1) changing the ringtone for a particular contact is an example of method overriding.

2) Putting whatsapp group on mute is an example of overriding.

## Polymorphism:-

1) Poly means many & morphism means forms.
2) One thing showing multiple behaviour is called as poly morphism.

Ex:- Polytechnique ——— Poly- ~~means~~ many
technique — technologies

Poly clinic ——— poly — many
clinic ——— treatments

Two types

① Compile time polymorphism

Ex:- Method overloading ie.,
```
run()
run(int i)
run(char ch)
run(String s, int i)
```

Def:- During compile time if one thing showing multiple behaviour then it is called Compile time poly----.

Ex:- Method overloading where during compilation compiler will decide which behaviour to be implemented so here name is same but depends on args it shows different behaviour.

Technical def:-

The process of resolving call to overloaded method during compile time depending on type of args is called as compile time polymorphism.

It is also called as "static polymorphism or compile time binding."

② Run time polymorphism :-

During Run/execution time, one thing showing multiple behaviour is called Run time polymorphism.

ex:- Method overriding

Where there are multiple methods with same name, during execution time only JVM will come to know which method to be executed depending on type of object.

```
class A
{
  public void run()
  {
    Sop("A class Method");
  }
class B extends A
{
  public void run()
  {
    Sop("B class Method");
  }
class Main
P S V m (String args[])
```

```
{
    A a1 = new A(); a1.run(); // It calls class A run()
    B b1 = new B(); b1.run(); //  "    "   class B run()
    A a2 = new B(); a2.run(); // It calls class B
}
```

Tech def:-

The process of resolving call to overridden method during run time depending on type of object we create is called as run time polymorphism.

Also called dynamic binding (or) dynamic polymorphism (or) dynamic method dispatch.

28/03/2020

Method Binding:-

The process of connecting method signature with method implementation is called method binding.

Ex:- p s v run( ) -------- { Sop("In run"); }

TYPES

① Compile Time Binding:-

The process of connecting method signature with method implementation during compile time is called as compile time binding.

⇒ Since binding is happening before execution so it is also called early binding.

⇒ All static and final methods will get binded during compile time ie, during compilation only method will come to know what is its implementation.

② Run time binding:-

The process of connecting method signature with method implementation during execution.

⇒ Static binding is happening at during execution, it is also called late binding.

⇒ All non static and abstract methods will get binded during run time ie, until program not executed method will not come to know what is it implementation.

That is the reason non static and abstract method can be overridden.

Ex:-
```
public void run()
{
    Sop("In run method");
}
```

| STATIC BINDING | DYNAMIC BINDING |
|---|---|
| 1) Also called Early binding | 1) Also called late binding |
| 2) It takes place at compile-time. | 2) Takes place at run-time. |
| 3) It uses overloading method | 3) It uses overriding method |

30 Mar 2020

# * ARRAY *

Def:- Array is collection of homogeneous elements.

=> Whenever we want to use multiple or group of elements or data at same time then we go for arrays.

for ex:- If we want to use 1 to 100 at same time its not possible to store in variables and use it because it is very leugthy porocess. So for such kind of situations java has given array.

Syntax:- (1) `ARRAYTYPE ARRAYNAME[]=new ARRAYTYPE[SIZE];`

Ex:- (i) `int a[] = new int[3];`

(ii) `double d[] = new double [size];`

(2) `ARRAYTYPE ARRAYNAME[] = {no. of elements};`

* Array stores the elements in index format which always starts from 0 (for above (i) ex) i.e., 0, 1, 2.

* Once we created an array default values will be present.

```
// Storing elements in array //
a[0] = 10;
a[1] = 20;
a[2] = 30;

double d[] = new double [4];
d[0] = 0.22;
d[1] = 0.33;
d[2] = 0.44;
d[3] = 0.55;
```

// for printing elements of array //

```
S o plu (a[0]);
S o plu (a[1]);
S o plu (a[2]);
S o plu (d[0]);
S o plu (d[1]);
S o plu (d[2]);
```

// When we already knows elements of array we can also create the array as //

arraytype arrayname[] = {elements};

```
int a[] = {10, 22, 33, 44, 555, 666, 7777, 88};
        Sop (a[0]);
        Sop (a[1]);
        Sop (a[2]);
```

// If we store elements more than decclared size we will get array index out of bounds exception.

Ex:

```
a[] = new int[3];
a[0] = 11;
a[1] = 22;
a[2] = 33;
a[3] = 44; // Aorray index out of bounds
                exception //
```

## Length variable :-

It provides length of array and length will always calculated from 1.

Ex: int a[] = new int[300];

S o p (a.length); // o/p is 300

WAP to create an integer array of size 5 and store elements as 12
24
35
56
78

and print first and last element

Prog :-

```
public class ArrInt
{
    p s v m (String args[])
    {
        int a[]= new int[5];
        a[0] = 10;
        a[1] = 20;
        a[2] = 30;
        a[3] = 40;
        a[4] = 50;

        S o p ("first element is" + a[0]);
        S o p ("last element is" + a[4]);
        S o p ("last element also print as" + a[a.length-1]);
    }
}
```

By using for loop

```
for(int i=0; i<a.length; i++)
{
    Sop(a[i]);
}
```

o/p:-
10
20
30
40
50

ASSIGNMENT:-

=>1) WAP to find sum of an array

Ex: Input is a[] = {10, 20, 30}

Output is 60

=>2) WAP to find average of an array {10, 22, 33, 40}

=>3) WAP to reverse an array

Ex: Input (10, 20, 33, 55)

o/p(55, 33, 20, 10)

=>4) WAP to greatest number {10, 2, 55, 66, 33, 22, 9}

=>5) WAP to find smallest number from {10, 2, 55, 66, 33, 22, 9}

① Prgm

```
public class Sample
{
    p s v m (String args[])
    {
        int a[] = {10,20,30};
        int sum = 0;
        for (int i=0; i<a.length; i++)
        {
            sum = sum + a[i];
        }
        S o p ("Sum of array is" + sum);
    }
}
```

②

Prgm :-

```
public class Sample
{
    p s v m (String args[])
    {
        float avg;
        avg = sum / a.length;
        S o p ("Average is" + avg);
    }
}
```

③

```
public class reverse
{
  p s v m (String args[])
  {
    int a[]= {10,33, 44,55, 66};
    for(int i= a.length-1; i>=0; i--)
    {
      S o p (a[i])
    }
  }
}
```

④ Prgm

```
public class greatest
{
  p s v m (String args[])
  {
    int a[] = {10, 33, 44, 55, 66};
    int min = 0;
    int max = 0;
    for (int i=0; i< a.length; i++)
    {
      if(max < a[i])
      {
        max = a[i];
      }
    }
      S o p ("Greates num is" + max);
  }
}
```

for min value

↓

written with black

⑥ WAP to find second highest number from an array {10, 22, 7, 99, 6}

⑦ WAP to find second smallest from an array {10, 22, 7, 99, 6}

⑥ pgm:

```
public class Secondhighest
{
  p s v m(String args[])
  {
    int a[] = {10, 22, 33, 444, 777, 1};
    int m=0, sm=0;
    for(int i=0; i<a.length ; i++)
    {
      if(a[i]>m)
      {
        sm=m;
        m=a[i];
      }
      else if(a[i]<m && a[i]>sm)
      {
        sm=a[i];
      }
    }
    Sop(m);
    Sop(sm);
  }
}
```

We can solve this in two ways
1) Sorted array
2) Unsorted array

Prgm :-

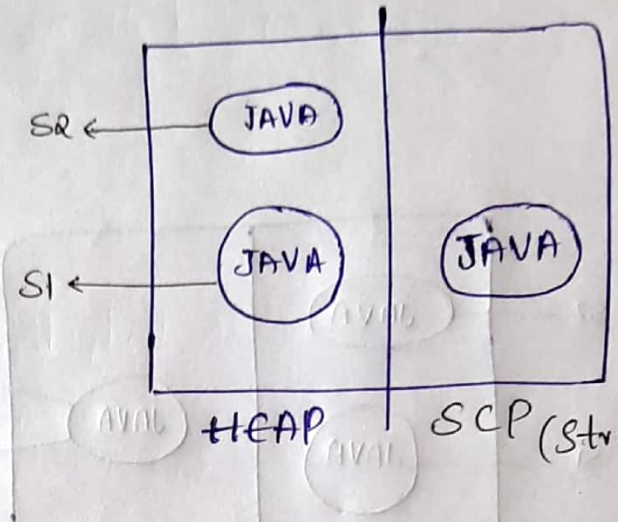
```
Package Arrayprog;
publit class sorting;
{
  P   S   V   m (String args[])
  {
     int a[] = {10, 22, 5, 77, 9, 8};
     int temp;
     for( int i=0; i<a.length; i++)
     {
       for( int j=0; j<a.length-1; j++)
       {
         if (a[j] > a[j+1]) // 10>22// 22>5 // 22>77//
         {
            temp = a[j];
            a[j] = a[j+1];
            a[j+1] = temp;
         }
       }
     }

     for (int k=0; k<a.length; k++)
     {
        Sop (a[k]);
     }
  }
```

## String :

It is a predefined class which is present in java.lang package

In java string is an

1) Object
2) datatype
3) class
4) group of characters




* Immutability

Two ways (Object creation

① Using new keyword : two objects will get created

(i) One is under heap area — ref variable assigned to it

(ii) Another is under string constant pool

Reference assigned to it is only for backup

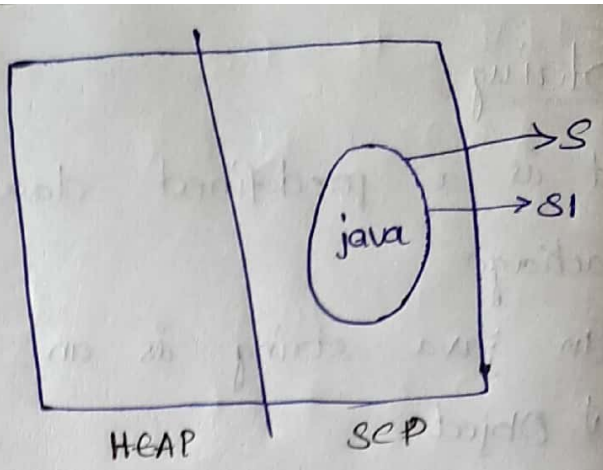
```
String s1 = new string ("java");
String s2 = new string ("java);
```

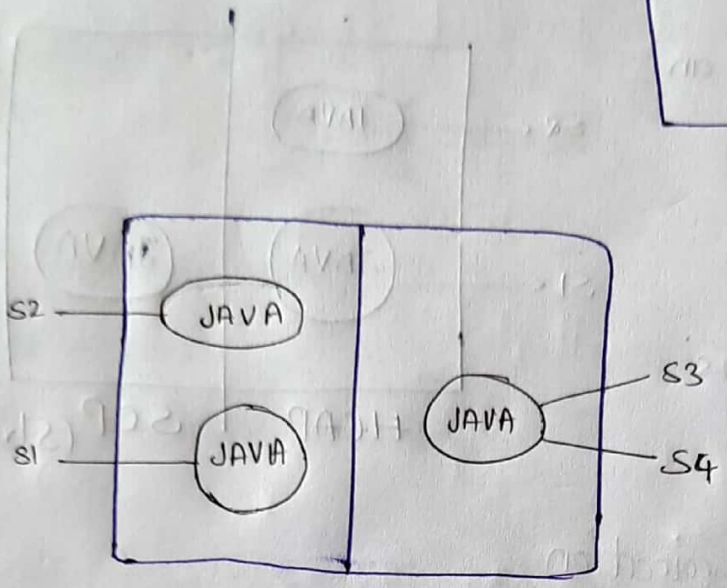
② Using literal : First JVM will go to SCP and check is there any object present with string data.

If it is there already it will assign a reference to the existing data.

If it is not there it creates new object and assign reference to it.

String s = "java";
String s1 = "java";



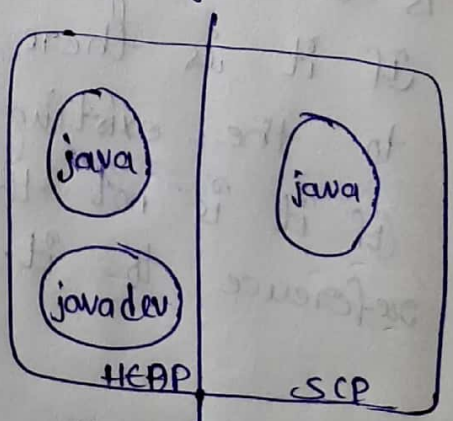




String s1 = new String ("java");
String s2 = new String ("java");
String s3 = "java";
String s4 = "java";

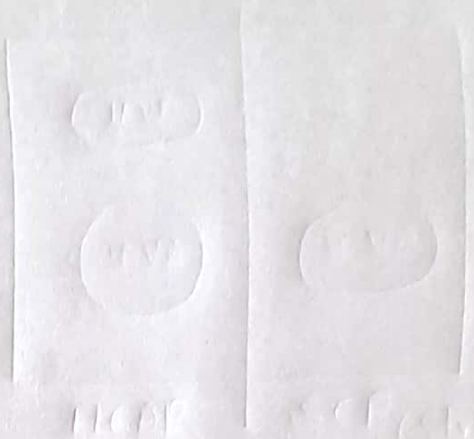
⇒ from the above example we can understand that for one string object there can be multiple reference assign to it.
If we changes data of one string object, it will effect to multiple.

```
public class demo
{
  p s v m(String args[])
  {
    String s1 = new String ("java");
    s1.contact (" dev ");
      Sop(s1);
  }
}
```

String s1 = new String ("java")

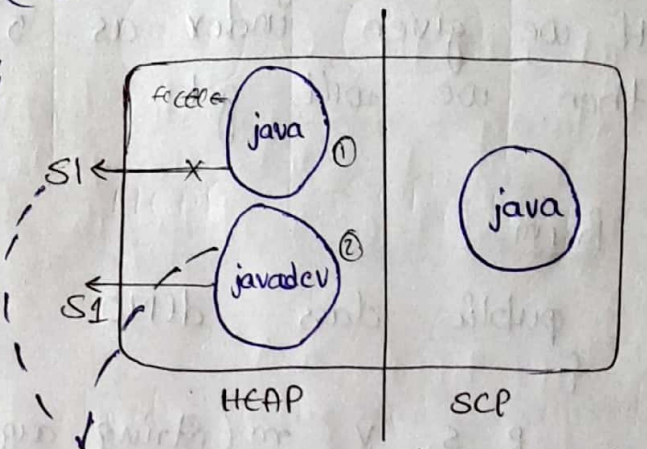



s1.contact (" dev ");
SOP (s1);



Pgm :-

```
public class DEMO
{
  p s v m(String args[])
  {
    String s1 = new String ("java");
    s1 = s1.concat ("dev");
    S o p (s1);
  }
}
```




