Generics & collections

HINDUSTAN LONG BOOK INDEX

NAME : Nitinkumar Vilas Takale

STD : ..... DIV : .....

SCHOOL / COLLEGE : ..... CLASS : .....

Nitinkumar
+91 9823442459

Thursday.
18/07/2013

* Object class :

- Object class is a root class for all classes in java.
- Every class in java directly or indirectly inherit object class.
- The properties (variables) & behaviour (methods) of object class used by JVM for manipulating objects.
- This class provide 11 methods.

1) public String toString()

- This method return String representation of the object.
- This method is executed by JVM when ever program print reference variable.

example:

```
class A
{
    int x=10;
}
class Demo
{
    public static void main (String args[])
    {
        A obj = new A()
        System.out.println (obj);
    }
}
```

Source code in toString():

```
public String toString()
{
    return getClass().getName() + "@" + Integer.toHexa
                                        String(hashCode());
}
```

friday
19/07

- tostring method is override inorder to print state(data) of object.

Example:

```
class Employee
{
    private int empno;
    private String ename;
    Employee (int empno, String ename)
    {
        this.empno = empno;
        this.ename = ename;
    }
    public String toString()
    {
        return empno+" "+ename+" "+super.toString();
    }
}

class Demo2
{
    public static void main(String args[])
    {
        Employee emp1 = new Employee (101,"Rama");
        Employee emp2 = new Employee (102,"Sita");

        System.out.println(emp1);
        System.out.println(emp2);
    }
}
```

exam

Friday
19/07/2013

2) Public boolean equals (Object) :-

- Indicates whether some other object is "equal to" this one.

- The equals method implements an equivalence relation on non-null object references:

1) It is reflexive:

for any non-null reference value x, x.equals(x) should return true.

2) It is symmetric:

for any non null reference values x & y, x.equals(y) should return true iff y.equals(x) returns true.

3) It is transitive:

for any non null values x, y & z, if x.equals(y) returns true & y.equals(z) return true, then x.equals(z). Should return true.

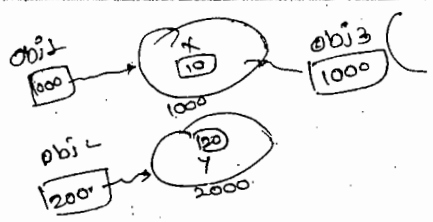
4) It is consistent:

- For any non null reference values x & y, multiple invocations of x.equals(y) consistently return false, provided no information used in equals comparisons on the objects is modified.

for any non null reference value x, x.equals(null) Should return false.

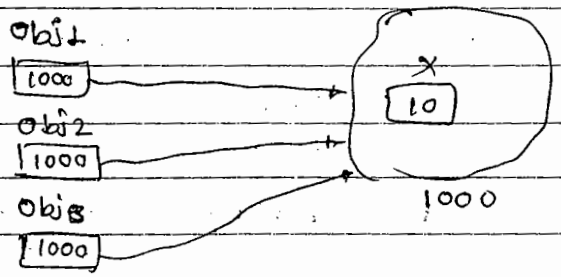
example:

```
1. class A
   {
      int x;
      A (int x)
      {
         this.x = x;
      }
   }

   A obj1 = new A(10);

   boolean b1 = obj1.equals(obj1);
   ... true.
```



2. A obj1 = new A(10);
A obj2 = new A(20);
boolean b1 = obj1.equals(obj2); --- false
A obj3 = obj1;
boolean b2 = obj1.equals(obj3); --- true.

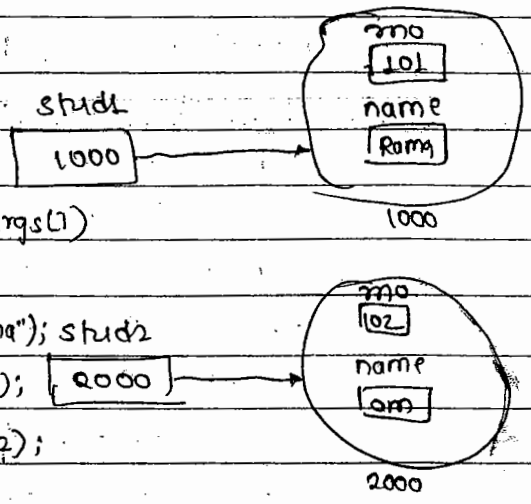
3.
A obj1 = new A(10);
A obj2 = ~~new A(20);~~ obj1;
A obj3 = ~~new A(30);~~ obj2;
boolean b1 = obj1.equals(obj2); --- true.
boolean b2 = obj1.equals(obj3); --- true..
boolean b3 = obj2.equals(obj3); --- true.



- equals method is override in order to compare contents of object.
- By default equals method compare references but not state.

example: Class ~~Demo~~ Student
{
private int no;
private String name;
Student(int no; String name)
{
this.no = no;
this.name = name;
}

```
public boolean equals( Object o)
{
    Student s = (Student)o; ... narrowing type conversion
    if (rno == s.rno)
    return true;
    else
    return false;
  }
}

class Demo3
{
   public static void main (String args[])
   {
    Student stud1 = new Student(101,"Rama"); stud2
    Student stud2 = new Student(102,"om"); 2000
    boolean bl = stud1.equals(stud2);
    System.out.println(bl);
   }
}
```



equals (=)	double equals (==).
1) It is a method of object class.	2) It is operator.
2) It is used for comparing reference types	2) It is used for comparing primitive & reference types.
3) This method by default compare object of references (hashcode). It can be override inorder to compare state of object.	3) It compares only object references. Java doesnot support operator overloading.
4)	

Saturday
20/07/2013

3) Public int hashCode():-

exam

- Returns a hash code value for object. This method is supported for the benefits of hashtable such as those provided by java.util.hashtable.

- The general contract of hashcode is:

Whenever it is invoked on same object more than once during an execution of a Java Application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer is need not remain consistent from one execution of an application to another execution of same application.

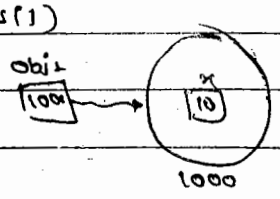
- If two objects are equals according to the equals (object) method, then calling the hashcode method one each of two objects must produce the same integer result.

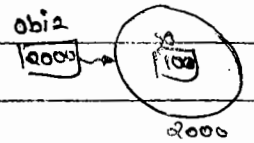
- It is not required that if two objects are unequal according to the equals (java.lang.Object) method, then calling the hashCode method on each of two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtable.

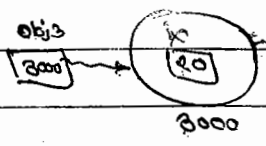
o/p:
true
fals
10
20
20
fal

example:

```
class A
{
    private int x;
    A(int x)
    {
        this.x = x;
    }
    public boolean equals(Object o)
    {
        A a = (A) o;
        if(x == a.x)
        return true;
        else
        return false;
    }
    public int hashCode()
    {
        return x;
    }
}
class Demo5
{
    public static void main(String args[])
    {
        A obj1 = new A(10);
        A obj2 = new A(10);
        A obj3 = new A(20);
        boolean b1 = obj1.equals(obj2);
        boolean b2 = obj1.equals(obj3);
        System.out.println(b1);
        System.out.println(b2);
        S.o.p(obj1.hashCode());
        S.o.p(obj2.hashCode());
        S.o.p(obj3.hashCode());
```







o/p:
true
false
10
20
20
false.

```
        boolean b3 = obj1 == obj2;
        S.o.p(b3);
    }
}
```

4) protected void finalize():-

- Called by the garbage collector on an object when garbage collection determines that are no more references to the object.
- Garabage collection is identifying the object & Garbage collector is removing the object.

example:

```
class A
{
    int x;
    protected void finalize()
    {
        S.O.P ("inside finalize");
    }
}
class Demo6
{
    public static void main (String args[])
    {
        new A();
        new A();
        new A();
        System.gc();
    }
}
```

5. getclass
6. wait()
7. wait(int)
8. notify()
9. notifyall()
10. clone()
11. wait(int, int)

Monday
22/07/2013

Generics

* Generics :-

- It is a new feature added in ~~used~~ Java 5.0.
- Generics are used for developing resuable classes and methods.
  - Generic class
  - Generic method
  - Generic constructor
  - Generic interface.
  - Generic enum.

Note:

It is called templates in C++.

- Generics programming means to write code than can be reused for object of many different types.
- Generics are parameterized types, which receives types as an argument.
- Generics are applied to only reference types.
- It is used for building classes, methods, constructors.

Advantage:

1) Code resuability.
2) rapid application development.
3) Type safe.
4) Not required any type casting.

* Generic class:

- Also called container class
- Generic class is a parameterized class, which receives one or more than one type.

Syntax:

```
class <class-typ-name> <typename,
                        typename...>
{
    variables;
    methods;
}
```

e.g:

```
class Stack <T>
{

}
```

where T - type
E - element type
K - key type
V - value type.

```
class A <T1, T2>
{

}
```

example:

```
Class Array <E>
{
    E a;
}
```

- type must be defined at the time of creating object.

Syntax for creating object of generic class:-

```
<class name> <type> reference name = new
                        <class-name> <type> ();
```

exam

exam

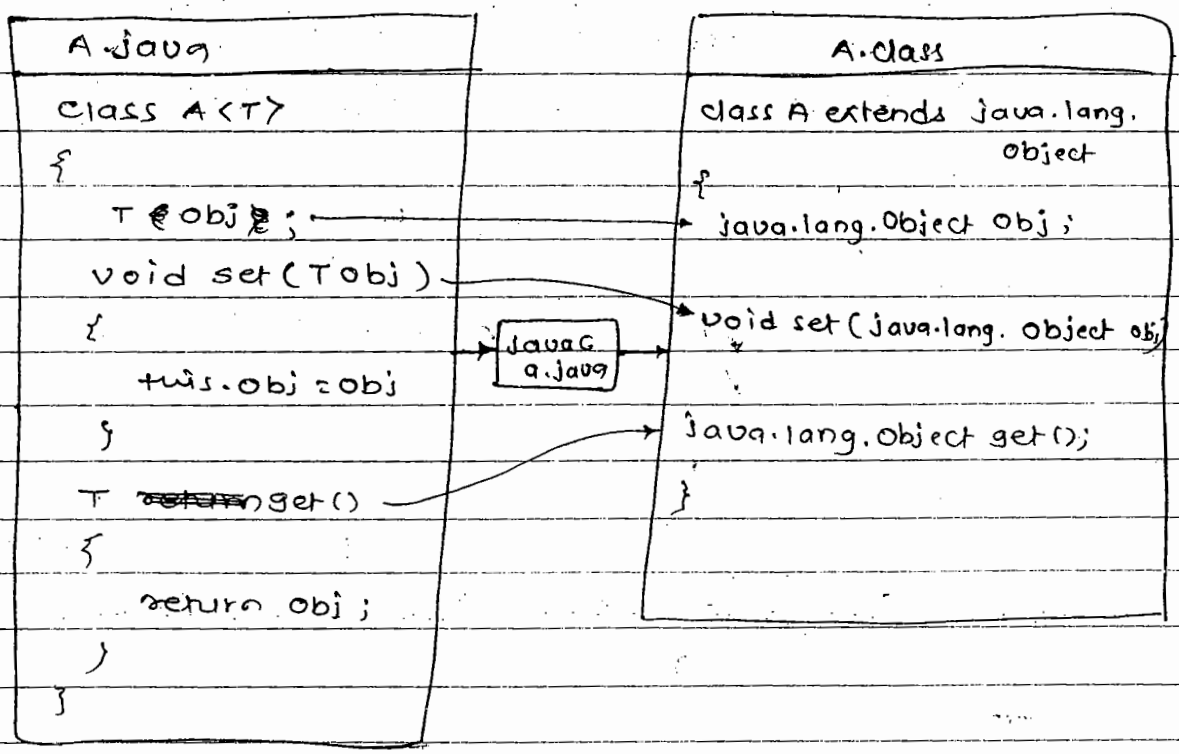
example: Stack<Integer> a = new Stack<Integer>();
A<Float> x = new A<Float>();

A<Integer> a = new A<~~Integer~~ Float>(); —⊗
A<Number> a = new A<Integer>(); —⊗
A<Integer> a = new A<Integer>(); —✓.

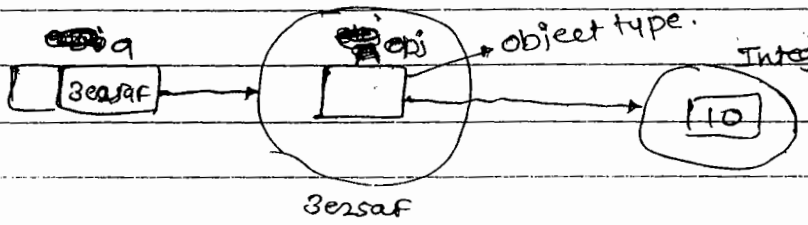
example:

```
class A<T>
{
    T obj;  → reference variable of type T.
    void set(T obj)
    {
        this.obj = obj;
    }
    T ~~void~~ get()
    {
        return obj;
    }
}
class GDemo1
{
    public static void main(String args[])
    {
        A<Integer> a = new A<Integer>();
        A<Float> b = new A<Float>();
        a.set(new Integer(10));
        b.set(new Float(1.5f));
        or get
        Integer x = a.get();
        Float y = b.get();
        System.out.println(x);
        System.out.println(y);
    }
}
```

Java p A

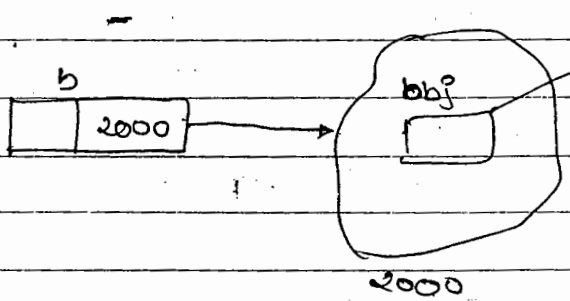


A<Integer> a = new A<Integer>();



~~a.set~~ a.set(new Integer(10)); ✓

a.set(new Float(1.5F)); ✗ CE



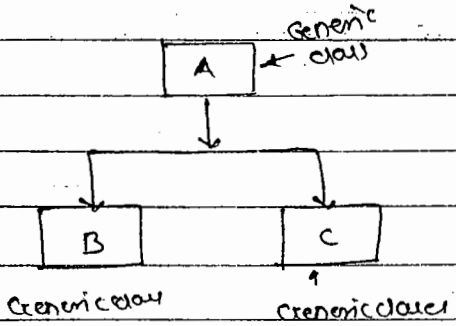
Tuesday
23/07/2013

* Generics and subtyping:

1) There is no inheritance relationship bet^n generic arguments of a generic class.

e.g: A<Number> obj = new A<Integer>(); // invalid.

2) Inheritance relationship between generic classes themselves still exists.



A<Integer> obj = new B<Integer>(); ✓

A<Integer> obj = new B<Integer>(); ✓

Collection<Integer> c = new ArrayList<Integer>(); ✓

ArrayList<Integer> a = new ArrayList<String>(); ✗

3) ~~Inheritance relationship between~~
3) Entries in Generic class maintain inheritance relationship.

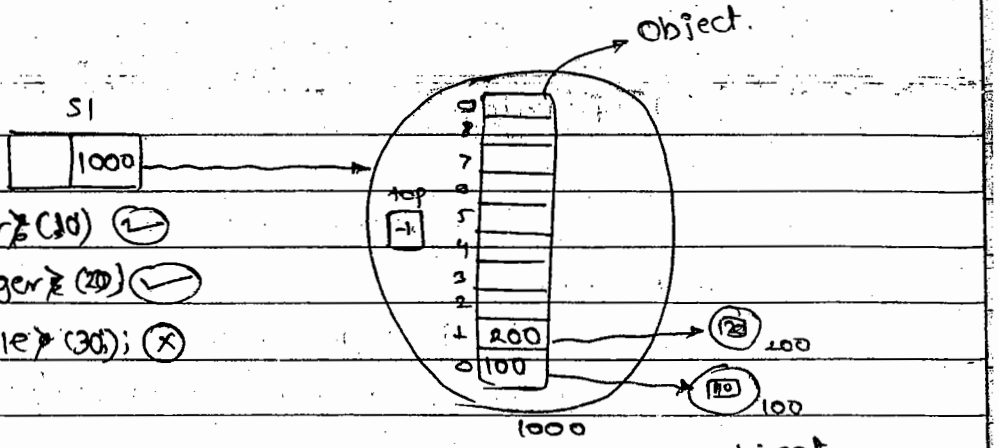
e.g:

A<Number> an = new A<Number>();
an.set = new<Integer>();

example:

```
Class Stack<T>
{
    Object o[] = new Object[10];
    int top;
    Stack()
    {
        top = -1;
    }
```

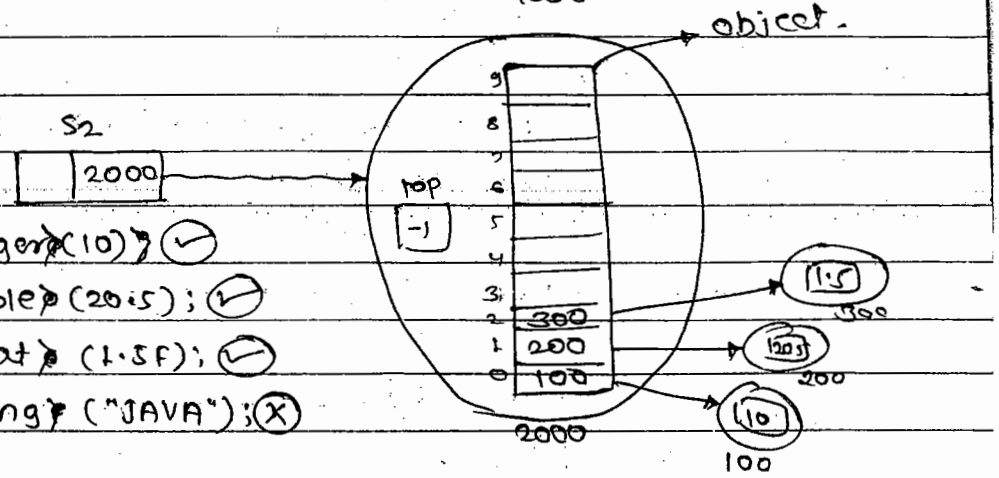
```
void push (T obj)                                          S1.
{                                                          S1.
    if (top>9)                                             S1.
    {
        System.out.println ("stack overflow");
    }
    else
    {
        o[++top]=obj;
    }
}                                                          S2.
void pop()                                                 S2.1
{                                                          S2.1
    if (top<0)                                             S2.
    {return null;
    else
    return o[top--];
    }
}

Class Demo3.
{
    public static void main (String args[])
    {
        Stack <Integer> s1 = new Stack <Integer> ();
        Stack <Number> s2 = new Stack <Number> ();

        S1.new push(
        s1.push (new Integer(10));
        s1.push (new Integer(20));
        System.out.println (s1.pop());
        System.out.println (s1.pop());
```



S1. push (new Integer (10)) ✓

S1. push (new Integer (20)) ✓

S1. push (new Double (30.5)); ✗



S2. push (new Integer (10)); ✓

S2. push (new Double (20.5)); ✓

S2. push (new float (1.5f)); ✓

S2. push (new String ("JAVA")); ✗

```
        S2. push (new Integer (10));
        S2. push (new float (1.5f));
        S2. push (new Double (10.5));
        System.out.println (S2.pop());
        System.out.println (S2.pop());
        System.out.println (S2.pop());
    }
}
);
();
```

* Wildcards:-

- Wildcards are used for creating generic references.
- This reference can be used for manipulating any type of object.

24/10/12

Problem:-

```
Stack<Number> s1 = new Stack<Integer>(); ⊗
Stack<Account> s2 = new Stack<SavingAccount>(); ⊗
Stack<Student> s3 = new Stack<MCAStudent>(); ⊗
```

- The above problem can be overcome using wildcards.

? → unknown type.

- Wildcard is defined using '?' symbol.

example:

```
class GenericDemo4
{
    static void print(A<?> obj)
    {
        System.out.println(obj.get());
    }
    public static void main(String args[])
    {
        A<Integer> o1 = new A<Integer>();
        A<Float> o2 = new A<Float>();
        A<Double> o3 = new A<Double>();
        o1.set(new Integer(10));
        o2.set(new Float(1.5F));
        o3.set(new Double(2.5));
```

```
print(o1);
print(o2);
print(o3);
}
}
```

```
T obj;
T obj[];
```
} valid.

```
T obj[] = new T[10];
T obj = new T();
```
} Invalid

24/07/2013

Bounded Wildcards:-

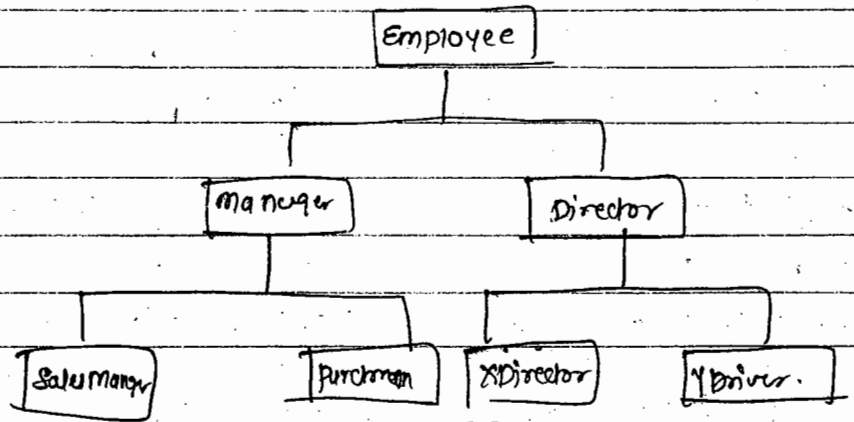
- wildcard which bind with specific type is called bounded wildcard.
- These are used for restricting type.
- wildcard is bind with specific type using two keywords.
  1. extends
  2. super

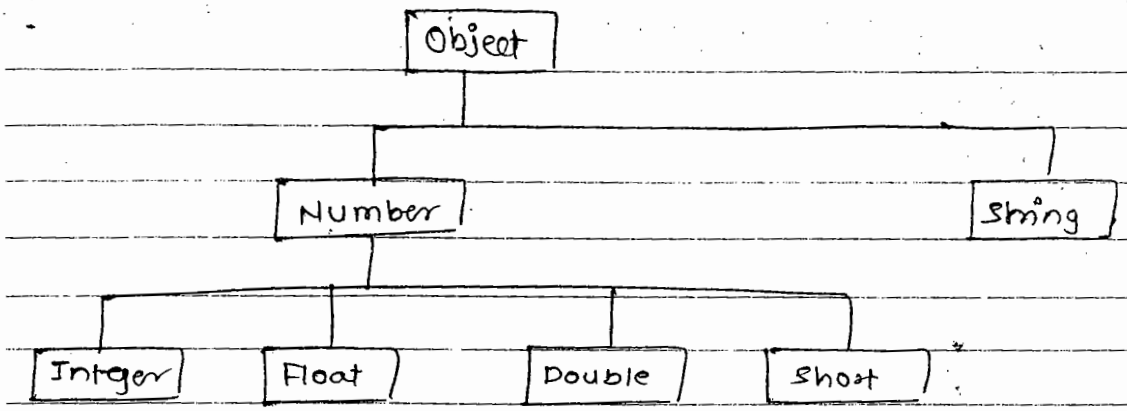
Syntax:

Syntax 1: ? extends type

Syntax 2: ? super type

1. If wildcard extends type, it allows all subtypes of given type.
2. If wildcard uses super type, it allows all super types of given type.



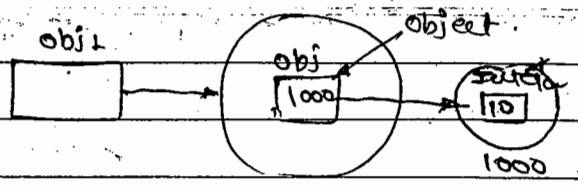


extends

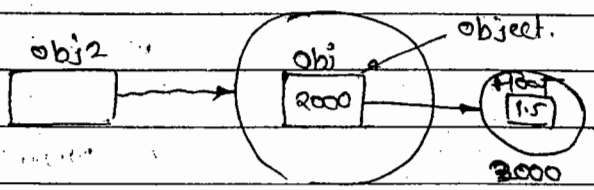
```
class A<T>
{
    T obj;
    void set(T obj)
    {
        this.obj = obj;
    }
    T get()
    {
        return obj;
    }
}
class GenericDemo5
{
    static void print(A<? extends Number> a)
    {
        System.out.println(a.get());
    }
    public static void main(String args[])
    {
        A<Integer> obj1 = new A<Integer>();
        A<Float> obj2 = new A<Float>();
        A<String> obj3 = new A<String>();
```

super

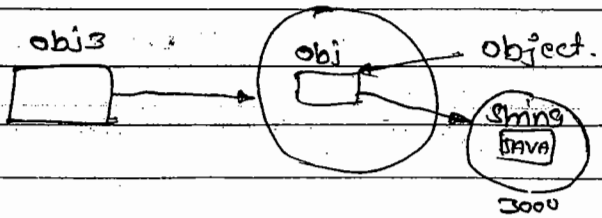
```
A<Integer> obj1 = new A<Integer>();
    obj1.set(new Integer(10)); ✓
    obj1.set(new Float(1.5F)); ✗
```




```
A<float> obj2 = new A<float>();
    obj2.set(new Float(1.5F)); ✓
    obj2.set(new Integer(10)); ✗
```




```
A<String> obj3 = new A<String>();
    obj3.set(new String("JAVA")); ✓
    obj3.
```




```
        obj1.set(new Integer(10));
        obj2.set(new Float(1.5F));
        obj3.set(new String("JAVA"));
        print(obj1);
        print(obj2);
        print(obj3); → gives CE.
    }
}
```

super

```
class GenericDemo6
{
    static void print(A<? super Integer> a)
    {
        System.out.println(a.get());
    }
```

exam

```
Public static void main(String args[])
{
    A<Integer> obj1 = new A<Integer>();
    obj1.set(new Integer(10));
    print(obj1);

    A<Number> obj2 = new A<Number>();
    obj2.set(new Float(1.5F));
    print(obj2);
}
}
```

* **Generic method :-**

- A method is parameterised with type.
- This type information is provided at the time of calling method
- It is a method which can be applied to various types (avoiding redundancy).

exam

Syntax :

```
[modifier] <type1, type2> <return type>
        <method-name> ([parameters])
{
    Statements;
}
```

```
static <T> void print(T obj)
{

}
```

Java compiler → compile →

```
static void print(Object obj)
{

}
```

example:

```
class GenericDemo7
{
    static <T> void compare(A<T> obj1, A<T> obj2)
    {
        System.out.println(obj1.get());
        System.out.println(obj2.get());
    }
    public static void main(String args[])
    {
        A<Integer> o1 = new A<Integer>();
        A<Integer> o2 = new A<Integer>();
        o1.set(new Integer(10));
        o2.set(new Integer(20));
        compare(o1, o2);
    }
}
```

example

```
class B
{
    static <T> void m1(T obj)
    {
        System.out.println(obj);
    }
}
class GenericDemo8
{
    public static void main(String arg[])
    {
        Integer x = new Integer(10);
        Float y = new Float(1.5F);
        Double z = new Double(2.5);
        B.<Integer>m1(x);   B.<Double>m1(2);
        B.<Float>m1(y);
```

25/07/2013

* Bounded Types:

- A type parameter inherits another type.
- A type parameter bind with another type.
- It is used for restricting types or allowing to use members of another type.

