Generics And Collections

Date

There are 3 types of interfaces in Collection → List, Map, Set

List	Map	Set
ArrayList	HashMap	TreeSet
Vector	HashTable	HashSet
LinkedList	LinkedHashMap	LinkedHashSet
	TreeMap (x)	

Priority Queue

Hashset not inserted
Hashmap replaces

ArrayList:

It is used to store elements of same or multiple data type.

Eg:

```
Class A {
psvm (-) {

ArrayList list = new ArrayList();
list.add(1);
list.add(str);
list.add(3.2);
sop(list);

}
}
```

O/p [1 str 3.2]

Date

1) Arraylist by default behaves as an Object and hence it can store all the values which are object.

2) We can change the behaviour of Arraylist and restrict it to only that type of elements.

→

2) Collection.sort works only for those classes which implement comparable interface and override compareTo() method

→

If a null value is passed to collection.sort it returns null pointer exception at runtime.

→

If the values given to collection.sort do not belong to same class it gives class Caste exception.

→

If ArrayList Behaves like type X it will return values which will be of type X. and hence we have to catch it in either X or downcast it.

Eg:

```
{ import java.util.*;
  public class A {
  psvm(-) {

    Arraylist list = new Arraylist();
    List.add(1);
    List.add(5);
    List.add(2);
                    } copy
int i = (Integer) list.get(0)
                    // Returns Object type and
                       so downcasted

sout(i); // 1
}

}
```

Date

Iterator Interface.

1) Iterator by default behaves like an object, but can be made to behave like other class.

2) It has 2 mtds:

1) hasNext mtd

This mtd returs true is element is present in list and false if not present in list.

2) ~~Next~~ next mtd

This mtd return next element in the list.

Iterator Program.

~~import~~

{

Paste

}

```
Iterator <Integer> itr = list.iterator();
                    // behaves like Integer

while (itr.hasNext()) {

    int x = itr.next(); // returns Integer
        sout (x);

  }
}
}
```

Date

Eg.

```
import java.util.*;

class X { }
class Y extends X { }
class Z extends Y { }

public class A {
psvm (-) {
    ArrayList<X> list = new ArrayList<X>();
    list.add(new X());
    list.add(new Y());
    list.add(new Z());

    Iterator itr = list.iterator();

    while (itr.hasNext()) {

        X x = (X) itr.next();
        Sout(x);

    }
}
}
```

Date

Case Study1

```
package School;

import java.util.*;

public class App {
    psvm (-) {
        Student s1 = new Student ("J", 80);
        --"-- s2 = new Student ("Ji", 60);
        s3 = --"-- ("G", 70);
        s4 = --"-- ("Ki", 50);

ArrayList <Student> List = new ArrayList
                           <Student>();

list.add (s1);
         (s2);
         (s3);
         (s4); } copy

Collection.sort (list);

System.out.print (list);
}
}
```

makes you happy?
ppyCollegeDays

#

Date

```
package School;

class Student implements Comparable
                          <Student>
{
  int marks;
  String name;

  Student (String name, int marks)
  {
     this.name  = name;
     this.marks = marks;
  }

   public String toString ()
   {
     return name + " --- " + marks;
   }
   @Override
   public int compareTo (Student s)
   {
     return s.marks - this.marks;
   }
}
```

Comparator Interface

If we cannot reach out to the original class (customer), then we have an option of using comparator interface and making collection.sort work as shown below.

```
{

Paste.

}

Collection.sort(list, new B());
    sout(list);
}
}
```

Date

```
class B implements Comparator<Student>
{
@override

public int compare(Student s1,
                   Student s2)

{
    return s1.marks - s2.marks;
}
}
```