Since s1 is referring to concatinated object, this reference is deleted

→ no s1 is refer to this object

Methods of string class :-

① Length() : int

* It provides length of the string
* Length will always calculated from 1

Ex:- String s = "java";

S o p (s.length());

O/P ⇒ 4

② charAt (index): char

It provides character at given index

Ex: String s = "java";

S o p (s.charAt (3));

O/P → a

If we given index as '5' ie, which does not exist then we will get

Ex:- Prgm:

```
public class DEMO
{
  p s v m (String args[])
  {
    String s = "java";
    Sop (s.length());
    Sop (s.charAt (2));
    Sop (s.charAt (5));
  }
}
```

O/P:- 4

v

Exception in thread "main" java.lang.StringIndex( )

① WAP to print no. of 'e' characters present in given string. String s = "Eewowoweeee"

② WAP to print all vowels from given string

String s = "javadevelopment"

③ WAP to count total no. of vowels from given string

String s = "javadevelopment"

④ WAP to count individual no. of vowels from given string

String s = "javadevelopment"

①
```
public class DEMO
{
  p s v m (String args[])
  {
    String s = "Eeeejsvee";
    int count = 0;
    for (int i=0; i<s.length(); i++)
    {
      if (s.charAt(i)=='e')
      {
        count ++;
      }
    }
    S o p (count);
  }
}
```

⑤
```
public class demo
{
  p s v m (Str...)
  {
    String s = "javadevelopment";
    for (int i=0; i<s.length(); i++)
    {
    if (s.charAt(i)=='a' || s.charAt(i)=='e'
        || s.charAt(i)=='i' || s.charAt(i)=='u')
    {
      S o p (s.charAt(i));
    }
  }
}
```

④

```
public class String3
{
 p s v m(String args[])
{
 int countofa = 0;
 int countofe = 0;
 int countofi = 0;
 int countofo = 0;
 int countofu = 0;
 String s = "javadevelopment";
  char[]ch = s.toCharArray();
 for(int i=0; i<ch.length; i++)
{
  if(ch[i]=='a')
  {
   countofa++;
  }
  if(ch[i]=='e')
  {
   countofe++;
  }
  if(ch[i]=='i')
  {
   countofi++;
  }
  if(ch[i]=='o')
  {
   countofo++;
  }
  if(ch[i]=='u')
  {
   countofu++;
  }
 }
```

```
 Sop(countofa);
 Sop(countofe);
 Sop(countofi);
 Sop(countofo);
 Sop(countofu);
 }
}
```

04/04/2020

③ Syntax: object1.equals(objecta): boolean

It checks whether two strings are same or not ie., compares two strings based on storing data

Ex:- 1) String s1 = "java";
String s2 = "java";
Sop(s1.equals(s2)); // true

2) String s1 = "java";
String s2 = "javadev";
Sop(s1.equals(s2)); // false

3) String s1 = "java";
String s2 = "JAva";
S o p (s1.equals(s2)); false
-> Because equals method consider case sensitivity

4) Syntax: object1.equalsIgnoreCase(object2);

Compare two things strings based on string data without considering case sensitivity.

* WAP to find reverse of a string & and also string is palindrome or not.

Prgm:-

```
public class reverse
{
  P S V m(String args[])
  {
    String s = "Mom";
    String s1 = "";
    for(int i=s.length()-1; i>=0; i--)
    {
      s1 = s1 + s.charAt(i);
    }
    S o p ("Reverse of a string is " + s1);
    if (s1.equalsIgnoreCase(s))
    {
      S o p ("Its paliudrome string");
    }
    else
      S o p ("Non paliudrome string");
  }
}
```

* WAP to count no. of words present in string i.e, i am java developer

Prgm:-

```
public class numsto
{
  P S V m(String args[])
  {
    String s = "java dev is easy";
    S o p(s);
    s = s.trim();
    S o p(s);
    int count = 1;
    for(int i=0; i<s.length(); i++)
    {
      if(s.charAt(i) == ' ' && s.charAt(i+1) != ' ')
      {
        count++;
      }
```

```
    }
    S o p ("count is" + count);
  }
}
```

O/P:- java dev is easy
java dev is easy
count = 4

⑤ trim(): String; Syntax

It removes white spaces from starting and ending of string trim() but not from middle.

Ex:- s = " java dev "
s.trim(); ⟶ "java dev"

⑥ Syntax: substring(int arg): String

It provides subpart of a string.

Ex:- String s = "java";
String s1 = s.substring(1); // ava
S o p (s1);

substring(int fromindex, int toIndex): String

Ex:- String s = "java";
String s1 = s.substring(1,3); // 1, 3-1 // av
Sop(s1);

Ex pgm:-

```
public class sss
{
  p s v m(String args[])
  {
    String s = "java";
    String s1 = s.substring(1);
    S o p (s1);
    String s2 = s.substring(1,3);
    S o p (s2);
    String s3 = s.substring(1,2);
    S o p (s3);
  }
}
```

O/P:- ava
av
a

* WAP to count forequency of substring in given string

String s = "we work to live and we live to be happy"
substring is Live
[have to count how many times live came]

Prgm:-

```
class QualitLabs
{
  p s v m(String args[])
  {
```

String str1 = "we want to be happy and live" + "to be happy live live live";

String word1 = "live";

check (str1, word1);

```
p s v check (String str, String word)
{
  String s[] = str.split(" "); int count = 0;
  for(int j=0; j<s.length; j++)
  {
    s o p (s[j]);
  }
  for(int i=0; i<s.length; i++)
  {
    if(word.equals(s[i]))
    {
      count++;
    }
  }
  s o p (count);
}
```

⑥ indexof() : syntax

indexof(args): int

indexof(String): int

indexof(String, from index): int

It provides index value for given string & character

for ex: String s = "java";

```
int i = s.indexof('a');
int j = s.indexOf("av");
int k = s.indexof("a", 2);
int l = s.indexof('z');
```

indexof() returns -1 if character & string is not present.

Prgm using indexof()

WAP to print all character only once from string

String s = "javajavajavadevdevdev"

o/p = javade

```
Public class dupl
{
  p s v m (String args[])
  {
    String s = "javajavajava development";
    String uni = "";
    for(int i=0; i<s.length(); i++)
    {
      char ch = s.charAt(i);
      if(uni.indexof(ch) == -1)
      {
        uni = uni + ch;
      }
    }
    s o p ("Non repeated chars are" + uni);
  }
}
```

⑦ split(args): String[]: Syntax

It converts & breaks the string into an array

Ex:-

```
String s = "i am developer";
String s1[] = s.split(" ");
```

O/P:- s1[0] = i
s2[1] = am
s3[2] = developer

```
string s = "java";
String ch[] = s.split("");
```

O/P:- ch[0] = "j"
ch[1] = "a"
ch[2] = "v"
ch[3] = "a"

WAP to calculate frequency of characters present in a string. I/P: javadev O/P:- j-1 d-1 a-2 e-1 v-2

Prgm:-

```
public class freq{
p s v m(  ){
  String s11 = "javadev";
String s = s11.toUpperCase();
char[] s1=s.toCharArray();
for(int i=0; i<s1.length; i++)
{  sop(s1[i]);
}
for(ch='A'; ch<='z'; ch++)
{
 int count = 0;
for(int i=0; i<s1.length; i++){
 if(ch==s1[i]){
   count++;
```

Syntax: toCharArray(): char[];

It converts string into character array.

Ex: String s = "javadev";
char ch[] = s.toCharArray();

O/P: ch[0] = 'j'
ch[1] = 'a'. . . . .

⑧ toUpperCase(): Converts string into uppercase.

⑨ toLowerCase(): Converts string into lowercase.

⑩ startsWith(String): boolean --> check whether string is starting with given string or not.

⑪ endsWith(String): boolean --> check whether string ends with given string or not.

⑫ Contains(String): boolean --> check whether string is present in given string or not.

⑬ .isEmpty: boolean --> 

Ex: Pgm

```
public class methods
{
  p s v m(String args[])
  {
    String s="javaDEvelopment";
    String up=s.toUpperCase();
    String lp=s.toLowerCase();
    String con=s.concat("is easy");
    boolean b1=s.startsWith("abcd");
    boolean b2=s.endsWith("corona");
    boolean b3=s.contains("java");
    boolean b4=s.isEmpty();
    Sop(s);
    Sop(lp);
    Sop(up);
    Sop(con);
    Sop(b1)
    Sop(b2);
    Sop(b3);
    Sop(b4);
  }
}
```

Replace():

Replace all occurances given character/sequence of character/string with replacement char/string respectively.

1) replace(givenchar, desirechar): string --> used to replace characters

2) replaceAll(givenString, desireString): String --> used to replace string

3) replaceAll(String regrex, String replacement)

regular expression

Ex: [0-9]
[a-3]
[A-Z]

WAP to replace e with a in give string "javadevelopment"

" " " "java" with core java in "javedev"

" " remove spaces from given string "java is easy".

" " " all capital letters. "jAavaADdev"

" " " " small letters "j1va234dev455"

" " " " vowels from string "javadevelopment"

Prgm

```
public class replacedemo
{
  p s v m(String args[])
  {
    String s="jAVA0EV1234ISEa sy";
    String s11=s.replace('e','a');
    String s1=s.replaceAll(" "," ");
    String s2=s.replaceAll("[0-9]","");
    String s3=s.replaceAll("[a-3]","");
    String s4=s.replaceAll("[A-Z]","");
    String s5=s.replaceAll("[aeiouAEIOU]","");
    S o p(s1);
    S o p(s2);
    S o p(s3);
    S o p(s4);
    S o p(s5);
    S o p(s11);
  }
}
```

Ex:-
```
public class repp
{
  P S V m (String args[])
  {
    String s = "I am a java developer and i am
                a good in java";
    String s1 = s.replaceFirst ("java", " ");
    S o p (S1)
  }
}
```

O/P:- I am a developer and i am good in java

Diff b/w equals and == :-

equals() ⟶ compare two strings based on String data.

== ⟶ compares two strings based on reference.

Ex:- Prgm

```
public class repl
{
  P S V m (String args[])
  {
    String s1 = new String ("java");
    String s2 = "java";
    String s3 = "java";
    S o p (s1.equals(s2)); // true
    S o p (s1 == s2); // false
    S o p (s2 == s3); // true
  }
}
```

## String Buffer and String Builder:

⇒ These are classes present in java.lang package.

⇒ String buffer and string builder objects can be created only by using new key word.

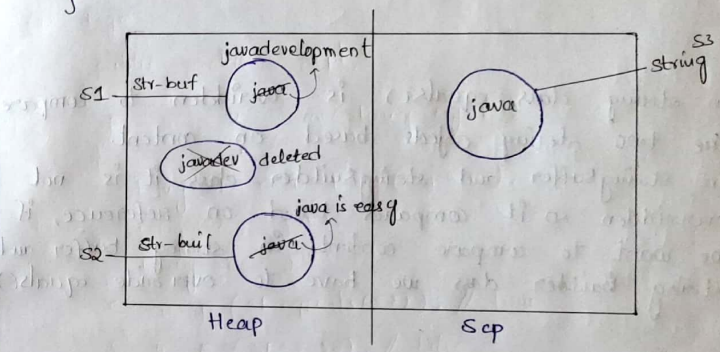
Ex: StringBuffer b1 = new StringBuffer ("java");
StringBuffer b2 = new StringBuilder ("java");
StringBuffer b3 = "java"; // invalid
StringBuffer b4 = "java"; // invalid

* String buffer & builder objects are mutable ie., once we created we can modify it.

* StringBuffer & StringBuilder objects will get created in heap area only.

Ex: Prgm

```
public class str
{
  p s v m( )
  {
    StringBuffer b1 = new StringBuffer ("java");
    StringBuilder b2 = new StringBuilder ("java");
    S o p (s1); // java
    S o p (s2); // java
    String s3 = "java";
    s3.concat ("dev");
    S o p (s3); // java
    s1. append ("development ");
    S o p (s1); // javadevelopment
    s2. append ("is easy");
    S o p (s2); // java is easy
  }
}
```




### equals( ):

⇒ In string class equal() compares based on string data.

⇒ In StringBuffer and StringBuilder class equals() compares based on reference.

```
public class str
{
  p s v m( )
  {
    StringBuffer s1 = new StringBuffer ("java");
    StringBuilder s2 = new StringBuilder ("java");
```

```
        String s3 = "java";
        String s4 = new String ("java");
    S   o   p (s1.equals(s2));//false - o3 equals() comparing references
    S   o   p (s3.equals(s4));//true - o3 equals() comparing content
  }
}
```

⇒ In string class equals() is overridden to compare the two string objects based on content.

⇒ In StringBuffer and StringBuilder class it is not overridden so it compares based on reference, if we want to compare content in string buffer and string builder class we have to override equals().

Access Specifiers :-

It provides the accessible permission to various fields of program.

Four types :-

1) Public
2) Private
3) Protected
4) Default

They are applicable to

1) Class
2) method
3) Variable
4) Constructors
5) Interface

Package: it is like a folder where all common classes kept together at one place.

Public specifier :-

=> It is the lowest level of specifier
=> public fields are accessable within class anywhere.
=> public fields are accessable within another class anywhere of same package.
=> public fields are accessable within another class of another package only after import statements.

import :-

=> It provides the privilige of using code of one class into another class

Syntax: import packagename.classname;
import packagename;

Ex:-

```
package Arraypprog;
public class d11
{
  public static int i=100;
  p s v m(String args[])
  {
    d22.fly();
    // accessable directly within class
    System.o.p(i);
  }
}
class d22
{
  public static void fly()
  {
    // i is accessable directly outside of class coz it is public.
    S o p(d11.i)
  }
}
```

⇒ Accessing public field in another class in another package but not only after import statement.

```
package dummy;
import Arrayprog.d11;
public class d22
{
    p s v m (String args[])
    {
        // TODO Auto-generated method stub
        S o p (11);
    }
}
```

⇒ If a class public then only it can be imported other wise we can't import public classes.

VALID
```
package java.ab;
ex:- public class A
     {
     }
     import java.ab.A;
```

INVALID
```
package java.ab;
ex:- class A
     {
     }
     import java.ab.A;
```

2) default specifier :-

⇒ In java there is no word called default ie., if we did not specify any modifier, JVM will consider it as default.

⇒ default fields can be accessable anywhere within class.

⇒ default fields are accessable outside of another class but that class belongs to same package.

⇒ default classes cannot be imported.

⇒ default specifier also called package specifier because it is restricted within package.

3) private specifier :-

⇒ It provides highest level of restrictions.

⇒ private members can also only be accessed within same class.

⇒ private fields cannot be accessed outside of decclared class.

⇒ A class can never be private.

⇒ Private methods can't be inherited and overridden.

protected specifier :

⇒ protected fields can be accessable anywhere within the class.

