17/04/19 :-

Abstraction :-

* Hiding Method implementation details and exposing method signature is called Abstraction.
* Abstraction can be achieved in 2 ways
  (i) by using interface
  (ii) by using abstract class.

Steps to achieve abstraction :-

(i) Create an interface.
(ii) Create an abstract method (incomplete method)
(iii) Create an implementation class.
(iv) Implement the abstract method (or) interface method.
(v) Create an helper class (it is also called as factory class)
(vi) Create an helper method (it is also called as factory method).

~~In-built Java library~~ :-

Helper method can be a static method or non static method.

Standard practise for helper method name :-

(i) Helper method name must be similar to interface name (or)
Helper method name should begin with get followed by interface name.

public void sample() (or) public void getSample();

(ii) Create an object of implementation class and upcasted to interface and return the object as interface type.

(iii) Generate the java document.

** In abstraction to call the interface method, we can create a object of implementation class. toget helper class and helper method (by using)

Program :-

```
interface Sample (i)
{
  public void test();
}
class Demo implements Sample
{
   public void test()
    {
      s.o.pln("In test");
    }
}
```

```
class Run // (ii) helper class
{
  public Sample sample() // helper method
  {
    sample s1 = new Demo();
    return s1;
  }
}
```

Java Document

Sample <interface>
test() : void

Run < class >
Sample() : Sample

Steps to use abstraction program :-

(i) We must have an idea about interface and helper class. (through java document)

(ii) We must call interface method.
Get the implementation class object through helper class and helper method.

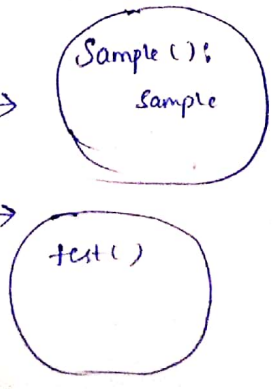
(iii) If helper method is static method use helper class name.

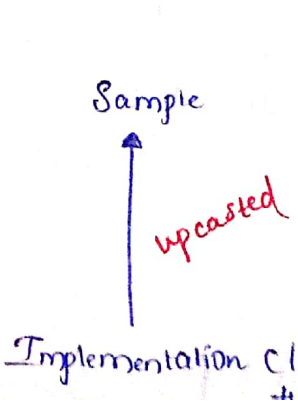
(iv) If helper method is non static method use helper object

(v) Invoke helper method and get the object of implementation class, helper method returns implementation class object which is upcasted to interface type.

(vi) use implementation class object and invoke interface methods.

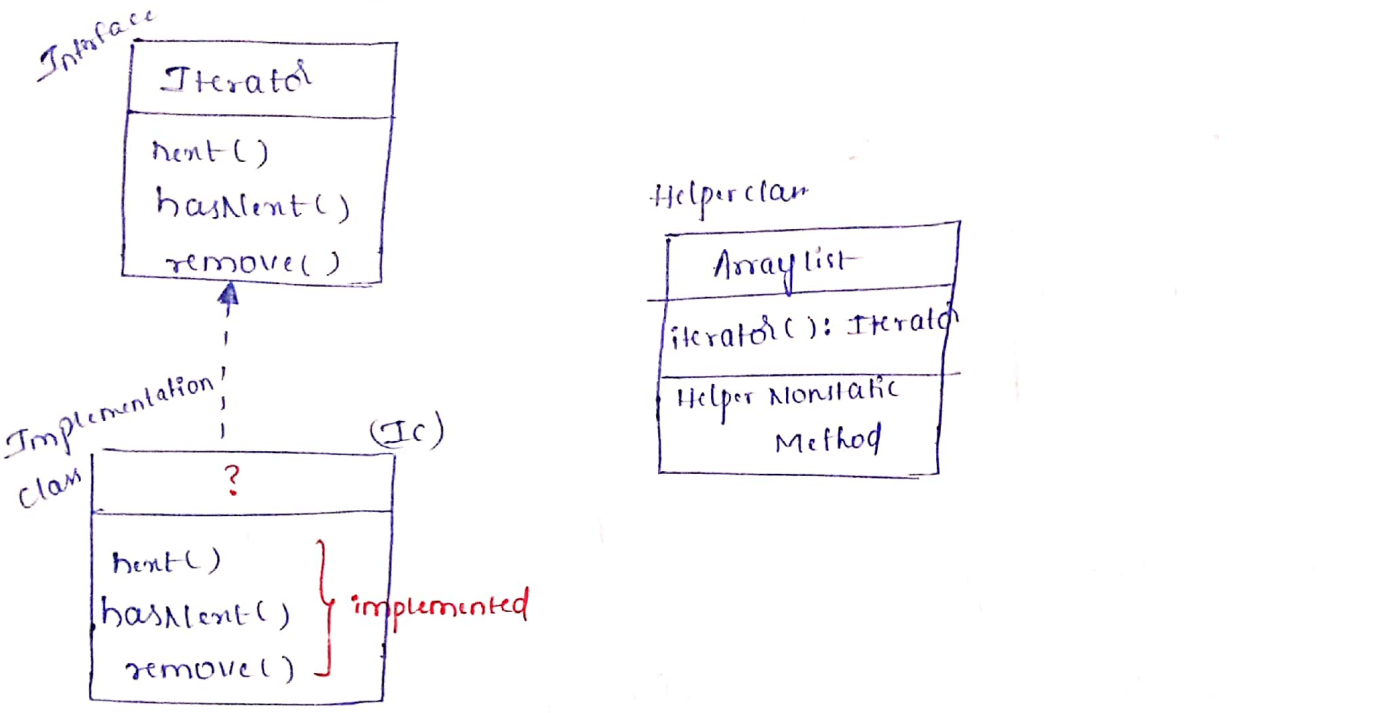
```
class UseAbstractionPgm
{
  public static void main(String[] args)
  {
    Run r1 = new Run();
    Sample s1 = r1.Sample();
    s1.test();
  }
```



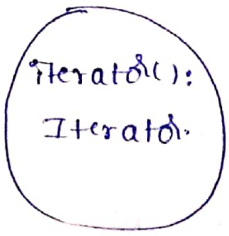


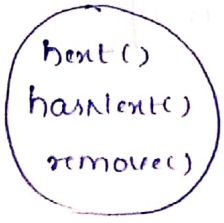
* If any one ask use the interface method and then you can ask what is the helper method. by using helper class method we should create an object and load the method of object

→ Inbuilt library example for abstraction.