Syntax:

```
<type extends type>
<type super type>
```

example:

```
class Math
{
    static <T extends Comparable> T max (T o1, T o2)
    {
        if (o1.compareTo(o2)>0)
        return o1;
        else
        return o2;
    }
}
class GenericDemog
{
    public static void main (String args[])
    {
        Integer a = Math.<Integer> max (new Integer(10),
                                        new Integer(20));
        System.out.println (a);
        Float b = Math.<Float> max (new Float(1.5f),
                                    new Float(2.5f);
    {   System.out.println (b);
    }
}
```

A.class

```
class A                                  class A
{                                        {
  static<T> T m1(T 01,T 02)                static Object m1(Object 01,
  {                                        {                Object 02)
  }                                        }
}                          Java compiler
static<T extends Comparable>             static Comparable m2(
        T m2(T01, T 02)                          Comparable 01,
{                                                Comparable 02)
}                                        {
                                         }
```

- A type can extends one or more than one type.

example:

```
class Student
{
    void
abstract class Student
{
    abstract void print();
}
class MCAStudent extends Student
{
    void print()
    {
        System.out.println("MCA student");
    }
}
class MBAStudent extends Student.
{
    void print()
    {
        System.out.println("MBA student");
    }
}
```

exar

```
class Printer.
{
    static <T extends Student> void printStudent (T s)
    {
        s.print();
    }
}
Class GenericDemo10
{
    Public static void main (String args[])
    {
        Printer.<MCAStudent> new
        printer.<MCAStudent> print new (MCAStudent());
        Printer.<MBAStudent> print new

        Printer.<MCAStudent> printStudent (new MCAStudate());
        Printer.<MBAStudent> PrintStudent (new MBAStudent());
    }
}
```

* **Generic Constructor:-**

- A constructor is parameterized with a type.

Syntax:

```
<T> <constructor-name> ([parameters])
{
    Statements;
}
```

- Generic constructor are helpful in developing typesafe constructors.

example:

```
Class A<T>
{
    T obj1;
    T obj2;
    A(T obj1, T obj2) // generic constructor.
    {
        this.obj1 = obj1;
        this.obj2 = obj2;
    }
    T getObj1()
    {
        return obj1();
    }
    T getObj2()
    {
        return obj2();
    }
}
Class GenericDemo11
{
    public static void main(String args[])
    {
        A<Integer> objInt = new A<Integer>(new
                Integer(10), new Integer(20));
        A<float> objfloat = new A<float>(new
                Float(1.5F), new Float(2.5F));
        Integer n1 = objInt.getObj1();
        Integer n22 objInt.getObj2();
        Float n3 = objfloat.getobj1();
        Float n4 = objfloat.getobj2();
        System.out.println(n1);
        System.out.println(n2);
        System.out.println(n3);
        System.out.println(n4);
    }
}
```

26/07/12

example:

```
class A
{
    <T extends Comparable> A (T obj)
    {
        System.out.println(obj);
    }
}
class GenericDemo12
{
    public static void main (String args[])
    {
        A obj1 = new A (new Integer(10));
        A obj2 = new A (new Float(1.5F));
    }
}
```

26/07/2013

* Generic Interface:

- An interface is parameterized with type.
- This interface is implemented by concrete class.

Syntax:

```
interface <interface-name> <type1, type2>
{
    abstract methods;
    constants;
}
```

o/p
10
1.5

example:

```
interface A<T>
{
    void set(T e);
    T get();
}
```

exa

(26)

new = class is known at compile time
class.forname = runtime.

(27)

26/07/2013

```
class C<T> implements A<T>
{
    T obj;
public void set (T e)
    {
        obj = e;
    }
Public T get()
    {
        return obj;
    }
}
class GenericDemo13
{
    public static void main (String args[])
    {
        C<Integer> cobj1 = new C<Integer>();
        C<Float> cobj2 = new C<Float>();
        cobj1. ~~new~~ set (new Integer (10));
        cobj2. set (new Float (1.5F));
        Integer a = cobj1. get();
        Float b = cobj2. get();
    }
}
```

er(10));

op
10
1.5

* Generic class extends another generic class:

Example:

```
class A<T>
{
    void print1 (T obj)
    {
        System.out.println (obj);
    }
}
```

```
class B<T1,T2> extends A<T1>
{
    void print2(T2 obj)
    {
        System.out.println(obj);
    }
}
class GenericDemo14
{
    public static void main(String args[])
    {
        B<Integer,Float> obj1 = new B<Integer,Float>();
        obj1.print1(new Integer(10));
        obj1.print2(new Float(1.5));
    }
}
```

op
10
1.5

* Generic class extends class

example:-

```
class A
{
    void print1(Integer a)
    {
        System.out.println(a);
    }
}
class B<T> extends A
{
    void print2(T b)
    {
        System.out.println(b);
    }
}
```

op 10
2

```
class GenericDemo15
{
    public static void main (String args[])
    {
        B<Float> objb = new new B<Float> ();
        Objb.print1(new Integer(10));
        Objb.print2(new Float(1.5f));
    }
}
```

op
10
1.5

* Class extends Generic class.

```
Class A<T>
{
    void print1 (T obj)
    {
        System.out.println(obj);
    }
}
class B extends A<Integer>
{
    void print2(Integer obj)
    {
        System.out.println(obj);
    }
}
Class GenericDemo16
{
    public static void main (String args[])
    {
        B objb = new B();
        objb.print1(new Integer(20));
        objb.print2(new Integer(20));
    }
}
```

op 10
20

29/07

* Generic enum :-

- enum is new feature added java 5.0.
- enum is a collection of constants.

Syntax:

```
enum <enum-name> <type1, type2>
{
    constants;
    methods;
}
```

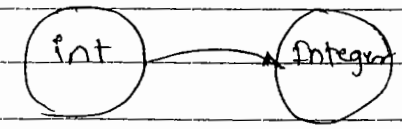
* Autoboxing and Autounboxing :

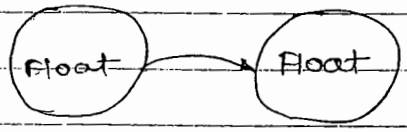
- Autoboxing :-

- Converting of premitive datatype to Reference datatype is called Autoboxing.
- This convension between similar types.

Java 1.4

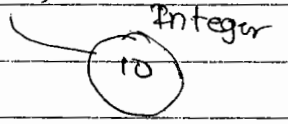
Integer a=10 ✗





Java 5.0

Integer a=10; ✓



Integer a=1.5; ✗

Integer a=1.5F; ✗

- Autounboxing :-

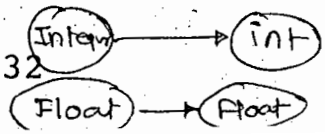
- Converting of reference type to primitive type is called autounboxing.
- This convension between similar types.

Java 1.4

```
Integer a = new Integer();
int b=a;  ---> ✗
```

Java 1.5

```
Integer a=new Integer();
int b = a;  ---> ✓
```



29/07/2013

* **Array :-**

- Array is reference datatype.
- Array is a collection of similartype data elements.
- Array is an object.
- For array superclass is Object.

- **Advantage:-**

1) Grouping & refering more than one value with one name.
2) Avoiding declaring of number of variable.
3) It is a container which can be used to pass a data one place to another place.
4) Allows sequential and random accessing.

- **Disadvantages:-**

1) There is no predefined method support to manipulate data exist with in array.
2) There is no underlying datastructure for array.
3) Arrays are fixed in size / once array is created the size of this array cannot increase or decreae. because of that there is a wastage of memory.

- **SingleDimensional Array :-**

- An array with one subscript is called single Dimensional array.
- Arrays are dynamic in java & created using new operator.

1. Delcare Array
2. Create Array

- When array is declared memory is allocated for reference variable.

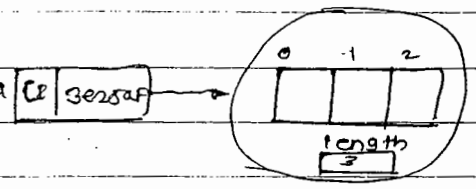
Syntax: datatype arrayname[];

e.g;

int a[];

Syntax: new datatype [size];

int a = new int[3];



int a[] = new int[3];
int b[] = new int[3];
boolean x = a.equals(b); (F)

example:-

```
class Arraydemo1
{
    public static void main (String args[])
    {
        int a[] = new int[10];
        int b[] = new int[5];
        System.out.println(a);
        System.out.println(b);
        System.out.println(a.toString());
        System.out.println(b.toString());
    }
}
```

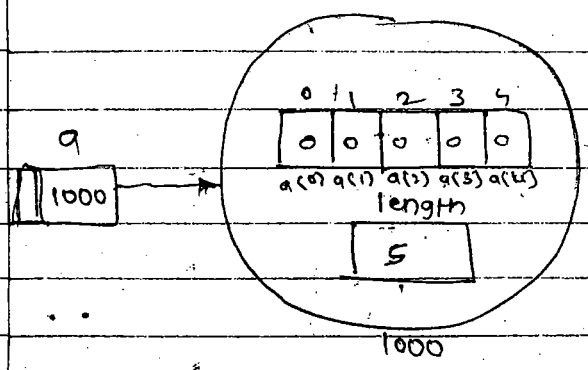
```
class Arraydemo2
{
    Public static void main (String args[])
    {
        int a = new int[5];
        int b = new int[5];
        boolean x = a.equals(b);
        int c[] = b;
```

a
1000

```
boolean y = b.equals (c);
System.out.println (a);
System.out.println (b);
System.out.println (c);
System.out.println (x);
System.out.println (y);
}
}

// storing & reading elements from array.
class ArrayDemo3
{
    Public static void main (String args[])
    {
        int a[] = new int[5];
        System.out.println (a.length);
        for (int i=0; i<a.length; i++)
        System.out.println (a[i]);
    }
}
```

* Enhanced for loop (Foreach loop):-

- Introduced in java 5.0.
- TO read element from array this loop is used.



Syntax:

```
for (variable : array)
statement;

for (variable : array)
{
    statements;
}
```

30/07

```
// program to find sum of elements
class ArrayDemo5
{
    public static void main (String args[]);
    {
      int a[] = {10,20,30,40,50};
      int sum=0;
      for (int n : a)
      sum=sum+n;
      System.out.printfn ("sum is %d", sum);
    }
}
```

* Creating an array of reference type:-

- An array of type primitives hold values.
- An array of type reference hold objects. exam

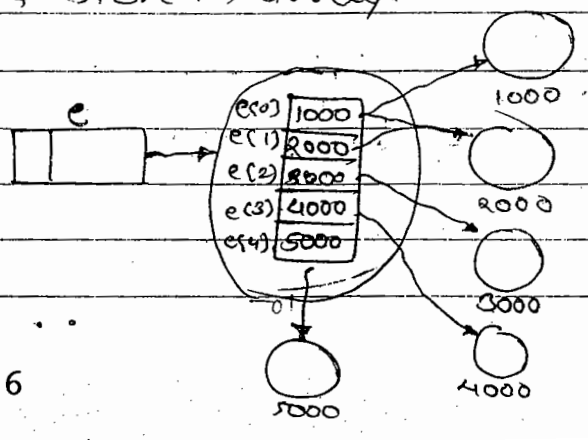
```
int a[]  // primitive type.
Employee e[]  // reference type.
Object o[]; // An array of Object.
T o[]; // an array of Object.
```

- In order to create an array of objects.

1. Declare array of reference type.
2. Create array.
3. Create object & store in array.

example:

```
Employee e[]
e=new Employee(5);
e[0]=new Employee();
e[1]=new Employee();
e[2]=new Employee();
e[3]=new Employee();
e[4]=new Employee();
```



Cannot create generic array.

30/07/2013

```
class Array <T>
{ int index;
  Object o[];
   Array()
   {
      o = new Object[5];
   }
   void add(T obj)
   {
      o[index] = obj;
   }
   T get()
   {
      return o[index]
```

```
T a[] = new T[10] //error
T a[] = new Object[10] //error
Object o = new Object[10] //✓
T a[]; // valid.
```

---

example:-

```
class Array<E>
{
   Object o[] = new Object[10];
   void set(E obj, int index)
   {
      o[index] = obj;
   }
   E get(int index)
   {
      return (E) o[index];
   }
}
class App
{
   Public static void main(String args[])
   {
      Array<Integer> a1 = new Array<Integer>();
```

30/07

```
ArrayList<Integer> a2 = new ArrayList<Integer>();
ArrayList<Integer> a3 = new ArrayList<Integer>();
a1.set(10,0);
a1.set(20,1);
a1.set(30,2);
System.out.println(a1.get(0));
System.out.println(a1.get(1));
System.out.println(a1.get(2));
Integer a = a1.get(0);
Integer b = a1.get(1);
Integer c = a1.get(2);
System.out.println(a);
System.out.println(b);
System.out.println(c);
}
}
```

O/P:

```
10
20
30
10
20
30
```

Que: How to verify upperbound and lowerbound in generics?

OR.

How bound checking is done in generices.

⇒

Using two keywords.

1. extends.
2. super.

30/07/2013

Collection Frameworks

(Data structures in java or java.util).

Q: What is Datastructure?

Datastructure :-

- Datastructure define set of rules and regulation for organization of data in memory.
- Datastructure define a method or algorithm in order to organize data in memory.

Q: ~~If~~ it every collection is datastructure?

⇒ Yes, every collection is having underlying datastructure.

Q: What is collections?

Collections :-

- A collection is "Object", sometimes called a container is simply an object that groups multiple elements into a single unit.
- collection are used to store, ~~retrieve~~ retrieve manipulate and communicate aggregate data.
- Typically, they represent data items that form a natural group, such as poker hand (a collection of cards), a mail folder (collection of letters) or telephone directory (a mapping names to phone numbers.

- collections are implemented using two methods
  1. Arrays
  2. Linked list.

31/07/2013

* Collection Framework :-

- Collection frameworks define a group of classes and interfaces which can be used for representing group of objects as a single entity.

(OR)

- A collection framework is a unified architecture for representing and manipulating collections.

- All collection frameworks contains;
  - Interfaces :
  - Implementations.
  - Algorithms.

- Collection framework is introduced ~~in~~ in Java 1.2.

- Before 1.2 version collections are called legacy collection classes.

Que: What is Diff. bet'n collection frameworks before 1.5 & after 1.5/5.0 onwards.

Before 1.5/5.0	1.5/5.0 onwards.
1) There are not generic collections. (Classes & interfaces not implement generics).	1) There are generic collections. (each class & interface available is parameterized or implements generics).
2) Not type safe collections	2) Type safe collections.
3) There doesn't support autoboxing & autounboxing.	3) There support autoboxing & autounboxing.
4) Typecasting required	4) typecasting not required.

* Benefits of framework collections:-

1) Reduces programming efforts.

2) Increases programs speed & Quality.

3) Allow interoperability among unrelated APIs.

- The collection interfaces are the vernacular by which APIs pass collections back & forth.

4) Reduce effort to learn & use new APIs.

5) Reduces effort to design new APIs.

6) Fosters software reuse.

- New datastructure that conform to the standard collection interfaces are by nature reusable.

1) Interfaces:

- collection interfaces are abstract data types that represent collections.

- collection interfaces are in the form of java interfaces.

- Interfaces allows collection to be manipulated independently of the implementation details of their representation. (polymorphic behaviour).

- In java programming language interfaces generally form a hierarchy.

2) Implementations:-

- These are the concrete implementations of the collection interfaces.

3) Algorithms:-

- These are the methods that perform useful computations such as searching & sorting, on objects that implement collection interfaces.

- The algorithm are said to be polymorphic, i.e.

same method can be used on many different implementations of the appropriate collection interface.
- Algorithms are reusable functionality.

* Types of Implementation:-
  - General-purpose Implementations
  - Special-purpose Implementations
  - concurrent Implementations
  - Wrapper Implementations
  - Convenience Implementation
  - Abstract Implementation.

04/08/2013

- Implementations are the data objects used to store collections which implements the interfaces.

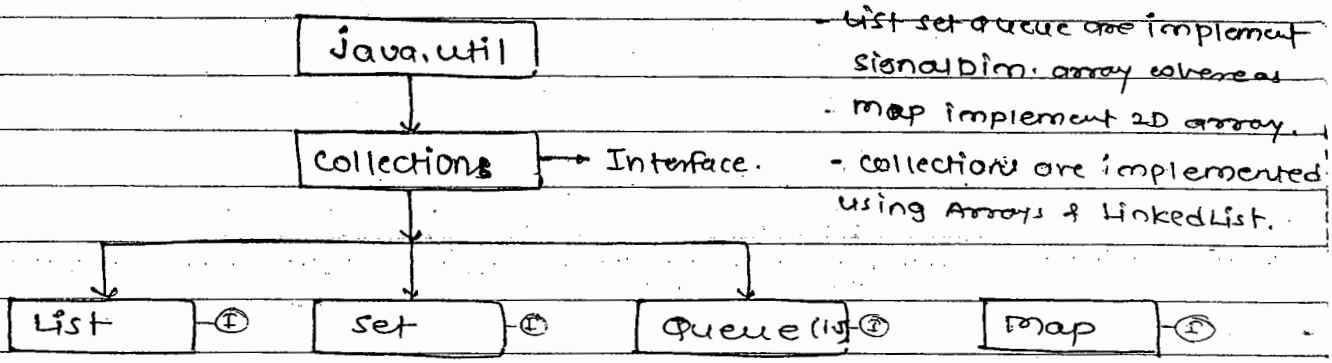
* Types of Collections:-
  1) List
  2) Set
  3) Queue
  4) Map.

- This classification is done based on underlying datastructure.

Arrays	Collections
1) Fixed in size.	1) Dyamic in size(size of collection can be increase/decrease during runtime.
2) No underlying datastructure	2) Have underlying datastructures.
3) Hold primitive & Referenece type	3) Hold only Reference types.
4) Hold similar type (homogenous)	4) Heterogenous. (Hold different

5. There is no predefined method support to manipulate data.	5) There is predefined method to manipulate data.

* **Collection Hierarchy :-**



- map - 2D array
- List - 1D array
- list set queue are implement signal dim. array whereas
- map implement 2D array.
- collections are implemented using Arrays & LinkedList.

* **Collection :-**

- Collection is root interface for all collections.
- The root of collection hierarchy.
- Is the least common ~~demo~~ denominator that all collection implement
- Every collection object is a type of collection interface.
- It is used to pass collection object around & to manipulate them when maximum generality is desired.
- JDK doesn't provide any direct implementation of this interface but provides implementation of more specific sub interfaces. such as Set and List.

Methods of Collection interface :-

- **boolean add (E e) :**

Ensures that this collection contains the specified element. 43

- boolean addAll (Collection <? extends E> c)
  - add all of the elements in the specified collection to this collection.

- void clear() :
  - Removes all of the elements from this collection.

- boolean contains (Object o)
  - Returns true if this collection contains the specified elements.

- boolean containsAll (Collection <?> c)
  - Returns true if this collection contains all of the elements in the specified collection.

- boolean equals (object o)
  - compares the specified object with this collection for equality.

- int hashCode () :
  - Returns hash code value for this collection.

02/08/2013

- boolean isEmpty () :
  - Returns an iterator over the elements in this collection

- Iterator <E> iterator() :
  - Returns an iterator over the elements in this collection.

- boolean remove (Object o)
  - Removes a single instance of the specified element from this collection if it present.

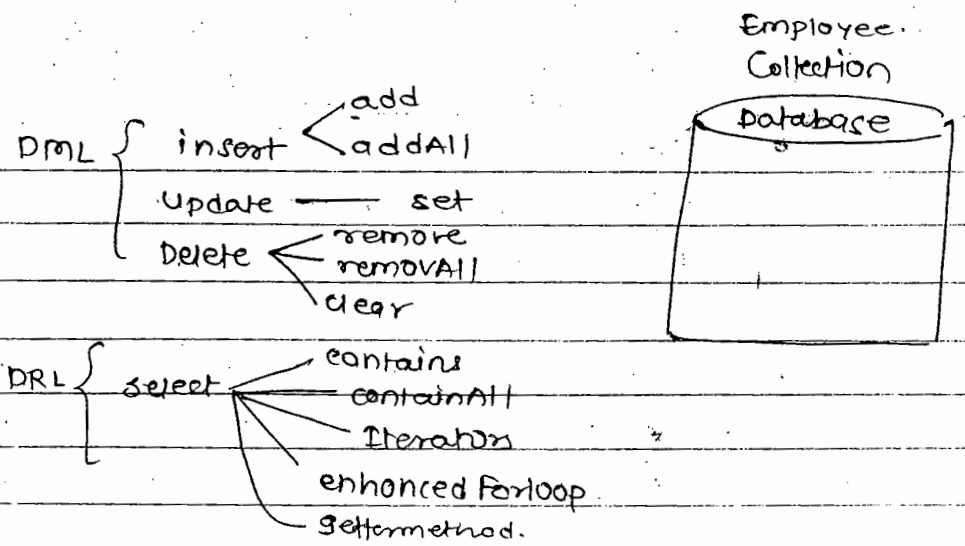
- boolean removeAll (Collection <?> c)
  - Removes all of this collection's elements that are also contained in the specified collection.

- boolean retainAll (Collection <?> c)
  - Retains only the elements in this collection that are contained in the specified collection (optional operation).

- int size()
  - Returns the number of elements in this collection.

- Object[] toArray()
  - Returns an array containing all of the elements in this collection.

- <T> T[] toArray (T[] a)
  - Returns an array containing all of the elements in this collection ; the runtime type of the returned array is that of the specified array.



02/08/2013

*1) List:

- An ordered collection (also known as a sequence).

- The user of this interface has precise control over where in the list of each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.

- List is a interface.

- List extends Collection.

- **void add (int index, E element)**

  - Insert the specified element at the specified position in this list.

- **E get (int index)**

  - Returns the element at the specified position in this list.

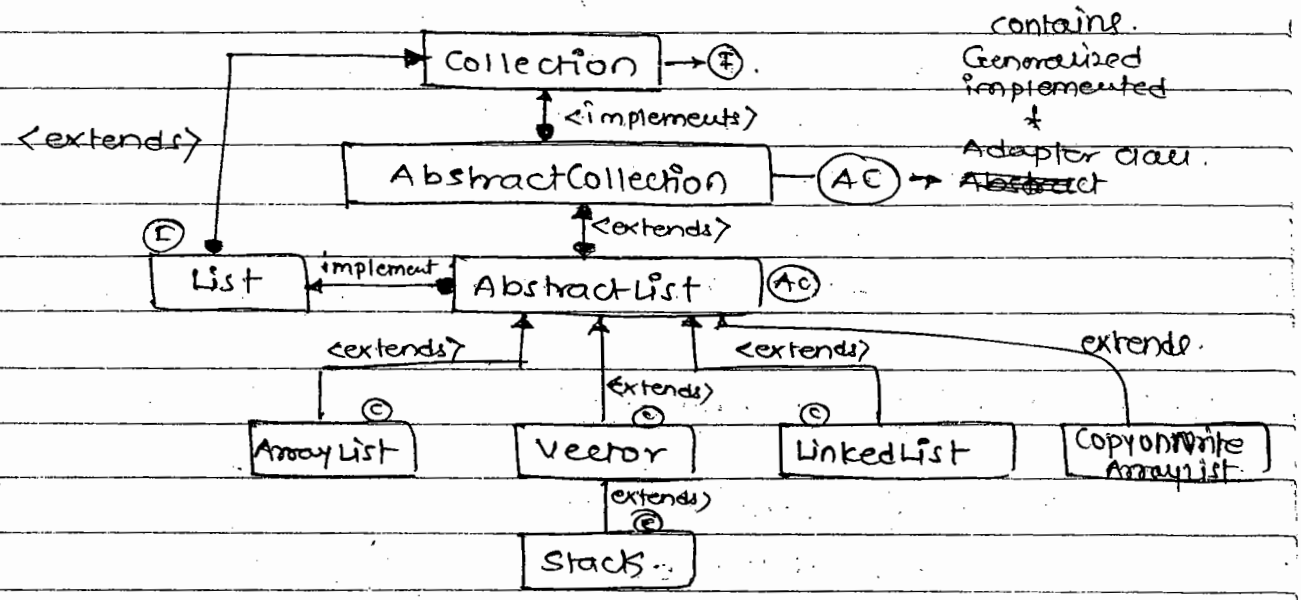
- **int indexOf (object o)**

  - Returns the index of first occurance of the specified element in this list, or -1 if this list does not contain the element.

- E remove (int index)
  - Removes the element at the specified position in this list.

- E set (int index, E element)
  - Replaces the element at the specified position this list with specified element.

- List<E> subList (int fromIndex, int toIndex)
  - Returns a view of portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.



Example: Class ArrayList extends AbstractList implements List.

Note:-

- Every collection is serializable and cloneable.
  (persited) (allow to create copy)

A) ArrayList:

public class ArrayList<E> extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable.

08/08/20

- Resizable-array implementation of the List interface. Implements all optional list operations & permits all elements, including null.
- In addition to implementing the List interface, this class provides methods to manipulate the size of the array i.e. used internally to store the list.

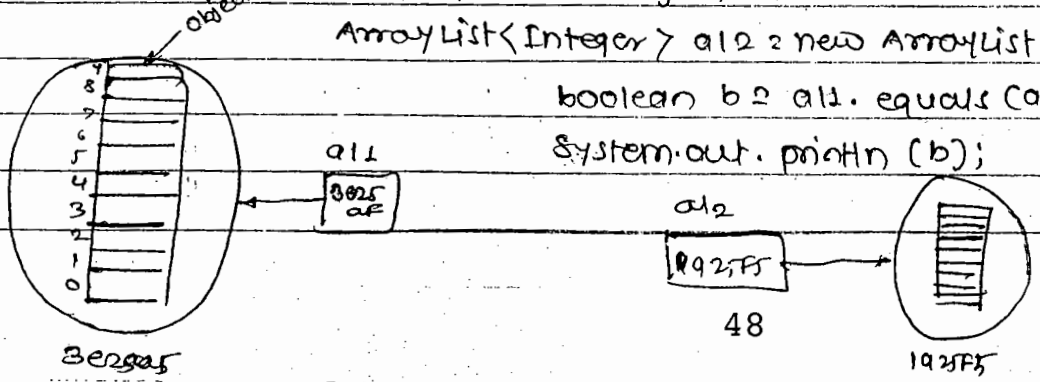
Properties / characteristics List Collection:

1) ordered collection
2) Index based collection
3) Allows duplicates
4) Allow null
5) Allows sequential access & random access.

O/P
false
E

Example:- Working with ArrayList:

```
import java.util.*;
class ArrayListDemo
{
    public static void main(String arg[])
    {
        ArrayList<Integer> al1 = new ArrayList<Integer>();
        ArrayList<Integer> al2 = new ArrayList<Integer>();
        boolean b = al1.equals(al2);
        System.out.println(b);
```



Overriding equals method comparing types.
In abstract class

O/P true

```
ArrayList<Integer> al3 = al1;
boolean b1 = al1.equals(al3);
System.out.println(b1);
}
}
```

08/08/2013

```
import java.util.*
class ArrayListDemo2
{
    public static void main(String args[])
    {
        ArrayList<Integer> a1 = new ArrayList<Integer>();
        HashSet<Integer> b2 = new HashSet<Integer>();
        boolean b = a1.equals(b2);
        System.out.println(b);
    }
}
```

O/P
false

Explanation:

- The O/p of above program is false, becoz equals method of List compare if this collection type is equal to any other collection type.

Working with ArrayList capacity:-

- Initial capacity of Arraylist is 10.
- This capacity incremented dynamically.

new capacity = ((old capacity * 3) / 2) + 1

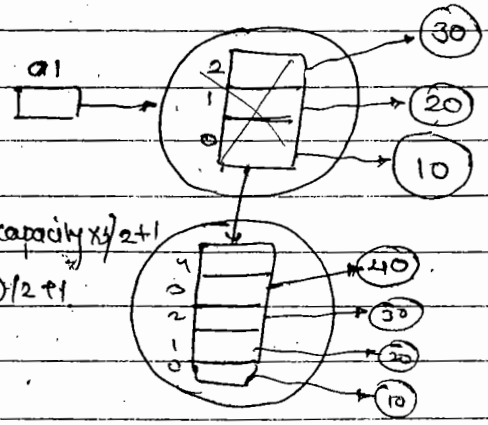
new capacity = (10 * 3) / 2 + 1

= 16.