⇒ protected fields can be accessable within another class of another package but only after inheritance

```
package Arrayprog
public class d11
{
  protected static int i=100;
```

⇒ A class can never be protected.

NOTE :

⇒ We can write multiple import statements to use multiple classes

Ex:
```
public class A   import A
{
}
public class B   import B
{
}
public class C   import c
{
}
```

```
class D
{
  // some code //
```

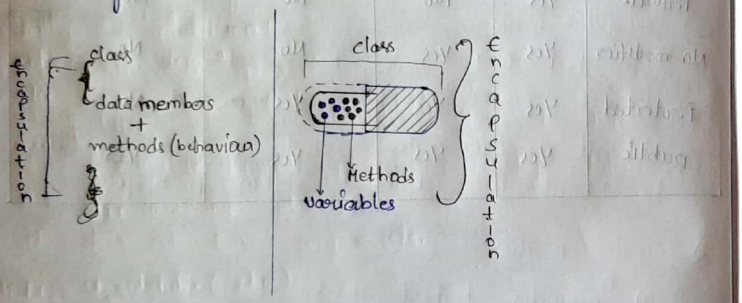
| Modifier | class | Package | Subclass | other classes |
|---|---|---|---|---|
| Private | Yes | No | No | No |
| No modifier | Yes | Yes | No | No |
| Protected | Yes | Yes | Yes | No |
| public | Yes | Yes | Yes | Yes |

10/04/2020

## Encapsulation :-

Def:- The process of wrapping the data member (variables) and member function (methods) into a single unit (class) is called encapsulation.

⇒ The main purpose of encapsulation is to achieve security of data.




What is the need of encapsulation?

⇒ We go for encapsulation to protect our data members from invalid user.

⇒ Ex:- If monthnum is not protected (encapsulated) user may issue the data member by assigning invalid values (like >12 & <1)

⇒ If data member is not protected (public) it is under user control ie., ~~under~~ user will decide what values to assign but those values can be valid & invalid.

```
package encapsulation;
public class demo
{
  p s v m(String args[])
  {
    App a1 = new App();
    a1.monthnum = 22;
    S o p("Month num is" + a1.monthnum);
  }
```

⇒ If datamember is protected (private) it is under developers control to decide which values to allowed

purpose :-

⇒ Protecting the datamembers by keeping them private and accessing through some special methods is nothing but encapsulation.

⇒ Rules for Encap :

① Declare all data members as paivate (if a datamember is paivate it is not accessable outside the class).

② Define separate setter and getter methods (it's not mandatory to define all only setter and getter we can define any user define methods).

```
package eucapsulation;
public class demo1 {
public s v m(String args[])
  {
   login ll=new login();
   // ll.username="john";

   ll.setdata("john");
    S o p(ll.veiwdata());
   ll.setpwd(9376);
    S o p(ll.veiwpwd());
  }
}

class login
{
  paivate String username;
  paivate int pwd;
  public void setdata(String username)
  {
    this.username= username;
  }
  public string viewdata()
  {
   if (username == "john").
   {
    return username;
   }
   else {
     return " bad person";
   }}
  public void setpwd(int pwd)
  {
   this.pwd = pwd;
  }
    public int viewpwd()
    {
      if (pwd == 9376)
      {
       return pwd;
      } else {
       return;
    }}
```

JAVA BEAN CLASS :-

If all the data members are private and there is seperate setter and getter method for each data member then that class is called java bean class.

Setter method :-

=> It is public in nature.
=> It doesn't have specific returntype.
=> It contains arguments same as datamembers.
=> Name of method is set followed by data member name.

Getter method :-

=> It is public in nature.
=> return type must be same as that of data member.
=> Name of method is get followed by data member name.
=> It does not have arguments.

NOTE :-

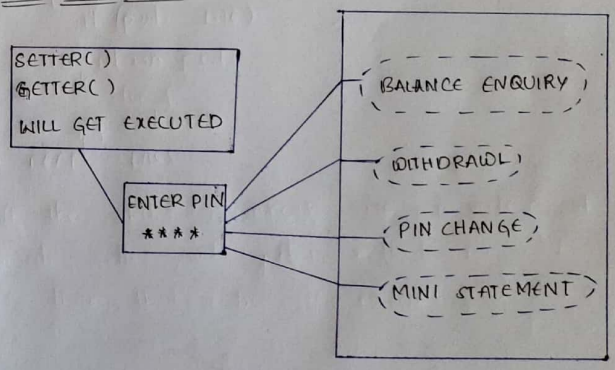
For every data member we have to define seperate setter and getter method.

Ex:-

```
package eucapsulation;
public class Adv
{
  p s v m(String args[])
  Employee e1=new Employee();
    e1.seteid(1234);
    S o p(e1.geteid());
  }
}
```

```
class Employee
{
  private int eid;
  private int eno;
    // set values
public void seteid(int eid)
{
   this.eid = eid;
} // get values
 public int geteid()
{
  return eid;
}
}
```

Ex of Eucapsulation



11/04/2020

## Examples of encapsulation :-

1) Protecting bank details from an invalid user to access is an example of encapsulation.
2) Keeping a password for mobile to protect data is an example of encap.
3) Protecting an online docs by keeping password is an ex of encap.
4) Securing house by keeping lock is an encap.

## fully encapsulated class :-

If all data members of a class are private and there is seperate methods to access those data such class is called as fully encapsulated.

```
class A
{
  private int i;
  private float f;
  private string s;
  // methods to access them //
}
```

## Partially encapsulated :-

If any data members of a class is non private such class is called as partially encapsulated.

```
class A
{
  private string name;
  public int age;
}
```

## No encapsulated class :-

If none of the data members of a class is private.

```
class A
{
  public string name;
  public int age;
}
```

## Advantages :

⇒ Securing the data
⇒ We can make our class as Write only by writing only setter method.
⇒ We can make our class as read only by defining only getter method.

## Super keyword :-

It is used to call superclass instance members (super class non static methods and non static variables).

⇒ We use super keyword when we want to call super class even after overriding occured.

Eg:-

```
class //SBI
{
  public void rateofinterest()
public class user
{
  p s v m(string args[])
{
  SBH SI= new SBH();
    SI.
  }}
class SBI
{
  public void rateofinterest()
  {
    S op ("7.5%");
  }
}
class SBH extends SBI
{
  public void rateofinterest()
  {
    S op ("8.5%");
  }
```

```
  public void interateSBI()
  {
    // rateofinterest();
    this.rateofinterest();
    super.rateofinterest();
  }
}
```

| Call to super [super()] | Super keyword |
|---|---|
| 1) It is used to call super class constructor. | 1) It is used to call super class instance members from subclass. |
| 2) super() must be first statement in constructor. | 2) Need not be the first statement. |
| 3) It is implicit as well as explicit | 3) It is explicit in nature. |

| Call to this [this()] | this keyword |
|---|---|
| 1) It is used to call one constructor from another constructor of same class | 1) It is used to indicate current object under execution. |
| 2) It must be first statement of the constructor. | 2) It can be any statement. |

| | |
|---|---|
| 3) This() can be used once in a constructor. | 3) It can be used multiple times in the constructor. |

## final Keyword :-

=> It is a keyword which indicates no more changes are allowed.

=> If we don't want data to be changed we can use final keyword.

final keyword is applicable with

1) Variable
2) Class
3) method

### final with variable :-

=> If a variable is declared as final we cannot reinitialise the value of it.

=> final is applicable with local var, static var and non static variable.

```
public class demo11
{
    final static int marks = 700;
    final float percentage = 60.5f;
    p s v m(String args[])
    {
        final String name = "pooja";
        S o p(name);
        name = "neha"; // not possible
        S o p(marks);
        marks = 75; // not possible
        demo11 d1 = new demo11();
        S o p(d1.percentage);
        d1.percentage = 66.05f // not possible
    }
}
```

NOTE :- Here it is mandatory to initialise final global variable during declaration, if we didn't initialize we will get CTE.

```
final static int marks ; // invalid
final float percentage // invalid
```

| STATIC | FINAL |
|---|---|
| 1) Static refers to single copy. | 1) final refers to fixed copy. |
| 2) Static keyword is applicable with global variable methods nested class and blocks. | 2) final keyword is applicable with class, method and variable. |

| | |
|---|---|
| 3) Static variable can be reinitialized. | 3) It cannot be reinitialise. |
| 4) It is not mandatory to initialise during declaration. | 4) It is mandatory during declaration. |
| 5) Ex: static String clg = "Qsp" | 5) Ex:- final float pie = 3.2145f; |

final with class :-

⇒ If a class is declared as final we cannot inherit it because final class

⇒ Is-A relationship is not applicable for final classes.

```
Ex:- public class demo
     {
       p s v m(String args[])
       {
        final class A
        {
        }
        class B extends A // we cannot in final class
        {
        }
       }
     }
```

∴ Super class cannot be final but sub class can be final.

final with method :-

⇒ If a method is final we cannot override it.

```
Ex:- public class demo11
     {
       p s v m (String args[]) {
        Flipkart fl = new Flipkart();
        fl.logo();
       }
     }
     class Amazon
     {
       final public void logo()
       {
       }
     }
     class Flipkart extends Amazon
     {
       final public void logo()
       {
       }
     }
```

* Can we inherit final methods?

⇒ Yes, we can inherit final methods but we cannot override it.

NOTE :-

⇒ We can have more than one access modifier for a method or variable.

⇒ All final methods can be static but all static methods cannot be final.

public static final void resultsofssc() // final & static cos no possibility of changing

public static void resultsoffirstsem() // static there is possibility to change.

14/04/2020

Object class :-

⇒ It is a super class to all predefine and user define classes.

⇒ Object class is present in java.lang package.

Methods of object class :-

1) getclass(): class ----->final
2) object1.equals(object2): boolean
3) tostring(): String
4) hashcode(): int
5) notify(): void//MT ------>final
6) notifyAll(): void/MT ------>final
7) wait(): void/MT ------>final
8) wait(long timeout): void/MT ------>final
9) wait(long timeout, int nanos): void/MT ------>final
10) finalise()
11) clone()

toString():

⇒ It is a method of object class. It provides complete information of an object.

⇒ Complete information consists of packagename.classname@objectaddress

Ex:- mypackage.Sample@33h3345

Ex :- pgm

```
package objectclsmethods;
{
public class sampl
{
  p s v m (string args[])
  {
    sampl s1 = new sampl();
    S o p (s1.tostring()); // s1.toString()
    // complete info of object.
    sampl s2 = new sampl();
    S o p (s2.tostring()); // s2.toString() ---> content of object
  }
}
```

o/p :- objectclsmethods.sample@
objectclsmethods.sample@

hashcode() :-

→ It's a method of object class. It prints the hashcode number for given object.

→ Hashcode number is simply a 32bit integer number.

→ It's a unique number allocated to every object by JVM.

→ If the object address are same they will have same hashcode number. Syntax:- public int hashcode()
{
return hashcodenum;
}

Ex:- Pgm

```
{
return hashcodenum;
}
package objectclsmethods;
public class Sample
{
  p s v m (string args[])
  {
    Sample s1 = new Sample();
    S o p (s1.hashcode());
    Sample s2 = new Sample();
    S o p (s2.hashcode());
    Sample s3 = s1;
    S o p (s3.hashcode());
  }
}
```

O/p:- 366712642
1829164700
366712642

Conclusion :-

① Output we get from toString() in practical there is no use of it, because if package changes and class changes and class changes output won't be same. So we it is programmer's responsibility to override hashcode() and get meaningful output.

② Output we are getting from hashcode() in practical there is no use of it to programmer because if object address changes it changes hashcode num. so it is our responsibility to override hashcode() and get meaningful output.

15/04/2020

Ex :- pgm

```
package objectclsmethods;
public class student
{
  String sname;
  int sid;
    p s v m (string args[])
  {
    Student s1 = new student ("john", 123);
```

```
        S o p (s1.toString());
        S o p (s1.toString());
    }
    public student (String sname, int sid)
    {
        this.sid = sid;
        this.sname = sname;
    }
    public String toString()
    {
        return "studentname:"+sname;
    }
    public int hashcode()
    {
        return sid;
    }
```

o/p :- studentname : john
123

equals() :-

⇒ It's a method of object class it compares two objects based on object address.

⇒ If object address is same output is true else it is false.

Ex :- program

```
public class Sample extends object
{
    sample s1 = new sample();
    sample s2 = new sample();
    sample s3 = s1;
    S o p (s1.equals(s2)); // false
    S o p (s2.equals(s3)); // false
    S o p (s1.equals(s3)); // true
}
```

Conclusion :-

Compares the object based on their object address which is not a proper way for comparison so it is programmers responsibility to override equals() so that it should compare based on features not based on address.

Type Casting :-

⇒ Converting one type of primitive datatype into another type of primitive datatype.

(or)

⇒ Assigning one type of value into other type is called type casting.

Divided into two types

1) Widening 2) Narrowing

① Widening:-

⇒ Converting smaller primitive data type into bigger primitive data type.

⇒ It is also called implicit casting.

byte → short → int → long → float → double

Since we are converting a smaller type into bigger type there is no loss of data.

Ex:- Pgm

```
package typeconversions.in;
public class widening
{
  p s v m(String args[])
  {
    byte b = 45;
    // widening
    short s = b;
    int i = s;
    long l = i;
    float f = l;
    double d = f;
      S o p(s);
      S o p(i);
      S o p(l);
      S o p(f);
      S o p(d);
  }
}
```

O/P:-
```
45
45
45
45.0
45.0
```

② Narrowing:-

⇒ Converting bigger primitive datatype into smaller primitive data type.

⇒ Narrow is also called as explicit casting.

byte ← short ← int ← long ← float ← double.

⇒ Since we are converting bigger type into smaller type there is loss of data.

Ex:- Pgm

```
package typeconversions.in;
public class narrowing
{
  p s v m(String args[])
  {
    double d = 163.45;
    float f = (float) d;
    long l = (long) f;
    int i = (int) l;
    short s = (short) i;
      S o p(d);
      S o p(f);
      S o p(l);
      S o p(i);
      S o p(s);
  }
}
```

O/P:-
```
163.45
163.45
163
163
163
```

## Life cycle of object :-

⇒ Objects can be created by using new keyword.

⇒ If an object is is no longer useful it is better make that object as abundon object.

⇒ Objects whose control is no longer with programmer or object who does not have refereuce or object whose refereuce is nullified, such objects are called abundon or objects.

For ex:

```
Dog d1 = new Dog();
Dog d2 = new Dog();
Dog d3;
Dog d4;
d3 = new Dog();
d4 = d3
d2 = null; // abundon obj
d1 = null; // abundon obj
```

16 Apr 2020

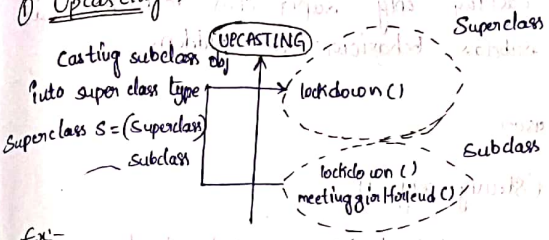
## Object castiug :-

Convertiug one type of object iuto auother type of object is called object castiug.