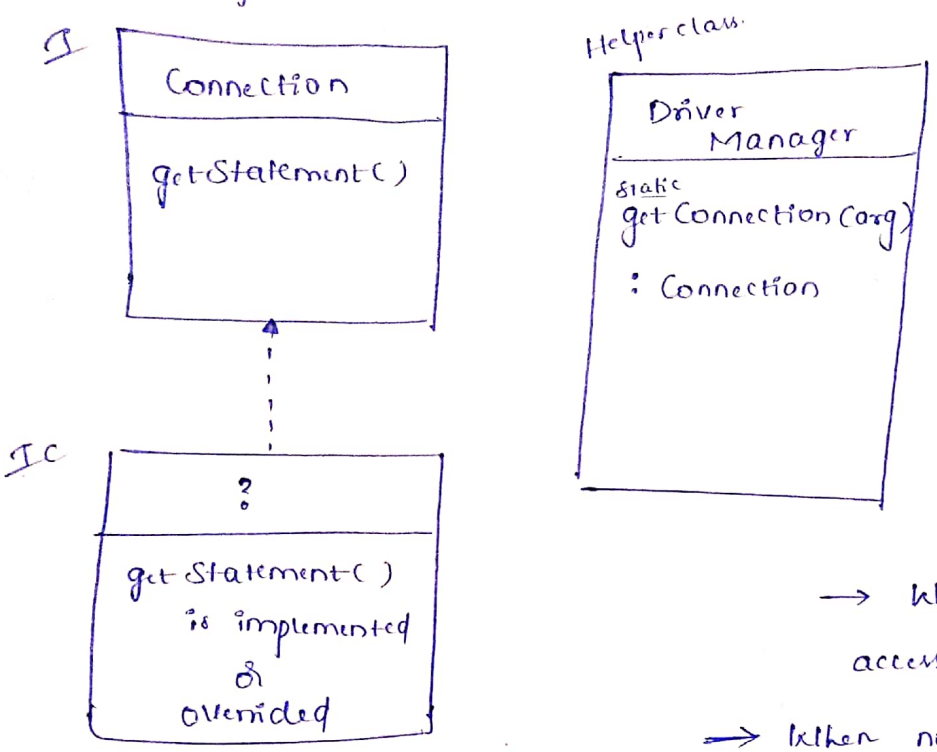

```
class useAbstractPgm2
{
  public static void main(String[] args)
  {
    Arraylist al = new Arraylist();
    Iterator il = al.iterator();
      il.next();
      il.hasNext();
      il.remove();
  }
}
```





Date : 17/04/19

2nd example for Inbuilt



→ When static methods are used access through interface (or) class

→ When nonstatic methods are used creating an object ~~then~~ by using helper class (or) helper method.

Program 1 :-

```
class useAbstractpgm2
{
   public static void main (String [] args)
   {
     Connection cl = DriverManager.getConnection(arg);
     cl.getStatement();
   }
}
```

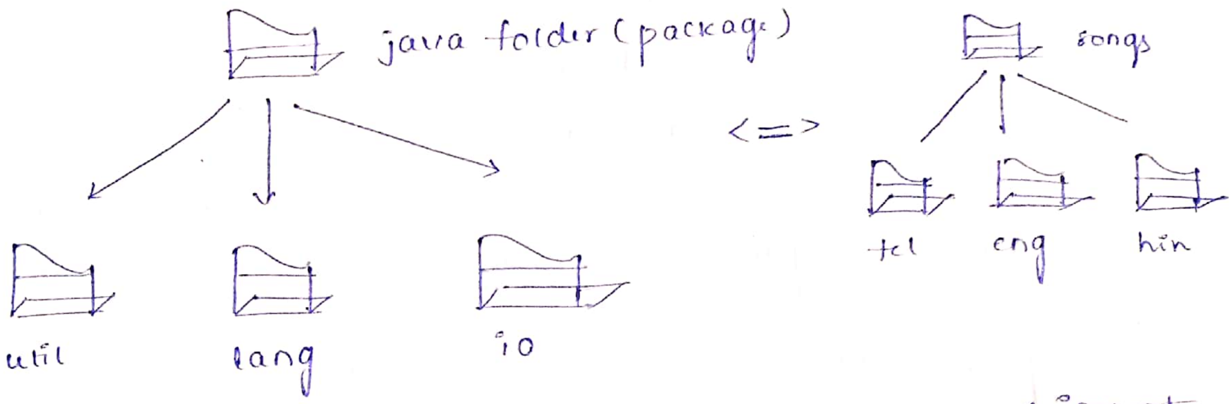
getStatement()

* Purpose of abstraction is to avoid unnecessary complexity.

Complexity : The user may get confused at the logic provided by the designer that may arise complexity.

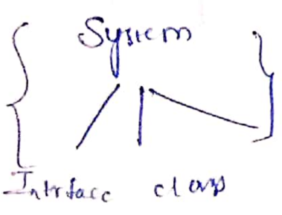
Inbuilt Library :-

(i) Inbuilt library consists of inbuilt classes and inbuilt interfaces.
(ii) All these inbuilt classes and interfaces belongs to inbuilt packages.
(iii) There are many inbuilt packages. for example
Ex: java.lang , java.util , java.io.



* Any classes & interfaces used from java.lang package import statement is not required because java.lang package classes and interfaces are imported by default.
* for any other package classes and interface import statement is compulsory.

System.out.println( ):-



System — class (✓), variable (×), Method (×)

By using class name naming format class name first letter should be capital and the remaining letters should be lower class so thats why System is a class.

out — class (×), variable (✓), Method (×)

variable — I.V (×), G.V — S.V, N.S.V (×)

S.V — primitive var (pv) (×), rv (reference variable) (✓)

out is a variable because variable naming format can be the word should be in lowercase. thats why out is an variable

out cannot be a local variable because it can only used in method then the Global variables can be used in any where in the program.

System.out (classname . variable) so this type of manner is used only for static variables.

In reference variable we are using "." operator. primitive variables does not contain any . operator thats why reference variable.

println
- class (x)
- variable (x)
- method (✓)
  - static (x)
  - Nonstatic (✓)

* println is a method and then it is declared as nonstatic because this can be used in multiple times (out.println) (var name . method) so this type of manner is only used for nonstatic method.

→ finally I conclude System is a final class present in java.lang package.

→ out is a final static reference variable of printStream type.

→ println is a overloaded nonstatic method present in printStream class.

Note: How many classes involved in System.out.println();

In System.out.println() two classes are involved i.e., System ∞ class and printStream class. (println belongs to printstream class)

Program :-

(ii)
```
package java.lang;
public final class System
{
  public final class static printStream out = new PrintStream();
}
System.out.println(56);
```

(i) package java.io;

public class PrintStream

{

public void println (String arg)

{ ——— ———

}

public void println (int arg)

{ ——— ——— ———

}

}

19/04/19:-

NOTE :-

* Access Modifiers in java :-

1. public (Green) 2. default (blue) (package) 3. private (Red) 4. protected (Yellow)

5. abstract 6. final 7. static 8. Nonstatic (no keyword).

* Access specifiers are also called as access Modifiers.

Object class :-

(i) It is superclass ~~through~~ to all the classes.

(ii) It is present in java.lang package.

(iii) Every class in java inherits members of object class.

Methods of object class :-

(i) toString() :-

toString() : String

Method name : Returntype

* toString() returns Complete information of current object.

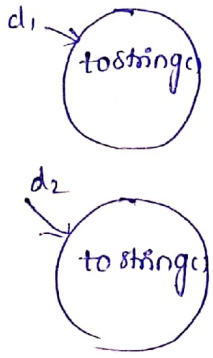
* Complete information Consists of "packagename.classname@ Object Address."

* Object Address can be changes.