ArrayList a1 = new ArrayList(5); ... 5.

ArrayList a2 = new ArrayList(); 10

```
ArrayList<Integer> al = new ArrayList<Integer>(3);
    al.add(10);
    al.add(20);
    al.add(30);
    al.add(40);
```



exa

* Methods of ArrayList:-

- int size();
  - Returns the number of element in this list.
- void trimToSize()
  - Trims the capacity of this ArrayList instance ~~to~~ be the list's current size.

exam

Capacity	Size.
1) How many objects can hold by collection or length of collection	1. count of objects exist in collection.

- ArrayList doesnot provide any methods to display capacity.

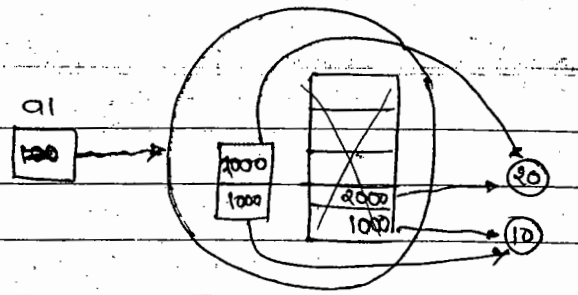
example:

```
import java.util.*;
class ArrayListDemo3
{
    public static void main(String args[])
    {
        ArrayList<Integer> al=new ArrayList<Integer>(5);
        al.add(10);
        al.add(20);
        System.out.println(al.size()); → 2.
```

al.tost
that-a

ou

```
        al.trimToSize();
    }
}
```



example: Reading element from ArrayList

- The contents of collection is read by using 3 methods

1) getter methods provided by collection
2) enhanced for loop.
3) cursor.

1) **Getter Method:** → randomly searching reading.

- List provide only one getter method.
- List is index based and read contents using index

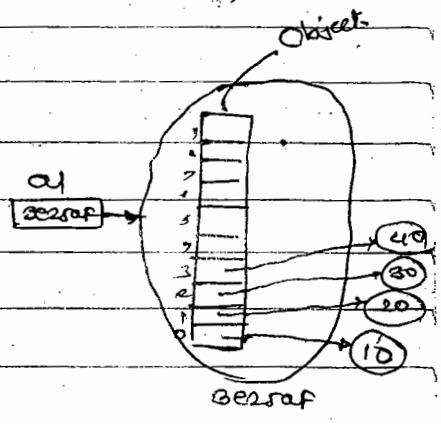
syntax: E get (index)

example:

- This getter is used to read elements of ~~set~~ list sequentially and randomly.

```
import java.util.*;
class ArrayListDemo4
{
    public static void main (String args[])
    {
        ArrayList<Integer> al = new ArrayList<Integer>();
        al.add(new Integer(10));
        al.add(20);  al.add(40);
        al.add(30);
        System.out.println(al);
    }
}
```

al.toString() / that way it display value



op 10
20
30
40

05/08

example

```
import java.util.*
Class MyInteger
{
  private int n;
  MyInteger (int n)
  {
    this.n=n
  }
}
```

```
class ArrayListDemo5
{
  Public static void main (String args[])
  {
    ArrayList<MyInteger> al=new
    ArrayList<MyInteger> ();
    al.add(new MyInteger(10));
    ArrayList<MyInteger> al=new
    ArrayList<MyInteger> ();
    al.add(new MyInteger(20));
    S.O.P.(al)
  }
}
```

```
/* public String toString()
{
  return String.valueOf(n);
} */
```

o/p 1000
2000
3000

10
20

example: Reading element using get();

```
import java.util.*;
class Employee
{
  private String name;
  private float salary;
  Employee(String name, float salary)
  {
    this.name=name;
    this.salary=salary;
  }
  public String toString()
  {
    return name+" "+salary;
  }
}
```

```
class ArrayListDemo6
{
  public static void main(String args)
  {
    ArrayList<Employee> al=new
    ArrayList<Employee> ();
    al.add(new Employee("ABC",10000F);
    al.add(new Employee("XYZ",12000F);
    al.add(null); al.add(new Employee("PQR",10000F);
    Scanner sc=new Scanner(System.in);
    System.out.println("Input empno");
    int empno=sc.nextInt();
    Employee e=al.get(empno);
    if(e==null)
    S.O.P("Invalid empno");
    else
    S.O.P.(e);
  }
}
```

05/08/2013

* Reading element using enhanced for loop:-

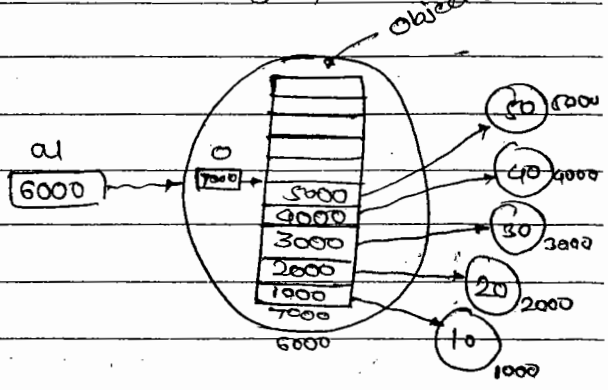
- This for loop is introduced in Java 5.0.
- It is reading elements from array. It is used with collections which implements array.

Syntax:

```
for (variable : array / collection)
{
    statements;
}
```

example: - It is used for reading elements in sequential order.

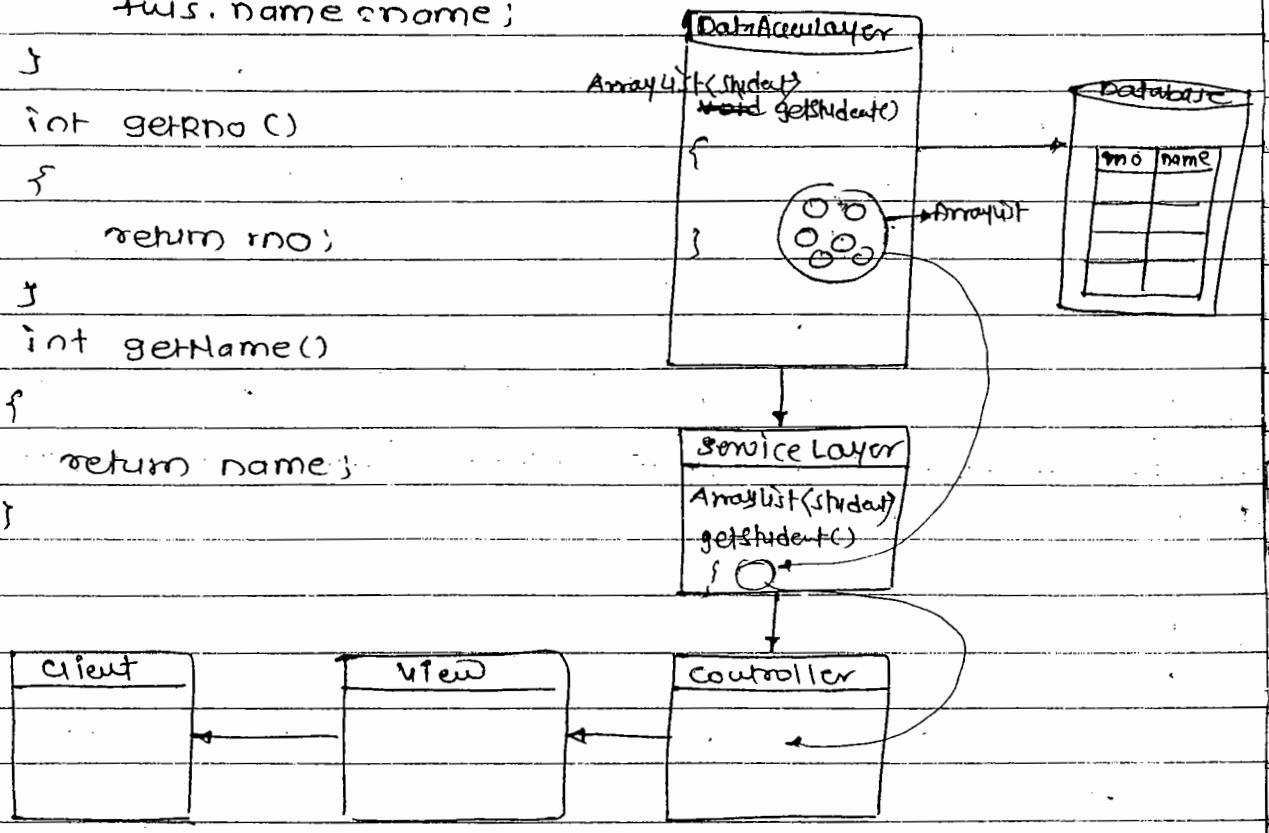
```
import java.util.*;
class ArrayListDemo7
{
    public static void main (String args[])
    {
        ArrayList <Integer> al = new ArrayList <Integer> ();
        al.add (10);
        al.add (20);
        al.add (30);
        al.add (40);
        al.add (50);
        for( Integer x : al)
        System.out.println (x);
    }
}
```



* Reading userdefined objects from Arraylist

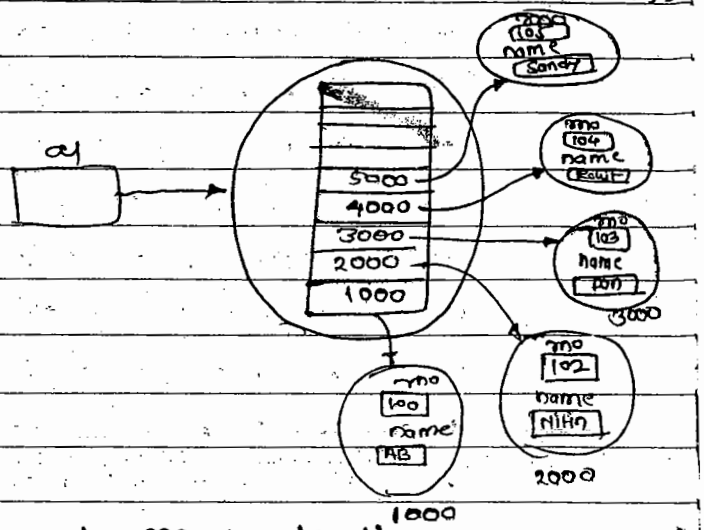
```
import java.util.*;
class Student
{
    private int rno;
    private string name;
    void setRno(int rno)
    { this.rno=rno; }
```

```
void setName(String name);
{
    this.name=name;
}
int getRno()
{
    return rno;
}
int getName()
{
    return name;
}
}
```




```
class ArrayListDemo8
{
    public static void main(String args[])
    {
        ArrayList<Student> al = new ArrayList<Student>();
        // Student S1 = new Student();
        // Student S2 = new Student();
        // Student S3 = new Student();
        Scanner sc = new Scanner(System.in);
        For(i=1; i<=5; i++)
        {   S.O.P("Enter Rollno :")
            int r = sc.nextInt();
            S.O.P("Enter Name:");
            String n = sc.next();
            Student s = new Student();
            s.setRno(r); s.setName(n); al.add(s);
        }
```

```
for (Student stud:al)
    System.out.println (stud.getRno() +" "+ stud.getName());
}
}
```



* Cursor:-

- Cursor is pointer to memory location.
- Cursor is holding reference of collection.
- Cursor is object wrapped with reference of collection

Types of cursors:-

1. Iterator (Interface)
2. ListIterator (Interface)
3. Enumeration (Interface)

1) Iterator:-

- public interface Iterator<E>

- An Iterator over a collection. Iterator takes the place of Enumeration in java collections framework. Iterator different from enumeration in two ways:

1) Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.

2) Method names have been improved.

- This interface is a member of Java Collection Framework since 1.2.

Que: Why Collection doesnot extends cloneable & serializable ?

⇒ - many collection implementation (including all of the ones provided by JDK) will have a public clone method

but it would be mistake to require it of all collections. For example, what does it mean to clone a Collection that's backed by a terabyte SQL database? Should method call cause the company to requisition a disk farm? Similar arguments hold for ~~for~~ serializable. If the client doesn't know actual type of collection. It's much more flexible and less error prone to have the client decide what type of Collection is desired, create an empty Collection of this type, & use the addAll method to copy the elements of the original collection into the new one.

Iterator:

- It is Forward only cursor. It allows to read contents of collection in Forward direction.
- It is updateable cursor, allows to perform remove operation.

O/P
10
20
30
40

How to get Iterator object?

- Collection object provide Iterator() method, which returns Iterator object.
- Iterator is an interface and implemented as inner class within Collection object.

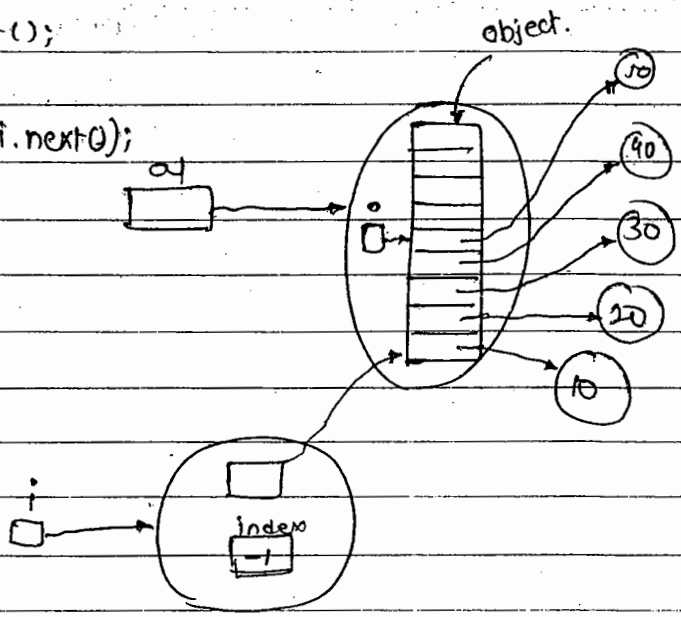
Where it is implemented?

-

```
public class AbstractList
{
    private class Itr implements Iterator
    {
    }
    public Iterator iterator()
    { return new Itr(); }
}
```

```
import java.util.*;
class ArrayListDemo9
{
    public static void main (String args[])
    {
        ArrayList <Integer> al = new ArrayList <Integer> ();
        al.add(10);  // al.add(40);
        al.add(20);  // al.add(50);
        al.add(30);
        Iterator <Integer> i = al.iterator ();
      // Integer x = i.next();
      //  SOP(x);
        System.out.println (i.next());
      // int y = i.next();
      // SOP(y);
        S.O.P (i.hasNext);
    }
}
```

O/P
10
20
30
true



* Reading user defined objects using iterator.

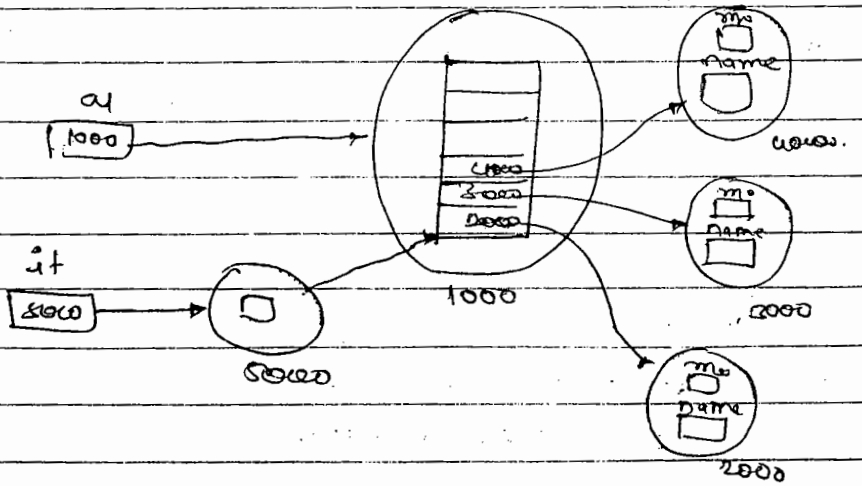
```
import java.util.*;
class ArrayListDemo10
{
    static void view (Iterator <Employee> i)
    {
        while (i.hasNext())
        {
            Employee e = i.next();
            S.O.P (e.getRno() + " " + e.getName());
        }
    }
```

employee replace student.
Student
59.getRno()

```
public static void main (String args[])
{
    ArrayList <Student> al = new ArrayList <Student> ();
    Scanner sc = new Scanner (System.in);
    for (i=0; i<=3; i++)
    {
        int eno = sc.nextInt();
        String en = sc.next();
        Student e = new Student(); Student();
        e.setEno (eno);
        e.setName (en);   al.add (e);
    }
    Iterator <Student> it = al.iterator();
    view (it);
}
}
```

O/P
20
30
40
50

07108P
V.IMP



Exampl

* Removing using Iterator

```
import java.util.*;
class ArrayListDemo11
{
    public static void main (String args[])
    {
        ArrayList <Integer> al = new ArrayList<Integer>();
        al.add (10);   al.add (30);   al.add(20);
        al.add (20);   al.add (40);
        Iterator <Integer> i = al.iterator ();
        i.next();
        i.remove();
        System.out.println (al);
    }
}
```

O/P
20
30
40
50

07/08/2019
V.Imp

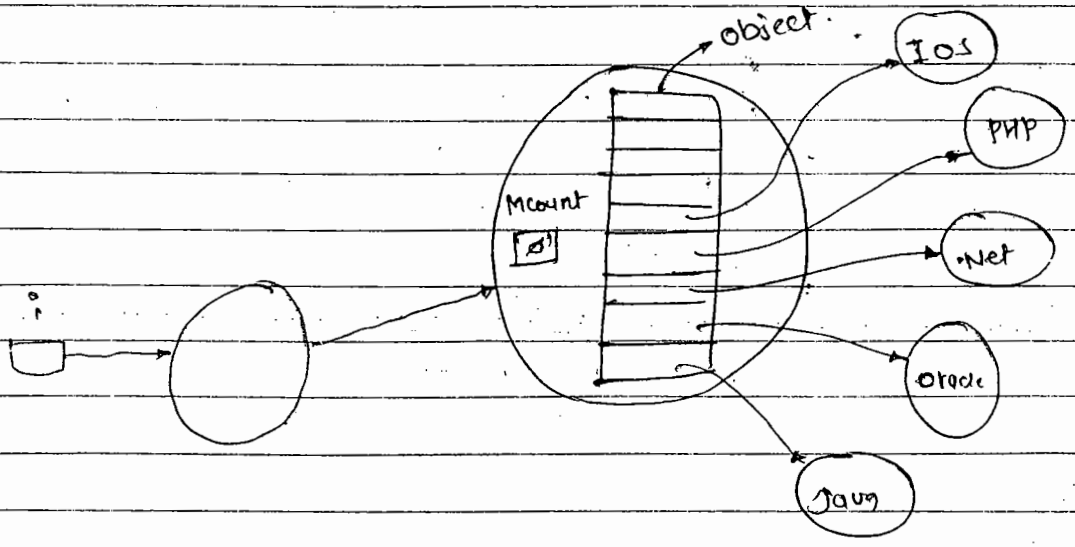
* Fail-Fast Iterators in Java:-

- As name suggest fail-fast Iterators fail as soon as they realized that structure of collection has been changed since iteration has begun.
- Structural changes means adding, removing or updating any element from collection ~~while~~ while one thread is iterating over that collection.
- Fail-fast behaviour is implemented by keeping a modification ~~count~~ count and if iteration thread realizes the change in modification count it throws the ConcurrentModificationException.

Example:

```
class ArrayListDemo12
{
    public static void main (String args[])
    {
        ArrayList<String> al=new ArrayList <String> ();
        al.add("Java");   al.add (".Net");
        al.add ("Oracle");   al.add ("PHP");
```

```
Iterator<String> i = al.iterator();
System.out.println(i.next());
al.add("IOS"); // runtime error
sop(i.next()); ---- throws exception (ConcurrentModificationException).
}
}
```



Diagram

* Fail-safe Iterator in Java :-

- contrary to Fail-Fast Iterator. Fail safe iterator doesn't throw any exception if collection is modified structurally while one thread is iterating over it because they works on clone of collection instead of original collection and thats why they are called as Fail safe iterator.

- Iterator of CopyOnWriteArrayList is an example of Fail safe Iterator also Iterator written by the ConcurrentHashmap keyset is also Fail-safe iterator & never throw ConcurrentModificationException in Java..

* CopyOnWriteArrayList : (5.0 onwards)

public class CopyOnWriteArrayList<E> extends Object implements List<E>, RandomAccess, Cloneable, Serializable.

- A thread-safe variant of ArrayList in which all

mutative operations (add, set...) are implemented by making a fresh copy of the underlying array.

- The "snapshot" style iterator method uses a reference to the state of the array at the point that the iterator was created. This array never changes during the lifetime of the iterator, so interference is impossible and the iterator is guranteed not to throw ConcurrentModificationException

Example:

```
import java.util.*; import java.util.concurrent.*;
class ArrayListDemo13
{
    public static void main(String args[])
    {
        CopyOnWriteArrayList list = new
                                CopyOnWriteArrayList<String> ();
        list.add ("Java");
        list.add ("Oracle");
        list.add (".PHP");
        Iterator <String> i = list.iterator ();
        System.out.println (i.next());
        list.add (".Net");
        System.out.println (i.next());
    }
}
```

2) ListIterator:-

public interface ListIterator<E> extends Iterator<E>

- An iterator for lists that allows programmer to traverse the list in either direction, modify the list during iteration, and obtain the iterator's current position in the list.

- A ListIterator has no current element; its cursor

position always lies between the element that would be returned by a call to previous() and the element that would be returned by a call to next().

Iterator	ListIterator
1) forward only cursor.	1) Bidirectional cursor.
2) It is used with List, set collection	2) It is used with only Lists. (Vector, ArrayList, LinkedList).
3) It maintain current cursor position	3) It is between two objects or there is no current cursor position.
4) It allows to remove element from underlying collection	4) It allows to remove, add & modify elements from underlying collection.

Methods:

- void add(E e)
  - Insert specified element into the list.
- boolean hasNext()
  - Returns true if this list iterator has more elements when traversing list in forward direction.
- boolean hasPrevious():
  - Returns true if this list iterator has more elements when traversing the list in reverse direction.
- E next():
  - Returns next element in list.
- int nextIndex();
  - Returns the index of element that would be returned by subsequent call to next.
- int previousIndex()
  - Returns the index of the element that would be returned by a subsequent call to previous.
- E previous();
  - Returns the previous element in the list.

O/P:
Java
Oracle
Orac
Jav

- void remove():
  - Removes from the list the last element that was returned by next or previous.
- void set(E e):
  - Replaces the last element returned by next or previous with specified element.

Q: How to get ListIterator Object?

- using listIterator() method of List.
- ListIterator interface is implemented within AbstractList class as an inner class.

```
public class AbstractList extends AbstractCollection
{                                   implements List.
    private class Itr implements ListIterator
    {
    }
    public ListIterator iterator()
    {
        return new Itr();
    }
}
```

```
// working with ListIterator:
import java.util.*;
class ArrayListDemo14
{
    public static void main(String args[])
    {
        ArrayList<String> al = new ArrayList<String>();
        al.add("Java"); al.add("Oracle"); al.add(".Net");
        al.add("PHP");
        ListIterator<String> itr = al.listIterator();
        System.out.println(itr.next());
        System.out.println(itr.next());
        System.out.println(itr.previous());
    }
}
```

O/P:
Java
Oracle
Oracle
Java

08/08/2013

3) Enumeration :-

- Enumeration is an interface

• public interface Enumeration <E>

- An object that implements the Enumeration interface generates a series of elements, once at a time. Successive calls to the nextElement method return successive elements of the series.

Methods:

• boolean hasMoreElements() :-

- Tells if this enumeration contains more elements.

• E nextElement() :-

- Returns the next element of this enumeration if this enumeration object has at least one more element to provide.

-

- Methods are provided to enumerate through the elements of vector, the keys of hashtables, & the values in hash table.

Legacy class :-

- The class which implements Serializable, Cloneable and provides synchronized method called Legacy class (collection class).

- The collection classes which are introduced in Java 1.0 are called legacy collection classes.

Iterator	Enumeration
1) Since 1.2 onwards	1) Since 1.0 onwards.
2) Updatable cursor, it allows to remove the element from collection.	2) Non-updateable cursor, it allow to read.
3) Iterator is Fail-Fast.	3) Enumeration is fail-safe.
4) Method names have been improved 5) contains three methods: hasNext(), remove(), next().	4) Method names are large. 5) contains two methods: nextElement(), hasMoreElements().

① Si
② Fo
③ Up
allo
④ Fo
⑤ 3
⑥ Ule
⑦ - cu
can
⑧ No

exam

- Enumeration interface is used by legacy classes. Vector.elements() & Hashtable.elements() method returns Enumeration.
- Iterator is returned by all Java collection framework classes. java.util.Connection.iterator() method returns an instance of iterator.

Iterator	ListIterator	Enumeration
① since 1.2	- since 1.2	- since 1.0
② forward only cursor	- bidirectional cursor	- forward only cursor.
③ updateable cursor, allow to remove	- updateble cursor, allow add, remove	- Non updateable cursor
④ fail fast	- fail fast	- fail-safe
⑤ 3 methods;	- Nine methods	- two methods.
⑥ used with List & set(map)	- used with List only	- used with legacy classes (vectors hashtable.
⑦ - current cursor position can be find.	- No current cursor position	- current cursor position
⑧ No methods to find index	- methods to find index	- No methods to find index.

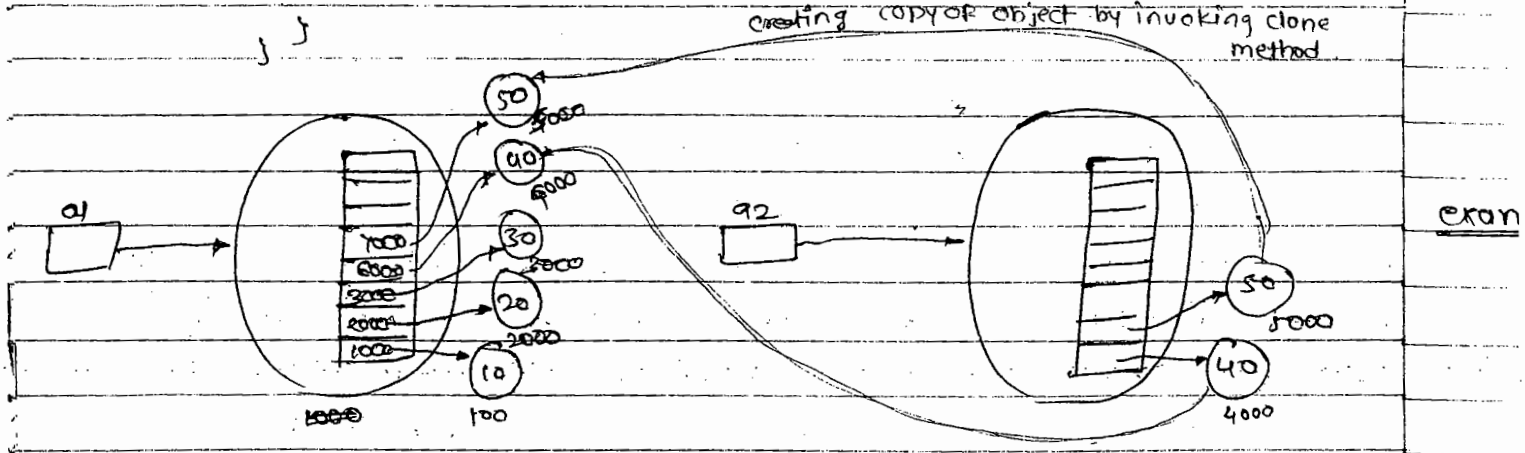
* Bulk methods of ArrayList / Bulk Operations:-

1) addAll (Collection<? extends E> c)
2) removeAll (Collection<? extends E> c)
3) retainAll (Collection<? extends E> c).

example: for addAll

```
import java.util.*;
class ArrayListDemo14
{
    public static void main (String args[])
    {
        ArrayList<Integer> a1 = new ArrayList<Integer>();
        a1.add(10); a1.add(20); a1.add(30);
        ArrayList<Integer> a2 = new ArrayList<Integer>();
        a2.add(40); a1.addAll(a2);
        a2.add(50); Sop(a1); Sop(a2); }}
```

```
a2.add(60);
SOP(a1);
a2.set(0,420);
SOP(a1);
}}
```



Example for removeAll

```
import java.util.*;
class ArrayListDemo15
{
  public static void main (String args[])
  {
    ArrayList <Integer> a1 = new ArrayList <Integer> ();
    a1.add(10); a1.add(20); a1.add(30);
    ArrayList <Integer> a2 = new ArrayList <Integer> ();
    a2.add(10); a2.add(50);
    boolean b = a1.removeAll (a2);
    SOP(b);
    SOP(a1);
  }
```

O/P true
{20,30}

Example for retainAll :

```
import java.util.*;
class ArrayListDemo16
{ public static void main (String arg[])
  { ArrayList <Integer> a1 = new ArrayList <Integer> ();
    a1.add(10); a1.add(20); a1.add(30);
    ArrayList <Integer> a2 = new ArrayList <Integer> ();
    a2.add(20); a2.add(
```

o/p - mule

```
        boolean b = al.retainAll(a2);
        sop(b);
    }
}
```

// How to convert ArrayList to Array?