Two types

1) Upcastiug 2) Douncastiug

## 1) Upcastiug :-




Ex:-

```
class police
{
  public void lockdoun()
  {
    S o p ("stay at home");
  }
}
class Boyfrieud extends police
{
  public void lockdoun()
  {
    S o p ("No i will go out");
  }
  public void meetiuggirlfrieud()
  {
    S o p ("want to see her");
  }
}
```

Def:- Converting subclass object type iuto super class object type

(or)

⇒) Create an object of subclass and store it in a refereuce of super class is called as upcastiug.

⇒ During type casting only super class behaviour is visible and subclass behaviour is hidden.

Prgm continuation:-

```
public class user
{
  p s v m (String args[])
  {
    Police Pl = new Boyfriend(); // upcasted object
    Pl.lockdown(); // compiles successfully
    // Pl.meetinggirlfriend(); // CTE because during upcasting
                               subclass behaviour is not visible
```

⇒ Since subclass contains the properties of superclass upcasting is possible directly ie., superclass can hold the reference of subclass

⇒ So upcasting is implicit in nature.

Downcasting:

⇒ Converting a superclass object type into subclass obj type is called as downcasting

(or)

⇒ Converting an upcasted object into normal form is called as downcasting.

⇒ During downcasting both subclass and normal form is ~~called as downcasting~~. superclass behaviour is visible.

⇒ Since super class does not contains property of subclass directly downcasting is not possible, explicitly we have to convert it.

⇒ So it is explicit in nature.

⇒ Only upcasted obj is downcasted, ie., direct downcasting is not possible.

Ex:- Prgm

```
public class user
{
  p s v m (String args[])
  {
    Pl.lockdown();
    Pl.meetinggirlfriend();
    Boyfriend bl=(Boyfriend)Pl;
    bl.lockdown();
    bl.meetinggirlfriend();
  }
}
class police
{
  public void lockdown()
  {
    S o p ("stay at home");
  }
}
class Boyfriend extends police
{
  public void meetinggirlfriend()
```

```
    {
        S o p ("want to see her");
    }
}
```

17/04/2020

Ex:-

Pgm :-

```
package typeconversion.in;

public class App
{
    p s v m (String args[])
    {
        Appuser a1 = new admin();
        a1.view();
        // a1.Edit(); CTE
        Admin a2 = (Admin) a1;
        a2.Edit();
        a2.view();
    }
}
class Appuser
{
    public void view()
    {
        S o p ("U can view details are feeded
                poopely");
    }
}
```

```
class Admin extends Appuser
{
    public void Edit()
    {
        S o p ("U can edit details of any user");
    }
}
```

Examples :-

```
class A                 class B extends A       class C extends B
{                       {                       {
    P1()                    P2()                    P3() ;
}                       }                       }

A a1 = new A()          B b1 = new B()          C c1 = new C()
a1.P1();                b1.P1();                c1.P1();
a1.P2();                b1.P2();                c1.P2();
a1.P3();                b1.P3();                c1.P3();
```

Upcasting

```
B b1 = new A()
b1.P1();
b1.P2();
b1.P3();
```

Upcasting

```
A a1 = new B();
a1.P1();
a1.P2();
```

```
A a1 = new B()
a1.P1();
a1.P2();
a1.P3();
```

Downcasting

```
C c2 = (C) b1;
c2.P1();
c2.P2();
c2.P3();
```

```
B b2 = (B) a1;
b2.P1();
b2.P2();
b2.P3();
```

Object a1 = new A();

a1.

## Abstraction :

⇒ It is the second pillar of oops concept.

⇒ The process of hiding the internal implementation and showing the neccessary data to the end user is called as abstraction.

⇒ In the other words we can say that we are not going to show method implementation we will just provide method header and method signature.

Examples :-

① When we send an email, only a send button is visible but how it is being send is hidden.

② Remote, where we can operate it through buttons, but how it is being operated is hidden.

③ Driver knows only to apply brakes and acceleration but it's working is hidden.

In JAVA abstraction can be achieved in two ways

1) Using Abstract class
2) Using Interface

## Abstract class :

⇒ Abstract is a keyword which indicate incompleteness.

⇒ Abstract class is a class which contains atleast one abstract method.

⇒ Normal / concrete / complete method : If a method have method signature as well as method body such method is called complete or concrete method.

Ex:-
```
public void run()
{
    Sop("In run");
}
public void fly()
{
    Sop("  ");
}
```

⇒ Abstract / Incomplete method : If a method contains only method signature ~~but~~ but not implementation.

Ex:- public void run()
public void fly()

⇒ Abstract method has to represented with abstract keyword and we have to add ; in the end of method declaration.

Ex:- abstract public void srun();
abstract public void fly();

Ex:- Prgm

```
abstract class vehicle
{
  abstract public void noofwheels()
}
```

⇒ It is mandatory to mention the abstract keyword for abstract methods and classes.

```
ie) abstract class fruit
{
  abstract public void taste();
}
abstract class loans
{
  abstract public void type();
  abstract public void rateofinterest();
}
```

* Can we create object of abstract class ?

⇒ We cannot create object of abstract class because it contains abstract methods, and which does not have body to execute.

Ex:-

```
public abstract class vehicles
{
  abstract public void noofwheels();
}
```

```
class A
{
  Vehicles v1 = new vehicles(); // Instantiation is not possible
  v1.noofwheels();
}
```

18 Apr 2020

⇒ If any child classes are there for abstract class that child class has to undergo with any of the one rule.

1) Complete all incomplete methods of abstract class by means of overriding.
2) Declare your class also as abstract class.

Ex:- Prgm

```
abstract class vehicles
{
  abstract public void noofwheels();
}
public class Twowheeler extends vehicles
{
  public void noofwheels()
  {
    S o p("No of wheels are 4");
  }
  p s v m(String args[])
```

```
// vehicles v1 = new vehicles(); // not possible because abstract
// class objects cannot be created //
        Twowheeler e1 = new Twowheeler();
        e1.rodwheels();
    }
}
```

Ex: Pgm

```
abstract class Employer
{
    // Incomplete methods //
    abstract public void salarybankacc();
    abstract public void projects();
    abstract public void trainingCenter();
    abstract public void dept();
    // complet method //
    public void pfDeductions()
    {
        S o p("Staudard deduction policies");   ②
    }
}

public abstract class Infosys extends Employer
{
    public void salarybankacc()
    {
        S o p("HDFC will process salary");   ④
    }
    public void trainingCenter()
    {
        S o p("It is in Odissa");   ⑤
    }
    public void dept()
    {
        S o p("It comprises all major of IT sector");   ①
    }
}
class BangloreDC extends Infosys
{
    public void projects()
    {
        S o p("We are focussing on serving for insurance and investment");   ③
    }
}

public class User1
{
    P S V m (String args[])
    {
        Employer e = new Employer();
        Infosys i = new Infosys();
        BangloreDC d = new BangloreDC();
        d.depts();
        d.pfDeductions();
        d.projects();
```

```
        d.salary bankacc();
        d.training centre();
    }
}
```

O/p:- com ①
②
③
④
⑤

⇒ Abstract class can have complete as well as incomplete methods.

⇒ Abstract class can have static, non static and final variables.

* Can we define a constructor in abstract class.

A) Yes we can define a constructor in abstract class.

Ex:- Pgm

```
abstract class Student
{
    String stdname;
    int stdmarks;
Super class// public student(String stdname, int stdmarks)
    {
        this.stdname = stdname;
        this.stdmarks = stdmarks;
    }
}
```

```
class Admin extends student
{
    public Admin()
    {
        Super("john", 70); // call to super()
    }
}
public class A
{
    p s v m(String args[])
    {
// It is not possible to create this obj // Student s = new student(); // Since its an abstract class
        Admin a1 = new Admin();
        S o p(a1.stdname + " " + a1.stdmarks);
    }
}
```

* Can we create the class as abstract even though it doesn't contains abstract method.

Yes, we can create in 3 cases

① If all the members of a class are static members then to access them we do not have to create an object we can call directly or through class name.

```
abstract class A
{
    static int i = 100;
    main()
    {
        SOP("In main");
        SOP(i);
        run();
    }
```

~~p s v m(String args[])~~

```
p s v run()
    sop("In run");
}}
```

2) In general we never create an object of super class we can declare super class we can declare super class as abstract class.

```
abstract class A
{
}
class B extends A
{
}
```

3) If you don't want anyone to create object of your class you can simply declare it as abstract (refer previous student ex).

20-04-2020

Q) Can we declare abstract class as final

A) No, we cannot declare an abstract class as final because if it is final, it cannot be extended & inherited.

Q) Can we declare abstract method as final?

A) No, because if it is final it cannot be overridden and we must have to override abstract method to complete it.

Q) Can we declare abstract method as static?

A) No, because if it is static it cannot be overridden since we must have to override abstract method to complete it.

Q) Can we achieve multiple inheritence through abstract class?

A) No, because one class cannot extend more than one abstract class

Conclusion :

Since abstract class contains complete as well as incomplete methods

We can able to achieve partial abstraction ie, 0 to 100% abstraction.

=> Therefore to achieve 100% abstraction we go for interface.

INTERFACE :

Prg def :

An interface is a type definition block (Block of codes) whose type convention is same as that of class.

```
Interface Animal
{
    void eat();
    void color();
    void species()
}
```

Tech def:-

⇒ From client point of view an interface is defined as set of services what his expecting from client.

⇒ From service provider point of view an interface is defined as set of services he offered.

⇒ Therefore interface is contract b/w client and service provider.

```
interface ATM
{
    void withdrawal();
    void deposit();
    void mst();
}
```

⇒ An interface has to be represented with interface keyword.

⇒ By default all the methods of interface are public and abstract whether we write or don't write.

```
interface A
{
void run();
public void fly();
abstract public void cry();
}
```

⟺

```
Interface A
{
public abstract void run();
public abstract void fly();
public abstract void cry();
}
```

⇒ By default all the variables of interface are public static and final. wether we write or don't write.

```
interface A
{
    int i=9; // public static final int i=9;
    static String s="java"; // " " " String s="java";
    final int k=88; // " " " int k=88;
    public static final float f=99.9f;
}
```

Example: Prog

```
interface A
{ int A;
  S o p(i);
}
```

⇒ CTE because all vars are final and it's mandatory to initialise final during declaration.

⇒ Constructors are not allowed in interface because there is no non static variable by default all variables are public and final.

```
interface A
{
  int abc;
  public A() // CTE because constructors are not allowed.
}
```

Q) Can we create an obj of interface?

A) No, because all methods are by defaultly abstract. But we can create a reference of interface.

Q) Can we initialise an interface?
(Q)
Can we create obj of interface?

A) No, because all methods are by default abstract. But we can create reference of interface.

Implements

Implements is a keyword. It will be used if any class wants to form a relationship with interface.

```
interface car
{
}
class Audi implements car
{
}
```

class ——— class ——→ extends

interface ——— interface ——→ extends

class ——— interface ——→ implements

interface ——— class ——→ Not possible

⇒ A class can extend only one class at a time.

⇒ An interface can extend multiple interface at a time.

⇒ A class can implement multiple interface at a time.

Ex: Pgm

```
interface RBI
{
  void deposits(); // public abstract void deposits();
  void withdrawol();
  void mainbal();
}
class SBI11 implements RBI
{
 public void deposits()
 {
    S o p("Deposit in SBI");
 }
public void withdrawol
{
   S o p("Withdraw an amount from SBI");
}
 public void mainbal
 {
    S o p("Main bal should be 3000/-");
}}

   public class USER11
   {
      p s v m(String args[])
      {
         SBI11 o SI = new SBI11();
```

```
        S1.deposits();
        S1.maiubal();
        S1.withdrawal();
    }}
```

21 Apr 2020

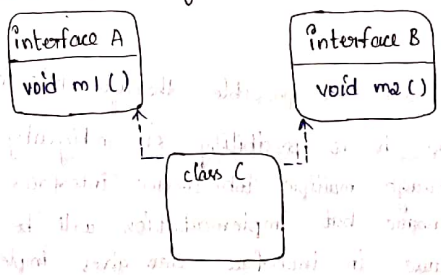
Ex: 2 Pgm

```
interface RBI
{
  void deposits();
  void withdrawal();
  void maiubal();
}
abstract class SBI11 implements RBI
{
  public void deposits()
  {
    S.o.p("Deposits in RBI");
  }
  public void withdrawal()
  {
    S.o.p("Withdrawal an amount fonom RBI");
  }}
class SBIHYD extends SBI11
{
    minbal
  public void xxxxxx()
  {
```

```
public
public class user11
{
  public static void main(Str...)
  {

    SBIHYD S1 = new SBIHYD();
      S1.deposits();
      S1.minbal();
      S1.withdrawal();
  }}
```

Ex pgm

One class implementing two interfaces at a time (also called multiple inheritance)




Pgm

```
interface A1
{
  void m1();
}interface B
{
  void m2();
```

```
class C implements A, B
{
  public void m1()
  {
    Sop("    ");
  }
  public void m2()
  {
    Sop("    ");
  }
}
public class Main
{
  p s v m(    )
  {
    C cl = new C();
    cl.m1();
    cl.m2();
  }}
```

Why multiple inheritance is possible through interface?

A) In Interface there is no possibility of ambiguity problem because even though multiple ~~inheritance~~ Interfaces contains same method name but implementation will be given only once because In 'interface' class gives implementation and it won't implement multiple methods with same name and same argument in single class.

Ex:-

```
Interface A1
{
  void m1();
}
```

```
Interface B
{
  void m1();
}
class C implements A, B
{
  public void m1()
  {
    Sop("    ");
  }}
public class Main
{
  p s v m(String args[])
  {
    C cl = new C();
  }} cl.m1();
```

⇒ Since Interface does not contain constructors there is no possibility of constructor chaining in Interface.

⇒ Therefore when there is no ambiguity problem and constructor chaining problem we can achieve multiple inheritance through Interface.

| ABSTRACT Class | INTERFACE |
|---|---|
| * abstract class A { } | * interface A { } |
| * Class keyword must be proceeded with abstract keyword. | * interface name must be proceeded with interface keyword. |
| * It have complete as well as incomplete methods. | * These are by default only abstract. |
| * Here we have public, private, protected, static, non static and non final variables | * All variables are by default static public and final |
| * Constructors are allowed in abstract class | * Not allowed. |
| * Using abstract class we can achieve 0 - 100% abstraction. | * We can achieve 100% abstraction. |
| * A class can extend only one abstract class at a time<br>abstract class A {}<br>abstract class B {}<br>class C extends either (A & B) | * Here implements multiple interfaces at a time.<br>interface A { }<br>interface B { }<br>class C implements A, B |

* Using abstract class we cannot achieve multiple inheritence.