Program :-

* learning inbuilt class means learning methods

```
package demopack1;
public class Demo
{
    public static void main (String[] args)
    {
        Demo d1 = new Demo();
        String s1 = d1.toString();
        S.O.pln (s1);
        Demo d2 = new Demo();
        S.O.pln (d2.toString());
    }
}
```



output:- demopack1.Demo@7852e922
demopack1.Demo@4e25154f.

* Whenever reference variables are printed there is an implicit call to toString() method. which returns complete information of current object.

```
Demo d1 = new Demo();
S.O.pln (d1.toString()); // explicit call to toString()
S.O.pln (d1);            // implicit call to toString():
```

output: demopack1.Demo@7852e922
demopack1.Demo@7852e922

d1 ref var gives only address but d1 implicit call to tostring method then gives the complete information.

(ii) equals (Object o1): boolean :-

equals (Object o1) : boolean
Method name : Return type

* equals method compares object based on object address.

Prog:

```
Package demopack1
Public class Sample
{
  public static void main (String[] args)
  {
    Sample s1 = new Sample();
    Sample s2 = new Sample();
    boolean status = s1.equals(s2);
    S.o.pln(status);
  }
}
```

o/p → false,
because address of s1 & s2 is different
s1 and s2 address is not equal.

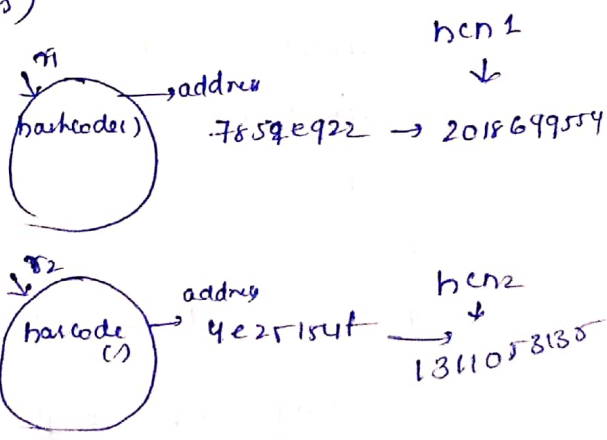
(iii) hashcode()

hashcode() : int
Methodname : Returntype

* hashcode method returns hashcode number for objects address
* hashcode numbers are unique & universal.
i.e., two objects hashcode number will not be same rather(unique)
but for same address in every system (laptops) hashcode number will b
same (universal)
* By using hashcode number to track the object (or) search the object

Prog:

```
Package demopack1;
public class Run
{
  public static void main (String[] args)
  {
    Run r1 = new Run();
    Run r2 = new Run();
    int hcn1 = r1.hashCode();
    int hcn2 = r2.hashCode();
    S.o.pln(hcn1);
    S.o.pln(hcn2);
  }
}
```



Output:—
2018699554
1311053135

Date : 20/04/19

Program :-

* By default toString() return packagename.classname@objectAddress. it is not useful information. It is developers responsibility to override toString() during the to return useful information
* By default equals() compares objects Address based on objectAddress. this is not proper way to compare objects, rather objects should be compared by using objects features.
* By default hashcode() returns hashcode number for object Address. there is no use of this hashcode number because if the object position is changes hashcode number also changes. in that case we cannot search the object in proper manner. so we must override hashcode method to return hashcode number for object itself.

Program :-

```
package demopack1;
public class Emp
{
    int empId;
    String empName;
    double empSal;
    public Emp(int empId, String empName, double empSal)
    {
        this.empId = empId;
        this.empName = empName;
        this.empSal = empSal;
    }
    public boolean equals(Object obj)
    {
        Emp rv1 = (Emp) obj;
        return this.empSal == rv1.empSal;
    }
```

```
public int hashCode()
{
    return this.empId;
}
public String toString()
{
    return this.empName;
}
public static void main(String[] args)
{
    Emp e1 = new Emp(12, "Sanju", 8500.0);
    Emp e2 = new Emp(16, "Sandhya", 9000.0);
    S.O.pln(e1);
    S.O.pln(e2);
    S.O.pln(e1.equals(e2));
    S.O.pln(e1.hashcode());
    S.O.pln(e2.hashcode());
}
}
```

22/04/19 :

2. toString()
2. equals()
3. hashcode

Other methods of object class :-

(i) clone() (ii) finalize() (iii) notify() (iv) notifyAll()
(v) wait() (vi) wait(int arg) (vii) wait(int arg, long arg)
(viii) getClass()

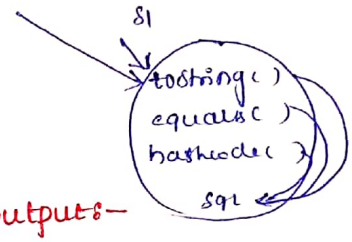
String class :-

* String is a class present in java.lang package.
* String is final class which cannot be inherited and used. rather only used.
* String class is used for storing String information.
* Object class is Super class for String class.
* within String class toString(), equals() and hashcode() are overided.

* toString() method is overrided to return String information as useful information.
* equals() method is overrided to compare string objects by using String information.
* hashcode() method is overrided to return hashcode number for String information.

Program :—

```
package demopack2;
public class sample
{
    public static void main(String[] args)
    {
        String s1 = new String("sql");
        String s2 = new String("sql");
        S.O.pln(s1);
        S.O.pln(s2);
        S.O.pln(s1.equals(s2));
        S.O.pln(s1.hashCode());
        S.O.pln(s2.hashcode());
    }
}
```



output:—

```
sql
sql
true
11426
11426
```

Explanation :—

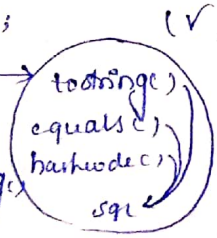
S.O.pln(s1); → implicit call to tostring() print the string data.
S.O.pln(s2);

S.O.pln(s1.equals(s2)); → Comparing the string data.

S.O.pln(s1.hashcode()); → Gives the address of variable as same because they have same data that data can be unique & universal.
S.O.pln(s2.hathcode());

→ String s1 = "sql"; (✓)

S.O.pln(s1);



s1 sql (x)

s1 is a reference var; implicit call to tostring() and gives the information as sql

If all the methods are overrided.