- Object toArray()
- <T> T[] toArray(T a[])

example:

```
import java.util.*;
class ArrayListDemo17
{
    public static void main(String args[])
    {
        ArrayList<Integer> al = new ArrayList<Integer>();
        al.add(10); al.add(30); al.add(50);
        al.add(20); al.add(40);
        Object a[] = al.toArray();
        Integer b[] = new Integer[10];
        al.toArray(b);
        sop(a);                          10,20,30,40,50.
        sop(b);                          5
        b[5] = new Integer(60);          10
        sop(a.length());
        sop(b.length());
    }
}
```

01/08/2018

B) Vector: (1.0).

- The vector class implements a growable array of objects. Like an array, it contains components that can be accessed using an integer index. However, the size of a vector can grow or shrink as needed to accommodate adding and removing items after the Vector has been created.

Vector extends AbstractList

Each vector tries to optimize storage mgmnt by maintaining a capacity & capacityIncrement. The capacity is always at least as large as vector size; it is usually larger because as components are added to the vector, the vector's storage increases in chunks the size of capacityIncrement.

- The Iterators returned by Vector's iterator and listIterator methods are fail-fast; if the Vector is structurally modified at any time after the Iterator is created, in any way except through the Iterator's own remove or add methods, the Iterator will throw a ConcurrentModificationException. The Enumeration returned by Vector's elements method are not fail-fast.

ArrayList	Vector
1) Since 1.2	1) Since 1.0
2) Not a Legacy class	2) Legacy class
3) Methods are not synchronized	3) Methods are synchronized.
4) Not thread safe	4) Thread safe.
5) Used in single threaded application	5) Used in multithreaded Applications.
6) optimization of storage mgmnt is not possible.	6) optimize storage mgmnt by maintaining a capcity & capacity increment.
7) Arraylist does not have any method returning enumeration	7) Enumeration returned by vector's

Que: Why should I always use ArrayList over Vector?

- Generally you want to synchronize a whole sequence of operations.
- Synchronizing individual operations is both less safe (if you iterate over a vector, for instance, you still need to take out a locks to avoid anyone else changing the collection at the same time.)

Que: How to convert ArrayList into Synchronized ArrayList?

- Collection class provide following method which return synchronized list.

- **static <T> List<T> synchronizedList(List<T> list)**
  - Returns a synchronized (thread safe) list backed by the specified list.

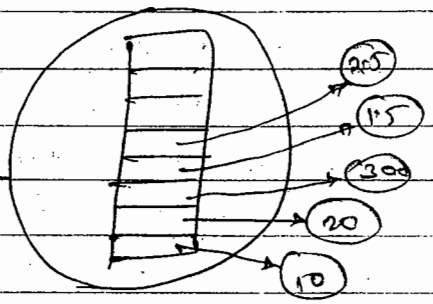
newcapacity = (oldcapacity * 3/2) + 1

newcapacity = (oldcapacity + capacityInc)

- Default capacity of vector is 10.

Ex

```
Vector v = new Vector();
v.add(new Integer(10));
v.add(new Integer(20));
v.add(new Integer(30));
v.add(new Float(1.5f));
v.add(new Float(2.5f));
```




```
Integer x = v.get(0); — (✗)
Integer x = (Integer) v.get(0) --- (✓)
```

- **Vector():-**

constructs an empty vector so that its internal data array has size 10 & its standard capacity increment is zero.

- **Vector(int initialCapacity, int capacityIncrement)**
  - constructs an empty vector with specified initial capacity & capacity increment.

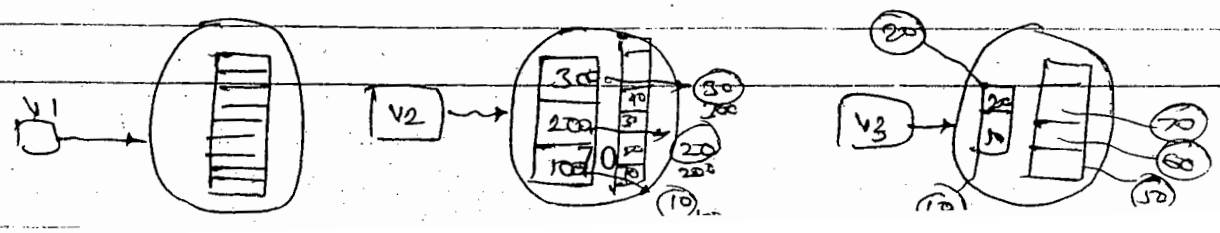
- **int capacity():**
  - returns the current capcity of this vector.

example:

```
import java.util.*;
class Vectordemo1
{
  public static void main(String args[])
  {
    Vector<Integer> v1 = new Vector<Integer>();
    System.out.println(v1.capacity());
    Vector<Integer> v2 = new Vector<Integer>(3);
    System.out.println(v2.capacity());
    Vector<Integer> v3 = new Vector<Integer>(2,2);
      v2.add(10);
      v2.add(20);
      v2.add(30);
      v2.add(40);
    System.out.println(v2);
    System.out.println(v2.capacity());
    v3.add(50);
    v3.add(60);
    v3.add(70);
  // v3.add(80);
    System.out.println(v3);
    System.out.println(v3.capacity());
  }
}
```

o/p :-

```
10
3
[10,20,30,40]
6
[50,60,70]
4
```



```
import java.util.*;
class VectorDemo2
{
  public static void main (String args[])
  {
    Vector box = new Vector ();
    box.add ("apple1");
    box.add ("apple2");
    box.add ("apple3");
    box.add ("Orange1");
    box.add ("orange2");
    box.add ("orange3");
    Vector applebox = new Vector();
    Vector orangebox = new Vector();
    for (Object o : box)
    {
```

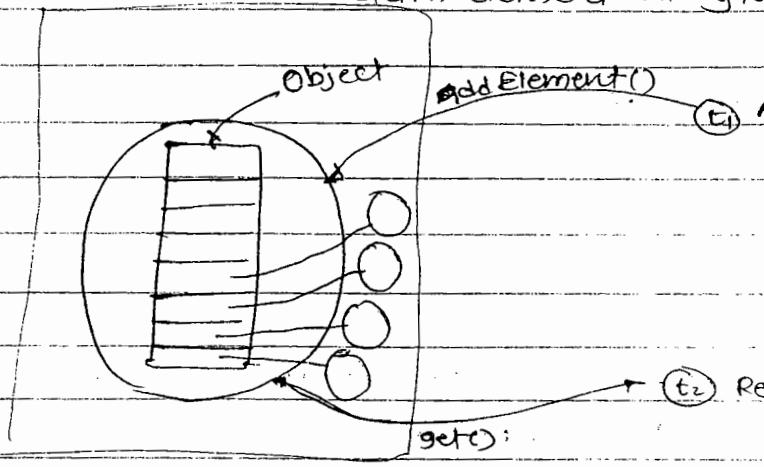
```
import java.util.*;
class VectorDemo2 Fr
{
}
class Apple extends Fr
{
}
class Orange extends Fr
{
}
class VectorDemo2
{
  p.s.v.m (String args[])
  {
    Vector <Fr> v = new Vector <Fr>();
    v.add (new Apple());
    v.add (new Apple());
    v.add (new Apple());
    v.add (new orange());
    v.add (new Orange());
```

```
Vector <Apple> vapple = new Vector <Apple>();
Vector <Orage> vorange = new Vector <Orange>();
for (Fr f : v)
{
  if (f instanceof Apple)
  {
    Apple a = (Apple) f;
    vapple.add (a);
  }
  else if (f instanceof Orange)
  {
    Orange o = (Orange) f;
    vorage.add (o);
  }
} sop (vapple); sop (vorange);
}
```

12/08/013

* Synchronized methods of Vector class:-

- synchronized void addElement (java.lang.Object)
  - This method append element to vector.
- synchronized void insertElementAt(java.lang.Object,int)
  - This method insert element at given position.
- synchronized void setElementAt(java.lang.Object, int)
  - This method replace existing element.
- synchronized void removeElementAt(int);
  - This method remove an element from vector using object.
- synchronized java.lang.Object get(int);
  - return element at given index.

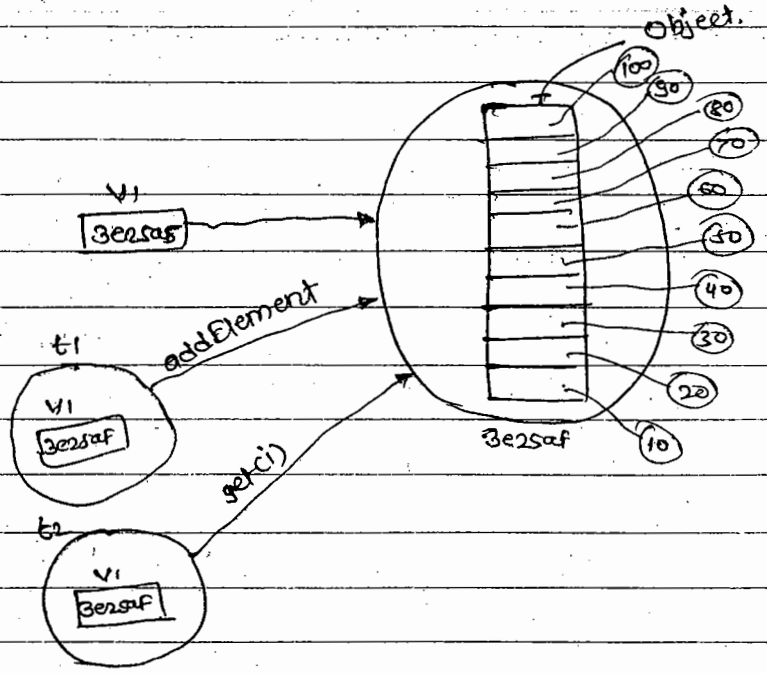



```
import java.util.*;
class AddThread extends Thread
{
   Vector<Integer> v;
   public void
AddThread ( Vector<Integer> v)
{
    this.v=v;
}
public void run()
{
   System.out.println (" Inside addthread");
   for(int i=0, i<10; i++)
   v.addElement();
}
```

```
class ReadThread extends Thread
{
   Vector (Integer> v;
   ReadThread (Vector<Integer>v)
   {
       this.v=v;
   }
   public void run()
   { S.O.P ("Inside readThread");
     for(i=0; i<10; i++)
     sop(v.get(i));
   }
```

exam

```
Class VectorDemo1
{
    public static void main (String args[])
    {
        Vector <Integer> V1 = new Vector <Integer> ();
        AddThread t1 = new AddThread (V1);
        ReadThread t2 = new ReadThread (V1);
        t1.start();
        t2.start();
    }
}
```



* Enumeration elements():

- This method returns enumeration object. It is cursor used for reading elements.

example: //

```
import java.util.*;
class VectorDemo2
{
    P.S.V.M (String args[])
    {
        Vector (Integer) V1 = new Vector (Integer) ();
        V1.addElement (10);
```

```
        V1.addElement(20);
        V1.addElement(30);
        V1.addElement(40);
        Iterator<Integer> y = V1.iterator();
        while(y.hasNext());
        S.O.P(y.next());

        Enumeration<Integer> e = V1.elements();
        while(e.hasMoreElements())
        S.O.P(e.nextElement());
    }
}
```

```
// UserDefined Enumeration
import java.util.*;
class IntegerList
{
    Integer a[] = new Integer[5];
    IntegerList()
    {
      a[0]=10;
      a[1]=20;
      a[2]=30;
    }
    class IntegerEnum implements
                Enumeration
    { int index=-1;
      public boolean hasMoreElements()
      {
        if(index<3)
          return true;
        else
          return false;
      }
      public Object nextElement()
      {
        if(index>2)
        throw new ArrayIndexOutOf
          BoundException();
        return(++index);
      }
    } //inerclass.
    public Enumeration elements()
    { return new IntegerEnum();
    }
} //outerclass.
class EnumDemo1.
{
  PSVM(String args[])
  {
    IntegerList I1 = new IntegerList();
    Enumeration e = I1.elements();
    SOP(e.hasMoreElements());
    SOP(e.nextElement());
  }
}
```

Que: How many ways Vector elements can be read?

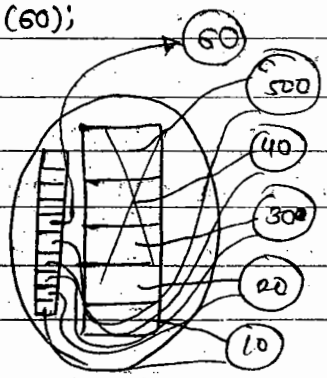
1) getter method
2) ~~List~~ Enhanced for Loop.
3) ~~Cursor~~ Iterator
4) ListIterator
5) Enumeration.

Que: How many way contents of ArrayList is read?

1) Getter method
2) Enhanced for loop
3) Iterator
4) List Iterator.

- void ensureCapacity (int)
- void setSize (int)

```
import java.util.*;
class VectorDemos
{
   P.S.V.M (String args[])
   {
      Vector <Integer> V1 = new Vector <Integer> (5);
      S.O.P (V1.capacity());
      V1.ensureCapacity (16);
      ~~S.O.P~~ V1.addElement (10);  V1.addElement (40);
            V1.addElement (20);  V1.addElement (50);
            V1.addElement (30);  V1.addElement (60);
      SOP(V1);
      S.O.P (V1.capacity());
   }
}
```



O/P: 3.
[10,20,30,40,50]
6

ensureCapacity():

- minimum capacity which has to maintained by server.
- if new capacity is less than ensureCapacity. it create new array with ensure capacity.
- if new capacity is greater than (>) ensure capacity it create new array with new capacity.

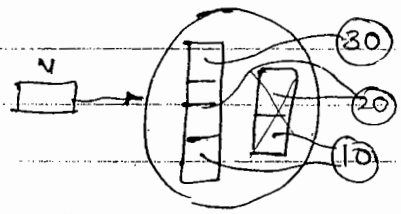
new capacity = (current capacity *2)

example:

```
import java.util.*;
Class VectorDemo4
{ p s v m (String args[])
  {
    Vector <Integer> V1 = new Vector<Integer> (4);
    System.out.println (V1.capacity());
    V1.ensureCapacity (6);
    System.out.println (V1.Capacity());
  }
}
```

① Vector<Integer> V = new Vector<Integer> (2); → double the capacity.

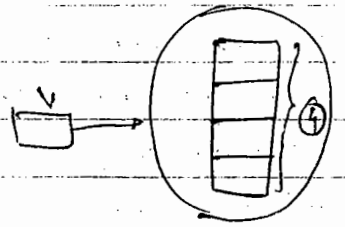
V.add(10)
V.add (20)
V.add (30)



② Vector <Integer> V = new Vector <Integer> (2); → create vector with new capacity (4)

V.ensureCapacity (3);

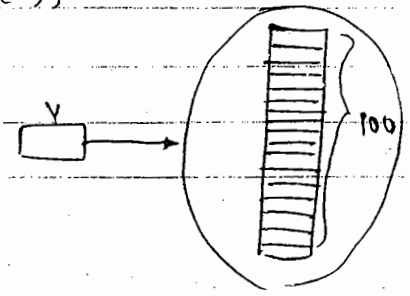
new capacity: current capacity *2.
= 2*2
= 4



③ Vector <Integer> V = new Vector <Integer> (5); → create vector with ensure capacity (>)

V.ensure.Capacity (100);

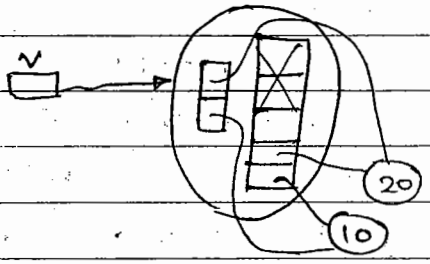
new capacity = current Capacity *20
= 5*2
= 10



size of reference variable = 4 byte

- **trimToSize():**
  - Capacity is decremented to size of vector.

```
Vector<Integer> v = new Vector<Integer>(5);
v.addElement(10);
v.addElement(20);
SOP(v.capacity());
v.trimToSize();
SOP(v.capacity());
```



- **setSize():**
  - Vector is created empty without null.
  - It assign null values to vector upto given size.

```
import java.util.*;
class VectorDemo5
{ p.s.v.m(String args[])
  {
    Vector<Integer> v1 = new Vector<Integer>();
    SOP(v1.capacity());
    v1.setSize(5);
    SOP(v1.capacity());
  }
}
```

O/P: []
[null, null, null, null, null]

example:

```
import java.util.*;
class Date
{ private int dd,mm,yy;
  Date(int dd, int mm, int yy)
  { this.dd=dd;
    this.mm=mm;
    this.yy=yy;
  }
  String getDate()
  { return dd+" "+mm+" "+yy);
  }
}
```

```
class VectorDemo6
{
  public static void main(String args[])
  { Vector<Date> d = new Vector<Date>();
    d.setSize(3);
    Date d1 = d.get(0);
    SOP(d1);
    SOP(d1.getDate()); → NullPointer Exception occur
    d.insertElement(new Date(10,6,2015));
    d.insertElement(new Date(10,6,2015),5);
    → ArrayIndexOutOfBoundEx.
    d.insertElement(new Date(10,6,2015),1);
  }
}
```

18/08/2013

C) * LinkedList :-

public class LinkedList<E> extends AbstractSequentialList<E>
implements List<E>, Deque<E>, Cloneable, Serializable.

- Linked list implementation of the List interface. Implements all optional list operations & permits all elements (including null). In addition to implementing List interface the LinkedList class provide uniformly named methods to get, remove and insert an element at the begining and end of the List. These operation allow linked list to be used as stack, queue or double-ended queue.

15/08/13

ArrayList	LinkedList.
1) It is represented as array	1) It is collection of nodes.
2) It is an index-based collection	2) It is not index-based collection.
3) It allows random & sequential access.	3) It allows sequential access.
4) Every insertion or deletion required to swift elements towards left or right.	4) It doesnot required swift elements.
5) It is not efficent in insertion & deletion. but efficent in reading	5) It is efficient in insertion & deletion but not efficient in reading.
6) It is created with Initial capacity	6) It is created with empty.
7) It is physical Representation.	7) It is logical Representation.

- LinkedList is collection of nodes & each node in Linked List is called Entry.
- Each entry object contain three values.

① Element ② NEXT ③ Previous.

- previous and next are reference variables of Entry type.
- This reference hold address of previous & next objects.
- element hold address of data object.
- LinkedList is a chain of objects.

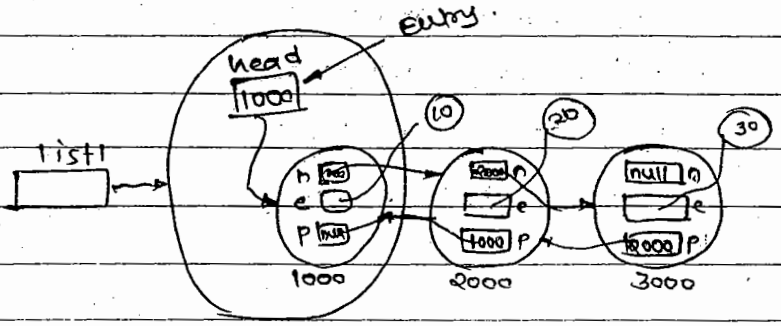
Constructors:

- **LinkedList ()**
  - constructs an empty list.
- **LinkedList (Collection <? extends E> c)**
  - constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

15/08/2013

example

```
LinkedList <Integer> list1 = new LinkedList <Integer> ();
list1.add (10);
list2.add (20);
list1.add (30);
```



Methods:

- **boolean add (E e)**
  - Appends specified element to end of lists.
- **void add (int index, E element)**
  - Insert the specified element at the specified position in list
- **void addfirst (E e)**
  - Insert specified element at begining of this list.
- **void addlast (E e)**
  - Apped specified element to end of this list.
- **Iterator <E> descendingIterator ():**
  - Returns an iterator over the elements in this deque in reverse sequential order.
- **E getfirst ()**
  - Returns first element in this list
- **E getlast ()**
  - Returns last element in this list.

16/08/1

- `ListIterator<E> listIterator(int index)`

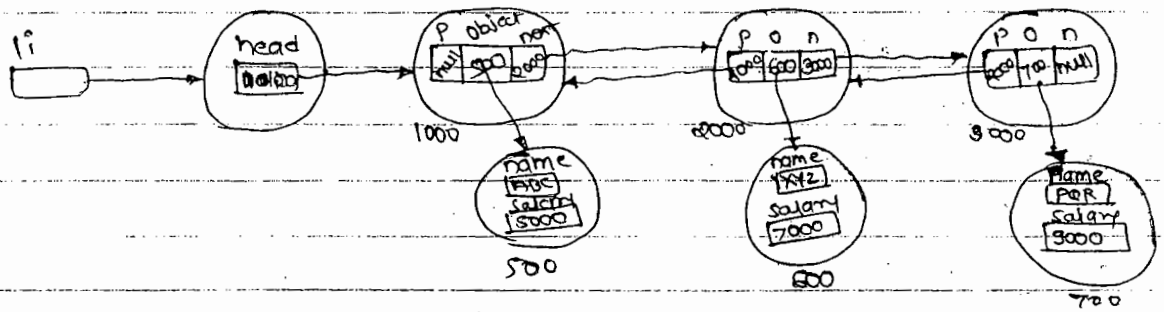
  -Returns a list-iterator of elements in this list (in proper sequence), starting at the specified position in the list.

example:

```
import java.util.*;
class Employee
{
  private String name;
  private float salary;
  Employee(String name, float salary)
  { this.name=name;
    this.salary=salary;
  public String toString()
  { return name+","+salary;
  }
}
```

```
class EmployeeDatabase
{ p s v main(String args[])
  {
    LinkedList<Employee> li = new
                LinkedList<Employee>();
    li.add(new Employee("ABC", 5000F));
    li.add(new Employee("XYZ", 7000F));
    li.add(new Employee("PQR", 9000F));
    System.out.println(li);
  }
}
```

P-previous
O-Object Entry
N-next
↳ Entry




```
- Employee e1 = list.get(0);
  SOP(e1);
  Employee e2 = list.get(2);
  SOP(e2);
```

```
get(int index)
{ Entry p = head;
  while (index > 0)
  { p = p.next();
    index--;
  }
  return p;
```

previous

```
- ListIterator<Employee> li = li.listIterator(0);
  while (li.hasNext()) { SOP(li.next()); }
- Iterator<Employee> i = li.descendingIterator();
  while (i.hasNext())
  { SOP(i.next());
  }
```

16/08/2013

B-tech) Stack:-

public class Stack<E> extends Vector<E>

- The stack class represents a Last in First out stack of object it extends class vector having 5 methods:

- **boolean empty():**
  - Test if stack is empty.
- **E peek():**
  - looks at the object of this stack without removing it from the stacks.
- **E pop():**
  - Removes the object at top of this stack & returns that object as the value of this function.
- **E push(E item):**
  - pushes an item onto the top of this Stack.
- **int search(Object o)**
  - Returns the 1-based position where an object is on this stack

example:

```
import java.util.*;
class StackDemo1
{
  public static void main(String args[])
  {
    Stack<Integer> stack1 = new Stack<Integer>();
    stack1.push(10);                    if(!stack1.empty())
    stack1.push(20);                    sop(stack1.pop()); → error (X)
    stack1.push(30);
    int x = stack1.pop();
    int y = stack1.pop();
    int z = stack1.pop();
    boolean b = stack1.empty();
    System.out.println(x+" "+y+" "+z);
    System.out.println(b);
  }
}
```

O/P: 30 20 10
true.

17/08/12

```
// Finding number of { are equal to }
// checking syntax errors.
import java.util.*;
class StackDemo2
{ public static void main(String args[])
  { boolean flag = true;
    Scanner sc = new Scanner(System.in);
    sop("Input any expression with { & }");
    String expr = sc.next();
    Stack <Character> stack1 = new Stack <Character> ();
    for (int i=0; i< expr.length(); i++)
    {
      char c = expr.charAt(i);
      if (c=='{')
        stack1.push(c);
      else
      if (c=='}')
      {
        if (stack1.empty())
        {
          flag = false;
          break;
        }
        else
        {
          stack1.pop();
        }
      }
    }
    if(stack1.empty() && flag == true)
      sop(" expression is valid");
    else
    if ( stack1.empty() && flag==false)
      sop(" expression is invalid");
    else if (!stack1.empty() && flag== true)
      sop("expression is invalid");
```

17/08/2013

27 Queue:

- Introduced in Java ~~1.5~~ 1.5.

-

public interface Queue<E> extends Collection<E>

- This interface is member of Java collection framework since 1.5.

- All known subinterfaces:

BlockingDeque<E>, BlockingQueue<E>, Deque<E>

- All known Implementing classes :-

AbstractQueue, ArrayBlockingQueue, ArrayDeque, ConcurrentLinkedQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingDeque, LinkedList, PriorityBlockingQueue, PriorityQueue, SynchronousQueue.

* Priority Queue:

- An unbounded priority queue based on a priority heap. The elements of the priority queue are ordered according to their natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used.

- A priority queue does not permit null elements. A priority queue relying on natural ordering also does not permit insertion of non-comparable objects (doing so may result in ClassCastException).

Comparable Interface:

- This interface imposes a total ordering on the objects of each class that implements it. This ordering is refered to as the class's natural ordering and the class's compareTo method is refered to its natural comparison method.

exam

exampl

impor

class

- int compareTo (T o):

- compares this object with specified object for order. returns one of -1, 0, or 1 according to whether the value of expression is negative, zero or positive.