Date

Syntax for Class Declaration

ArrayList = AL

AL list = new AL(); // behaves like object ✓

List list = new AL(); ✓

List<Integer> list = new AL(); ✓

List list = new List(); X

List list = new Vector(); ✓

List list = new LinkedList(); ✓

~~AL<object> list = new ArrayL~~

AL<object> list = new AL<Integer>(); X

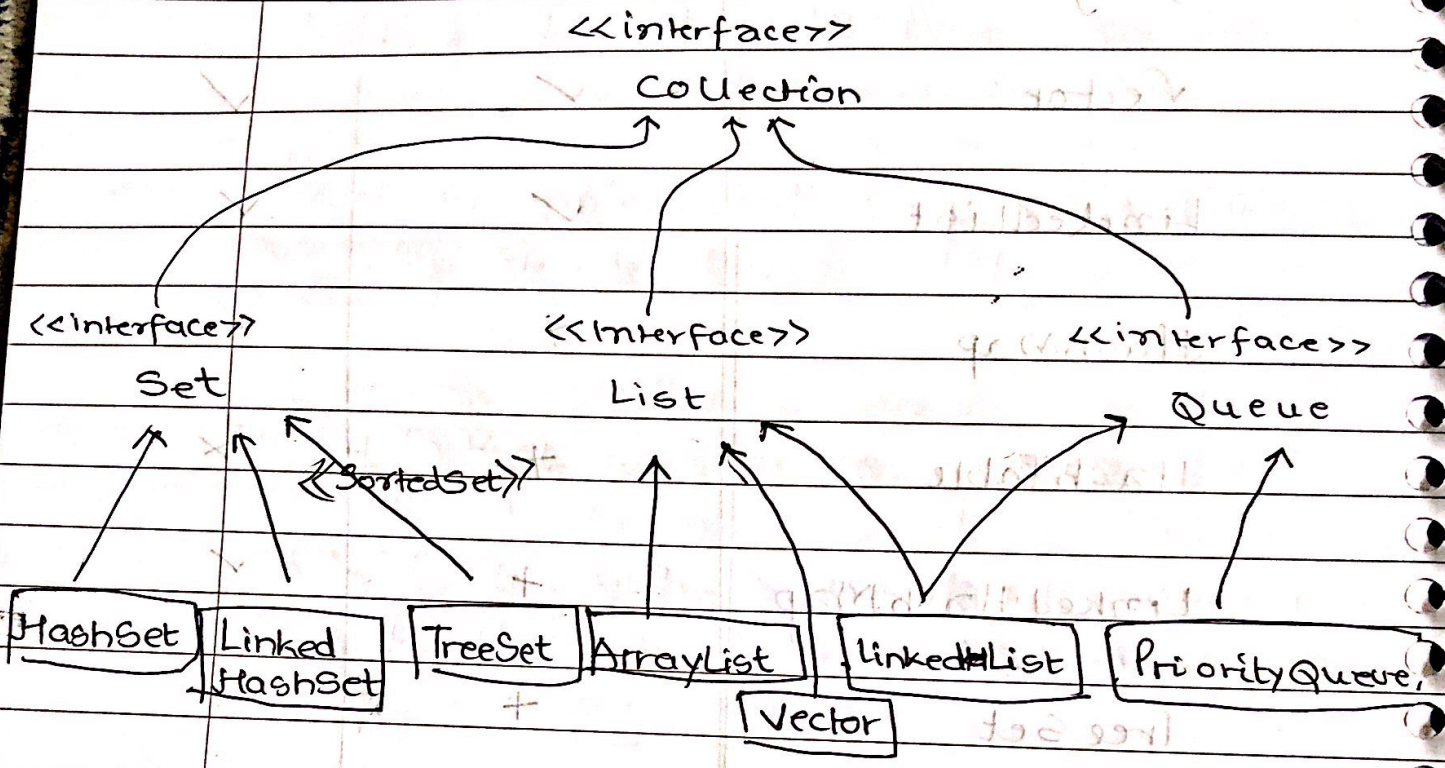
AL<int> list = new AL<Integer>(); X

#

#

Date

for ~~Collections~~



What makes you happy?
#HappyCollegeDays

#
#

✓ ↑ if no large add rem.

Date

ArrayList VS LinkedList VS Vector

From the hierarchy diagram, they all implement list interface. They are very similar to use. ~~The main~~ Their main difference is implementation which causes different performance ~~for~~ different operations.