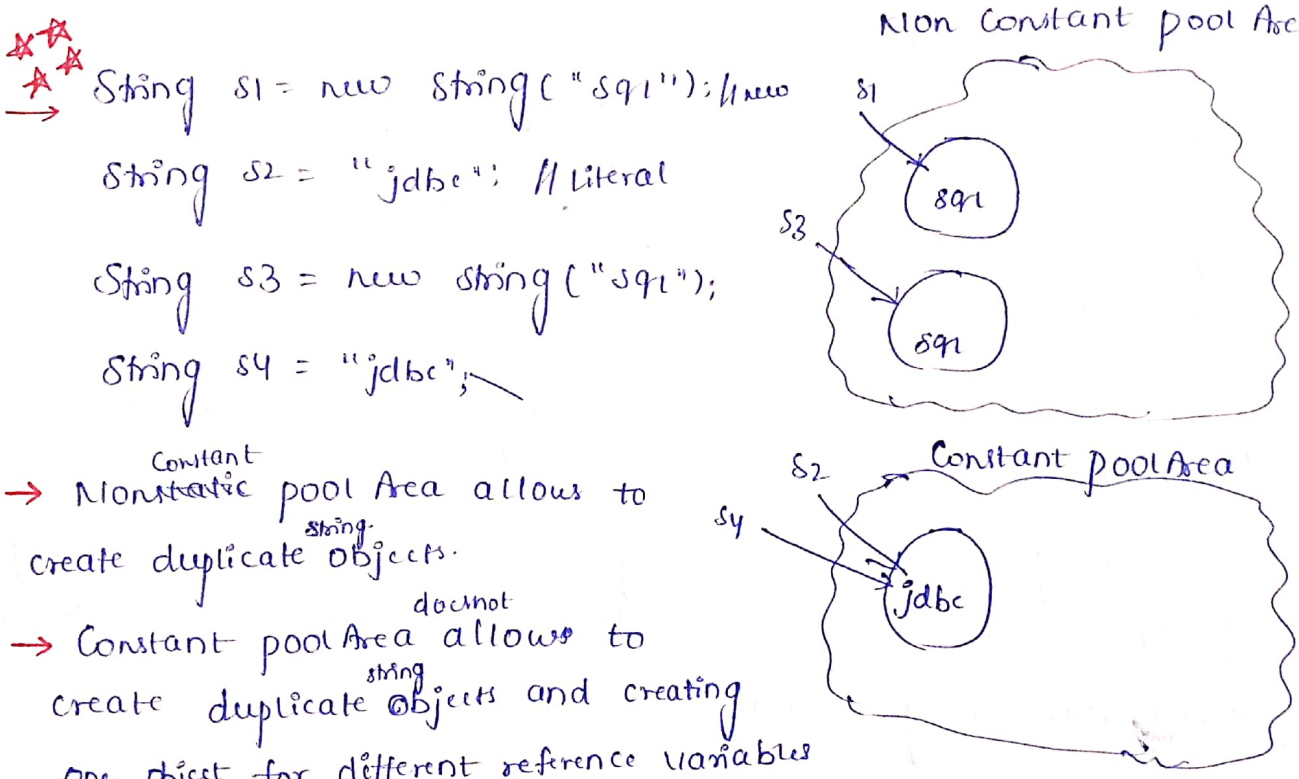
→ When we declare String object method, it can be defined in two ways

(i) by creating an object (ii) literal → (" ");

String s1 = new String("sql"); String s1 = "sql";

(Correct assumption) (wrong assumption)

When s1 is a reference variable, all the methods are overrided implicit call to toString() method and gives the information as sql.



→ String s1 = new String("sql"); // new

String s2 = "jdbc"; // Literal

String s3 = new String("sql");

String s4 = "jdbc";

→ Non Constant pool Area allows to create duplicate String objects.

→ Constant pool Area does not allow to create duplicate String objects and creating one object for different reference variables

→ new creating an object for Non Constant pool Area.

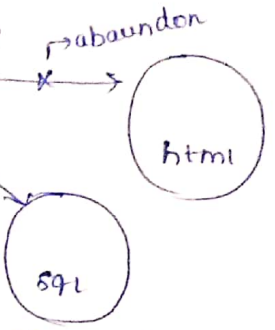
→ literal creating an object for Constant pool Area.

String immutable property :- (immutable = Constant) same)

* immutable means String data present in String object cannot be changed. this is called immutable property.

* Even if we try to change new String object is created.

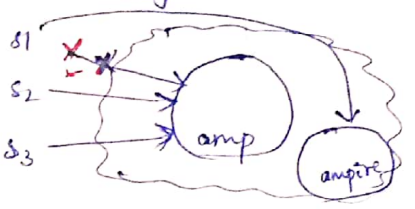
Ex:- String s1 = "html";
s1 = "sql";



→ When we creating an String object this data cannot be changed suppose if we trying to change the data, previous object will be "abaundon" and before the creating an new object for same reference variable.

→ When the object contains same data that data can stored in different reference variables for same object.

Ex:3: String s1 = "amp";
String s2 = "amp";
String s3 = "amp";
String s1 = "ampire";



* String is immutable because same String object can have multiple reference variables.
* In case String is mutable, if any reference variable makes changes to String object, the changes will reflect to all other reference variables.
* To avoid this String is immutable. (Ex:3).

24/04/19

String class methods will be used in String programs:-

Wrapper classes:-

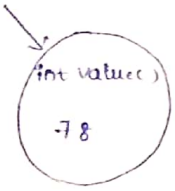
* To represent primitive datatype as object as an object we use wrapper class.
* All the wrapper classes are present in java.lang package.
* All the wrapper classes are final classes.
* Within all the wrapper classes object class methods (toString(), equals(), hashCode()) are overrided.

* For every primitive datatype there is a corresponding wrapper class.

Primitive Datatype	Wrapper class.
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean.

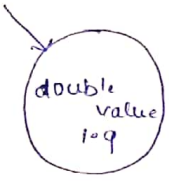
* By using Wrapper class we can perform boxing and unboxing.
* Process of converting primitive data into object is called boxing.
* Process of converting object back to primitive data is called unboxing.

(i) `Integer i1 = new Integer(78);` [boxing]



`int k = i1.intValue();` [unboxing]

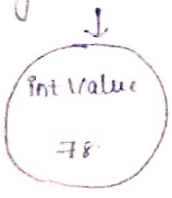
(ii) `Double d1 = new Double(1.9);` [boxing]



`double j = d1.doubleValue();` [unboxing]

* from jdk 1.0 to jdk 1.4 boxing and unboxing is explicit. We have to perform it in the above manner.
* from jdk 1.5 till the latest version boxing and unboxing is implicit.

`Integer i1 = 78;`

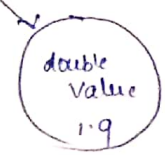


`int k = i1;`

↓

k variable cannot store the address of the i1 variable it can store the value of variable i1. (implicit call)

`Double d1 = 1.9;`



`double j = d1;`

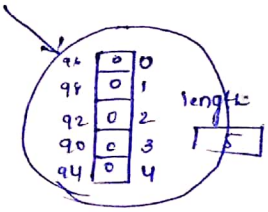
* Integer i1 = 78;
  i1 value is not assign to "K" - there is implicit call to the value present in i1 address.

Arrays :-

* Arrays are objects in java.
* Syntax to declare Array.
  * arraytype [ ] refvar = new arraytype [int size]
    ↓ datatype
  (i) arraytype [ ] arrayName = new arraytype [int size] (or)
  (ii) arraytype [ ] arrayName [ ] = new arraytype [int size]

Note :- arraysize, index number, length value will always be integer type. irrespective of arraytype.