- Every wrapper class implement Comparable interface and provide implementation of compareTo method.

example;

```
Class Demo
{ public static void main (String args[])
  {
    Integer a = new Integer (10);
    Integer b = new Integer (20);
    int x = a.compareTo (b);
    System.out.println (x); // -1.
    int y = b.compareTo (a);
    System.out.println (y); // 1
    int z = b.compareTo (new Integer (20));
    System.out.println (z); // 0.
  }
}
```

(1>2) → -1
(2>1) → 1
== → 0

example:

```
import java.~~lang~~util.*
Class PriorityQueueDemo1
{ p s v main (String args[])
  {
    PriorityQueue<Integer> q = new PriorityQueue<Integer> ();
    q.add (40);
    q.add (30);
    q.add (10);
    q.add (50);
    q.add (20);
    System.out.println (q);
```

example:

```
import java.util.*;
class PriorityQueueDemo2
{ public static void main (String args[])
  {
    PriorityQueue<String> q = new PriorityQueue<String>();
    q.add ("xyz");
    q.add ("abc");
    q.add ("pqr");
    q.add ("mno");
    System.out.println(q);
  }
}
```

O/P
abc
mno
pqr
xyz

Example:

```
import java.util.*;
class Book implement comparable
{
  String bname, aname, edition;
  Book (String bname, String aname, String edition)
  {
    this.bname = bname;
    this.aname = aname;
    this.edition = edition;
  }
  public String toString()
  { return bname + "" + aname + "" + edition); }
  public int compareTo ( Object o)
  { Book b = (Book) o;
    if (bname.compareTo (b.bname) > 0)
      return +1
    else if (bname.compareTo (b.bname) < 0)
      return -1;
    else if (bname.compareTo (b.bname) == 0)
      return 0;
  }
}
```

```
class BookQueue
{   public static void main (String args[])
    {
        PriorityQueue <Book> q2 = new PriorityQueue<Book>();
        q.add(new Book ("Java", "James", "4.0"));
        q.add (new Book ("Oracle", "Scott", "5.0"));
        q.add (new Book ("Java", "James", "5.0"));
}   }
```

19/08/2013 - Comparable & comparator interface are used to sort collection or array of objects.

Comparable	Comparator
1) Comparable interface is used to provide the natural sorting of objects & we can use it to provide sorting based on single logic.	- Comparator interface is used to provide different algorithms for sorting & we can choose the comparator we want to use to sort the given collection of objects. It is used to modify existing comparator objects.

example:

```
import java.util.*;
class PriorityQueueDemo3
{   public static void main (String args[])
    {
        Scanner sc=new Scanner(System.in);
        PriorityQueue <String> q = new PriorityQueue <String> ();
        String name="null";
        while (!name.equals ("stop"))
        {
            SOP("Input name");
            name= scan.next();
            if (!name.equals("stop"))
            q.add (name);
        }
        SOP(q);
    }
}
```

// Example for Comparator Interface:

- A comparison function, which imposes a total ordering on some collection of objects.

- int compare(T o1, T o2):-
  - compares its two arguments for order.
- boolean equals(Object obj):-
  - Indicates whether some other object is "equal to" this comparator.

```
// organizing string in descending order:
import java.util.*;
class StringOrder implements Comparator
{ public int compare(Object o1, Object o2)
  {
    String s1 = (String) o1;
    String s2 = (String) o2;
    if (s1.compareTo(s2)>0)
    return -1;
    else if (s1.compareTo(s2)<0)
    return +1;
    else
     return 0;
  }
public boolean equals(Object o)
{
  // ;
   if (this == o)
   return true;
   else
    return false;
  }
}
```

```
class PriorityQueueDemo4
{ psvm(String args[]);
  PriorityQueue<String> q = new
  PriorityQueue<String>(5,
          new StringOrder());
  q.add("bca");
  q.add("cba");
  q.add("abc");
  sop(q);
}
```

o/p [bcba, bca, abc]

WAP add string into PriorityQueue based on length.

```
// Custom objects / User Defined objects
import java.util.*;
class Client implements Comparable
{
  private string name;
  private string type;
  Client (string name, string type)
  { this.name=name;
    this.type=type;
  }
  public int CompareTo(Object o)
  { int n1=n2=0;
    Client c=(Client)o;
    if (type.equals ("vip"));
    n1=1;
    else
    n1=2;
    if (c.type.equals ("VVIP"))
    n2=2;
    else
    n2=1;
    if(n1>n2)
    return -1;
    else if (n1<n2)
    return +1;
    else
     return 0;
  }
  public string toString()
  {
    return name+" , "+type;
  }
}
class Server
{ public static void main (String args[])
  {
   PriorityQueue <Client> q=new
   PriorityQueue <Client> ();
   q.add (new Client ("abc","vip"));
   q.add (new Client ("xyz","vvip"));
   sop(q);
  }
}
```

✳ DQueue:

- public interface Deque <E> extends Queue (E)

- A linear collection that supports element insertion and removal at both ends. The name deque is short for "double ended queue" and is usually pronounced "deck".

	First Element (Head)		Last Element (Tail)	
	Throws Exception	Special value	Throws Exception	Special value
Insert	addFirst(e)	offerFirst(e)	addLast(e)	offerLast(e)
Remove	removeFirst()	pollFirst()	removeLast()	pollLast()
Examine	getFirst()	peekFirst()	getLast()	peekLast()

- There are two types of Deque

1) ArrayDeque
2) LinkedBlockingDeque

31/08/2013

Deque
BlockingDeque
NavigableSet
NavigableMap

1) ArrayDeque: (1.6)

- ArrayDeque is a class that implements Deque. It has no capacity restrictions. It will perform faster than Stack when used as stack and faster than linked list when used as queue. ArrayDeque is not thread safe.

Constructors:

1) ArrayDeque()

- constructs an empty array deque with an inital capacity sufficient to hold 16 elements.

- Iterator <E> iterator():

- Returns an iterator over elements in this

```
import java.util.*;
class ArrayDequeDemo1
{
  public static void main(String args[])
  {
    ArrayDeque dq = new ArrayDeque();
    dq.addFirst("CoreJava");
    dq.add("AdvancedJava");
    dq.add("Oracle");
    String c1 = dq.peekFirst();
    String c2 = dq.peekLast();
    sop(c1);
    sop(c2);
    sop(dq);
    String c3 = dq.removeFirst();
    String c4 = dq.removeLast();
    sop(c3);
    sop(c4);
    sop(dq);
  }
}
```

```
Iterator i = dq.iterator();
sop(i.next());
```

O/P

```
CoreJava
Oracle
[CoreJava, AdvancedJava, Oracle]
CoreJava
Oracle
[AdvancedJava]
```

1) BlockingDeque:

- ~~BlockingDeque is similar to Deque and provide~~

2) BlockingQueue: (1.5)

- A queue that additionally supports operations that waits for the queue to become non-empty when retrieving an element, and wait for space to become available in queue when storing an element.

	Throws exception	Special value	Blocks	Timeout
Insert	add(e)	offer(e)	put(e)	offer(e, time, unit)
Remove	remove()	poll()	take()	poll(time, unit)
Examine	element()	peek()	not applicable	not applicable.

- A BlockingQueue does not accept null elements. Implementation throw NullPointerException an attempts to add, put or offer a null.

- This interface is available in java.lang.concurrent package. exam

* What is BlockingQueue:-

- java.util.concurrent.BlockingQueue is a Queue that supports operations that wait for the queue to become non-empty when retrieving and removing an element, & wait for space to become available in queue when adding an element. for for

BlockingQueue interface is part of java collections framework and its primarily used for implementing producer consumer problem. pub { p

to solve producer-consumer problem

- we don't need to worry about waiting for the space to be available for producer or object to be available for consumer in BlockingQueue as its handled by implementation classes of BlockingQueue. Java provides several BlockingQueue implementation such as:

1) ArrayBlockingQueue
2) LinkedBlockingQueue
3) PriorityBlockingQueue
4) SynchronousQueue.

- A BlockingQueue may be capacity bounded. At any given time it may have a remainingCapacity beyond which no additional element can be put without blocking.

- A BlockingQueue without any inherent capacity constraints always reports a remaining capacity of Integer.MAX_VALUE. (2^{32}).

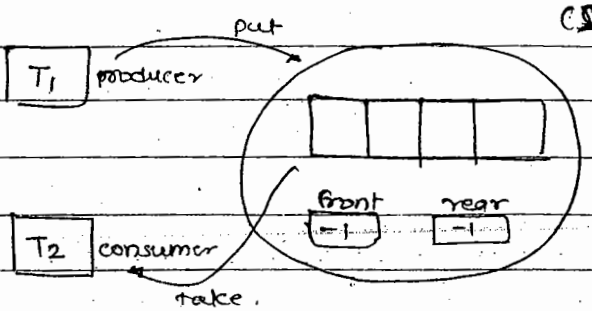
1) ArrayBlockingQueue:

- It is one of the implementation of java.util.concurrent.BlockingQueue which internally store the queue elements inside an array. The Array

elements for the size defined when the object is initialized by the constructor. once it size is defined it cannot be change or resize.

Example:

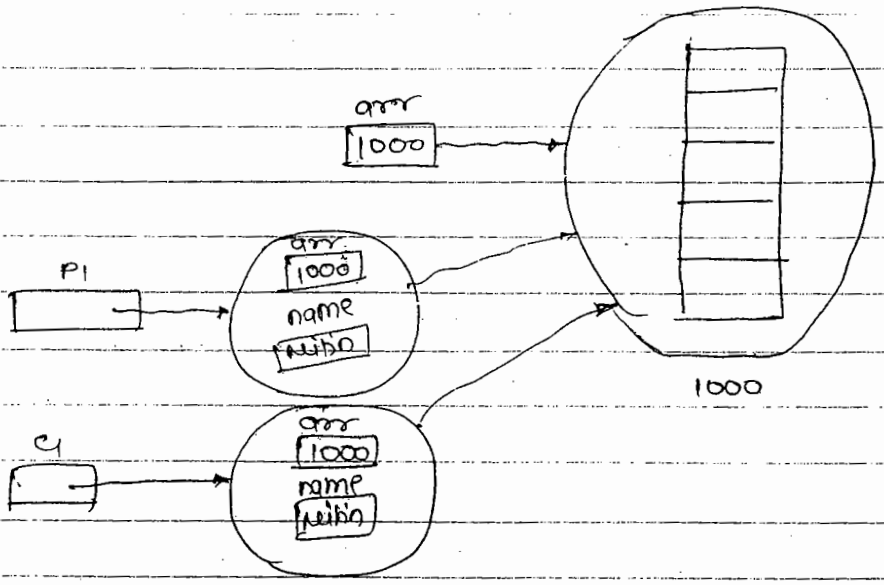
```
ArrayBlockingQueue <String> aq = new ArrayBlockingQueue (String)(5);
```




```
import java.util.*;
import java.util.concurrent

public class ABQExample
{
  private BlockingQueue <String> sq = new ArrayBlockingQueue (String)(5);
   p. s. v. main (String args[])
   {
      new ABQExample.createProducerConsumer();
   }
   private void createProducerConsumer()
   {
      new Thread(new Runnable(){ :
          public void run () {
           while (true) {
           sop(" Thread. currentThread(). getName());
           try {
                sq: put ("DATA");
                Thread. sleep (500);
           } catch (InterruptedException e){
                e. printStackTrace();
                }
      } } }
      }, "producer Thread"). start();
```

22/08/20

```
new Thread (new Runnable() {
public void run(){
 while (true){
  Sop(" Thread.currentThread().getName() + "=>");
  try {
      Sop( sq.take());
      Thread.sleep(1000);
    } catch (InterruptedException e)
    { e.printStackTrace(); }
  }
 }
}, " consumer Thread-1").start();
}
}
```



22/08/2013

```
import java.util.concurrent.*;
class Producer extends Thread
{
    ArrayBlockingQueue<String> aq;
    String name;
    Producer (ArrayBlockQueue<String> aq, String name)
    {
        this.aq = aq;
        this.name=name;
    }
    public void run()
    {
        while(true)
        {
            try
            {
                aq.put(name + "Data");
                Thread.sleep(500);
            }
            catch(Exception e)
            {}
        }
    }
}
class Consumer extends Thread
{
    ArrayBlockingQueue<String> aq;
    String name;
    Consumer(ArrayBlockingQueue<String>, String name)
    {
        this.aq=aq;
        this.name= name;
    }
```

```
public void run()
{
    while(true)
    {
        try
        {
            String s = aq.take();
            SOP(s+name);
            Thread.sleep(500);
        }
        catch(Exception e)
        {}
    }
}
```

Forex

```
class ArrayBlockingQueueDemo1
{   PSVM(String args[])
    {  ArrayBlockingQueue<String> aq = new ArrayBlockingQueue
                                        <String>(5);
       Producer P1 = new Producer(aq, "producer");
       Consumer C1 = new Consumer(aq, "Consumer");
       P1.start();
       C1.start();
    }
}
```

2) LinkedBlockingQueue:

- An optionally-bounded blocking queue based on linked nodes. This queue orders elements (FIFO). The head of queue is that element that has been on the queue the longest time. The tail of the queue is that element that has been on the queue the shortest time.

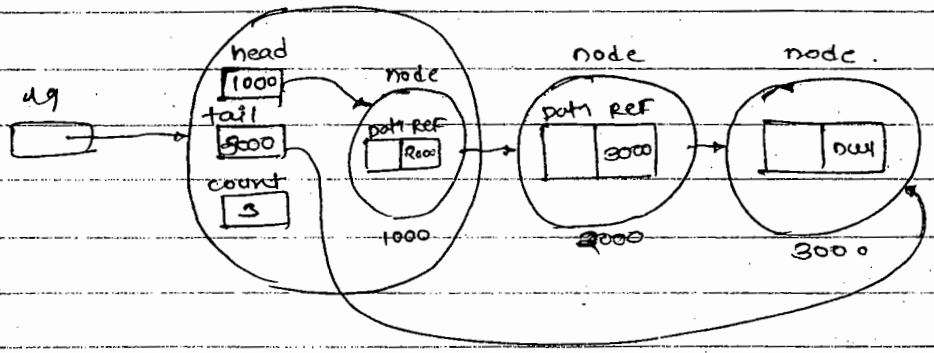
- Elements are inserted at tail of the queue & the queue retrieval operations obtain elements at head of queue.

- **LinkedBlockingQueue():-**
  - Creates a LinkedBlockingQueue with a capacity of Integer.MAX_VALUE.

- **LinkedBlockingQueue(int capacity)**
  - creates a LinkedBlockingQueue with given fixed capacity.

Forexample

```
LinkedBlockingQueue<String> lq = LinkedBlockingQueue<String>(3);
```



- **void put(E e)**
  - Insert the specified element at the tail of this queue, waiting if necessary for space to become available.

- **E take()**
  - Retrieves & removes the head of this queue, waiting if necessary until an element becomes available.

```
import java.util.concurrent.*;
import java.util.*;
class WriteStringThread extends Thread
{ Scanner sc = new Scanner(System.in);
  LinkedBlockingQueue<String> q;
  WriteStringThread(LinkedBlockingQueue
                    <String> q)
  { this.q = q;
  }
  public void run()
  {
    String s = "start";
    while(!s.equals("stop"))
    {
      SOP("Input string");
      s = sc.next();
    try{q.put(s);}catch(Exception e){};
    }
  }
}

Class ReadStringThread extends Thread
{
  LinkedBlockingQueue<String> q;
ReadStringThread(LinkedBlockingQueue
{                   <String> q)
  this.q = q;
}
  public void run()
  {   String s=null;
      while(true)
      { try{
          String s = q.take();}catch
          SOP(S.toUpperCase());
      }
  }
}
```

```
Class LinkedBlockingQueueDemo
{ P.S.V.M(String args[])
{ LinkedBlockingQueue<String> q =
      new LinkedBlockingQueue<String>(3);

WriteStringThread t1 = new WriteStringThread();
ReadStringThread r1 = new ReadStringThread();
t1.start();
r1.start();
}
}
```

30/07

End of Collection

Nithinkumar
+919823442459

23/08/2013

* PriorityBlockingQueue:

- An unbounded blocking queue that uses the same ordering rules as class priorityqueue & supplies blocking retrival operation.
- While this queue is logically unbounded attempted addition may fail due to resource exhaustion (CallingOutOfMemoryError). This class does not permit null value. The priorityQueue relays on natural ordering also does not permit insertion of non comparable object (doing so result is ClassCastException).

ArrayBlockingQueue	PriorityBlockingQueue
1) Insertion order is preserved.	1) Insertion order is not preserved.
2) It allows multiple types of objects / heterogenous collection	2) It allows similar types of objects / homogenous.
3) It allows null ~~values~~	3) It does not allows null.
4) There is no order	4) There is order using Comparable or Comparator

```
import java.util.concurrent.*;
class PriorityBlockingQueueDemo1
{
    p.s.v.main(String args[])
    {
Final String name[] = {"pqr", "abc", "xyz", "mno", "ijk"};
Final PriorityBlockingQueue<String> PQ = new
                PriorityBlockingQueue<String>(10);
```

local class acces only local final variables.

```
class Producer extends Thread
{
    public void run()
    {
        For (int i=0; i<name.length; i++){
        try
        { pq.put (name[i]
          // sleep(500);
        }
        catch (Exception e){}

        }
    }
}

class Consumer extends Thread
{
    public void run()
    {
        For (int i=1; i<=5; i++)
        {
            try
            {
            sop(pq.take());
            sleep sleep(500);
            }
            catch (Exception e){}
        }
    }
}

Producer p1 = new Producer();
Consumer c1 = new Consumer();
P1.start();
C1.start();
}
}
```

example

inner class

* Synchronous Queue:

- A Blocking queue in which each insert operation must wait for a corresponding remove operation by another thread, and vice versa.
- A synchronous queue does not ~~ha~~ have any internal capacity, not even a capacity of one.

- void put (E e)
- E take()

example:

```
import java.util.concurrent.*;
class SynchronousQueueDemo
{
    public static void main(String args[])
    {
    final SynchronousQueue<Integer> s = new
                SynchronousQueue<Integer>();
    Thread t1 = new Thread()
    {
        public void run()
        {
            for(int i=1; i<=10; i++)
            {
                try
                { s.put(i);
                  Sleep(500);
                }
                catch(Exception e){}
            }
        }
    };
```

anonymous class →

```
Thread t2 = new Thread()
{
    public void run()
    {
        for (int i=1 ; i<=10; i++)
        {
            try
            {
                sop ("i=" + s.take());
                sleep(500);
            }
            catch (Exception e) {}
        }
    }
};

t1.start();
t2.start();
}
}
```

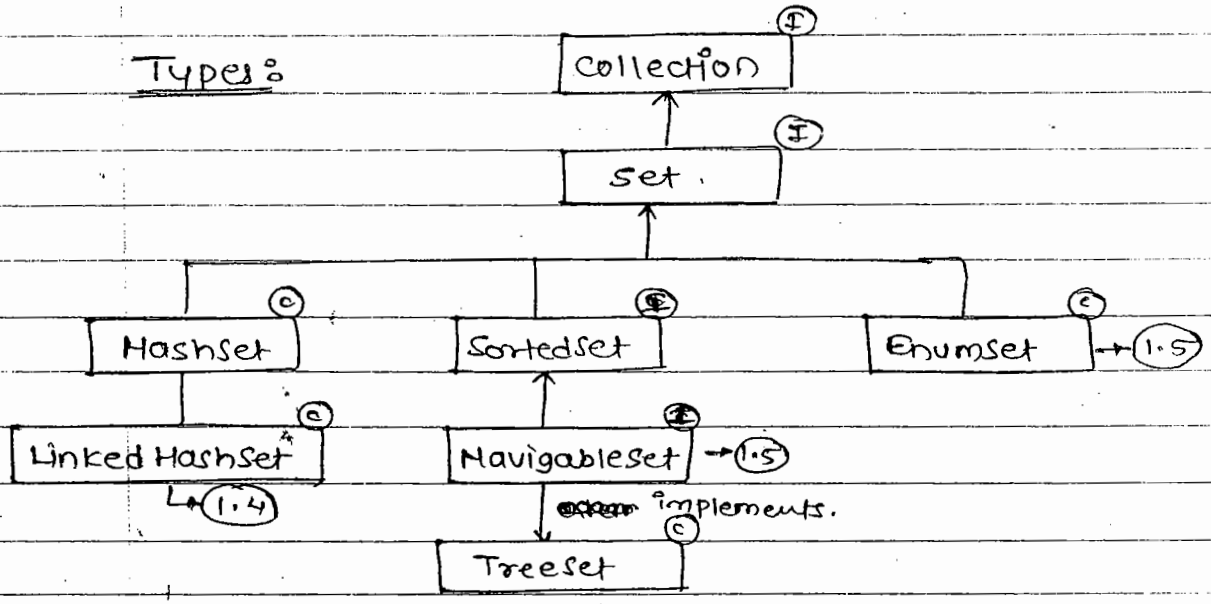
1)
co
e)
3)
4)
5)

3) Set: (1.2 onwards)

- A collection that contains no duplicate elements. More formly, sets contain no pair of elements e1 + e2 such that e1.equals(e2), and at most one null element. As implied by its name, this interface models the mathematical set abstraction.

List	Set.
1) It is an index based collection	1) It is not a index based collection
2) Ordered collection	2) Unordered collection.
3) Insertion order is Preserved	3) Insertion order is not preserved.
4) List allows duplicates.	4) does not allows duplicates.
5) allows more than one null	5) It allows at most one null.

Types:



addAll -
containsAll
removeAll
clear
retainAll → intersection

24/08/2015

Page No. 6
Date

- Set inherit all the methods of Collection interface.
- Set doesnot have its own methods.
- The implementation of add, addAll doesnot allows duplicates.

1) Hashset:

— This class implements the Set interface, backed by a hash table (actually Hashmap instance). It makes no gurantees that the order will remain constant over time. This class permits the null elements.

loadfactor
capacity
increment.

• Hashset():

- constructor a new empty set, the backing Hashmap instance has the defaut initial capacity (16) and load factor (0.75).

• Hashset (int initalCapacity)

- constructs a new, empty set; the backing Hashmap instance has the specified initial capacity and defaut load factor (0.75).

• Hashset (int initalCapacity, Float loadfactor)

- Constructs a new ~~capacity~~, ~~the~~ empty set, the backing Hashmap instance has the specified initial capacity and specified load factor.

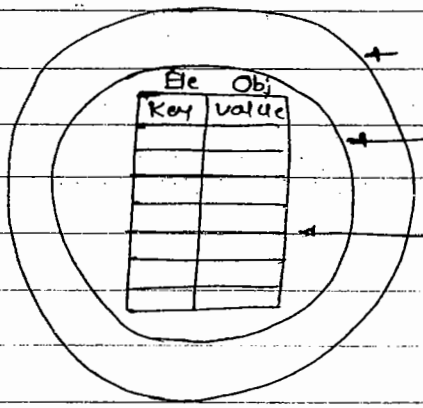
Note: - Collection want to accept unique object then use set other wise use list to accept duplicates.

Q) How data is represented in Hashset?

- Hashset represents the data by creating Hashmap object which is an implementation of hashtable.

- Hashmap contain pair of values; key, value.

e-g



→ Hashset Object.

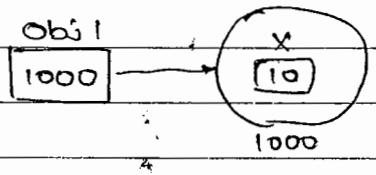
Hashmap object

inital capacity (16).

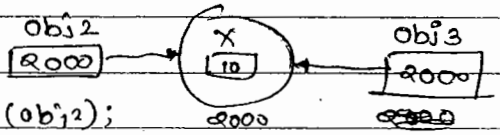
Hashset hs = new Hashset();

key - doesn't allow duplicates.

```
Class A
{ int x=10;
}
A obj = new A();
```




```
A obj2 = new A();
boolean b = obj1.equals(obj2);
   sop(b); → false
```



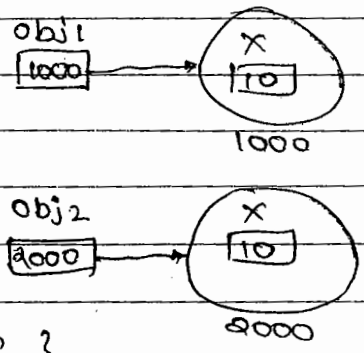

```
A obj3 = obj2;
boolean b1 = obj2.equals(obj3);
   sop(b1); → true.
```

- **Overriding equals method:**

```
class A
{
  int x=10;
  Public boolean equals (Object o)
  {
     A a=(A) o;
     if (x==a.x)
return true;
     else
     return false;
  }
}
```

Note:
Whenever you override equals method you have to override hashCode method.

```
A obj1= new A();
A obj2 = new A();
boolean b= obj1.equals(obj2);
  Sop(b) → true;
  Sop(obj1.hashCode()); → 1000
  Sop(obj2.hashCode()); → 2000
```



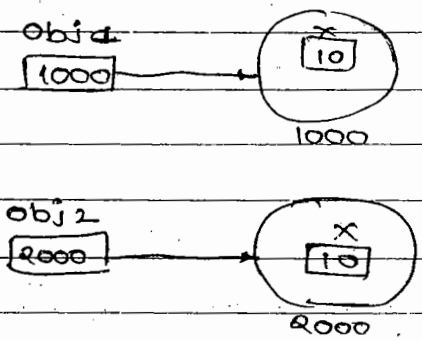
→ This is violating rules of object.

- **Overriding equals & hashCode method:**

```
class A
{
  int x=10;
public boolean equals (Object o)
  {
     A a= (A) o;
     if (x==a.x)
     return true;
     else
     return false;
  }
```

```
public int hashCode()
{
    return x;
}
```

```
A obj1 = new A();
A obj2 = new A();
boolean b = obj1.equals(obj2);
  sop(b); → true.
sop(obj1.hashCode()); → 10
sop(obj2.hashCode()); → 10
```



thod
ide
d.

example:

```
Class EqualsDemo1
{
    p.s.v.m()
    {
        String s1 = "Java";
        String s2 = "Java";
        boolean b = s1.equals(s2);
         sop(b); → true.
        sop(s1.hashCode()); → 2301506
        sop(s2.hashCode()); → 2301506.
} }
```

- All wrapper classes override equals and hashcode methods of Object class.
- classes likes String, StringBuffer, StringBuilder override equals and hashcode methods.

Example: Adding predefined objects in Hashset.

```
import java.util.*;
class HashsetDemo1
{
   public static void main (String args[])
 { HashSet <String> hs = new HashSet<String> ();
   hs.add ("Java");
   hs.add ("Java");
   hs.add ("Oracle");
   hs.add ("Oracle");
   hs.add (".net");
   hs.add (".net");
   SOP (hs);
 } }
```

O/P

[.net, Oracle, Java].

↓

order is based on the hashcode.

27/08/2013

Java Object Law For HashCode() & equals:

The API docs for class object state the rules you MUST follow:

1) If two objects are equal, they MUST have matching hashcodes.

2) If two objects are equals, calling equals() on either object MUST returns true. In other words if (a.equals(b)) then (b.equals(a)).

3) If two objects have the same hashcode value, they are NOT ~~required to~~ required to be equal. But if they're equal, they MUST have the same hashcode value.

4) So, if you override equals(), you must override hashcode.