* We go for abstract class when we know partial implementation

Ex
```
abstract class A
{
  public void getsal()
  {
    SOP( );
  }
  abstract p v getincentives();
}
```

* Using interface we can achieve multiple inheritence.

* We go for this when we know only specification but not implementation.

Ex
```
interface A
{
  void getsal();
  void getincentives();
}
```

22 Apr 2020 Ex Prgm

```
package DEMO.IN;
interface RBI
{
  void deposits;
  void withdrawal;
public void
  public void default aadharlink()
{
    S o p ("Link your aadhar");
```

```
}
  public default void mainbal ()
  {
    S o p("check with ur concerned banks");
  }
}
class SBI implements RBI
{
  public void withdrawal ()
  {
    S o p("Withdrawal from SBI");
  }
  public void deposits ()
  {
    S o p ("deposits in SBI");
  }
  public void aadharlink ()
  {
    S o p ("Please check aadhar");
  }
}

class AXIS implements RBI
{
  public void deposits ()
  {
    Sop ("Deposits in Axis");
  }
```

```
  public void withdrawal ()
  {
    S o p ("Withdrawal from axis");
  }
}
public class Bank
{
  p s v m (String args[])
  {
    SBI s1 = new SBI ();
      s1. withdrawal (); s1. deposits (); s1. aadharlink();
      s2. minbal ();
    AXIS s2 = new AXIS ();
      s2. withdrawal (); s2. deposits (); s2. aadharlink();
      s2. minbal ();
  }
}
```

O/P:-

```
Withdrawal from SBI
deposits in SBI
link your aadhar
check with ur concerned bank
Withdrawal from axis
deposits in axis
link your aadhar
check with ur concerned banks
```

## Default methods from 1.7v of java :-

⇒ From 1.7v of java, the oracle developers of java has added a new feature to interface i.e., default method.

⇒ In interface we can not only write abstract method but we can define default method also.

⇒ Default method is complete method.

⇒ The advantage of default method is it does not force classes to complete it.

## Generalisation AND Specialisation :-

| | | | | |
|---|---|---|---|---|
| METHOD DEF | run() | run(int i) | run(char ch, int i) | run(Employee, s) |
| METHOD CALL | run() | run(100) | run('A', 600) | run(new employee) |

### Specialisation :-

The process of developing a method which handles only one type of object such methods are called as special methods and the process called specialisation.

### Ex:- Prgm

```
public class genRspec
{
  p s v m( )
  {
    dogdetails(new Dog());
    catdetails(new cat());
  }
  public static void dogdetails(Dog d) // Dog d=new Dog();
  {
      d.eat(); d.makesound();
  }
  public static void catdetails(cat c) // Cat c=new cat();
  {
      c.eat(); c.makesound();
  }
}
interface Animal
{
  void eat();
  void makesound();
}
class Dog implements Animal
{
  public void eat()
  {
    S o p("Dog eats biscuits");
  }
```

```
    public void makesound ()
    {
        S o p (" Bow Bow ");
    }
}
class cat implements Animal
{
    public void eat ()
    {
        S o p (" cat eats chicken ");
    }
    public void makesound ()
    {
        S o p (" meow meow ");
    }
}
```

```
    public void makesound()
    {
        S o p("Bow Bow");
    }
}
class cat implements Animal
{
    public void eat()
    {
        S o p("cat eats chicken");
    }
    public void makesound()
    {
        S o p("meow meow");
    }
}
```

23 Apr 2020

Generalisation :-

⇒ The main advantage of specialisation is methods will handle one type of object.

⇒ For an instance in above example if we add monkey class we need to add another method which handles monkey object.

i.e.,
```
public void monkeydetails(monkey m)
{
    m.eat();
    m.makesound();
}
```

NOTE:-
⇒ To overcome this, we go for generalisation.

Generalisation :-

The process of developing a method which can handles any type of object, is called as general method and the process is called generalisation.

Ex:-
```
interface Animal
{
    void eat();
    void makesound();
}
class Dog implements Animal
{
    public void eat()
    {
        SOP("Dog eats biscuits");
    }
    public void makesound()
    {
        SOP("Bow Bow");
    }
}
class cat implements Animal
{
    public void eat()
    {
        SOP("cat eats chicken");
    }
    public void makesound()
    {
        SOP("meow meow");
    }
}
class monkey implements Animal
{
    public void eat()
    {
        SOP("monkey eats fourit");
    }
    public void makesound()
    {
        SOP("woowoo");
    }
}
public class genRspec {
    P S V m( ) {
animal deatails(new Dog());
animaldeatails(new cat());
animaldetails(new monkey());
}
public static void animaldetails(Animal a)
    //Animal a = new Dog()//
    // Animal a = new cat()//
    //Animal a = new Monkey()//
    {
        a.eat();
        a.makesound();
    }
}
```

24-Apr 2020

## EXCEPTION HANDLING :-

An exception is an unexpected & unwanted condition which disturbs our noml flow of execution.

Ex:- Lockdown exception
Coronovirus exception

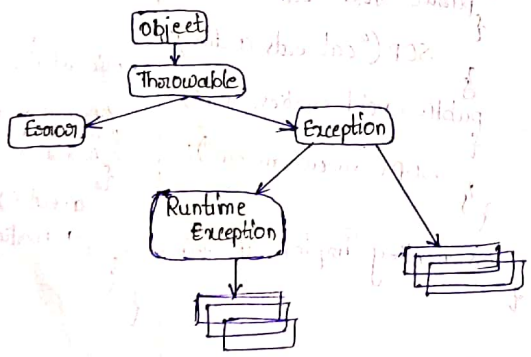
=> Once exception occured remaining part of program will not be executed.

=> So it is our responsibity to handle the exception.

=> Exception handling doesn't means we are resolving an exception it is just like providing an alternate solution so that even though exception happens our program should work properly.

## Exception Hierarchy :-

Exception Hierarchy




=> Object class is a super class to all the predefine and userdefine class of java.

=> Throwable class is a super class to Exception class and error class.

=> Exception class is a superclass to runtime exception class and other exception classes.

=> Depending on hierarchy exceptions are divided into two types.

① Checked exception
② Unchecked exception

### 1) Checked exception :-

Exceptions which are checked during compile time by compiler such type of exceptions are checked exception.

(OR)

Exception classes which are directly inheriting exception class except runtime exceptions is called as checked exceptions.

=> Checked exceptions are also called compile time exception.

Examples :-

* Interrupted Exception
* ClassNotFoundException
* SQL Exception
* FileNotFoundException

2) Unchecked exception :-

⟹ Exceptions which are checked during Runtime, such type of exceptions are unchecked exceptions.

⟹ Incase of unchecked exception our program will atleast compiles successfully.

⟹ Unchecked exceptions are also called as Runtime exceptions.

⟹ Runtime exception is super class to all our exception classes.

Examples :-

* Airthematic Exception
* ArrayIndexOut of Bounds Exception
* NullPointer Exception
* Class Cast Exception

Error :-

⟹ An error is an irrecoverable condition ie, if error occured it is not under programmer's control to get over it.

Ex: If we develop any program whose size is 4gb but our system storage is 3gb so such condition is not in programmers control and such situation is refer as error

Examples :-

* StackOverFlow Error
* VirtualMemory Error
* 404 page not-found

27 Apr 2020

| ERROR | EXCEPTION |
|---|---|
| 1) An error is caused due to lack of system resources | 1) An exception is caused because of some problem in code. |
| 2) An error is irrecoverable i.e., it is a critical condition cannot be handled by code of program. | 2) An exception is recoverable i.e., we can have some alternate code to handle exception. |
| 3) There are no ways to handle error. | 3) We can handle exception by means of try and catch block. |
| 4) As error is detected program is terminated abnormally. | 4) As exception is occured it can be thrown and caught by catch block. |
| 5) There is no classification for error | 5) Exceptions are classified as checked and unchecked. |
| 6) Errors are defined in java.lang.Error package. | 6) Exceptions are define in java.lang.Exception package. |

## What happens when an exception occured?

```
class Sample
{ Public static void div() {
    int a=10, b=0, c;
    c= a/b; // 10/0
    SOP(c);
}
  public static void main(String args[])
  {
      div();
  }
}
```

⇒ In the above program execution begins from main method and it calls to div().

⇒ In div() there is an unexpected statement ie., c= 10/0. When this statement is encountered div() creates an exception object, which includes

NAME:
description:
location:

and handover it to JVM, Now JVM will check is there any exception handling code present in div() since there is no exception handling code present in div(). It checks with caller method in above program ie., main().

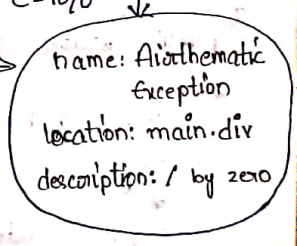
So it checks in main() is there any exception handling code present or not. Since there is no exception handling code present in main() also JVM will calls to default exception handler ie., printStackTrace() and that default exception handler provides the complete information of exception.

ie., Exception occured in thread main "java.lang.ArithmeticException: / by zero" at Sample.div(Sample.java:13)

So if we want the above program to be executed we have to handle the exception.

⇒ It can be handled by using try and catch block.

Exception object creation — c=10/0




### try and catch block :-

Exception can be handled by using try and catch block.

```
try
{ // risky statements which cause exception //
}
catch(Exceptionname refvar)
{
   // Alternate sol
}
```

→ try block is used to keep a code which causes an exception.

→ Once exception occured in try block remaining part of code will not be executed, so we should keep only statements which causes exception under try block.

→ Once exception occured in try block JVM will immediately make a search for corresponding catch blocks.

→ Catch block is where we will caught the exception.

⇒ Under catch block we poovide some statements which works as alternate solution for exception.

Ex: pogm

```
class Sample
{
  public static void main(String args[])
  {
    int a=10, b=0, c;
    try
    {
    // put a code which causes execution //
    c=a/b; // An exception object gets occured
              by main() and handover to JVM
    }
    catch (ArithmaticException e) // e is ref which
                holds ArithmaticExp obj
    {
      SOP("Exception is being noticed");
    }}}
```

```
class Sample
{
  P. S V m(String args[])
  {
    int a=10, b=0, c;
    try
    {
    // put a code which causes exception //
    c=a/b; // An exception obj gets created
              by main() and handover to JVM
      SOP("under try block"); // unexpected statme
    }
    // CTE because we cannot write any b/w try
       and catch block //:
       SOP("B/w try and catch block");
    catch (ArithmeticException e) // e is ref which
                holds ArithExp obj
    {
      SOP("exception is being noticed");
    }}}
```

Q) Can we write any statement b/w try and catch block?

A) No we cannot write any statement. Immediately after try block there should be "catch" or "finally" block only.

Q) If we dont know exception type, what type should we mention in catch block?

A) We can mention it as a Exception Class type. Because Exception is a super class to all the class and during upcasting we

studied that superclass can hold reference of subclass.

Ex: Exception e = new Arithmetic Exception ()

28 Apr 2020

NOTE:-

* Single program can have multiple entry statements i.e., one try block can have multiple exception statement.
* In that case only one exception will get caught because one try block can define one exception at a time.
* If we want to handle multiple exceptions it is always recommended to define seperate try and catch blocks.

```
class Sample
{
  p s v m (String args[])
  {
     int a=10, b=0, c;
     int d[] = new int[3];
    try
    {
    d[3]=55; // beyond declared size so an
        exception obj gets created.
     c= a/b; // unexecuted
    }
    catch (Arithmetic Exception e)
    {
        SOP("Exception is caught");
    }
```

```
    catch (ArrayIndexOutofBoundsException f)
    {
        SOP("Exception caught for array");
    }}}
```

O/P:-

Exception caught for array

⇒ This above program is always recommended to write in below manner.

```
class Sample
{
  p s v m (      )
  {
    int a=10, b=0, c;
    int d[] = new int[3];
    try
    {
      d[3]=55; // beyond declared size
    }
    catch (ArrayIndexOutofBounds Exception f)
    {
        SOP("Exception caught for an array");  ①
    }
    try
    {
      c= a/b;
    }
    catch (Arithmetic Exception e)
    {
        SOP("Exception is caught");  ②
    }}
```

O/P:- ① - - - -
② - - - -

## Single try with multiple catch blocks :-

It is allowed to write single try with multiple catch blocks but it should be most specific to most general.

```
class Sample
{
    p s v m ( )
    {
        int a=10, b=0, c;
        int d[] = new int[3];
        try
        {
            d[3]=55;
        }
// most specific catch block
catch (ArrayIndexOutofBoundsException f)
                    //AIOBE f=new AIOBE()
        {
            SOP ("Exception caught for array");
        }
// general check block
    catch (Exception e) //Exception e = new AIOBE()
    {
        SOP ("Exception is caught");
    }
```

⇒ for the above program we cannot define catch blocks as ~~specific~~ most general to most specific if we did we will get compile time error.

```
// general catch block
    catch (Exception e)
    {
        SOP ("Exception is caught");
    }
// most specific catch block
catch (ArrayIndexOutofBounds Exception f)
                    // AIOBE f=new AIOBE()
    {
        SOP ("Exception caught for array");
    }
```

we will get error as exception ArrayIndexOutofBounds exception has already been caught catch (ArrayIndexOutofBounds Exception f)
//AIOBE f= new AIOBE()

## finally Block :-

* It is a block which will get executed irrespective of
1) exception occured or not
2) exception occured and handled.
3) exception occured and not handled.

* Basically it is used to keep an important code which should not be skip like closing of data base connection or closing of opened file etc.

ex:-

```
class Sample
{
  p s v m( )
  {
    int a=10, b=0, c;
    int d[]= new int[3];
   try
   {
    d[3]=55;
   }
  catch (ArrayOfIndexOutofBoundsException f)
                    //AIOBE f= new AIOBE()
    {
      SOP("Exception caught for array");
    }
  finally
    {
      SOP("final block");
    }
  }}
```

29 Apr 2020

⇒ following combinations are valid for try, catch and finally block.

① try
{}
catch()
{}

② try
{}
finally
{}

③ try
{}
catch
{}
finally
{}

THROWS Keyword :-

```
class Test
{
 p s v m( )
 {
    SOP("Go to Sleep");
   Thread.sleep(1000);
    SOP("Awake");
 }}
```

⇒ In the above program we have called sleep() of thread class.

⇒ Means we are making current thread ie. main() to go in sleeping state which means once main() should stop execution once sleep() is invoke.

⇒ But when it is in sleeping state there will be a chance of other threads trying to interrupt main thread from sleeping.

⇒ So above program we will get exception as

Test.java:6:
error: unreported exception InterruptedException; must be caught or declared to be thrown.