ArrayList is implemented as a resizable array. As more elements are added to ArrayList, its size is increased dynamically. It's elements can be accessed directly by using get and set methods, since ArrayList is essentially an array.

LinkedList is implemented as a double linked list. It performance on add and remove is better than ArrayList, but worst on get and set method.

Vector is similar with ArrayList, but is synchronised.

What makes you happy?
#HappyCollegeDays

#_____

#_____

Date

HashSet vs TreeSet vs LinkedHashSet

HashSet is implemented using a hash table.
Elements are not ordered.

TreeSet is implemented using a tree structure
The element in set are sorted

LinkedHashSet is Between HashSet and TreeSet. It is implemented as a hash table with a linked list running through it, so it provides the order of insertion.

Date

HashMap vs HashTable vs LinkedHashMap

HashMap is implemented as a hashtable, and there is no ordering in keys~~value pair~~ or values. Its unsynchronized and permits null.

HashTable, has no ordering but sync... and does not permits null.

LinkedHashMap is a subclass of HashMap. That means it inherits the features of HashMap, In addition it preserves the insertion order.

#

#

Map Interface

HashMap:

Type 1:

Iterating through Map.

```
import java.util.*;

public class App
{
    psvm(-)
    {
        HashMap<Integer, String> map = new HashMap();
        map.put(1, "John");
        ——v—— (3, "Emma");
        ——v—— (4, "Jiu");
        ——v—— (5, "Jack");

        for(int o : map.keySet())
        {
            sout(o);
        }
        for(String s : map.values())
        { sout(s);
        }.
    }
```

cops { ... } cops

#_____
#_____

Type 2

~~Comm~~ Connecting map to list and set.

```
import java.util.*;

public class App
 {
   psvm(-)
   {
    ~~HashMap<Integer, Stri~~
```

Paste

```
   }
   Set<Integer> set = new TreeSet(map.keySet());
ArrayList list = new ArrayList(map.values());
Iterator<String> its = list.iterator();

while(its.hasNext()) {
      String s= ~~new~~ its.next();

   }
 }
```


#_____

#_____

Type 3.

Using Iterator directly.

```
import java.util.*;
{

faste }

Iterator itr = map.entrySet().iterator();
while( itr.hasNext()) {
Map.Entry<Integer, String> e =(Map.Entry<Integer, String>)
                                itr.next();

    Sout (e.getKey());
    Sout (e.getValue());
  }
}
```

What makes you happy?
#HappyCollegeDays

#_____
#_____

Generics

1) Generic Method
2) Generic Class

Generic Method

Write a program to display and iterate different array types using a single generic method.

```
import java.util.*;

psvm (-)
{
  Integer [] i = {1,3,4};
  Float [] f = {1.2f, 3.4f};

  display (i);
  display (f);

}
```

Date

```
static <T> void display (T[] i) {

Error   //T t1 = new T(); // No object
Error   // static T t1 = ; // No static

    for (T i1 : i)
    {
      sort (i1);
    }
}

}
```

Date

Generic Class

```
import java.util.*;

class School<T>{
    T t;
    void add(T t)
    {
        this.t = t;
    }
    T get(){
        return t;
    }
}
public class App{
    psvm(){
    School<Integer> s1 = new School();
    School<String> s2 = new School();

    s1.add(1);
    s2.add("hello");

    sout(s1.get());
    sout(s2.get());
}
```

Priority Queues.

Priority Queue sorts strings as per natural ordering.

It has 2 notable methods.

i) Peek() ii) Poll()

i) Peek()

It returns element which is at top of the Queue.

ii) Pou()

It returns element which is at top of Queue and deletes it.

Date

Eg.

```
public class App
{
    psvm(-)
    {
        PriorityQueue<String> pq = new PriorityQ-
                                   ueue();

        pq.add("banana");
        ——————("apple");
        ——————("grapes");

        sout(pq.peek());
        pq.poll();
        sout(pq.peek());

    }
}
```

O/p apple
banana.