5) The default behaviour of hashCode() is to generate a unique integer for each object on the heap. So if you don't override

hashCode() in a class, no two objects of that type can EVER be considered equal.

```
// Adding userdefined objects in Hashset
import java.util.*;
class Mystring
{
private String str;
    mystring (String str)
    {
        this.str = str;
    }
    public String getString()
    {
      return str;
    }
}
class HashSetDemo2
{  p s v m (String args[])
   {
     HashSet<Mystring> hs1 = new HashSet<String>();
     Mystring s1 = new Mystring ("Java");
     Mystring s2 = new Mystring ("Oracle");
     Mystring s3 = new Mystring ("Java");
     Mystring s4 = new Mystring ("Oracle");
     hs1.add (s1);
     hs1.add (s2);
     hs1.add (s3);
     hs1.add (s4);
     System.out.println (hs1);
   }
}
```

O/P:

```
Java
Oracle
Java
Oracle
```

// Overriding hashCode method:-

```
import java.util.*;
class MyInteger
{ private int n;
  MyInteger (int n)
  {
  } this.n=n;
  public int hashCode()
  { return n;
  }
}
class HashSetDemo3
{  P.S.V.M (String args[])
   {
     HashSet<MyInteger> hs1 = new HashSet<MyInteger>();
     MyInteger a1 = new MyInteger (10);
     MyInteger a2 = new MyInteger (20);
     MyInteger a3 = new MyInteger (30);
     hs1.add (a1);
     hs1.add (a2);
     hs1.add (a3);
     System.out.println (hs1);
} }
```

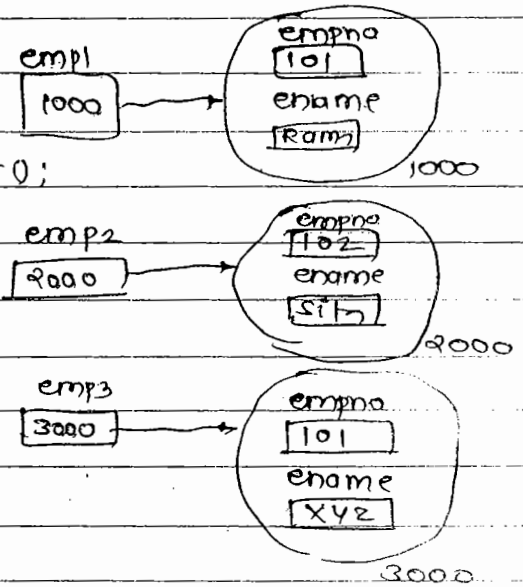
// Overriding equals() & hashCode() method:

```
import java.util.*;
class Employee
{
   private int empno;
   private String ename;
   Employee ( int empno , String ename)
   { this.empno = empno;
     this.ename = ename;
   }
```

```
public boolean equals (Object o)
{
    Employee e = (Employee) o;
    if (empno == e.empno)
    return true;
    else
    return false;
}
public int hashCode()
{  return empno;
}
}
```

```
public String toString()
{
    return empno + " " +
           ename;
}
```

```
class HashSetDemo4
{  p.s.v.m (String args[])
   {
     HashSet <Employee> emp = new HashSet<Employee>();
     Employee emp1 = new Employee (101, "Rama");
     Employee emp2 = new Employee (102, "Sita");
     Employee emp3 = new Employee (103, "XYZ");
     emp.add (emp1);
     emp.add (emp2);
     emp.add (emp3);
     sop (emp);
     .
     .
Iterator <Employee> i = emp.iterator();
 while (i.hasNext())
  sop (i.next());
   }
}
```



Note:

// performing Set operations

```
import java.util.*;
class Student
{
    private String name;
    private String course;
    Student (String name, String course)
    { this.name=name;
      this.course = course;
    }
public boolean equals(Object o)
{
    Student s = (Student) o;
    if (name.equals(s.name) && course.equals(s.course));
    return true;
    else
    return False;
}
public int hashCode()
{
    return name.length+course.length;
}
}
Class HashSetDemo5
{  p s v m. (String args[])
   {
     HashSet (Student) hs1 = new HashSet<Student>();
     hs1.add ( new Student ("abc", "Java");
     hs1.add (new student ("XYz", "-net");
     hs1.add (new student ("pqr", "oracle");
     hs1.add (new Student ("anbo", "php");
     System.out.println(hs1);
     HashSet<Student> hs2 = new HashSet<Student>();
```

```
public String toString()
{
   return name+" "+course;
}
```

```
hs2.add(new Student("abc","java"));
hs2.add(new student("mno","c++"));
hs2.add(new student("asd","c"));
hs2.add(new student("xyz",".net"));
System.out.println(hs2);
// retainAll (intersection)
boolean b1 = hs1.retainAll(hs2);
System.out.println(b1); → true.
System.out.println(hs1);
// removeAll (minus)
boolean b2 = hs1.removeAll(hs2);
sop(b2);
sop(hs1);
sop(hs2);
// containsAll (subset)
boolean b3 = hs1.containsAll(hs2);
sop(hs1);
sop(b3); → false
// addAll (union)
boolean b4 = hs1.addAll(hs2);
sop(hs1);
sop(b4); → true.
// clear
sop(hs1.clear());
sop(hs2.clear());
}
}
// Hashset with null
import java.util.*;
class HashsetDemo6
{ p.s.v.m(String args[])
{
Hashset<String> hs = new Hashset<String>();
hs.add(null);
hs.add(null);
sop(hs);
```

O/P
[null]

28/08/2013

Page No. 16
Date

2) LinkedHashSet:

- Hash table & linked list implementation of Set interface, with predictable iteration order.
- This implementation differs from HashSet in that it maintains a doubly-linked list running through all of its entries.
- This linked list defines the iteration ordering, which is the order in which elements were inserted into set. (insertion-order).

LinkedHashSet = LinkedList + HashSet.

Que) What is the difference between HashSet & LinkedHashSet?

HashSet	LinkedHashSet
1) HashMap and Array imple.	1) Hashtable + LinkedList imple.
2) Insertion order is not preserved	2) Insertion order is preserved.
3) HashSet is introduced in 1.2	3) LinkedHashSet is introduced in 1.4.

* Constructors:

- LinkedHashSet():
  - Constructs a new, empty linked hashset with the default initial capacity (16) & load Factor (0.75).

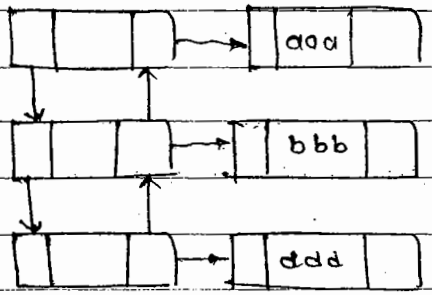
example:

```
import java.util.*;
class LinkedHashSetDemo1
{
    p.s.v.m (String args[])
    {
        HashSet<String> hs = new HashSet<String>();
        LinkedHashSet<String> lhs = new LinkedHashSet<String>();
        hs.add("aaa");4
```

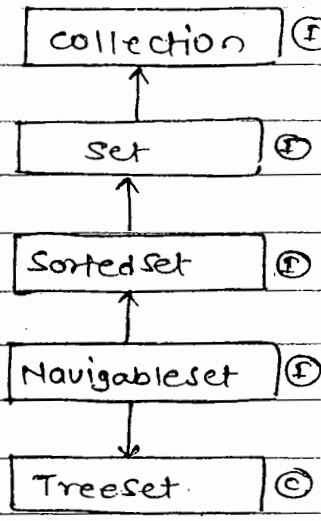
```
        hs.add("bbb");          lhs.add("aaa");
        hs.add("ddd");          lhs.add("bbb");
        hs.add("ccc");          lhs.add("ddd");
        hs.add("eee");          lhs.add("ccc");
                                lhs.add("eee");
        sop(hs);                sop(lhs);
    }
}
```

o/p: [add, ddd, bbb, ccc, eee]

[aaa, bbb, ddd, ccc, eee]



3) TreeSet:



→ 1.5.

- A NavigableSet implementation based on TreeMap. The elements are ordered using their natural ordering, or by a Comparator provided at set creation time, depending on which constructor is used.

Constructors:

- **TreeSet():**

  Constructs a new empty tree set sorted according to natural ordering of its elements.

Page No. 112

Q) What is difference between Hashset and Treeset.

Hashset	Treeset.
1) HashMap implementation.	1) TreeMap implementation.
2) Hash based Datastructure.	2) Tree based Datastructure.
3) It is a hetrogenouse collection	3) Hemogenous collection
4) Compare is done using equals & hashCode method.	4) Comparing is done using Comparator or Comparable.
5) Unordered collection	5) Order collection, order is defined using comparator or comparable.
e.g. Hashset hs = new Hashset hs.add (new Integer (10)); hs.add (new Float (1.5F));	e.g. Treeset ts = new Treeset(); ts.add (new Integer (10)); ts.add (new Float (1.5f)); ⊗.
7) It does not provides navigation methods.	7) It provides navigation methods.

Methods:

- E ceiling (E e)

  - Returns the last element in this set greater than or equal to given element, or null if there is no such element.

- Iterator<E> decendingIterator()

  - Returns an iterator over the elements ~~the~~ in this set in descending order.

- NavigableSet<E> descendingset()

  - Returns a reverse order view of the elements contained in this set.

- E First()

  - Return the First (lowest) element currently in this set.

- E floor(E e)

- Returns the greatest element in this set less than or equal to the given element or null if there is no such element.

example:

```
import java.util.*;
class TreesetDemo1
{
    public static void main(String args[])
    {
        Treeset<Integer> ts = new Treeset<Integer>();
        ts.add(60);     ts.add(40);
        ts.add(30);     ts.add(20);
        ts.add(50);
        System.out.println(ts);
    }
}
```

o/p (20,30,40,50,60)

Note: It allows to add only those object which implement Comparable or Comparator.

29/08/2013

// Adding User Defined objects into Treeset.

```
import java.util.*;
class Student implements Comparable
{
    private int rno;
    private String name, course;
    Student(int rno, String name, String course)
    {
        this.rno = rno;
        this.name = name;
        this.course = course;
    }
```

```
public int compareTo (object o)
{
    Student s = (Student)o;
    if(rno > s.rno)
    return +1;
    else if (rno < s.rno)
    return -1;
    else 0;
}
  String getStudent()
  {
    return rno+" "+name+" "+course;
  }
}
class TreeSetDemo2
{
  public static void main (String args[])
  {
    TreeSet<Student> ts = new TreeSet<Student> ();
    ts.add (new Student (102,"abc","java"));
    ts.add (new Student (101,"xyz","Oracle"));
    ts.add (new Student (103,"pqr",".net"));
    System.out.
    Iterator <Student> i = ts.iterator();
    while (i.hasNext())
    {
      Student s1 = i.next();
      System.out.println (s1.getStudent());
    }
    Iterator <Student> i1 = ts.descendingIterator();
    while (i1.hasNext())
    {
      Student s2 = i1.next();
      System.out.println (s2.getStudent());
    } } }
```

O/P

```
101 abc java
102 xyz oracle
103 pqr .net
103 pqr .net
102 xyz oracle
101 abc java
```

Reading set of elements from Treeset:-

1) SortedSet<E> tailSet(E fromElement)

- Returns a view of the portion of this set whose elements are greater than or equal to fromElement.

(1.5)→ 2) NavigableSet<E> tailSet(E fromElement, boolean inclusive)

- Returns a view of portion of this set whose elements are greater than (or equal to, if inclusive is true) fromElement.

3) NavigableSet<E> subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive)

- Returns a view of portion of this set whose elements range from fromElement to toElement.

4) SortedSet<E> subSet(E fromElement, E toElement)

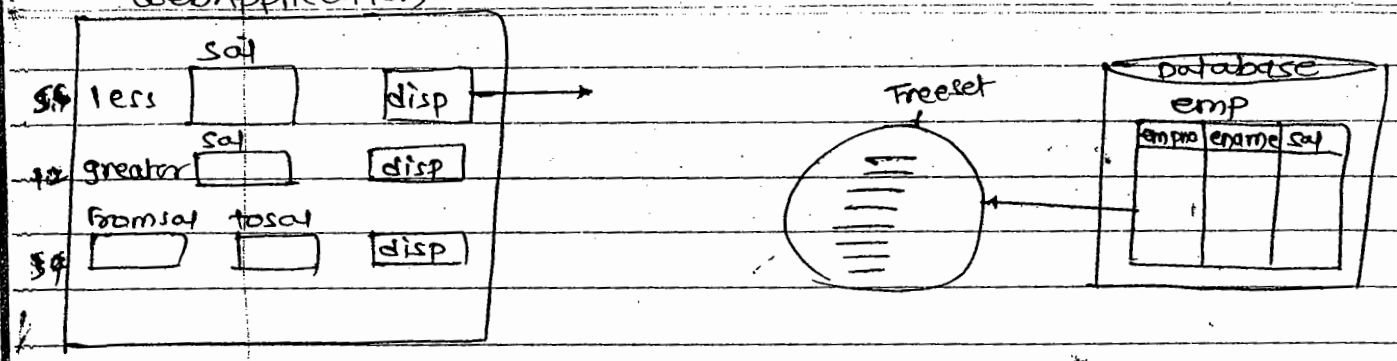
- Returns a view of portion of this set whose elements range from fromElement, inclusive toElement, exclusive.

5) SortedSet<E> headSet(E toElement)

- Returns a view of portion of this set whose elements are strictly less than toElement.

6) NavigableSet<E> headSet(E toElement, boolean inclusive)

- Returns a view of the portion of this set whose elements are less than (or equal to if inclusive is true) toElement.



example:

```
import java.util.*;
class TreesetDemo3
{
  psvm(String args[])
  {
    Treeset<Integer> ts = new
    Treeset<Integer>();
    ts.add((Integer)(40));
    ts.add(50);
    ts.add(10);
    ts.add(20);
    ts.add(30);
    sop(ts);
```

```
NavigableSet<Integer> n1 =
                ts.tailSet(30, true);
System.out.println(n1);
NavigableSet<Integer> n22
                ts.headSet(30, true);
System.out.println(n2);
NavigableSet<Integer> n32
        ts.subSet(20, true, 40, true);
System.out.println(n3);
}
}
```

O/P: [10, 20, 30, 40, 50]
tailset → [30, 40, 50] → greaterthan
headset → [10, 20, 30] → lessthan
subset → [20, 30, 40] → between

example: (Floor & Ceiling)

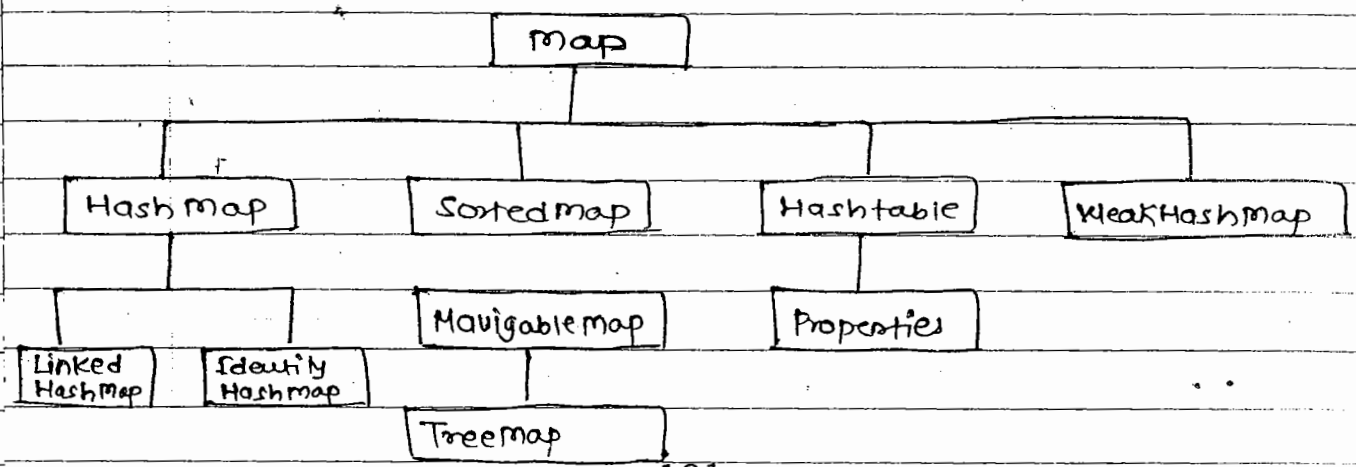
```
import java.util.*;
class Treesetdemo4
{
  P.S.V.M(String args[])
  {
    Treeset<Float> ts = new
    Treeset<Float>();
    ts.add(1.5F);
    ts.add(1.56F);
    ts.add(1.57F);
    ts.add(1.58F);
    ts.add(1.59F);
    ts.add(1.6F);
    System.out.println(ts);
    Float a = ts.floor(1.57F);
    System.out.println(a); → 1.57
    sop(ts.floor(1.53F)); → 1.53
    Float b = ts.ceiling(1.53))
    System.out.println(b); → 1.56
  }
}
```

Map:

- Map is an interface.
- An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.
- The Map interface provides three collection views, which allow a map's content to be viewed as a set of keys, collection of values, or set of key value mappings.
- The order of map is defined as the order in which the iterators on the map's collection views return their elements.
- searching is more efficient.

Que) What is the difference bet^n List, set and Map.

List	Set	Map
1) Index based collection	1) Not an index based collection	1) key based collection.
2) Ordered collection	2) Unordered collection	2) Unorder collection.
3) allows duplicates	3) it does not allow duplicate.	3) It allows duplicate value but does not allows duplicate keys.
4) allow more than one null	4) allows almost one null	4) allows at more one null.
5) Single Dimensional	5) Single Dimensional	5) MultiDimensional.



30/08/2013

Page No. 24
Date

Methods of Map interface:

- void clear()
  - Removes all of the mappings from this map (optional operation).
- boolean containsKey (Object key)
  - Returns true if this map contains a mapping for the specified key.
- boolean containsValue (Object value)
  - Returns true if this map maps one or more keys to specified value.
- Set<Map.Entry<K,V>> entrySet()
  - Returns a set view of the mappings contained in this map.
- boolean equals (Object o)
  - compares the specified object with this map for equality.
- V get (Object key)
  - Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
- int hashCode()
  - Returns the hashcode value for this map.
- boolean isEmpty()
  - Returns true if this map contains no key-value mappings.
- Set<K> keySet()
  - Returns Set view of the keys contained in this map.
- V put (K key, V value)
  - Associates the specified value with the specified key in this map (optional operation)

- Void putAll (Map<? extends K, ? extends V> m)
  - copies all of the mappings from the specified map to this map (optional operation).
- V remove(Object key)
  - Removes the mapping for a key from this map if it is present (optional operation).
- int size()
  - Returns the number of key-value mappings in this map.
- Collection<V> values()
  - Returns a Collection view of the values contained in this map.

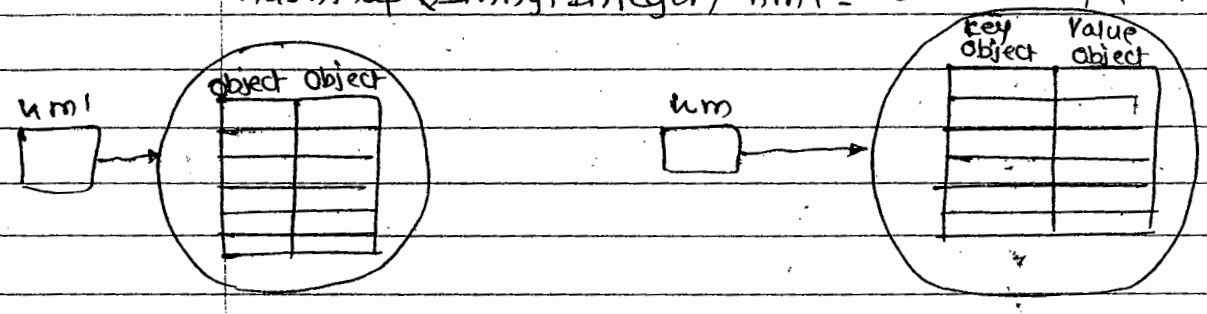
*) HashMap:

- Hash table based implementation of Map interface. This implementation provides all of the optional map operations and permits null values and the null key.
- The HashMap class roughly equivalent to Hashtable, except that it is unsynchronized & permits nulls.
- This class makes no guarantees as to the order of map; in particular it doesnot guarantee that the order will remain constant over time.

Constructor:

- HashMap():
  - constructs an empty HashMap with the default initial capacity (16) and the default load factor (0.75).

HashMap hm = new HashMap();
HashMap<String, Integer> hm1 = new HashMap<String, Integer>();



example:

```
import java.util.*;
class Hashmapdemo1
{
  P.S.V.M(String args[])
  {
   HashMap<Integer, String> hm = new HashMap<Integer, String>();
   hm.put(1, "one");     hm.put(4, "four");
   hm.put(2, "two");     hm.put(5, "five");
   hm.put(3, "three");   hm.put(6, "six");
   SOP(hm);
  }
}
```

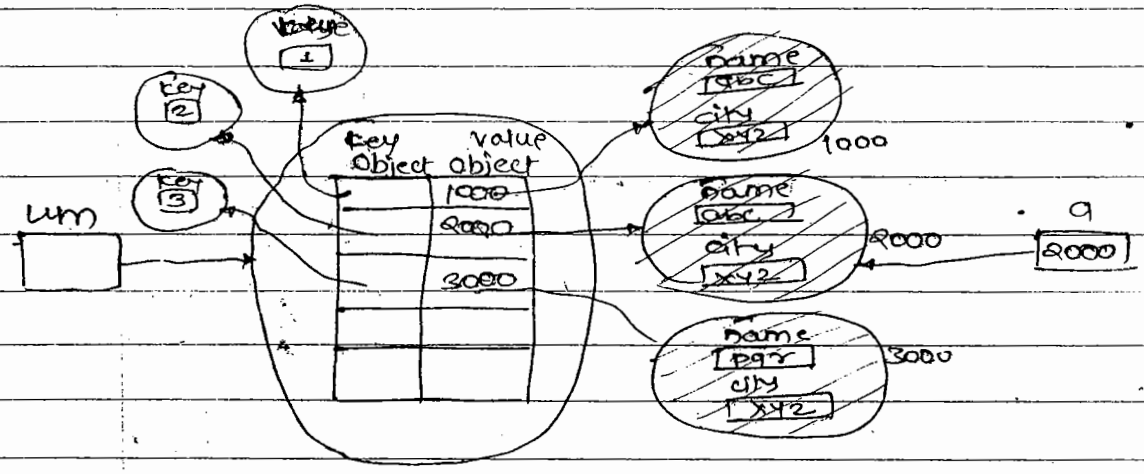
Note
put method is used to adding as well as modifying element.

O/P: {1=one, 2=two, 3=three, 4=four, 5=six}

// How to add userdefined objects or custom objects:

```
import java.io.*;
class Address
{
private String name, city;
  Address(String name, String city)
  {
    this.name=name;
    this.city=city;
  }
  String getAddress()
  {
    return name+" "+city;
  }
}
```

```
Class HashMapDemo2
{
    P.S.V.M(String args[])
    {
        HashMap<Integer, Address> hm = new HashMap<Integer, Address>();
        hm.put(1, new Address("abc", "xyz"));
        hm.put(2, new Address("abc", "xyz"));
        hm.put(3, new Address("pqr", "xyz"));
        System.out.println(hm);
        Scanner sc = new Scanner(System.in);
        System.out.println("Input Phone No. =");
        int phno = sc.nextInt();
        Address a = hm.get(phno);
        if(a==null)
            { SOP("phone number not exists"); }
        else
            { System.out.println(a.getAddress()); }
    }
}
```



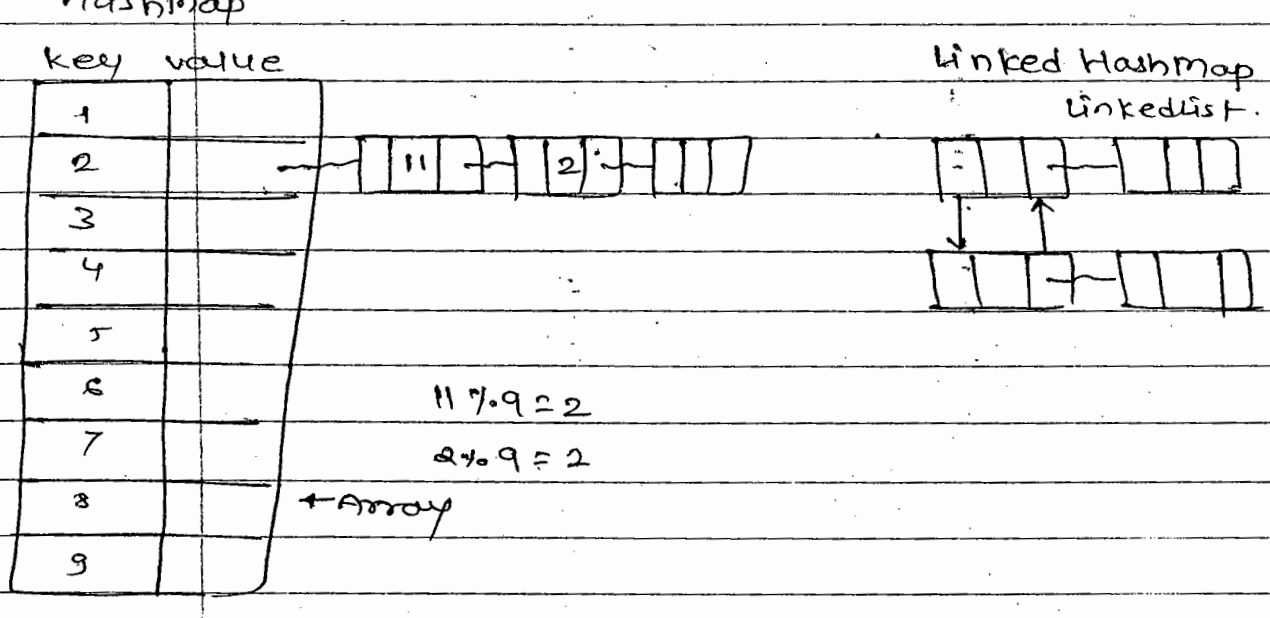

```
// Reading values from map without using key
import java.util.io.*;
class HashMapDemo3
{ P S V M (String args[])
  { HashMap<Integer, String> hm = new HashMap<Integer, String>();
    hm.put(1,"one"); hm.put(3,"Three"); hm.put(5,"Five");
```

```
Collection <String> c = hm.values();
Iterator <String> i = c.iterator();
while (i.hasNext())
{
    System.out.println (i.next());
}
}
```

31/08/2013

2) LinkedHashMap:

- Hashtable and Linkedlist implementation of the map interface, with predictable iteration order. This implementation differ from Hashmap in that is maitain a doubly Linkedlist running through all of its entries. This linkedlist defines the iteration ordering which is normally the order in which keys were inserted into map (insertion-order).



Que: What is the difference bet^n HashMap & LinkedHashMap.

HashMap	LinkedHashMap.
1) Insertion order is not constant over time.	1) Insertion order is constant.
2) Hash table with array implementation.	2) Hashtable with LinkedList implementation.

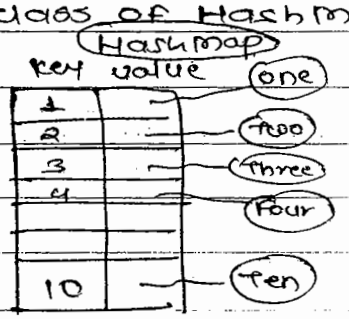
- LinkedHashMap is subclass of HashMap.

example:

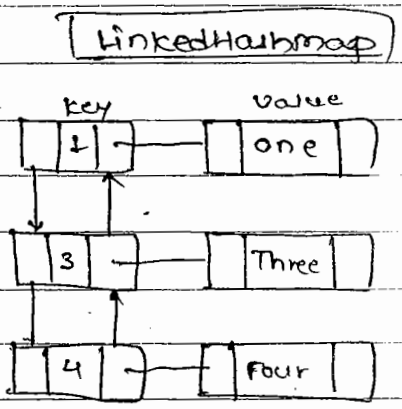
```
import java.util.*;
class LinkedHashMapDemo1
{
    psvm (String args[])
    {
    LinkedHashMap<Integer,String> lhm=
    new LinkedHashMap<Integer, String>();
    lhm.put(1,"one");
    lhm.put(3,"Three");
    lhm.put(4,"Four");
    sop(lhm);
    lhm.put(2,"Two");
sop(lhm);
}}
```

op

```
[1=one, 3=Three, 4=Four]
[1=one, 3=Three, 4=Four, 2=Two]
```



insertion order will change over time.