Conclusion :-

1. In case of checked exception user have two options

1) Caught the exception: use try and catch block
2) Declare the exception: use throws keyword.

* Throws keyword is used to declare a checked exception.
* Throws keyword is used with method declaration.

p v fly() throws ExceptionName

* When we use throws keyword we are indicating current method will not handle exception rather calling method or its caller will handle the exception.

```
Class Test
{
 p s v m(  ) throws InterruptedException
 {
    SOP("Go to sleep");
    Thread.sleep(1000);
    SOP("Awake");
 }}
```

=> In the above program main method is declared an exception using throws keyword that means it is telling that I won't handle exception rather it is responsibility of my caller to handle it.

=> So in the above program caller of main() is JVM.

Ex:- Exception Propagation:-

```
Class Test
{
 p s v m(  )
 {
    SOP("Main thread Go to Sleep");
    Thread.sleep(1000);
    try
    {
     a1();
    }
    catch(InterruptedException e)
    {
       SOP("catch block");
    }
 public static void a1() throws InterruptedException
 {
    Thread.sleep(1000);
    SOP("a1 Awake");
    a2();
 }
 public static void a2()throws InterruptedException
 {
    Thread.sleep(1000);
    SOP(1000);
    SOP("a2 Awake");
 }}
```

=> In the above program, a2() is telling that I won't handle exception, I will declare it as it is. My caller ie., a1 should handle it.

=> a1() is telling that I won't handle exception I will declare as it is.
My caller ie, main() should handle it.

⇒ So this process where exception is passed from one method to another method this process is known as exception propagation.

30 Apr 2020

throw Keyword :-

```
class Demo
{
p s v m( )
{
    SOP(10/0);
}}
```

In the above program main method creates an exception object and handover it to JVM since there is no exception handling code JVM will handover that object to default exception handler and it prints exception message as

Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Demo.main(Demo.java:5)

⇒ But if we want we can create our own exception object by using throw keyword.

⇒ Throw keyword is used to explicitly creating an exception object.

⇒ Syntax : throw new Exceptiontype (description of exception);

⇒ throw keyword is mainly used for user-define exceptions.

Ex : 1

```
class Demo
{
  p s v m (String args[])
  {
    throw new ArithmeticException ("My exception");
  }}
```

Exception in thread main

Ex : 2

```
class Demo
{
p s v check
        m (int age)
{
  if (age<18)
  {
    throw new ArithmeticException ("He is small");
  }
  else {
    SOP ("Go out with gf ");
  }}
p s v m( )
{
  check (15);
}}
```

O/p :- Exception in thread "main" java.lang.ArithmeticException
:He is small
at Demo.check (Demo.java:7)
at Demo.main (Demo.java:15)

## User define Exception (or) Customized Exception :-

⇒ When predifine exceptions are not fulfills our requirement we will go for user define ie, we can create our exceptions.

⇒ Such type of exceptions are called as user defined exceptions.

## Rules for creating user define Exception :-

⇒ Create our exception class and that class should be extending either throwable or exception or Runtime Exception classes. (Preferable to extend RuntimeException).

⇒ Define constructor whenever requires.

⇒ Throw exception as per our own requirement.

Ex:-

```
package helper;
class AgeGapException extends RuntimeException
{
public AgeGapException (String msg)
{
    super(msg); // It calls default exception handler ie, printStackTrace( )
    // SOP(msg);
}}
  public class Demo
  {
    p s v m(      )
    {
      int bage=10, gage=27;
      if (bage<15 && gage>=27)
      {
    throw new AgeGapException ("He is small");
      }
      else
      {
        SOP ("Go out with gf");
      }}
```

O/P:-

```
Exception in thread "main" helper.AgeGapException:
    He is small   at helper.Demo.main(Demo.java:16)
```

Explanation:

Execution started from main method, under that a condition is given. If that condition satisfied then an we create an explicit exception obj with details as.

Name: AgeGapException
description: He is small
Location : Demo.main

⇒ Our exception objects created it calls to AgeGapException constructor and pass msg as argument.

⇒ Under that constructor we have written, super(msg) ---> because it calls printStackTrace() which is present in throwable class whose job is to give the details of exception msg message includes packagename.classname.Exception details

Assignment :-

⇒Create an userdefine exception class as MinBal-exception inherit RunTimeException.

⇒define constructor and write call to super(msg)

Create class

* define main()
* Create obj of scanner class (to take inputs)
* Input 1 ⟶ double ⟶ amtdeposit
* Input 2 ⟶ double/int ⟶ amtwithdrawl
* Check condition if(amtwithdrawl > amtdeposit || amtwithdrawl == amtdeposit)

throw new MinBalException ("Ur Balance exceeds limit because ur current bal: "+amtdeposit)

else

print message as "collect ur amount"

Pogm:-

```
Package helper;
import java.util.scanner;
{
 public class Demo
 {
   p s v m (      )
   {
    scanner s = new scanner(system.in);
 double amtdeposits = s.nextDouble();
     SOP ("Available bal in account: "+amtdeposit);
 double amtwith= s.nextDouble ();
 if (amtwith > amtdeposit || amtwith == amtdeposit)
 {
 throw new MinBalException (" Ur exceeds limit
        because current bal is"+ amtdeposit)
 }
 else
 {
    SOP ("Please collect ur amt");
 }}}
class MinBalException extends RuntimeException
 {
    public MinBalException (string msg)
     {
        super (msg);
     }}
```

Using try & catch blocks

```
import java.util.Scanner;
public class Sample
{
  p  s  v  m (      )
 {
   Scanner s = new scanner (System.in);
 double amtdeposit = s.nextdouble ();
     SOP ("Available bal :" + amtdeposit);
   double amtwith= s.nextDouble ();
  try
 { if (amtwith > amtdeposit || amtwith == amtdeposit)
   {
throw new MinBalException ("Ur exceeds limit
    because current bal is: " + amtdeposit)
```

```
}
else
{
    SOP("please collect ur amount");
}}
catch(MinBalException e)
{
    SOP("Caught the exception");
}}
class MinBalException extends RunTimeException
{
    public MinBalException(String msg)
    {
        super(msg);
    }}
```

04 May 2020

## THROW

1. It is used to create exception obj explicitly.
2. It is used inside the method.
3. Ex: throw new Exception-name(Expdescription)
   throw new Airthmetic-Exception("exception");
4. It is mainly used for RunTimeException/Userdefine.
5. Using "throw" keyword we can throw only one exception at a time.
6. throw new MinBalException("zero");

## THROWS

1. It is used to declare the exception.
2. It is used with method declaration.
3. Ex: method declaration throws Exceptionname
   public void f1()() throws InterruptedException
4. It is mainly used for checked exception.
5. Using "throws" keyword we can declare multiple exceptions at a time.
6. pv check() throws Interrupted-Exception, SQL Exception.

## Wrapper classes :-

⇒ In order to represent primitive datatype as an object we use data type.

⇒ For every primitive data type there is a corresponding wrapper class available in java.

⇒ Since java is 100% obj oriented we have to represent everything as object.

⇒ Therefore to represent primitive datatype as object we use wrapper classes.

| datatype | Wrapper classes |
|---|---|
| byte | Byte |
| boolean | Boolean |
| char | character |
| short | Short |
| float | Float |
| int | Integer |
| double | Double |
| long | long |

NOTE

* All wrapper classes are final classes.
* All wrapper classes are present in java.lang package.

Examples :-

```
int a = 100;
Integer a = new Integer(100);   (or) Integer a = 100;

double d = 22.45;
Double d = new Double(220.45);   (or) Double d = 445.5;
```

```
int a = 100;
Integer a = new Integer(100);
Integer a = 100;

double d = 22.45;
Double d =
```

## Constructors :-

```
Integer a = new Integer(int);
Integer a1 = new Integer(String);
byte b1 = new Byte(byte);
byte b2 = new Byte(String);
```

For all wrapper classes there are two constructors but for float we have three constructors.

double, float & String

```
Float f1 = new Float(double);
Float f2 = new Float(String);
Float f3 = new Float(float);
```

⇒ In all wrapper classes toString() is over-ridden and it prints value of object not complete info of object.

Ex:
```
Integer a = new Integer(100);
SOP(a.toString()); // 100
SOP(a); // 100
```

⇒ In wrapper class, String value is automatically converted into Integer format.

for ex:
```
Integer i = new Integer("100");
SOP(100+i); // 100+100 ⟶ 200
Float f = new Float("200");
SOP(200.2f + 200); // 400.2
Double d = new Double("bcdef");
SOP(233.3+d); // NumberFormat Excep...
```

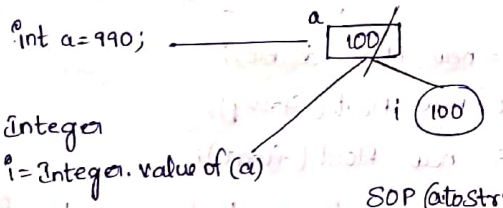
⇒ Using wrapper classes we can perform boxing and unboxing.

**Boxing :-**

The process of converting primitive type into wrapperclass is called as boxing.

Boxing can be performed by using valueOf()

Ex:
```
int a = 990;
Integer i = Integer.valueOf(a);
```




```
SOP(a.toString()); // CTE
SOP(i.toString()); // 100
```

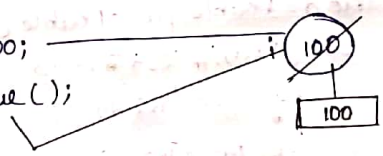
**UnBoxing :-**

The process of converting wrapperclass into primitive type is called as UnBoxing.

It is performed by using xxxValue()

(xxx ⟶ primitivedatatype)

Ex: 1)
```
Integer i = 100;
int a = i.intValue();
```




2)
```
Double d = 22.23;
double a2 = d.doubleValue();
```

**NOTE :-**

From 1.5v of java we can perform boxing and unboxing directly so they are referred as autounboxing and autoboxing respectively.

Ex:
```
int i = 100;
Integer a = i; // Autoboxing
Integer i = 122;
int k = i; // Autounboxing
```

05 May 2020

**parse() :-**

It is a method of wrapper class, it converts the storing data into double or float.

But will work only if storing holds proper data otherwise we will get CTE.

Ex:-
```
String s="123", String s1 = "223.33";
int i = Integer.ParseInt(s);// converts string into integer
double d = Double.parseDouble(s1);// Converts string to double
    SOP(i+d); // 123+223.33
int i = Integer.parseInt(s);// converts string into integer
double d = Double.parseDouble(s1);// String into double
    SOP(i+d);// 123+223.33
    String s2="122.2f";
int k=Integer.parseInt(s2);// NumberFormatException- because we are converting float value as string into integer
    SOP(k);
}
}
```

06 May 2020

## COLLECTION FRAMEWORKS :-

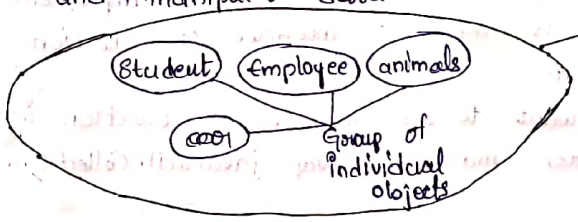
| ARRAYS | COLLECTIONS |
|---|---|
| ① Arrays are used to store collection of homogeneous data (similar). | ① Collections is used to store homogeneous and heterogeneous data. |
| ② Arrays are fixed in size. ie, int a[]=new int[3] | ② Collections are numerous. |
| ③ Arrays can deal with primitive as well as wrapper classes. int a[]; Integer a[]; | ③ Purely deals with (objects) wrapper classes ArrayList<int> a1; (Invalid) ArrayList<Integer> a1; (valid) |
| ④ It does not have any underlying data structure | ④ Every class of collection have data structure. |
| ⑤ Arrays does not contain predefine methods (add, sorting, removing, replacing) which makes manipulation of data difficult. | ⑤ In collections 80% of support is by predefine API's. |
| ⑥ We should use arrays when we already knows the elements in advance. | ⑥ We should go for collection when we don't know the elements in advance. |
| ⑦ Memory wise array is not preferred. | ⑦ Memory wise collections is preferred. |
| ⑧ Preferred for performance wise. | ⑧ Performance wise collection is not preferred. |

### Collection:-

⇒ Collection is nothing but a group of objects represents as a single unit.

⇒ Its main purpose is to store huge amount of data.

⇒ It provides multiples API (methods) to store and manipulate data.



## Collections frameWorks :-

It provides the group of classes and interfaces which is used to store multiple objects as single unit.

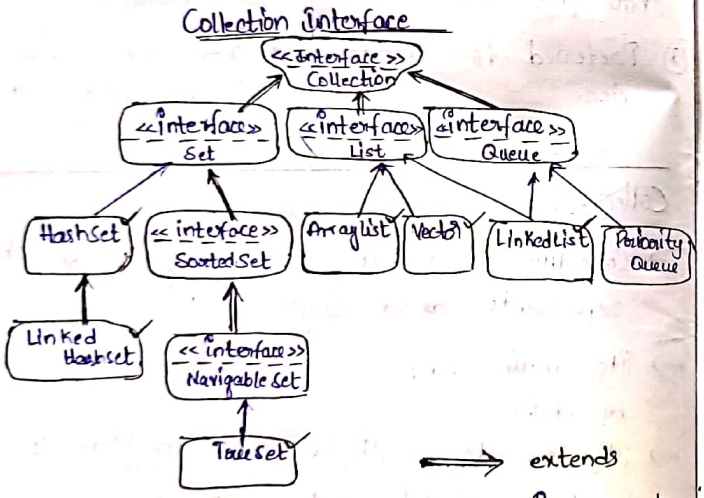
CollectionsFrameWorks ——→ Concept

Collection ——→ Root Interface

Collections ——→ class

⇒ All the classes and interfaces of collections-framework are present in java.util package

Collection Hierarchy :-




⇒ Collection is the root interface of Collection-FrameWork.

⇒ If we want to see methods of collection interface enter cmd as javap java.util.Collection.

⇒ Collection interface is extended by 3 different interfaces.

1) List < Interface >
2) Set < Interface >
3) Queue < Interface >

⇒ Learning about collection means learning about classes and its characteristics

Differ class of collection are :-

1) ArrayList
2) LinkedList
3) Vector
   implements List Interface
4) PriorityQueue
5) LinkedList
   implements Queue Interface
6) TreeSet
7) HashSet
8) LinkedHashSet
   implements Set Interface

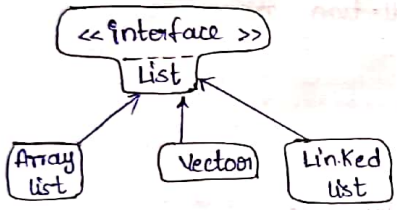
⇒ We have some characteristics to study for every class in collections. they are

## Characteristics :-

* Version
* homogeneous (or) heterogeneous
* Null object is possible or not.
* duplicates allowed or not.
* Insertion order is preserve or not
* data structure
* cursors

07 May 2020

## List Interface :-




List <interface> (1.2 V)

(1.2 V) ArrayList — LinkedList (1.2) <classes>

Vector <class> (1.0 V)

⇒ List is the interface which extends collection Interface.