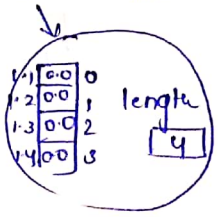
Ex: int[] a1 = new int [5];




```
a1[0] = 96;
a1[1] = 98;
a1[2] = 92;
a1[3] = 90;
a1[4] = 94;

for(int i = 0; i <= a1.length-1; i++)
{
  System.out.println(a1[i]);
}
```

double[] d1 = new double [4];



0,1,2,3 → index values

```
d1[0] = 1.1;
d1[1] = 1.2;
d1[2] = 1.3;
d1[3] = 1.4;

for (int i = 0; i <= d1.length-1; i++)
{
  System.out.println(d1[i]);
}
```

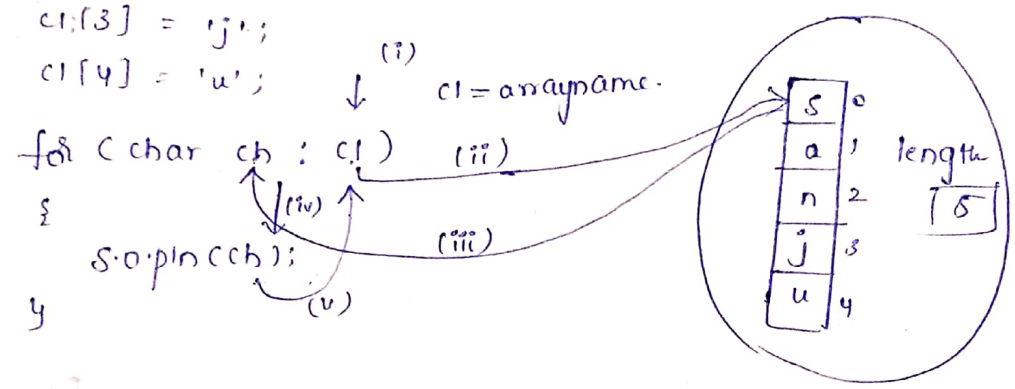
25 24/04/19

* In case of array we can also retrieve data by using foreach loop.

Syntax: for (arrayelement type varname : array name) { - - - }

```
for ( char ch : c1)
  {
  ____
  ____
  }
```

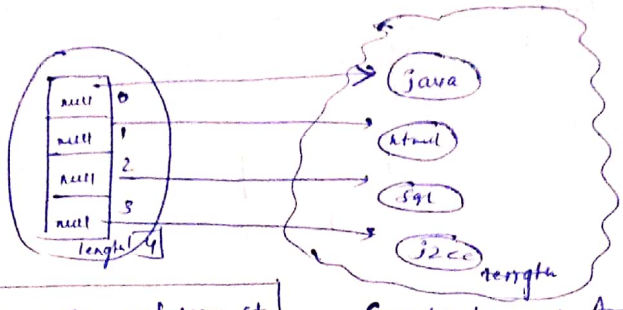
prog:
```
class prog1
{
  public static void main (String[] args)
  {
    char[] c1 = new char[5];
    c1[0] = 's';
    c1[1] = 'a';
    c1[2] = 'n';
    c1[3] = 'j';
    c1[4] = 'u';
    for ( char ch : c1 )
    {
      S.o.println(ch);
    }
  }
}
```



→ first control goes to the c1 (arrayname) and fetches the first element initialite it to the variable name and does the operation and then goes to the arrayname. Hence the process repeats.

prog:-
```
class prog2
{
  public static void main(String [] args)
  {
    String[] S = new String[4];
    S[0] = "Java";
    S[1] = "html";
    S[2] = "sql";
    S[3] = "j2ee";
```

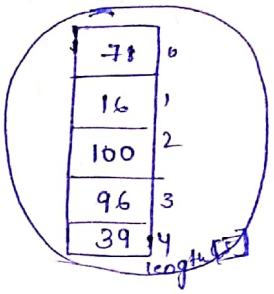
```
for ( String st : s )
{
    s.o.pln (st);
}
}
}
```



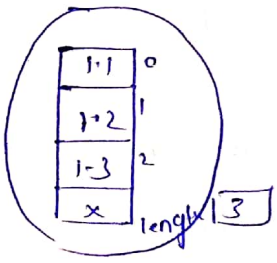
→ I assumed that String is not a primitive but the String data is not stored in object rather it is stored in Constant pool Area object. because String is derived within array object reference variables are created which holds address of objects in Constant pool Area.

Second way to create Array :-

(i)
```
int [] a1 = new int[] { 78, 16, 100, 96, 39 };
for (int i1 : a1 )
{
    s.o.pln (i1);
}
```



(ii)
```
double [] d1 = new double[] { 1.1, 1.2, 1.3 };
for (double d : d1 )
{
    s.o.pln (d);
}
```



(iii)
```
char[] c1 = new char[] { 's','a','n','j','u' };
for ( char c : c1 )
{
    s.o.pln (c);
}
```

(iv)
```
String[] s1 = new String[] { "java", "html", "sql", "j2ee" };
for ( String s : s1)
{
    so.pln (s);
}
```

Third way to create array :-

(i)
```
int[] a1 = {1, 2, 3, 4, 5};
for (int i : a1)
{
    s.o.pln(i);
}
```

(ii)
```
double[] d1 = {1.1, 1.2, 1.3, 1.4};
for (double d : d1)
{
    s.o.pln(d);
}
```

(iii)
```
char[] c1 = {'s', 'a', 'n', 'j', 'u'};
for (char c : c1)
{
    s.o.pln(c);
}
```

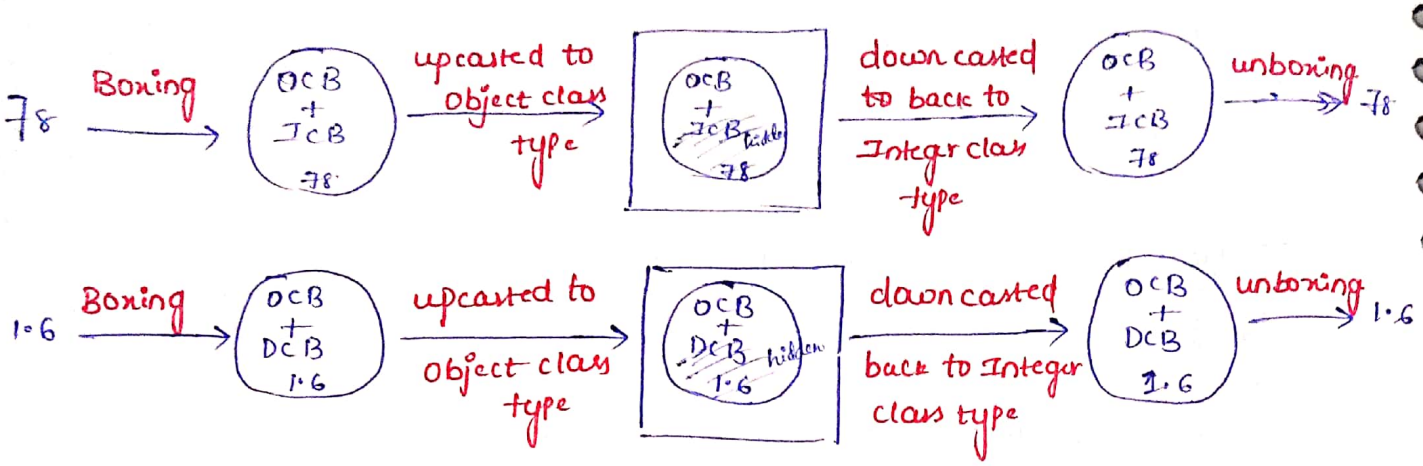
(iv)
```
String[] s1 = {"java", "html", "css", "j2ee"};
for (String s : s1)
{
    s.o.pln(s);
}
```