- The element which is added as key in Hashmap must override equals and hashCode methods.

example:

```
import java.util.*;
class PhoneNumber
{   private int phno;
    PhoneNumber(int phno)
{ ...
```

```
public boolean equals (Object o)
{
PhoneNumber p = (PhoneNumber) o;
if (phno == p.phno)
return true;
else
return false;
}
public int hashCode()
{
   return phno;
}
public String toString()
{ return String.valueOf(phno);
}}
```

```
class LinkedHashmapDemo2
{
P.S.V.M. (String args[])
{
LinkedHashmap<PhoneNumber, String>
lhm = new LinkedHashmap<PhoneNumber, String>();
lhm.put(new PhoneNumber(1), "XYZ");
lhm.put(new PhoneNumber(2), "abc");
lhm.put(new PhoneNumber(3), "pqr");
llm.put(new PhoneNumber(1), "mno");
SOP(lhm);
}
}
```

=> ~~Hashtable~~

Que: What is difference between Hashmap, ~~Hashtable~~, Hashset.

Hashset:

- Hashset does not allow duplicate values.
- It provides add method rather put method.
- You also use it's contain method to check whether the object is already available in Hashset.
- Hashset can be used where you want to maintain a unique list.

Hashmap:

- It allows null for both key & value
- It is unsynchronized.

=> Hashtable:

- This class implements a hashtable which map key to values. Any non-null object can be used as a key or as a value.

- To successfully store & retrieve objects from a hashtable, the objects used as keys must implement the hashCode method & equals() method.

Que: What is the difference bet^n HashMap & Hashtable.

HashMap (1.2)	Hashtable (1.0)
1) Not a legacy class.	1) Legacy class
2) methods are not synchronized	2) methods are synchronized.
3) Not Thread safe	3) Thread safe.
4) SingleThread applications	4) Multithreaded applications.

Constructor:

- Hashtable()

- constructs a new, empty hashtable with a default initial capacity (11) & load factor (0.75).

example:

```
Class HashtableDemo1
{
    P.S.V.m(String args[])
    {
    Hashtable<String, Integer> ht = new Hashtable<String, Integer>();
    ht.put("one", 1);
    ht.put("Two", 2);
    ht.put("Three", 3);
    ht.put("Four", 4);
    SOP(ht);
    }
}
```

O/p: {one=1, Three=3, Four=4, Two=2}

* Properties:

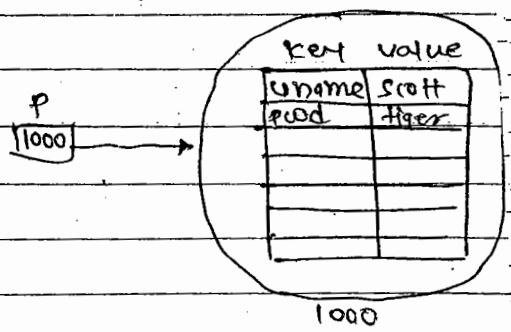
- Properties is a subclass of Hashtable.

- The properties class represent persistent set of properties. The Properties can be saved to a stream or loaded from a stream. Each key and its corresponding value in the property list is a String

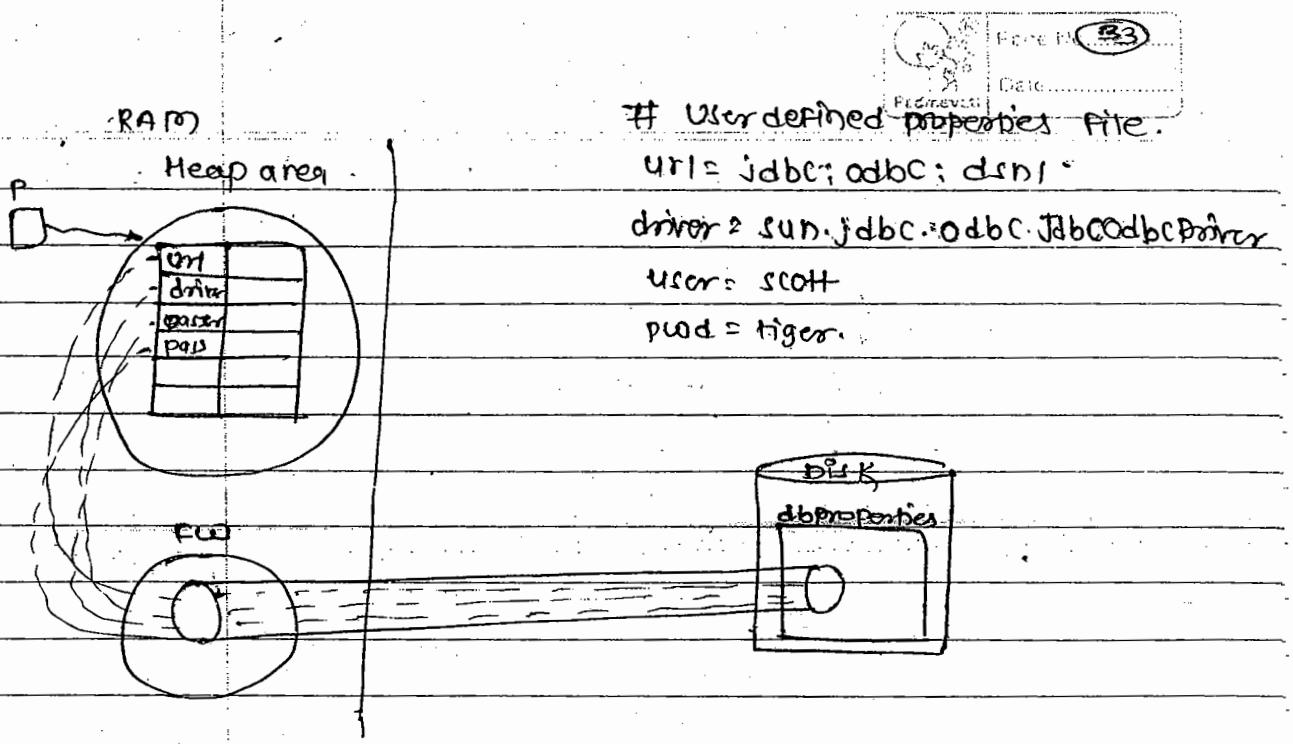
Que: what is the difference between Hashtable & Properties

Hashtable	Properties
1) Allows any type of key & value.	1) Allows only String type key & value
2) Not any methods for persistence	2) provide methods for persistence

```
// Creating Properties
import java.util.*;
class PropertiesDemo1
{
    public s.v.m.(String args[])
    {
        Properties p = new Properties();
        P.setProperty("uname","scott");
        p.setProperty("pwd","tiger");
        System.out.println(P);
        SOP(P.getProperty("uname"));
        SOP(P.getProperty("pwd"));
    }
}
```




```
// Persisting / saving Properties inside file.
import java.util.*;
import java.io.*;
class PropertiesDemo2
{
    PSVM(String args[]) throws Exception
    {
        Properties P = new Properties();
        P.setProperty("uname","scott");
        P.setProperty("pwd","tiger");
        FileWriter fw = new FileWriter("dbproperties");
        P.store(fw," This is database Properties");
        fw.close();
    }
}
```

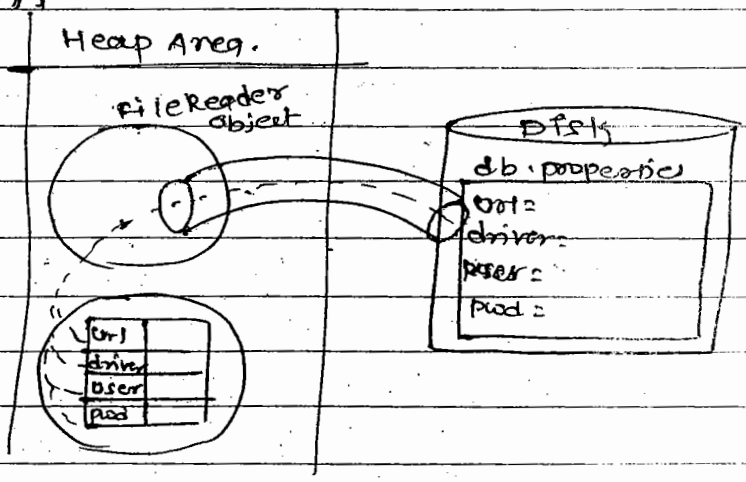


User defined properties file.

```
url = jdbc:odbc:dsn1
driver = sun.jdbc.odbc.JdbcOdbcDriver
user = scott
pwd = tiger
```

// loading properties from db.properties

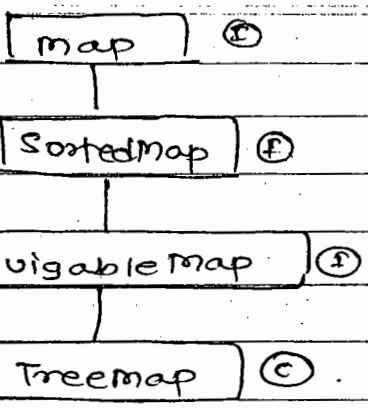
```
import java.util.*;
import java.io.*;
Class PropertiesDemo02
{
    p.s.v.m(String args[]) throws Exception.
    {
        FileReader fr = new FileReader("db.properties");
        Properties p = new Properties();
            p.load(fr);
Class.forName(p.getProperty
        ("driver"));
Connection con=DriverManager.
  getConnection(p.getProperty(
```



```
Connection con = DriverManager.getConnection(P.getProperty("url"),
        P.getProperty("user"), P.getProperty("pwd"));

    SOP(P.getProperty("url");
    SOP(P.getProperty("driver");
    SOP(P.getProperty("user");
    SOP(P.getProperty("pwd");
}
}

// Persistence/saving Properties inside File in xml
format.
import java.io.*;
import java.util.*;
class PropertiesDemo4
{
    P.S.V.M(String args[]) throws Exception
    {
        Properties P = new Properties();
        P.setProperty("unome", "scott");
        P.setProperty("pwd", "tiger");
        FileOutputStream fos = new FileOutputStream("db.xml");
        P.storeToXML(fos, "This is database Properties");
        fos.close();
    }
}
```

- **Void loadFromXML(InputStream in)**
  - loads all of the properties represented by XML document on specified input stream into this properties table.

37 TreeMap



- A Red-Black tree based NavigableMap implementation. The map is sorted according to the natural ordering of its keys or by a Comparator provided at map creation time, depending on which constructor is used.

Que: What is difference between HashMap & TreeMap.

HashMap	TreeMap.
1) Underlying datastructure is hashtable.	1) Underlying datastructure is binary tree.
2) It is Heterogenous collection	2) It is homogenous collection.
3) Comparing is done by overriding equals & hashCode method.	3) Comparing is done using Comparable or Comparator.
4) It is unordered collection	4) Elements are order in ascending or descending.
5) It doesnot provide any navigation methods	5) It provide navigation methods.

```
// Key added inside Treemap must implement
Comparable or Comparator.
import java.util.*;
Class Treemappemo1
{
    P.S.V.M. (String args[])
    {
        Treemap<Integer,String> tm = new Treemap<Integer,
                                        String>();
        tm.put(5, "Five");
        tm.put(3, "Three");
        tm.put(1, "one");
        tm.put(4, "Four");
        tm.put(2, "Two");
        SOP(tm);
        SOP(tm.get(1));
    }
}
```

O/P

{1=one, 2=Two, 3=Three, 4=Four, 5=five}
one.

	Treemap	Treeset.
-	headmap	headset
-	tailmap	tailset
-	subMap	subSet
-	FloorMap	Floor
-	ceilingmap.	ceiling

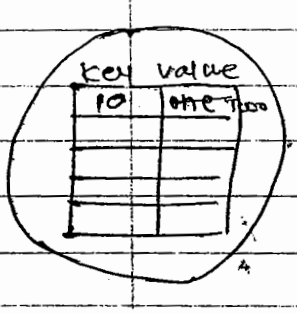
* IdentityHashMap:

- This class implements the Map interface with a hash table, using reference equality in place of object-equality when comparing keys (and values).

--In other words, in an IdentityHashMap two keys k1 & k2 are considered equal iff (k1==k2).

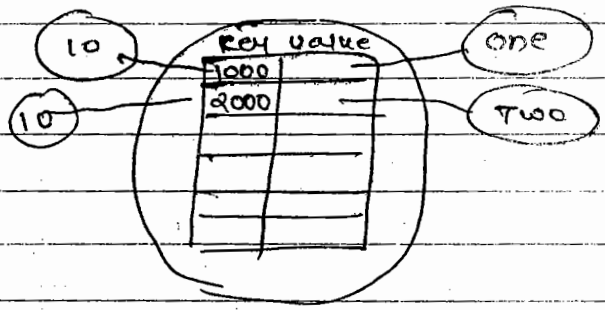
HashMap

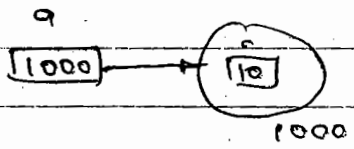
```
HashMap<Integer, String> hm = new
HashMap<Integer, String>();
Integer a = new Integer(10);
Integer b = new Integer(10);
hm.put(a, "one");
hm.put(b, "Two");
```

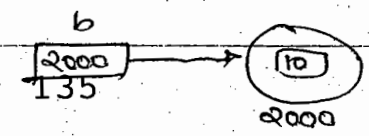


IdentityHashMap

```
IdentityHashMap<Integer String>
idm = new IdentityHashMap<Integer,
                          String>();
Integer a = new Integer(10);
Integer b = new Integer(10);
idm.put(a, "one");
idm.put(b, "Two");
```







example;

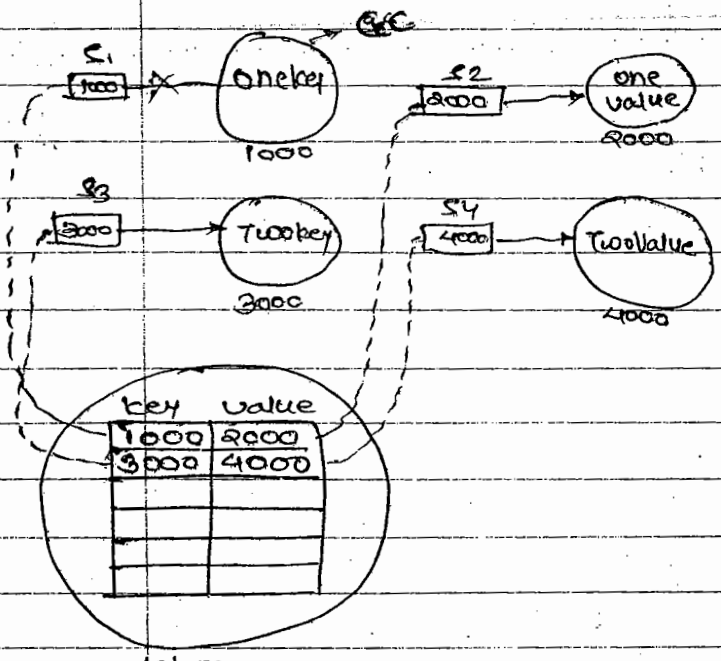
```
import java.util.*;
class IdentityHashmapDemo1
{
  P.S.V.M (String argsLL)
  {
    IdentityHashmap <Integer, String> ihm = new
        IdentityHashmap <Integer, String> ();
    HashMap <Integer, String> hm = new HashMap <Integer, String>();
    Integer a2 new Integer (10);
    Integer b= new Integer (10);
    ihm.put (a,"One");
    ihm.put (b,"Two");
    hm.put (a, "One");
    hm.put (b, "Two");
    SOP( ihm);
    SOP(hm);
  }
}
```

O/P { 10=one , 10=Two }
{ 10=Two }

* WeakHashMap:

- A hashtable based map implementation with weak keys.
- An entry in a WeakHashMap will automatically be removed when its key is no longer in ordinary use.

```
WeakHashMap<String, String> whm = new
        WeakHashMap<String, String>();
String s1 = new String ("one key");
String s1 = new String ("one value");
String s3 = new String ("Two key");
String s4 = new String ("Two value");
whm.put (s1, s2);
whm.put (s3, s4);
s1 = null;
```



example:

```
import java.util.*;
class MyInteger
{
   int n;
   MyInteger()
  { this.n=n; }
public boolean equals (Object o)
{  MyInteger m = (MyInteger) o;
   if (this.n == m.n)
   return true;
   else
} return false;
public int hashcode()
{  return n;
}
protected void Finalize()
{
   sop ("Inside Finalize method");
}
public String toString()
{  return String.valueOf(n);
}
```

```
class WeakHashMapDemo1
{ psvm(String args[]){
  WeakHashMap<MyInteger, String> whm = new
        WeakHashMap<MyInteger, String>();
  MyInteger a = new MyInteger(10);
  MyInteger b = new MyInteger(20);
  whm.put(a, "Hello");
  whm.put(b, "Bye");
  SOP(whm);
  b = null;
  System.gc();
  SOP(whm);
}}
```

O/P :- {10=Hello, 20=Bye}
{10=Hello}
Inside Finalize method.

* **Collections:**

- This consists exclusively of static methods that operate on or return collection. It contains polymorphic algorithms that operate on collections "wrappers" which return a new collection backed by a specified collection, and a few other odds and ends.

- Collections is a class.

1. Methods for converting Nonsynchronized collection to synchronized collection.

- Static <T> List<T> SynchronizedList(List<T> list)

- Returns a synchronized (thread-safe) list backed by specified list.

• Static <K,V> Map<K,V> synchronizedMap (Map<K,V> m)

- Returns a synchronized map backedup the specified map.

• static <T> set<T> synchronizedSet (set<T> s)

- Returns a synchronized set backed by the specified set.

example:

```
import java.util.*;
class ArrayListSync
{
    P.S.V.M (String args[])
    {
     ArrayList <Integer> al=new ArrayList<Integer>();
     al.add (10);
     al.add (20);
     al.add (50);
     al.add (40);
     List <Integer> list1= Collections.synchronizedList(al);
     System.out.println (list1);
    }
}
```

O/P: {10, 20, 50, 40}.

• Static <T extends Comparable <? super T>>
void sort (List<T> list)

- Sorts the specified list into ascending order according to the natural ordering of its elements.

example:

```
import java.util.*;
class SortArrayList
{
  P S V M (String args[])
  {
    ArrayList <Integer> al = new ArrayList <Integer>();
    al.add (30);
    al.add (20);
    al.add (40);
    al.add (50);
    al.add (10);
    Sop (al);
    Collections.sort (al);
    Sop (al);
  } }
```

O/p {30, 20, 40, 50, 10}
{10, 20, 30, 40, 50}

How to create executable jar file :-

Jar utility :-

- Java provide "jar" command to create compressed files.
- This compressed files can be executable or non executable (lib).

```
jar -cvf <jar-file-name> <includefiles>
jar -cvmf <jar-file-name> <manifest-files>
        <includefiles>
```

```
import java.awt.*;
import java.applet.*;
public class Frame1 extends Frame.
{
    Public static void main (String args[])
    {
        Frame1 f = new Frame();
        f.setSize(400,400);
        f.setVisible (true);
    }
}
```

Extract.
jarfile.

```
> jar -cvf jar1.jar Frame1.class
added manifest
```

```
> jar -cvmf META-INF\MANIFEST.MF jar1.jar Frame1.class.
        (dir)    (file)
```

↓

```
manifest-version 1.0
main-class : Frame1..
```

12/08/2013

Reflections: (Reflection API)

- API is an application Programming Interface, which is collections of packages.
- Reflection API is introduced in Java 1.0
- Java is a dynamic programming language, linking of executable modules done at runtime.
- Java achieves dynamic linking using reflections.
- Reflection API provide set of classes used for reading / invoking details of class at runtime.
- Runtime type identification is done by using reflections.

Need of Reflections:

1) Developing loosly coupled application / runtime polymorphism
2) Developing complex framworks.
3) Dynamic linking

Class:

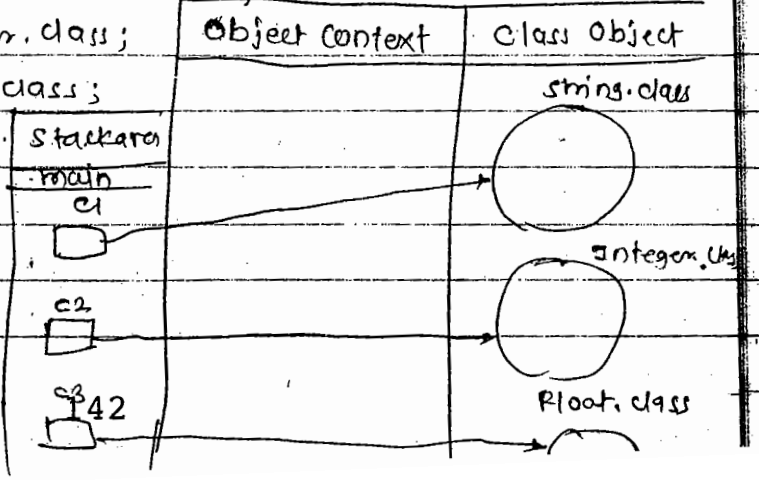
- Class is available java.lang package.
- Class represents class object.
- Class represents class object which is loaded during runtime.

Example:

```
class ClassDemo1
{
    psvm(String args[])
    {
        Class c1 = String.class;
        Class c2 = Integer.class;
        Class c3 = Float.class;
        S.o.p(c1);
        S.o.p(c2);
        S.o.p(c3);
    }
}
```

Class c = new Class() // error

c holds string class object



- The constructor of class is private & cannot used for creating object.

Class a = new Class(); → error

13/08/2013

Adv: of Reflection

1) Loosly coupled applications.
2) Debuggers
3) Class Browsers
4) Tools.

Example:

```
class A
{
  void m1()
  {
    sop("m1");
  }
  void m2()
  {
    sop("m2");
  }
}
```

```
class RefDemo1
{
  public static void main(String args[])
  {
    Class c = A.class();
    s.o.p(c);  s.o.p(c.hashCode());
  }
}
```

* Methods of Class:

1) public String getName():

- return name of the class.

```
class RefDemo2
{
  public static void main(String args[])
  {
    Class c = java.lang.String.class;
    sop(c.getName());
  }
}
```

2) - Returns the name of entity (class, interface, array class, primitive types or void) represented by this Class object as a String.

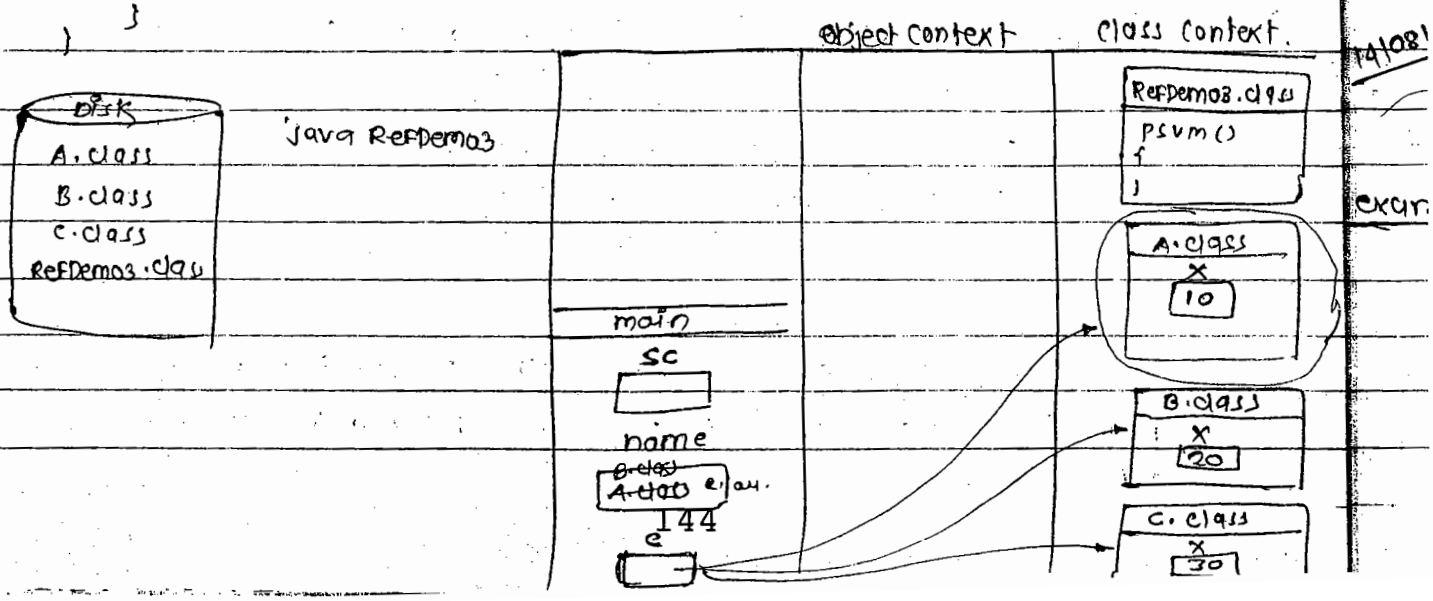
2) public static Class forName(String)

- This method loads the class from secondary to primary.
- It load the class by representing classname as string.
- It is used for loading class dynamically.

example: import java.util.Scanner;

```
class A                    class C
{                          {
  static int x=10;           static int x=30;
}                          }

class B
{
  static int x=20;
}

class RefDemo03
{ public static void main (String args[]) throws Exception
  {
      Scanner sc= new Scanner(System.in);
      S.O.p("Enter Class Name :");
      String name= sc.next();
      Class c= Class.forName(name);
      System.out.println(c);
  }
}
```

Class
ClassLoader
URLClassLoader



3) public Object newInstance() throws InstantiationException, IllegalAccessException.

- This method is used for creating object.

new	newInstance()
1) It is a keyword	1) It is a method.
2) required class at compile time.	2) required class at runtime.

```
// Write a program to create object of any class.
Class RefDemo4
{   public static void main (String args[]) throws Exception
    {
        Scanner sc = new Scanner (System.in);
        System.out.println ("Enter class Name :");
        String name = sc.next();
        Object o = Class.forName (name).newInstance();
    /* (OR)
        Class C = Class.forName (name);
        Object o = C.newInstance();
    */
        System.out.println (o);
    }
}
```

19/08/2013

- The primitive Java types and keyword void are also represented as class objects.

Example:

```
Class RefDemo5
{
    public static void main (String args[])
    {
        Class c = int.class(); sop(c); sop(c.hashCode);
        Class c1 = Float.class(); sop(c1); sop(c1.hashCode);
    }
}
```

14/08/2013

- newInstance method create an object of loaded class by invoking default constructor.
- If their is no default constructor inside loaded class, newInstance() method throws InstantiationException

4) String getCanonicalName():

- Returns the canonical name of underlying class as defined by Java Language specification.

example:

```
class RefDemo6
{ public static void main(String args[])
  {
    Class c1 = Object.class;
    Class c2 = int.class;
    Class c3 = System.class;
    Class c4 = String.class;
    SOP(c1.getName()+" "+c1.getCanonicalName());
    SOP(c2.getName()+" "+c2.getCanonicalName());
    SOP(c3.getName()+" "+c3.getCanonicalName());
    SOP(c4.getName()+" "+c4.getCanonicalName());
  }
}
```

O/P

```
java.lang.Object  java.lang.Object
int  int
java.lang.System  java.lang.System
java.lang.String  java.lang.String
```

exam

```
cla
{
}
```

* Class.forName("String").newInstance() → (X).

```
Class c2 = String.class
String name = c.getCanonicalName();
Class.forName(name).newInstance();
```

forName method required fully qualified class name along with package name.

```
Connection c = DriverManager.getConnection(url, user, pwd);
```

// How to findout class name of created object.

```
class A                          class RefDemo7
{                                {  public static void main (String args[])
}                                {    A obja = new B();
class B extends A                    Class c = obja.getClass();
{                                    SOP (c.getName());
}                                }  }
```

* getClass(): It is a method of Object class.
  - public final class <?> getClass()
  - Returns the runtime class of this object.