⇒ List is implemented by 3 different classes.

1) ArrayList
2) LinkedList
3) Vector (class vector extends stack implements list)

⇒ List follows index based process.

⇒ List allows heterogeneous objects.

⇒ List allows null objects.

## Useful methods :-

* add (Object) :- Used to add objects.
* addAll (collection) :- Used to copy one collection objects into another.
* remove (Object) :- Used to remove objects
* remove (index) :- Used to remove the object at given index.
* set (index, obj) :- it replaces the old obj with new obj at given index.
* get (index) :- Retrives the obj at given index.
* isEmpty () :- Check whether collections empty or not.
* contains (obj) :- check whether obj is present or not.
* containsAll (collection) :- check whether content of one collection is present in another collection or not.
* removeAll (collection) :- Removes whole objects of collection.
* Size () :- Provides size of collection (always calculates from 1)
* Collections.sort() :- sort the collection in ascending order (in case of chars as per Unicode order)
* Collections.reverse () :- Sort in reverse way

## Array List :-

⇒ It is a class which implements list interface.

→ For checking methods:

javap java.util.Arraylist

## Characteristics of Arraylist :-

⇒ Arraylist stores heterogeneous data.

⇒ It is possible to add NULL obj's in arraylist.

⇒ It allows duplicate objects.

⇒ In Arraylist insertion order is preserved ie, the way we added objects the same way it will be printed.

⇒ It follows data structure as growable size array.

⇒ Iterator and ListIterator cursors are used.

Ex:- Pgm

```
import java.util.Arraylist (or) java.util.* (or) java.util.collection.Array
public class first
{
 p s v m( )
 {
  // creation of Arraylist //
  Arraylist al = new Arraylist();
  // adding obj's in arraylist
  al.add("java");
  al.add("SQL");
  al.add("Apti");
  // befoor 1.5v
  Integer i = new Integer(100);
  al.add(i);
  // from 1.5v autoboxing is performed directly
  al.add(300); // al.add(Integer.valueOf(10))
  al.add('A'); // al.add(Character.valueOf(A))
  SOP(al); // al.toString()
  SOP(al.toString());
 }
}
```

O/P :- [java, SQL, Apti, 100, 300, A]

[java, SQL, Apti, 100, 300, A]

⇒ In collection toString() is overridden by default ie, when we call toString() it prints content of object.

⇒ In collection from 1.5v whenever we add data it is being converted into object type (Autoboxing).

08 May 2020

```
import java.util.Arraylist
public class second
{
p s v m( )
{
 Arraylist al = new Arraylist();
  al.add("SQL"); //0
  al.add(100); //1
  al.add('A'); //2
```

```
SOP(a1);
SOP(a.size());  //3
a1.add(0, "java");
a1.add(1, "Apti");
SOP(a1.size()); //5
SOP("After adding @ partpost: " + a1);
a1.set(1, "J2EE");
a1.set(2, "Selenium");
SOP("After set() (Replacing: "+ a1);
a1.set(1, a1.get(4));
a1.set(a.get(7)); // Out of Bounds Exception
```

// Printing arraylist objects using get() and for loop

```
SOP("for LOOP --- get()");
for(int i=0; i<a1.size(); i++){
    {
        SOP(a1.get(i));
    }
}
```

O/P :

```
[SQL, 100, A]
3
5
After adding @ partpost: [java, Apti, SQL, 100, A]
After set() (Replacing): [java, J2EE, Selenium, 100, A]
A
for LOOP ---- get()
java
J2EE
Selenium
100
A
```

⇒ Arrays are Typesafe but collections are not type safe.

Ex: 1) `int a[] = new int[3];`
`a[0] = 100;`
`a[1] = "java"; // CTE`

2) `Arraylist a = new Arraylist(3)`
`a.add("java");`
`a.add(100);`
`a.add('A');`

## Generics :-

Since collections are not type safe ie., programmer can add any type of objects without any restriction for that, to overcome this drawback JAVA has given generics concept where we can create an object of specific type, if we add any other type of object than specified, we will get CTE

1) `Arraylist<Integer> a1 = new Arraylist<>();`
`a1.add("java");`
`a1.add(100);`
`a1.add('A');`

2) `Arraylist<String> a1 = new Arraylist<>();`
`a1.add("java");`
`a1.add(100);`
`a1.add('A');`

3) Arraylist a1 = new Arraylist <> ();

a1.add
a1.add
a1.add

Ex:- Pogrm

```
package Collection.com;
import java.util.*;
public class fourth
{
    p s v m ( )
    {
        Arraylist <Integer> a1 = new Arraylist<>();
        a1.add (100);
        a1.add (200);
        a1.add (300);
        // a1.add ("java"); → // [100, 200, 300]
        SOP ("Integer arraylist is :" + a1);
        SOP (a1.contains (250)); // false
// Arraylist<object> a2 = new ArrayList();
        a2.add ("java");
        a2.addAll (a1); // To copy one Arraylist into another
        SOP ("New Arraylist is :" + a2); // [Java, 100, 200, 300]
        a2.removeAll (a1); // remove all a1 obj's from only a2
        SOP (a1.isEmpty()); // false
        SOP (a2); // [java]
        SOP (a1); // [100, 200, 300]
    } SOP (a.isEmpty() + "because A1 arraylist is:" + a1);
}
```

Since obj is a superclass to all classes.

O/P:-

Integer arraylist is : [100, 200, 300]
false
New Arraylist is : [java, 100, 200, 300]
false
[java]
[100, 200, 300]
true because A1 arraylist is : []

Ex:- Pogrm

foreach syntax:

```
for (arraytype varname : arrayname)
                (or)
for (Collectiontype varname : Collectionname)
{
    SOP (varname);
}
```

Pogrm :-

```
Package collection.com;
import java.util.Arraylist;
import java.util.Collections;
public class Sortport
{
    p s v m ( )
    {
        Arraylist <String> a21 = new Arraylist<>();
        a21.add ("Apti");
        a21.add ("Z");
```

```
a21.add ("Sql");
a21.add ("sql");
Collections.sort(a21); // It's a method of collection class

SOP(a21);

Collections.reverse(a21);

SOP(a21);

for(String data : a1)
{
    SOP(data);
}
```

09 May 2020

Cursors :-

⇒ This is the special characteristics given for collection concepts.

⇒ Using cursors we can retrive the objects in collection one by one.

Two types

1) Iterator
2) ListIterator

① Iterator :-

⇒ It is an interface which is used to traverse the list in forward direction.

⇒ Basically it provides the privilige to access the objects without using index.

```
interface Iterator
{
    public boolean hasNext();
    public object next();
    public void remove();
}
```

hasNext() : It returns true if there is object available.

next() : It returns current object and move the cursor to next object.

Ex:- Pogm

```
package Collection.com;
import java.util.*;
public class std
{
   p  s  v  m(  )
   {
      Arraylist al = new ArrayList();
       al.add("John");
       al.add("Riya");
       al.add("Pooja");
       al.add("Rohan");
// for using iterator its object
  Iterator i1 = al.iterator();
// Iterator is an obj interface and its obj cannot
 be created but every classes of collections
 contains iterator() which gives object of
 iterator && interface
while(i1.hasNext())
{
      SOP(i1.next());
}
}
```

O/P:-

John
Riya
Pooja
Rahul

2) ListIterator :-

⇒ It is an interface which provides the facility to traverse the list in forward as well as backward direction.

```
interface ListIterator
{
   public boolean hasNext();
   public object next();
   public void remove();
   public boolean hasPrevious();
   public object previous();
   public void add();
}
```

hasPrevious:- Returns true if obj is available to iterate from previous direction

previous(): Points the current obj and move cursor to next obj in previous direction.

hasNext():
It returns true if obj available.

next():-
It returns current obj and move the cursor to next object.

Ex:- Pogm

```
Package collection.com;
import java.util.*;
public class Std1
{ P s v m ( )
  { ArrayList<String> al = new ArrayList<>();
      al.add ("John")
      al.add ("Riya");
      al.add ("Pooja");
      al.add ("Rahul");
  // for traversing using list iterator cursor
  ListIterator<String> l1 = al.listIterator();
    // for traversing in forward
    SOP ("--- forward direction ---");
    While (l1.hasNext())
    {
        SOP (l1.next());
    }
    // for traversing in reverse direction
    SOP ("--- Reverse direction ---");
    while (l1.hasPrevious())
    {
        SOP (l1.previous());
    }
  }
```

O/P:- --- forward direction ---
John
Riya
Pooja
Rahul

--- Reverse direction ---
Rahul
Pooja
Riya
John

Assignment :-

* Create an arraylist to add all subjects of 4th year as objects and iterate it using for loop.
* Create the Arraylist and add 7 colours as objects and traverse it using
  1) for loop
  2) foreach loop
  3) Iterator
  4) ListIterator
* Create an Arraylist and add 5 students as objects and create one more array -list and
  1) Sort it in order
  2) Print alternate objects
  3) remove even objects

12 May 2020

| Iterator | ListIterator |
|---|---|
| * It is an interface use to retrive objects in forward direction. | * It is an interface use to retrive objects in forward as well as backward direction. |
| * It is used for all collection classes. | * It is only used for classes of list interface, Arraylist, Linked list, vector. |
| * Iterator obj we will get using iterator(). | * ListIterator obj we will get by using listIterator() |
| * Using iterator interface methods we can traverse only in forward direction i.e. hasNext() and next() | * Using list iterator we methods we can traverse in both forward and backward direction. |

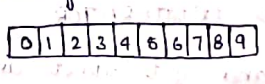
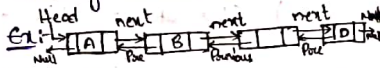
## Linked List :-

→ Intro 1.2 v ~~vector is also refereed as legacy~~
→ Heterogeneous objects are allowed.
→ Insertion order is preserved.
→ Duplicate objects are allowed.
→ Data structure is doubly linked list.
→ Cursors - ~~Linked~~ List Iterator and Iterator.

* Same process to write programs, only in place of arraylist object create object of LinkedList.

## Vector :-

→ Intro in 1.0V so vector is also refereed as legacy class.
→ Heterogeneous objects are allowed.
→ Null objects are allowed.
→ Insertion order is preserved.
→ Duplicate objects are allowed.
→ Data structure is growable array.
→ Cursors — ListIterator and Iterator.
// Programs are same, only in place of arraylist use vector.

| ARRAYLIST | LINKED LIST |
|---|---|
| * Default size of arraylist is 10 | * Default size of linked list is 0. |
| * Data structure is growable size array. | * It's data structure is doubly linked list. |
| Ex:-   | Ex:-   |
| * It is good if our operation is retrival. | * It is good if our operation is insertion and deletion. |
| * While adding data in middle arraylist is slower | * Linked list is faster when we add obj in middle. |
| * Arraylist is faster when we add obj in the end. | * It is slower when we add obj in end. |

| ARRAYLIST | VECTOR |
|---|---|
| * Default size is 10 | * Default size is 10. |
| * It is 1.2 so non legacy class. | * It is 10 so legacy class. |
| * When it reaches it saturation point size increases by 3/2+1<br>Ex:- $\frac{10*3}{2}+1 = 16$ | * When it reaches its saturation point increases by its equal size.<br>Ex:- 10 + 10 = 20 |
| * Performance is higher in Arraylist (Not thread safe) | * Performance is poor in vector (Thread safe). |

ASSIGNMENT :-

1) Create a linke list and 5 objects and print it
   A) FORLOOP
   B) FOREACH
   C) CURSORS – LIST AND LISTITERATOR

2) Create a linked list and get first and last obj added.

HINT : Use getFirst() and getLast()

3) WAP to convert
   (a) Arraylist into array
   (b) LinkedList into arraylist

② Pgm

```
package Collection_com;
import java.util.*;
public class linkedlistdemo
{
    p s v m( )
    {
        LinkedList < Object > ll = new LinkedList<>();
        ll.add("java");
        ll.add("sql");
        ll.add("Selenium");
        ll.add("sql"); // checking duplicates are allowed are not
        ll.add(null); // checking null obj is....
        SOP("LinkedList obj's are:" + ll);
        SOP("First obj:" + ll.getFirst());
        SOP("last obj:" + ll.getLast());
    }
}
```

③ Pgm: Arraylist into Array.

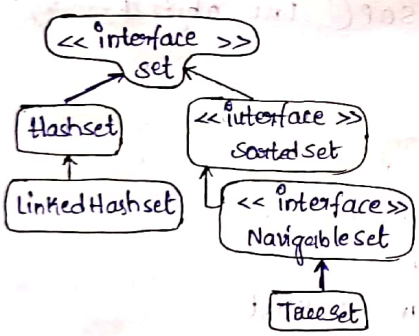
```
Package Collection.com;
import java.util.*;
public class LinkList
{
    p s v m( )
    {
        Arraylist < String > al = new ArrayList<>();
        al.add("SQL");
        al.add("Java");
        al.add("Apti");
```

```
String s[] = new String[al.size()];
for (int i=0; i<s.length(); i++)
{
    s[i] = al.get(i);
}
for (int j=0; j<s.length; j++)
{
    SOP(s[j]);
}
}
```

O/P :-

SQL
java
Api

Set Interface :-




→ Set is an interface which is implemented by 3 different classes.

① hashSet
② treeSet
③ LinkedhashSet

Hashset :-

→ It is a class which implements set interface

features :-

→ Introduced in 1.2 V.
→ Heterogeneous obj's are allowed.
→ Duplicates are not allowed. In case if we add we won't get CTE

Methods of set interface :-

javap java.util.set

public interface java.util.Set<E> extends java.util

## HashSet :-

It is a class which implements set interface

### features :-

→ Introduced in 1.2v.
→ Heterogeneous objects are allowed.
→ Duplicates are not allowed. In case if we add we won't get CTE.
→ Only one "null" object is allowed.
→ Data structure is hashtable.
→ Insertion order is preserved (depends on hashcode).
→ Set is only unidirectional so it supports only iterator.

Ex:- Pgm

```
package collection.com;
import java.util.*;
public class hashdemo
{
    p  s  v  m(  )
    {
        Hashset<Object> hl = new Hashset();
        hl.add("Rohan");
        hl.add("Riya");
        hl.add('A');
        hl.add("pooja");
        hl.add(445);
        SOP("Hashset objects are:"+hl);
        hl.add("Riya");
        hl.add(445);
        SOP("After add duplicates:"+hl);
        hl.add(null);
        hl.add(null);
        SOP("After add null objects:"+hl);
        SOP("foorward direction---");
        Iterator<object> i1 = hl.iterator();
        while(i1.hasNext())
        {
            SOP(i1.next());
        }
    }
```

// Duplicate objs are allowed.

