method excepts object reference of implementing class of functional Interface. on the basis of implementation of apply method. From function interface we can provide or perform operation on the element.

(#) Function Interface:-

* function is a functional interface from java.util.function package. this interface has two generics.
  1. For input data. type of the Apply method.
  2. As a return type value.

(#) Apply Method:-

* Apply method is from function functional interface. It is an Abstract method.
* on the basis of data which is taken as argument of this method we can modify the return type.

Eg:- import java.util.ArrayList;

public class TestMap2 {