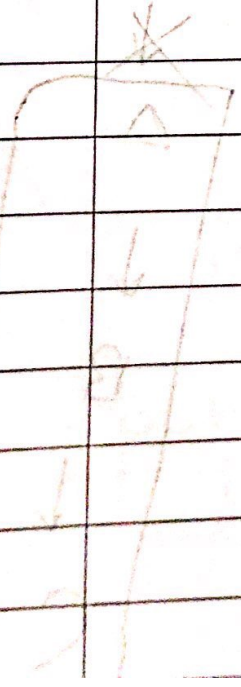
Thread wait() and notify()

```
class A extends Thread
{
  String str = "";
  public void run ()
  {
    synchronized (str) {
    Sout ( Thread. currentThread ().getName());
    str.notify();

    try
    {
     str.wait();
    }
    catch (InterruptedEx ex)
    {
     }
     sout (Thed.cu  gtN);
     Str.notify();
  }
}
```

Date

```
psvm (--)
{
    Thread t1 = new Thread (new A(), "A");
    Thread t2 = new Thread (new A(), "B");
    t1.start();
    t2.start();
}
```



Date

IS-A HAS-A Relationship

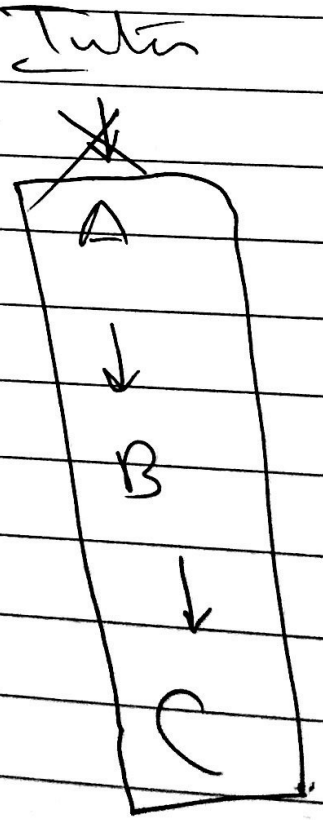
```
interface Inter { }
class A implements Inter {
    C c b;
}
class B extends A { }
class C extends B { }
```

✓ A has a C

✗ A is a Inter

✗ B is a C

✓ C is a A



Date

Enumeration. (Enums)

→ Enum is a data type which is assign to a variable to restrict it to the values that programmer wants.

→ Enums are abstract entities, that can have variables mtds and object inside it.

Version 1

```
enum College{
    ABC(90), XYZ(80), PQR(70);

College (int percent)
  {
    this.percent = percent;
  }

  private int percent;

public int getPercent()
  {
    return percent;
  }
}
```

Date

```
class App
{
  static College col;

  psvm(-)
  {
    sout(col.xyz.getPercent());
  }

}
```

Note:

As enums are abstract entities they cannot be instanciated.

Ex: College c = new College(); ✗

Version II

→ Every enum object can have its own variables and methods provided the signature is available to the enum.

```
enum College
{
  ABC {
    void welcome()
    {
      Sout("Welcome to ABC");
    } },
  XYZ {
    void welcome()
    {
      Sout("welcome to XYZ");
    }
    },
  PQR {
    void welcome()
    {
      Sout("welcm to PQR");
    }
    };
  abstract void welcome();
}
```


#_____

#_____

Date

```
class APP {

    static College col;

    psvm(-)
    {
     col.ABC.welcome();

     }

}
```

→ Enums can be created inside the class and outside the class.

Outside the Class

→ If we create enums outside the class, we have to declare enum reference as static as use it inside main as shown in the above program.

Inside the Class

→ if enum is created inside the class, we can use object of the class and access enum variable we do not have to mark it as static.

~~Example~~

Ex:

```
class APP {

    enum College {

    ABC, PQR; // Semi Colon Optional
    }

    College col;

    psvm (-)
    {

    Sout (" new APP().col.ABC);
    }

}
```

O/p : ABC.