O/P :-

```
Hashset objects are : [A, Riya, pooja, 445, Rohan]
After add duplicates : [A, Riya, Pooja, 445, Rohan]
After add nullobjects : [null, A, Riya, pooja, 445, Rohan]
foorward direction---
null
A
Riya
Pooja
445
Rohan
```

## Linked Hashset :-

It is a class which extends HashSet and implements set interface.

### features :-

→ Introduced in 1.2v.
→ Heterogeneous objects are allowed.
→ Duplicates are not allowed even if we add it will printed only once but no CTE.

→ Data structure is Linked list.
→ Insertion order is preserved.
→ Only one null obj is allowed.
→ Set is only uni directional so it supports only Iterator.

// ex pgm is same but output is as per insertion order.

Treeset :-

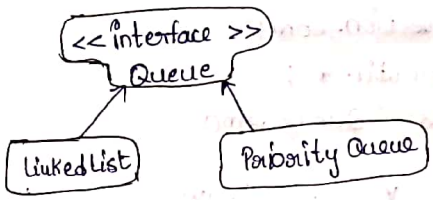
It is a class which implements set interface.

Features :-

→ Introduced in 1.2v.
→ Heterogeneous objects are not allowed, if we add we will get class cast exception.
→ Duplicates are not allowed even though if we add no CTE, it will add only once.
→ Only one null object is allowed.
→ Data structure is tree.
→ Insertion order is preserved (depends on hashcode).
→ Set is only unidirectional so it supports only iterator.

// ex pgm is same only th heterogeneous objects are not allowed and o/p will be in sorted order.

14 May 2020




Queue Interface :-

→ It is an Interface implemented by 2 different classes
1) PriorityQueue
2) LinkedList

Priority Queue :-

→ Introduced in 1.2v.
→ It allows only homogeneous objects.
→ Basically data structure of Queue is FIFO but in priority queue internal sorting happens.
→ Duplicate objects are allowed.
→ Null objects are not allowed.
→ For retriving of data they have given special methods called peek() and poll().
→ Peek() :- (i) It retrives the head element with out deleting it.
(ii) It returns null if there is no head element present.
→ Poll() :- It retrives head element (first element) and delete it.

Ex:- Prgm

```
package collection_com;
import java.util.*;
public class Queuedemo
{
    p s v m (String args[])
    {
        PriorityQueue q1 = new PriorityQueue();
        q1.add("aptitude");
        q1.add("ADP");
        q1.add("dell");
        q1.add("Banglore");
        // q1.add(334);  } Classcast exception
        // q1.add('A');  }
        SOP(q1.peek());
        SOP(q1.poll());
        SOP(q1.contains("ADP"));
        while(q1.peek() != null)
        {
            SOP(q1.poll());
        }
        SOP(q1);
    }
}
```

O/P:-
ADP
ADP
false
Banglore
aptitude
dell
[] → empty

LinkedList :-

→ It is a class which implements List as well as Queue interface

→ Queue q1 = new LinkedList();
Here q1 will exhibits behaviour of queue.

NOTE:-

→ All features are same, only but in linked-list implementing Queue interface objects will not be sorted rather insertion order is maintained.

Multithreading :-

Introduction

Multitasking:- The process of performing more than one task parallelly is called as multitasking it is of two types

i) Process based multitasking :-

The process of executing more than one task where it is independent of other.

ex:- borowsing on google — Task 1
watching youtube — Task 2
Using Excel — Task 3
Typing something on word — Task 4

15 May 2020

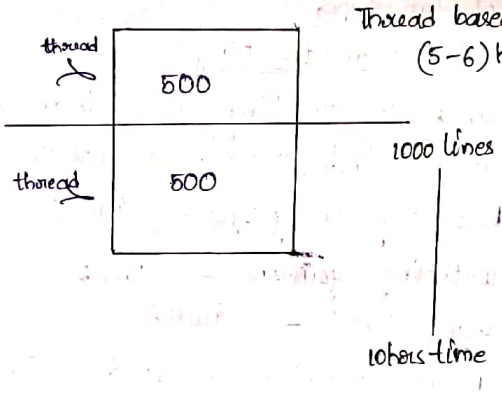
## 2) Thread based multitasking :-

The process of executing more than one task parallelly where each task is an independant part of same program.

This type of processing is called as thread based multitasking or thread programming.

Ex:- If we have a program which contains 1000 lines of code but what we observed is next 500 is independant of first 500, but still it has to wait until first 500 finishes its execution, so due to this

→ Performance is decreasing
→ execution time is increasing
→ CPU utilisation time is decreasing




Thread based tasking (5-6) hrs

→ Therefore in order to overcome the above situation we will go for thread programming.

## Thread :-

→ It is a flow of execution (or)
→ It is a small, part of an app (or)
→ It is a light weight process.

→ We will create 1-thread for first 500 lines
→ We will create 2-thread for second 500 lines and run both the thread parallelly.
→ Programs which contains multiple threads is called as multi threaded program and such process is called multi threading.

## Creation of thread :-

→ Thread can be created in two ways.
① By extending thread class
② By implementing runnable interface.

NOTE :-
→ In every program there is always one default thread called main thread. (thread main())

Ex:- Pogm (1st way)

```
package Multithreading.com;

class Mythread extends thread
{ //override run() of thread class for defining thread
    public void run()
    {
        // JOB of thread
        for(int i=0; i<=5; i++)
        {
            SOP("My thread");
        }
    }

public class User
{
    // default thread of every pogm
    p s v m( )
    {
        // JOB of main thread
        Mythread t1 = new Mythread();
        t1.start(); // creates thread by call run()
// Now two threads will under execution i.e., main and mythread
        for(int i=0; i<=5; i++)
        {
            SOP("Main thread");
        }
    }
```

O/P:-

Mythread
5 times
Main thread
5 time

(or) vise verse } depends on JVM ie., thread scheduler

Explanation:-

* In the above pogm exe starts from main method and till t1.start() there is only one thread under execution. Once t1.start() is invoke it creates a thread and calls run().
* Now two threads are under execution ie., main thread and my thread.
* So out of two which thread will execute is decided by thread scheduler.
* Thread scheduler is nothing but JVM and it is upto JVM whichever algorithm it follows and select a thread for execution.
* Since we don't know on which basis thread scheduler picks a thread for execution we cannot predict the output of the program.

Creating program by implementing Runnable interface:- (2nd way)

```
package Multithreading.com;

class Mythread implements Runnable
{ // override run() of thread class for defining
  thread
  public void run()
  {
      // JOB of thread
      for(int i=0; i<=5; i++)
      {
          SOP("My thread");
      }
```

```
}}
public class User
{
    // default thread of every pgm.
    p s v m (String args[])
    {
        // JOB of main() thread
        Mythread t1 = new Mythread();
// t1.start() // CTE ~~Job of main() thread~~
        Thread t2 = new Thread(t1);
        t2.start(); // creates thread by call run()
// after this start two threads under exe ie; main and mythread
        for(int i=0; i<=5; i++)
        {
            SOP("Main thread");
        }}
    }}
```

18 May 2020

Pogm

```
package Multithreading.com;
public class User
{
  p s v m(String args[])
  {
    Mythread1 t = new Mythread1();
    Mythread11 t1 = new Mythread11();
      t.start();
      t1.start();
      for(int i=0; i<5; i++)
      {
```

```
        SOP("pooja 1.0");
    }}
}}
class Mythread1 extends thread
{
    public void run()
    {  for(int i=0; i<5; i++)
       {  SOP("pooja 2.0");
       }}}
}
class Mythread11 extends thread
{
    public void run()
    {  for(int i=0; i<5; i++)
       {  SOP("pooja 3.0");
    }}}
```

Q) Can we restart a thread or not?

A) No, we cannot restart a thread, if we do we will get IllegalThreadStateException.

Q) Out of two ways which is preferable?

A) Second way of thread creation is preferable because when we create a thread by implementing runnable interface at same time we can extend another class also but if we create a thread by extending thread class by we cannot extend any other class.

## Life cycle of thread :-

Depends on different phases a thread will be in any of one phase:

1) New
2) Runnable / Ready
3) Running
4) Blocked
5) Terminated

**New:**
When we create obj of our thread class

```
Mythread m1 = new Mythread();
```

**Ready:**
When we call run() by using start(), but before thread schedule picks that thread for execution.

```
m1.start();
```

**Running :-**
When thread scheduler (CPU) picks thread for execution.

**Blocked :-**
If the running thread goes to sleeping state

**Terminated (or) dead :-**
When execution of run() is completed.

19 May 2020

## MAP Interface :-

→ MAP is not a subpart of collection rather itself is collection.
→ MAP is an interface. which stores the objects as a key value pair.
→ Each key, value pair is called as entry.
→ So map is also referred as collection of entry objects.
→ Whenever we want to represents objects as key value pair we go for maps.

MAP interface is implemented by 3 different classes

1) HASHMAP
2) LINKEDHASHMAP
3) TREEMAP

**Useful methods are :-**

**put (K, V)** : Used to add objects as key value pair into MAP.
**Keyset ()** : Returns all key's available in MAP.
**Values ()** : Returns all value's available in MAP's.
**ContainsKey (key) & ContainsValue (value)** :
Checks key and value is present or not.

## HashMap :-

It is a class which implements Map interface.

Features:-

→ Introduced in 1.2 v.
→ Heterogeneous data allowed.
→ Datastructure is hashtable.
→ Duplicate keys are not allowed but values can be duplicate.
→ Random order based on hashcode.
→ Only one null key is allowed and multiple null values are allowed.

ex: Pgm

```
package maps.demo;
import java.util.*;
public class Hashmapdemo
{
    p s v m( )
    {
        HashMap<Integer, String> m1 = new HashMap<>();
        m1.put(101, "Rahul");
        m1.put(102, "Riya");
        m1.put(103, "pooja");
        // m1.put("334", 1030);
        SOP(m1); // {101=Rahul- - - -
        SOP(m1.keySet()); // [101, 102, 103]
        SOP(m1.values()); // [Rahul, Riya, pooja]
        SOP(m1.get(101)); // Rahul
        SOP(m1.get(334)); // null
        SOP(m1.containsKey(1002));
        SOP(containsValue("java"));
        m1.put(101, "sanju");
        m1.put(103, "pooja1.0");
        SOP("After adding duplicate key:"+m1);
        m1.put(null, "pooja2.0");
        SOP(m1);
        m1.put(null, "pooja3.0");
        SOP(m1);
    }
}
```

o/p:-

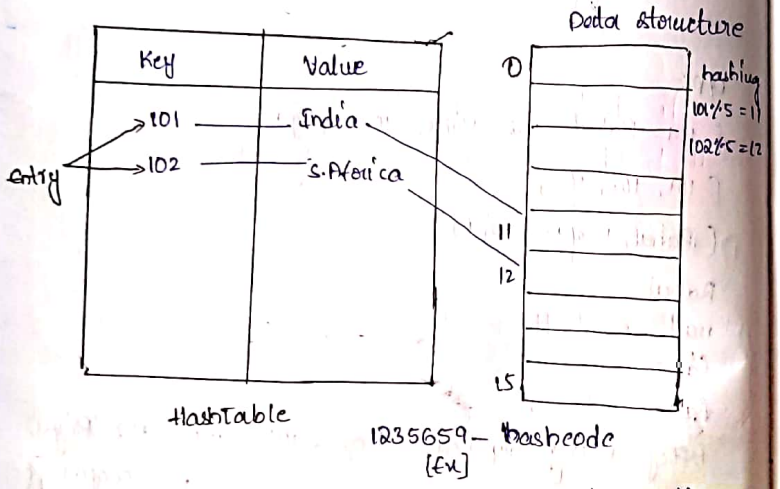
```
{101=Rahul, 102=Riya, 103=pooja}
[101, 102, 103]
[Rahul, Riya, pooja]
Rahul
null
false
false
After adding duplicate keys {101=sanju, 102=Riya, 103=pooja1.0}
{null=pooja2.0, 101=sanju, 102=Riya, 103=pooja1.0}
{null=pooja3.0, 101=sanju, 102=Riya, 103=pooja1.0}
```

## Hash Table :-

→ It is a datastructure (like an array) which stores the objects as key value pair by means of Hashing.

→ Default size of hashtable is 11.

→ Once hashtable reaches the 3/4 of default size




101 % 15 = 11
102 % 15 = 12
103 % 15 =

Ex:-

```
package mps.demo;
import java.util.*;
public class Hashdemo
{
    p s v m( )
    {
        HashMap<Integer, String> m1 = new HashMap<>();
        m1.put(101, "Rahul");
        m1.put(102, "priya");
        m1.put(103, "pooja");
        // m1.put("334", 445);
        SOP(m1); // {101=Rahul, 102=Riya, ....}
        SOP(m1.KeySet()); // [101, 102, 103]
        SOP(m1.values()); // [Rahul, Riya, pooja]
        SOP(m1.get(101)); // Rahul
        SOP(m1.containsKey(1002)); // false
        SOP(m1.containsvalue("java")); false
        SOP(m1.put
        m1.put(101, "sanju");
        m1.put(103, "pooja 2.0");
        SOP("after adding duplicate keys:" + m);
        m1.put(null, "pooja3.0");
        SOP(m1);
        m1.put(null, "pooja 4.0");
        SOP(m1);
    }
}
```

O/P:-

```
{101=Rahul, 102=Riya, 103=pooja}
[101, 102, 103]
[Rahul, Riya, pooja]
Rahul
false
false
After adding duplicate keys {101=sanju, 102=
                             Riya, 103=pooja}
```

{ null = pooja 3.0, 101 = sayju, 103 = pooja 2.0 }

{ null = pooja 4.0, 101 = sayju, 102 = Riya, 103 = pooja 1.0 }

## Linked Hashmap:-

It is a class which extends hashmap and implements map interface.

features:-

→ Introduced in 1.4V

→ Heterogeneous data allowed

→ Duplicate Keys are not allowed but ~~if we~~ values can be duplicate, if we add duplicate Key it replace with original one.

→ As per insertion order.

→ Only one null Key and multiple null values are allowed.

// Only differ is o/p is as per insertion order //

## TreeMap:-

It is a class which implements map interface.

features:-

→ Introduced in 1.2V.

→ Only homogeneous data allowed, if we add heterogeneous object we will get cas exception

→ Data structure is hash table

→ Duplicate Keys are not allowed but values can be duplicate.

→ Random order based on hashcode.

→ We can add null Key if tree is empty, if treemap contains any data and if we add null Key we get NullpointerException.

Ex:- 1) TreeMap m = new TreeMap();
m.put(null, 1233); // valid because treemap does not have any data.

2) TreeMap m1 = new TreeMap();
m1.put(101, "India");
m1.put(null, "sss"); // NullPointerException