25/04/19

Drawbacks of Array :-

* In case of array, we must compulsory provide size of array.
* In case of array, types of data must be same [Homogenous types]

Collection frame Work



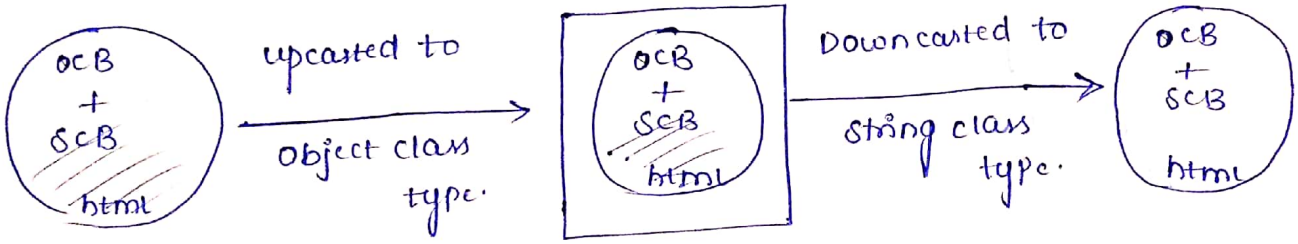
OCB : object class Behaviour

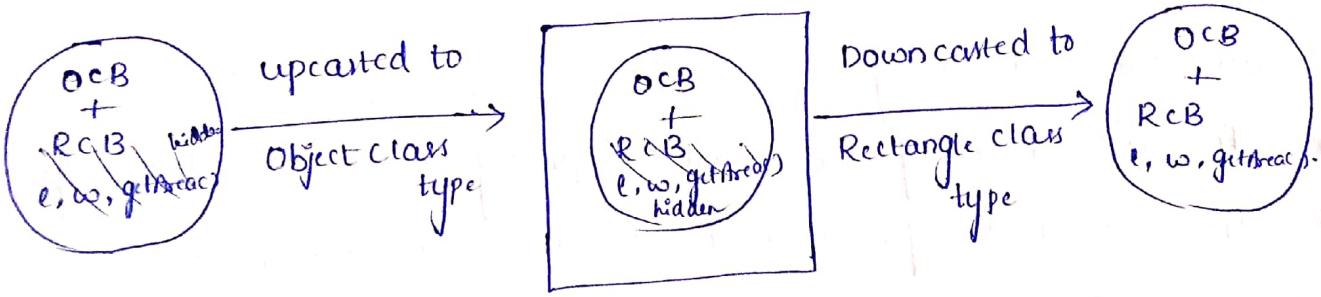
IcB : Integer class Behaviour

DcB :- Double class Behaviour.

* In case of primitive collection works in the below manner.

primitive data → Boxing → upcasting → store in collection → retrieving → downcasting → unBoxing → primitive Data.





* In case of collection, non primitives (derived types) works in below manner.

object of derived type → upcasting → store in collection → retrieving → down casting → object of derived type.

Note :- Incase of collection in and out, every thing is object.

* learning collection means learning about methods.
* We have different types of collection type.

1. List :-

* There are three types of list.
  - (i) ArrayList
  - (ii) Vector.
  - (iii) LinkedList.
* List is an index based.
* List mantains insertion order.
* List allows duplicate object.
* List allows null values.
* List allows different types of object (heterogenous type)

add : To add objects

size : Count no. of objects including null values.

prog:

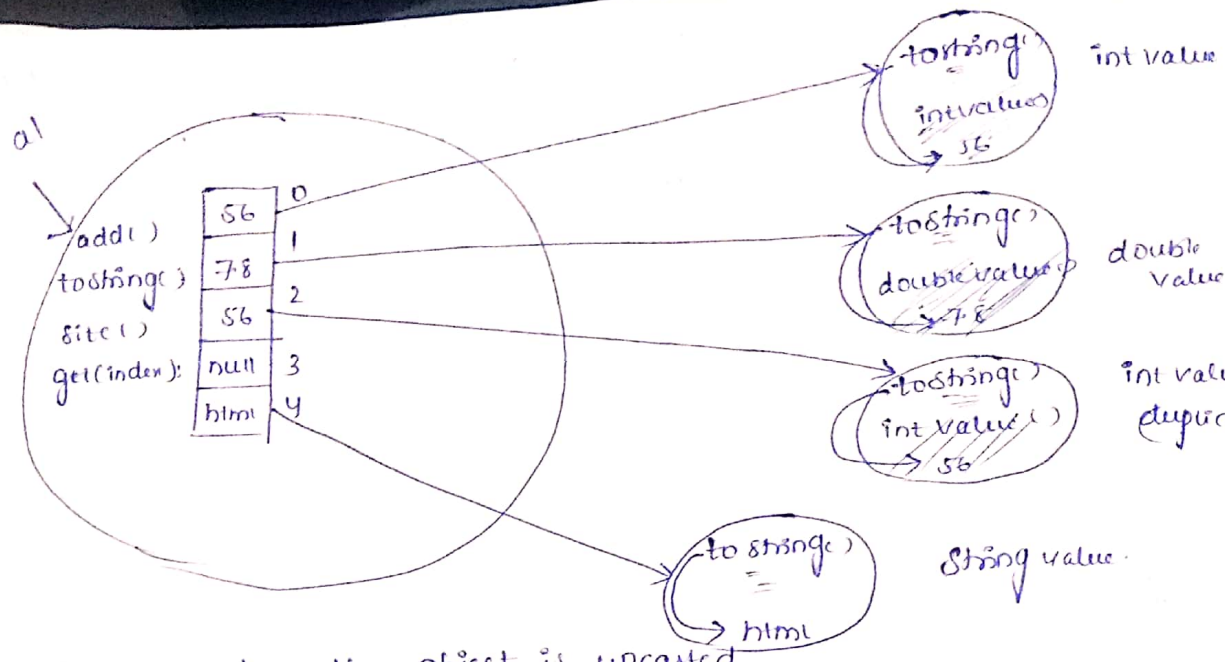
```
package demopack1
import java.util.ArrayList;
public class Sample
{
  public static void main (String[] args)
  {
    ArrayList al = new ArrayList();
    al.add(56);
    al.add(7.8);
    al.add(56);
    al.add(null);
    al.add("html");
    S.O.pln(al); // implicit call to tostring()
    s.o.pln(al.size());
    s.o.pln(al.get(2));
  }
}
```

Output:

[56, 7.8, 56, null, html]

5.

56.



→ tostring() :— when the object is upcasted tostring method is overrided to 56 as the useful information.