The returned class object is object that is locked by static synchronized methods of the represented class.

example:

```
class RefDemo8
{  public static void main (String args[])
   {
      Number n1 = 0;
      SOP (n1.getClass().getName());
      Number n2 = 1.5;
      Number n3 = 1.5F;
      Object o = "XYZ";
      SOP (n2.getClass().getName());
      SOP (n3.getClass().getName());
      SOP (o.getClass().getName());
   }  }
```

O/P
java.lang.Integer;
java.lang.Double.
java.lang.Float
java.lang.String

* Accessing Fields Information
  - Field getField (String name)
    - Returns a Field object that reflects the specified public member Field of the class or interface represented by this Class object.

• Field[] getDeclaredField():

- return an array of Field obj reflecting all the field declared by the class or interface this represented by this object.

• Field[] getFields()

- Returns an array containing field objects reflecting all the accessible public fields of the class or interface represented by this class object.

15/08/2013

- All the classes which belongs to reflection available in java.lang.reflect package. This package is introduced in java 1.1. Reflection API or patage is used to manipulate class information at runtime.

* Field:

- A field provides information about dynamic access to, a single field of a class or an interface. The reflected field may be class (static) field or an instance field.

Methods:

• String getName():

- Returns name of field represented by this Field object.

exan

• int getModifiers():

- Returns the Java language modifiers for the field represented by this field object as an integer.

example

```
// WAP to display the fields exist in particular class.
import java.lang.reflect.*;
import java.util.Scanner;
class RefDemo9
{   p.s.v.main(String args[]) throws Exception.
    {
        Scanner sc = new Scanner(System.in);
        S.O.P("Enter class name");
        String name = sc.next();
```

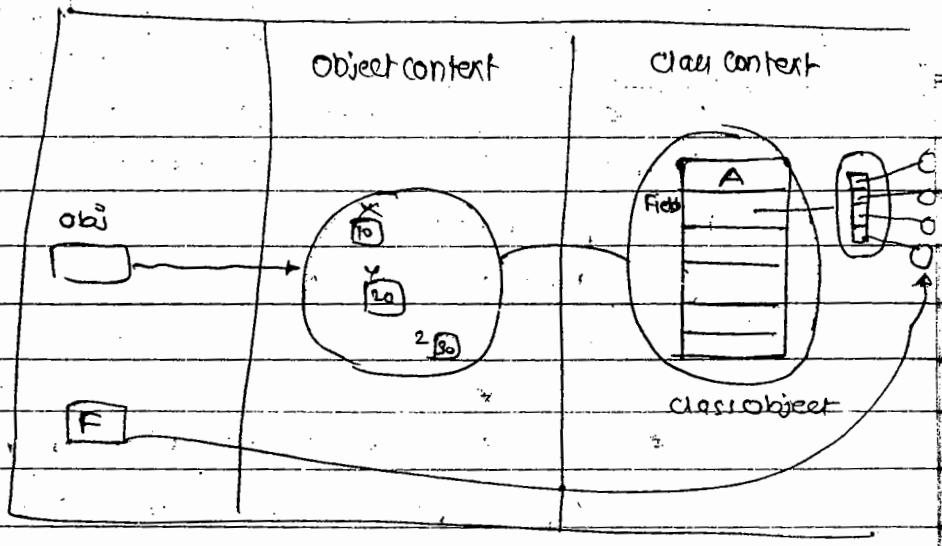
```
Class c = Class.forName(name);
Field f[] = c.getDeclaredFields();
for (Field fl : f)
SOP (fl.getName());
}
}
```

- How to read value of given field at Runtime :
  - boolean getBoolean (Object obj)
    - Gets the value of static or instance boolean field.
  - byte getByte (Object obj)
    - Gets the value of static or instance byte field
  - char getChar (Object obj)
    - Gets the value of a static or instance field of type char or of another primitive type convertible to type char via a widening conversion.
  - double getDouble (Object obj)
  - float getFloat (Object obj)
  - int getInt (Object obj)

example: // Reading value of given field.

```
import java.lang.reflect.*;
import java.util.Scanner;
Class RefDemo10
{ p.s.v.m (String args[]) throws Exception
  {
    Scanner sc = new Scanner (System.in);
    Class c = Class.forName ("A");
    A obj = (A) c.newInstance();
    SOP ("Enter Filed name");
    String name = sc.next();
    Field f = c.getDeclaredField (name);   f.setAccessible (true);
    SOP (f.getInt (obj));
  }
}
```

```
Class A
{ public int x = 10;
  public int y = 20;
  public int z = 30;
  private int a = 40;
}
```



// How to access private members & modify values:-

```
import java.lang.reflect.*;
import java.util.*;
class A
{
    private int x=10;
    private int y=30;
}
class Reflemo11
{ public static void main(String args[]) throws Exception
  {
      Scanner sc = new Scanner(System.in);
      A obj = new A();
      Class c = obj.getClass();
      Field f1 = c.getDeclaredField("x");
      Field f2 = c.getDeclaredfield("y");
      f1.setAccessible(true);
      f2.setAccessible(true);
      S.o.p(f1.getInt(obj));
      S.o.p(f1.getInt(obj));
      f1.setInt(obj, 40);
      (f2.setInt(obj, 60));
      sop(f1.getInt(obj));
      sop(f2.getInt(obj));
  }
}
```

150

- Field class provide setter methods to change the value of field related to an object.
  - setInt(Object, int)
  - setFloat(Object, float)
  - setDouble(Object, double)

Accessing Modifiers of Fields:

- public int getModifiers()
  - Return the Java language modifiers for the field represented by this Field object as an integer. The modifier class should be used to decode the modifiers.

16/08/2013

* Working with methods:
  - Method getDeclaredMethod(String name, Class<?>... paramTypes)
    - Returns a method object that reflects the specified declared method of the class or interface represented by this class object.

  - Method[] getDeclaredMethods():-
    - Returns an array of method objects reflecting all the methods declared by the class or interface represented by this class object.

  - Method getMethod(String name, Class<?>... param.types)
    - Returns a method object that reflects the specified public member method of the class or interface represented by this class object. ~~reading~~

(...)- variable length Arguments.

- Method[] getMethods()

  - Returns an array containing Method objects reflecting all the public member methods of the class or interface represented by this class object, including those declared by the class or interface & those inherited from superclass & supe interfaces.

- Method getMethods(String name, Class<?>... paramTypes)

  name of the method.

  parameter types of method.

  - Returns one Method object.

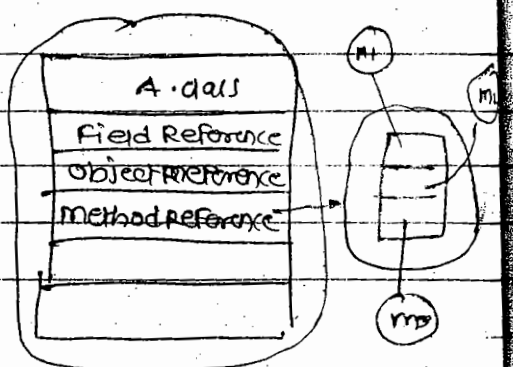
- Method[] getMethods()

  - Returns array of Method object.

example:

```
ClassA
{
  public void m1()
  {
    sop("m1");
  }
  public void m2()
  {
    sop("m2");
  }
  public void m3()
  {
    sop("m3");
  }
}
```

```
class RefDemo12
{
  public static void main(String args[])
  {
    Class c = A.class;
    Method m[] = c.getMethods();
    SOP(m);
    SOP(m[0].getMethods());
    SOP(m[0].getMethods());
    SOP(m[2].getMethods());
  }
}
```



* Method class:-

- Method class is available java.lang.reflect package
- A Method provides information about and access to a single method on a class or interface. The reflected method may be a class method or an instance method (including an abstract method).

• String getName():

- Returns the name of the method represented by this Method object, as a String.

example:

```
import java.lang.reflect;
class A
{
public void m1()
{ sop("m1");
}
public void m2()
{ sop("m2");
}
void m3()
{ sop("m3");
}
void m4()
{ sop("m4");
}
}
```

```
class RefDemo13
{
  public static void main(String args[])
  {
    Class c = A.class;
    Method m1[] = c.getMethods();
    Method m2[] = c.getDeclaredMethods();
    for (Method m : m1)
      sop(m.getName());
    for (Method m : m2)
      sop(m.getName());
  }
}
```

o/p: m1
m2
m1
m2
m3
m4

Note:
- getmethods returns only public methods in current class + object class.
- getDeclared method returns all the methods in current class.

// Reading one method by giving method Name at Runtime.

```
import java.lang.reflect.*;
import java.util.Scanner.*;

class A
{
    public void m1()
    {
        sop("m1");
    }
    public void m2()
    {
        sop("m2");
    }
}
```

```
class RefDemo14
{
    public static void main(String args[]) throws
                                        Exception
    {
        Scanner sc = new Scanner(System.in);
        sop("Enter method Name");
        String name = sc.next();
        Class c = A.class;
        // Method m = c.getMethod(name);
        Method m = c.getDeclaredMethod(name);
        sop(m.getName());
        sop(m.getModifiers());
        sop(m.getReturnType());
    }
}
```

O/P Enter method name
m1
Method Name - m1
Modifier - 1 (public).
ReturnTyp - void.

- Object invoke(Object obj, Object... args) :
  - Invokes the underlying method represented by this method object, on specified object with specified parameters.

// Invoking methods whose name is given at runtime.

```
import java.lang.reflect.*
import java.util.Scanner;
class Reports
{
    public void monthly()
    {
        sop("Inside Monthly Report");
    }
    public void yearly()
    {
        sop("Inside yearly Report");
    }
}
```

```
class RefDemo15
{
    public static void main(String args[])
                                throws Exception
    {
        Scanner sc = new Scanner(System.in);
        sop("Enter Report type");
        String name = sc.next();
        Class c = Reports.class;
        Reports r = (Reports)c.newInstance();
        Method m = c.getDeclaredMethod(name);
        m.invoke(r);
    }
}
```

```
                                        m.setAccessible(true);
private void weekly()                   m.invoke(r);
{                                       }
  SOP("Inside weekly Report");      }
}
```

```
Integer i[] = {10,20}
m.invoke(obj,i);
```

07/08/13

// Invoking method having parameters:

```
import java.lang.reflect.*;
import java.util.io.*;
Class A
{
  void max(int x, int y)
  { SOP if(x>y)
      SOP("%d is max", x);
    else
    SOP("% is max", y);
  }
  void min(int x, int y)
  {
    if(x<y)
    SOP("%d is min", x);
    else
    SOP("%d is min", y);
  }
}
```

```
class Reflemo16                                    throws Exception
{ public static void main(String args[])
  {
    Scanner sc = new Scanner(System.in);
    Class c = Class.forName("A");
    A obj1 = (A) c.newInstance();
    SOP("Input method Name = ");
    String name = sc.next();
    Method m = c.getDeclaredMethod(
    name, new Class[] { int.class, int.class});
    Object o[] = new Object[2];
    o[0] = new Integer(10);
    o[1] = new Integer[20];
    m.invoke(obj1, o);
  }
}
```

* Constructors:

- Class provide two methods inorder to read constructor of a class at runtime.
- Constructor<T> getConstructor(Class<?>... paramTypes)
  - Returns a constructor object that reflects the specified public constructor of class represented by this class obj.
- Constructor<?>[] getConstructors()
  - Returns an array containing Constructor objects reflecting all the public constructor of the class represented by this Class object.

- Constructor <T> getDeclaredConstructor(Class<?>... parameterTypes)
  - Returns a constructor object that reflects the specified constructor of the class or interface represented by this object.
- Constructor <?>[] getDeclaredConstructors()
  - Returns an array of constructor object reflecting all constructors by the class represented by this Class object.

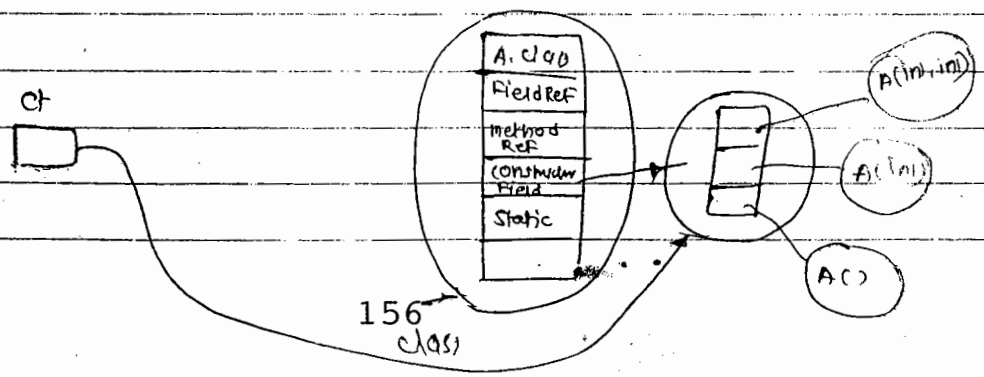
Constructor Class:

- This class available in java.lang.reflect package.
- constructor provides information about, and access to, a single constructor of a class.

example: // Program to Display constructor available in given Class.

```
class A
{
  A()
  {
    sop("D.C");
  }
  A(int x)
  {
    sop("O.P.C");
  }
  A(int x, int y")
  {
    sop("T.P.C");
  }
}
```

```
class ReflDemo17
{
  public static void main(String args[])
  {
    Class C = A.class;
    Constructor ct[] = C.getDeclaredConstructors();
```



• String getName()

- Returns the name of the constructor as a String.

• Class<?>[] getParameterTypes()

- Returns an array of Class objects that represent the formal parameter types in declaration order, of the constructor represented by this constructor object.

• T newInstance (Object... initargs)

- Uses the constructor represented by this Constructor object to create and initialize a new instance of the constructor's declaring class, with a specified initialization parameters.

Example:

```
import java.lang.reflect.*;
import java.util.*;
class RefDemo18
{ public static void main(String args[]) throws Exception
  {
     Scanner sc=new Scanner (System.in)
     System.out.println (" Input class name");
     String name= sc.next();
     Class c2 class.For Name (name);
    Constructor cn[] = c.getDeclaredConstructors();
    for ( Constructor c1 : cn)
   { SOP ("constructor Name " + c1.getName());
    Class types[] = c1.getParameterTypes();
    for ( class type : types)
     SOP (type);
  }
 }
}
```

// creating object of class getting constructor information at runtime.

```
import java.lang.reflect.*;
import java.util.*;
class RefDemo19
{ public static void main (String args[]) throws Exception.
  {
    Scanner sc=new Scanner (System.in);
    SOP (" Input Class Name");
    String name = sc.next ();
    Class c = Class.forName(name);
    Constructor cn = c.getDeclaredConstructor();
    cn.setAccessible(true);
    Object o = cn.newInstance();
    SOP(o);
  }
}
```

19/08/2013 * ClassLoader :-

Methods used to verifying types;

- boolean isArray()
  - Determines if this class object represent an array class.
- boolean isInstance();
  - Determines if the specified object is assignment compatible with the object is represent by this class.
- boolean isInterface();
  - Determines if the specified class object represent by an Interface class.
- boolean isLocalClass();
  - Returns true iff the underlying class is a local class.
- boolean isMemberClass();
  - Returns true iff the underlying class is member class.
- boolean isPrimitive();
  - Determine if the specified class object represents a primitive types.

- boolean isAnonymousClass();
  - Returns true iff the underlying class is Anynomemous class.

example:

```
import java.util.*;
class RefDemo20
{ public static void main(String args[]) throws Exception
  {
   Scanner sc=new Scanner(System.in);
   SOP("Input class Name");
   String name= sc.next();
   Class c= Class.forName(name);
   if(c.isInterface())
   SOP(" Not a valid Type");
   else if (c.isClass())
  { SOP("
   Object o= c.newInstance();
   SOP (o.hashCode());
}}
```

- int getModifiers()
  - Returns the Java language modifiers for this class or interface encoded in an integer.

examples:

```
import java.util.*;
class RefDemo21
{ psvm main (String args[])
  {
   Scanner sc=new Scanner(System.in);
   SOP(" Input class name");
   String name= sc.next();
   Class c= Class.forName(name);159
   int m= c.getModifiers();
   if(m==1024)
   SOP("It is an abstract class");
   else if(m==512)
   SOP("It is an Interface");
   else if(m== java.lang.reflect.Modifier.FINAL)
   SOP(" It is final class");
   else
   SOP("It is class"); }}
```

* URLClassLoader:

20108

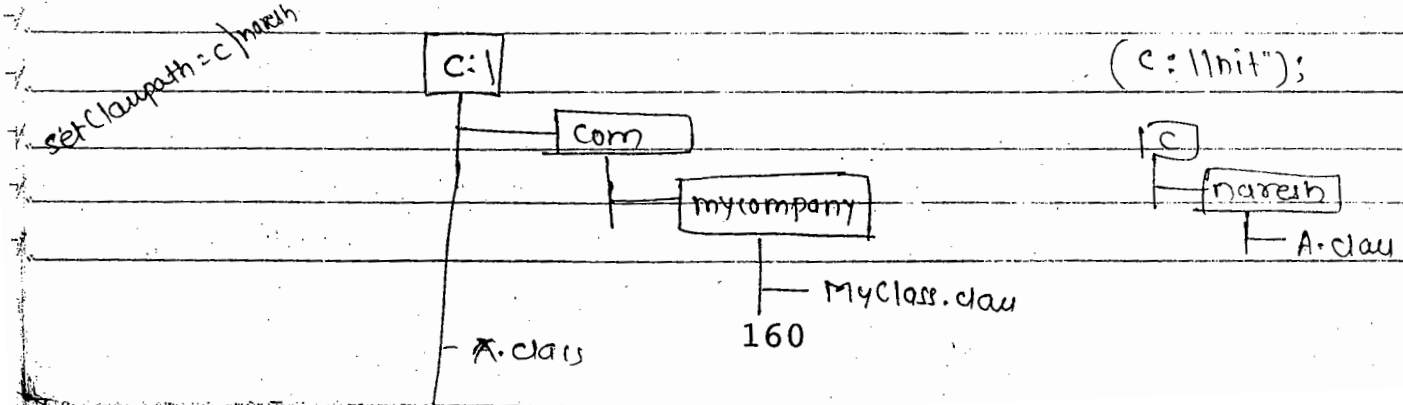
- This class is available in java.net package.
- This class loader is used to load classes & resources from a search path of URLs refering to both JAR files & directories. Any URL that ends with a '/' is assumed to refer to directory.

• URLClassLoader (URL[] urls)

- constructs a new URLClassLoader for the specified URLs using the default delegation parent ClassLoader.

example:

```
// A URLClassLoader can be used to load classes in any directory.
import java.io.File;
import java.net.URL;
import java.net.URLClassLoader;
class Main{
  Public static void main (String args[])
  {
    File File = new File ("C:\\");
    URL url = File.toURI.toURL();
    URL[] urls = new URL[]{url};
    ClassLoader cl = new ClassLoader (urls);
    Class cls = cl.loadClass ("com.mycompany.MyClass");
    System.out.println (cls);
  }
}
```



20/08/2013

* Annotations *

- It is a new feature introduced in java 5.0
- Annotations allows declarative programming
- Annotation is a special type
- Annotation is a interface.
- Annotation a form of metadata, provide data about a program i.e. not part of program itself.
- Annotation have no direct effect on the operation of the code they annotate.
- Annotations have a no. of uses, among them:

① Information for the compiler

- Annotations can be used by compiler to detect errors or suppress warnings.

② Compile time + deployment time processing

- software tools can process annotation info. to generate code, XML files & so on.

③ Runtime processing:

- some annotations are availble to examined at runtime.

- A set of annotation types are predefined in Java SE API. some annotation types are used by Java compiler, & apply to other annotations.
- The predefined annotation types defined in java.lang are
  1) @Deprecated
  2) @Override
  3) @SuppressWarnings.
  4) @Inherited.
- All this are ~~often~~ known as simple annotations.

1) @Deprecated:

- This annotation indicates that the marked element is deprecated and should no longer be used. The generates warnings whenever a program uses a method

class or field with @Deprecated annotation. whenan element is deprecated, it should be also documented using Javadoc @deprecated tag.

Syntax:

example

```
class A                          class B extends A
{                                {
  @Deprecated                      public static void main(String args[])
  void m1()                        {
  {                                  A obja = new A();
    sop("Hello");                    obja.m1();
  }                                }
}                                }
@Deprecated
class B extends A
{
}
```

1) What is deprecated?

- A method, class or fields which are modified or removed in future versions is declared with Deprecated annotation.

2) @Override:

- This annotation informs the compiler that the element is meant to override an element declared in a superclass.

example:

```
class A                          class B extends
{                                {
  void m1(int x)                   @Override
  {                                void m1(int x)
    sop("Inside A");               {
  }                                  sop("override method");
}                                }
                                 }
```

javac -xlint AnnEx.java.

- Java compiler verifies m1() method of subclass is available in superclass.
- If not compiler generate an error.

example

```
class A
{
    @Override
    public String toString()
    {
        return "Hello";
    }
}
```

→ Compile time error, because 'A' class does not override toString method of Object class.

3) @SuppressWarnings:-

- This annotation tells the compiler to supress specific warnings that it would otherwise generate.

example:

```
import java.util.*;
class AnnEx
{
  public static void main(String args[]);
  {
    ArrayList al = new ArrayList();
      al.add(10);
      al.add(20);
      SOP(al);
  }
}
```

⇒ unchecked warning

```
class A
{
  @Deprecated
  void m1()
  { SOP("m1"); }
}
class B extends A
{
  P S V m(String args)
  {
    A obj = new A();
    obj.m1();
  }
}
```

⇒ deprecation warning.

Syntax: @SuppressWarnings ("warning")

example

```
import java.util.*;
class AnnEx2
{
    @SuppressWarnings ("unchecked")
    public static void main (String args[])
    {
        ArrayList al = new ArrayList ();
        al.add (10);
        al.add (20);
        SOP (al);
    }
}
```

22/08/2013

* Meta Annotations:

- All meta annotations are available in java.lang.annotation package.
- Meta annotation define the properties & behaviour of annotation.

1) @Retention meta Annotation:

- How long annotation information is kept.
- Enum RetentionPolicy
  - > SOURCE -
    - It indicates information will be placed in source file but will not be available from class file.
  - > CLASS (Default);
    - It indicates that info. will be placed in class file but will not be available at runtime through reflection.

> RUNTIME :

It indicates that information will be stored in class file and made available at runtime through reflective APIs.

Syntax:

```
@ Retention (value = SOURCE)
@ interface MyAnnotation
{
}
@ Retention (value = RUNTIME)
@ interface MyAnnotation1
{
}
```

2) @ Target Meta Annotation :-

- Restrictions on use of this annotation.
- Enum ElementType
  - > TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE, ANNOTATION_TYPE, PACKAGE.

Syntax:

```
@ Target (value = METHOD)
@ Target MyAnnotation
{
}
```

e.g:

```
@ MyAnnotation
class A
{
}
```

⇒ invalid.

* Types of Annotations :-

1) marker Annotation
2) Single value Annotation
3) Normal Annotation.

1) ~~Marker Annotation~~:

* How to Define Annotation Type:

- Annotation types definitions are similar to normal Java interface definitions.
  - > An at-sign (@) precedes the interface keyword.
  - > Each element declaration defines an element of the annotation type.
  - > method declaration must not have any parameters or a throws clause
  - > Return types are restricted to premitives, String, Class, enums, annotations and arrays of preceding types.
  - > methods can have default values.

1) Marker Annotation :

- An Annotation type with no elements

• Definition

Indicates that the specification of annotated API element is preliminary and subject to change.

```
public @interface Preliminary { }
```

Usage:

```
@preliminary
public class TimeTravel
{
...
}
```

DBA.java. X

example: import static java.lang.annotation.RetentionPolicy.*;

import java.lang.annotation;*; import static java.lang.annotation.ElementType.*;

```
@Target (value = TYPE)
@Retention (value = RUNTIME)
@interface DBA
{
}
```

```
import java.lang.annotation.*;
@Target(value = ElementType.TYPE);
@Retention (value = RetentionPolicy.RUNTIME);
@ interface DBA.
{
}
```

```
@DBA
class A
{
}
class B
{
}
```

```
import java.lang.annotations.*;
class DBAOperations
{
    void drop(object o)
    {   boolean flag= false;
        Class c = o.getClass ();
        Annotation a[] = c.getAnnotations();
        for ( Annotation a1 : a)
        {
            if (a1 instanceof DBA)
            // sop (" Elg. to drop");
            flag= true;
        }
        if(flag== true)
        { sop(" Elg to drop"); }
        else
        sop (" Not Elg. to drop");
    }
}
```

```
class AnnEx7
{
    public static void main (String args[])
    {
        A obja = new A();
        B objb = new B();
        DBAOperations db = new DBAOperations ();
        db.drop (obja);
        db.drop (objb);
    }
}
```

23/08/2013

2) Single Valued Annotation:

- single element : single element or single value type annotation provides piece of data only. this can be represented with a data = value pair or simply with the value only with in parenthesis

Example

```
public @interface MyAnnotation
{
    String doSomething();
}
```

Usage:

```
@ MyAnnotation ("what to do");
public void myMethod()
{
    ...
}
```

example

```
import java.lang.annotation.*;
@Retention (RetentionPolicy.RUNTIME)
@ interface Todo
 {
    String value();
 }
import java.lang.annotation.*;
@ Todo ("Hello World")

public class Example
{
    P.S.V.main (String args[])
    {
        Todo task = Example.class.getAnnotation (Todo.class);
        SOP ("Found Annotation :" + task.value());
    }
}
```

O/P: Found Annotation : HelloWorld

Example

```
import java.lang.annotation.*;
@ Retention ( RetentionPolicy.RUNTIME)
@ interface Store
{
    String value();
}

import java.lang.annotation.*;
@Store ("This is Employee Class")
class Employee
{

}
```

```
class Supplier
{

}
```

```
import java.util.*;
import java.io.*;
import java.lang.annotation.*;
Public class Tool
{
    P.S.V.main(String args[]) throws Exception.
    {
        Scanner sc=new Scanner(System.in);
        SOP("Input class name");
        String name=sc.next();
        Class c=Class.forName(name)
        Store s=(Store)c.getAnnotation(Store.class);
        FileWriter fw=new FileWriter("File1");
        IF(s!=null)
        fw.write(s.value());
        fw.close();
    }
}
```

o/p:

```
> java Tool
Input class name
Employee
> type File1
This is Employee class.
```

Example:

```
@ main("HelloWorld")
class A
{
}
```

3) Full value or Multi-value Annotation: (Normay Annotation)

- Full value type annotations have multiple data members.
- Therefore you must use a Full data : value parameter syntax for each member.

e.g

```
public @interface MyAnnotation
{
    String doSomething();
    int count;
    String date();
}
```

Usage:

```
@MyAnnotation (doSomething = "What to do", count = 1,
                                    date = "09-09-2005")
public void myMethod()
{
  ...
}
```

example: Data.java

```
import java.lang.annotation.*;
@Retention (RetentionPolicy.RUNTIME)
@interface Data
{
    String value1();
    String value2();
}
```

AnnEx8.java

```
import java.lang.annotation.*;
@Data (value1 = "10", value2 = "20")
class AnnEx8
{   p.s.v.main(String args[]) throws Exception
    {
```

```
Class c = AnnEx8.class;
Data d = (Data) c.getAnnotation (Data.class);
String s1 = d.value1();
String s2 = d.value2();
SOP (s1);
SOP (s2);
}
}
```

* apt.exe:

- java provide apt.exe, which is called annotation processing tool.
- This tool process predefined annotations provided java like @override, @Deprecated, @Target, @Retention....
- User Defined annotations not processed by apt.exe.
- Need to develop program to process user defined annotations.

* @Inherited:

- Indicates that an annotation type is automatically inherited.
- If an inherited meta-annotation is present on an annotation type declaration, and the user queries the annotation type on a class declaration, & the class declaration has no annotation for this type, then the class's superclass will automatically be queried for the annotation type.

example

```
@Inherited
interface A
{ void m1()
}
```