→ get(index) :— To get the specific information for the particular index value. is used

→ add() :— add method to add the objects

→ size() :— size method is used to count the objects including null values.

```
package demopack2;
import java.util.ArrayList;
public class Run
{
  public static void main (String[] args)
  {
    ArrayList al = new ArrayList();
    al.add(76);
    al.add(23);
    al.add(21);
    al.add(97);
    al.add(14);
    S.O.pln(al);  // 1.
```

```
al.add ( 2, 35 );      // (index, element)   2.
S.O.pln(al);           // 2
al.set (4, 99) ;       // set(4.99)   3. remove older value. enter newer value (replace)
S.O.pln (al);          // 3
al.remove (3);         // remove the 3rd index value.  4
S.O.pln (al);          // 4
al.contains S.O.pln (al.contains (23));   // 5.
  al.clear();          // 6
  S.O.pln (al.isEmpty ());   // 6
  }
}
```

Output :-

[78, 23, 21, 97, 17]

[78, 23, 35, 21, 97, 17]

[78, 23, 35, 21, 99, 17]

[78, 23, 35, 99, 17]

true.

true.

→ add (index, element) :- This method is used to add the element in specific index.

→ set (index, element) :- This method is used to replace the element based on index value.

→ remove (index / element) :- This method is used to remove the object present at the index value.

→ contains (element) :- when the eleem To check the element is present in the Arraylist & not.
if it present gives boolean value (True) otherwise -false.

→ clear() :- To remove all the elements present in the Arraylist.

→ isEmpty() :- If the Arraylist is empty returns boolean value True otherwise false.

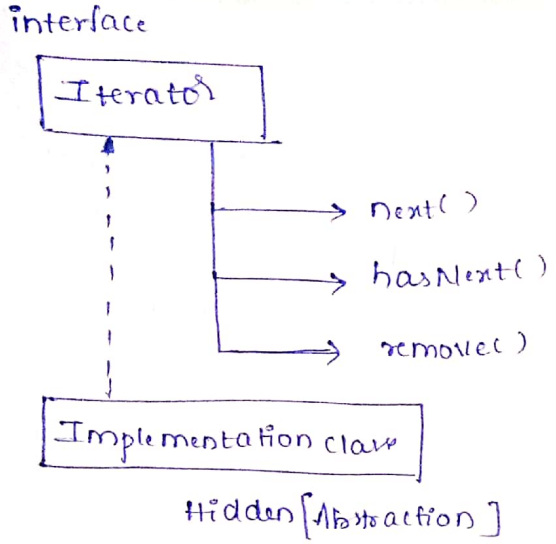
* We can also retrieve elements from Arraylist by using "Traversing Technique." i.e., methods of Iterator (next(), hasNext(), remove()).

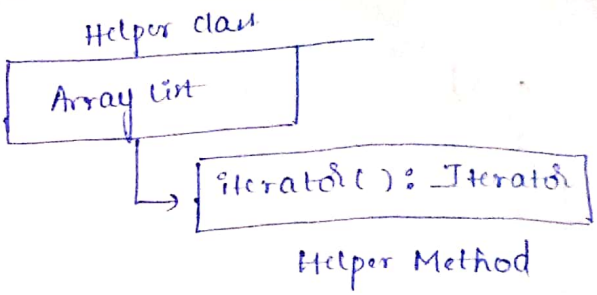
* Traversing one Step to another Step.
* interface (Iterator)

Prog:-

```
package demopack1;
import java.util.Arraylist;
import java.util.Iterator;
public class Run
{
  public static void main (String[] args)
  {
    Arraylist al = new Arraylist();
    al.add(78);
    al.add(23);
    al.add(21);
    al.add(97);
    al.add(17);
    Iterator il = al.iterator(); // Abstraction.
    while (il.hasNext())  // it gives boolean values true/false.
    {
      S.O.pln (il.next());
    }
  }
}
```

output:
78
23
21
97
17





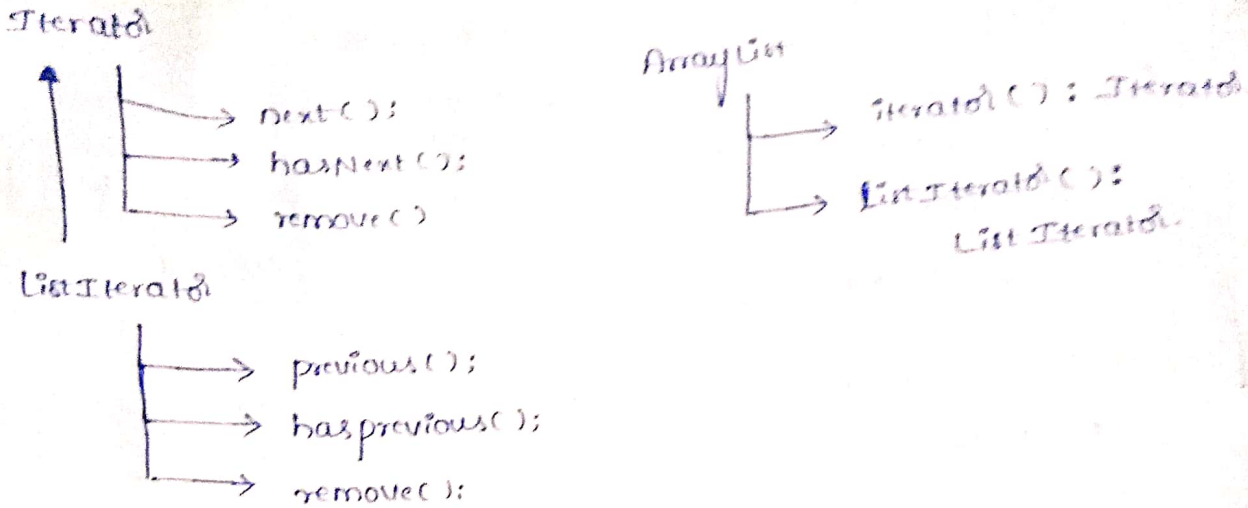
Date : 26/04/19

On list Arraylist we can also perform backward direction traversing. by using "ListIterator" interface.

```
package demopack1
import java.util.Arraylist;
import java.util.ListIterator;
Public class prog3
{
  public static void main (String[] args)
  {
    Arraylist al = new Arraylist();
    al.add(78);
    al.add (23);
    al.add (21);
    al.add (97);
    al.add ( 17);
    ListIterator li = al.listIterator();
    while (li.hasNext()).
    { s.o.pln (li.next());
    }
    while (li.hasprevious())
    { s.o.pln (li.previous());
    }
  }
}
```

Output :-

78
23
21
97
17
17
97
21
23
78.



Generic :-

We can convert any collection type to handle single type of object.

Prog:-

```
package demopack1
import java.util.ArrayList;
import java.util.Iterator;
public class prog1
{
  public static void main(String[] args)
  {
    ArrayList<String> al = new ArrayList<String>();
    al.add("html");
    al.add("java");
    al.add("jdbc");
    al.add("sql");
    Iterator<String> it = al.iterator();
    while(it.hasNext()).
    {
      s.o.p(it.next());
    }
  }
}
```

* In this case Arraylist allows only String object. if we want any specific object then it must be mention in between < > brackets
↳ angular

1. Write a program to create Arraylist and add 4 integers, retrive it by using bidirection Traversing Technique.

Prog:-
```
package demopack1
import java.util.Arraylist;
import java.util.Iterator;
public class prog6
{
  public static void main (String[] args)
  {
    Arraylist <Integer> a1 = new Arraylist <Integer> ();
    a1.add (23);
    a1.add (35);
    a1.add (12);
    a1.add (29);
  List Iterator <Integer> il = a1.iterator();
    while (il.hasNext())
    {
      S.O.pln (il.next());
    }
    while (il.hasprevious())
    {
      S.O.pln (il.previous());
    }
  }
}
```

29/04/19

Difference between Arraylist and Linked list

27/04/19

Difference between Arraylist and Linked list :-

(i) When Arraylist object is created initial capacity is 10

(i) when linked list object is created initial capacity is zero.
[capacity : no. of memory blocks that are reserved.]

(ii) Incase of Arraylist all memory blocks are reserved in a continous manner.

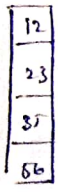
(ii) Incase of linkedlist memory blocks are scattered randomly but each memory block is linked in double linked list fashion [each memory block remembers next memory block and previous memory block].

(iii) Arraylist is faster when elements are added & removed at the end.

(iii) linked list is faster when elements are added & removed in between.

(iv) Incase of Arraylist we can define initial capacity by calling overloaded arraylist constructor.

Ex: Arraylist al = new Arraylist();
al.add(12);
al.add(23);
al.add(35);
al.add(56);

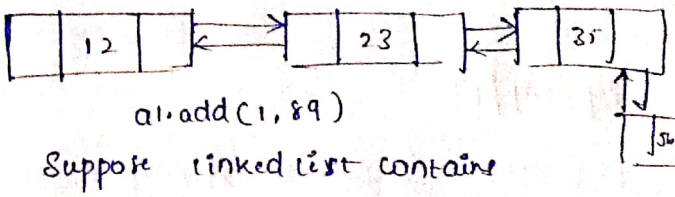


al.add(1,89);

Suppose Arraylist contains 100 elements adding index at 89 then remaining elements are shifted to another position. it takes more time.

(iv) Incase of linkedlist we cannot set the initial capacity.

Ex:
linkedlist l1 = new linkedlist();
l1.add(12);
l1.add(23);
l1.add(35);
l1.add(56);



al.add(1,89)

Suppose linked list contains

* Difference between Arraylist and linkedlist is similar to Vector and linked list.

Difference between Arraylist and Vector :-

Arraylist	Vector.
(i) Arraylist was introduced in jdk 1.2 version	(i) Vector was introduced in jdk 1.0 version.
(ii) Arraylist objects are not thread safe. [Explanation: one object is used by multiple users at called a time is called not thread safe Ex: Road	(ii) Vector objects are thread safe. [Explanation :- one object is used by multiple users by following the rule one by one at any point of time is called thread safe. Ex: chair.
(iii) Arraylist object is not thread safe methods are not synchronized.	(iii) Vector objects are thread safe methods because methods are synchronized.
(iv) Incase of Arraylist there might be data inconsistency. because same object can be used by multiple threads at the same time.	(iv) Incase of Vector there is no data inconsistency problem. because
(v) Incase of Arraylist performance is faster. because all the methods used at a time.	(v) Incase of Vector performance is slower. because if the one method is used another method is waitforition to used.
(vi) Arraylist increases by half of its initial capacity.	(vi) Vector increases by double of its initial capacity.

Queue :-

* There are two types of queue
  (i) linked list
  (ii) priority queue

Note :- LinkedList can behave like list type as well as Queue type.

1. List ll = new LinkedList();
2. Queue ql = new LinkedList();

1. LinkedList is upcasted to list type it can behaves like list type

2. LinkedList is upcasted to Queue type it behaves like Queue type.

Properties of Queue :-

* It is not an index based.
* Duplicate objects are allowed.
* null values is not allowed
* Storage order of elements depends on type of queue.
* Type of objects allowed depends on type of queue.
* Only iterator traversing technique can be used.
* By using poll() method we can retrieve and remove element from the queue.
* By using peek() method we can only retrieve the element.
* LinkedList as a queue, objects are stored in FIFO fashion.
* priorityqueue as a queue, objects are stored in sorting order.
* LinkedList as a queue, it allows heterogenous objects.
* priorityqueue as a queue allows homogenous objects.

Prog:-

```
package demopack1
import java.util.LinkedList;
import java.util.Queue;
public class Run
{
  public static void main (String[] args)
   {
     Queue q1 = new LinkedList();
     q1.add (45);
     q1.add (67);
     q1.add (13);
     q1.add (99);
     while (q1.peek() != null)
      {
        S.O.pln (q1.poll());
      }
   }  S.O.pln (q1.size());
}
```

Output:

45
67
13
99
0

Prog:-

```
package demopack1
import java.util.LinkedList;
import java.util.Queue;
public class Run
{
  public static void main (String[] args)
  {
    Queue q1 = new PriorityQueue();
     q1.add(45);
     q1.add (67);
     q1.add (13);
     q1.add (99);
    Iterator il = q1.iterator();
    while (il.hasNext())
     {
       SO.pln (il.next());
     }
```

```
package demopack1;
import java.util.LinkedList;
import java.util.Queue;
Public class Run
{
  public static void main (String[] args)
  {
    Queue q1 = new priorityQueue();
    q1.add(45);
    q1.add(67);
    q1.add(83);
    q1.add(99);
    while (q1.peek() != null)
    {
      S.O.pln (q1.poll());
    }
    S.O.pln (q1.size());
  }
}
```

Set :-

* There are three types of sets.
  - (i) HashSet
  - (ii) LinkedHashSet
  - (iii) Tree Set
* Set is not an index based.
* Set doesnot allows duplicate objects.
* Set allows only one null value.
* There is no dedicated method to retrieve objects from set. rather we must use iterator traversing technique to retrieve objects from Set.
* In hashset unordered, linkedhashset insertion order, Treeset sorting order.
* Hash Set and LinkedhashSet allows heterogenous objects but Tree set doesnot allows heterogenous objects.

Prog: public class Run1

```
{
  public static void main (String[] args)
  {
    HashSet s1 = new HashSet();
    s1.add(34);
    s1.add (56);
    s1.add ("abc");
    s1.add (true);
    s1.add ('t');
    Iterator il = s1.iterator();
    while (il.hasNext())
    {
      s.o.pln (il.next());
    }
  }
}
```

output: — unorder

[abc , true , 36, 56 , t]

Prog:-

```
public class Run2
{
  public static void main (String[] args)
  {
    LinkedHashSet s1 = new LinkedHashSet();
    s1.add (34);
    s1.add (56);
    s1.add ("abc");
    s1.add (true);
    s1.add ('t');
    Iterator il = s1.iterator();
    while (il.hasNext())
    {
      s.o.pln (il.next());
    }
  }
}
```

output : Insertion order

[34 , 56 , abc , true , t]