Date

Arrays

1D

```
int [] i = { 1, 2, 3 };        ✓
int [] i = new int [3];        ✓
int [3] i = new int [3];       ✗
int i [] = new int [3]         ✓
int [] i = new int [] {1, 2};
    i.[0] = 1
    i [1] = 2
int [] i = new int [] { };
    i[1] = NPE // (Null Pointer Ex)
```

Date

20

```
int [][] a = new int [2][3];

a[0][0]
a[0][1]
a[0][2]

a[1][0]
a[1][1]
a[1][2]

a[1][3] : ArrayIndexOutOfBoundsException

a[0][2] = 0
```

x

```
int [][] a = new int [3][];

a[0] = new int [2];
a[1] = new int [3];
a[2] = new int [1];

a[0][0]
   [1]

a[1][0]
   [1]
   [2]

a[2][0]
```

What makes you happy?
#HappyCollegeDays

Date

3D

```
int [][][] a = new int [2][2][2];
```

a[0][0] [0]
[1]

a[0][1] [0]
[1]

a[1][0] [0]
[1]

a[1][1] [0]
[1]

```
int [][][] a = new int[3][1][];
```

```
a[0][0] = new int [3];
```

a[0][0] [0]
[1]
[2]

```
a[1][0] = new int [1]
```

a[1][0][0]

```
a[2][0] = new int [2]
```

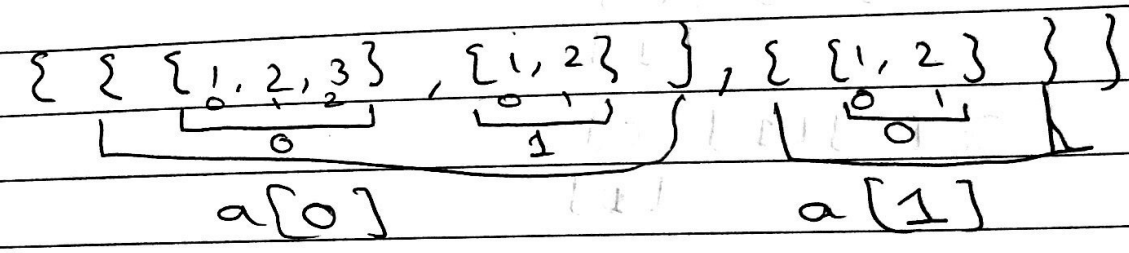
a[2][0][0]
[1]

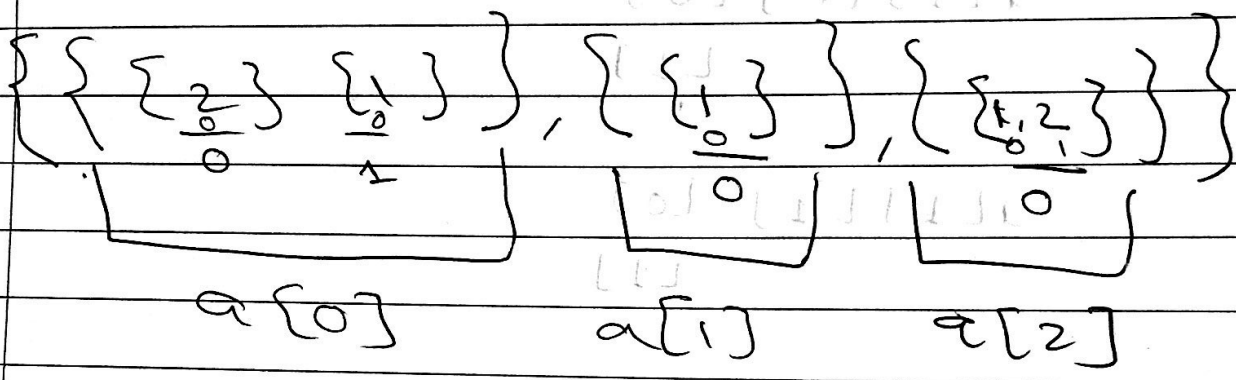
What makes you happy?
#HappyCollegeDays

#_____
#_____

~~int [][]~~

int [][][] a = new int [][][]

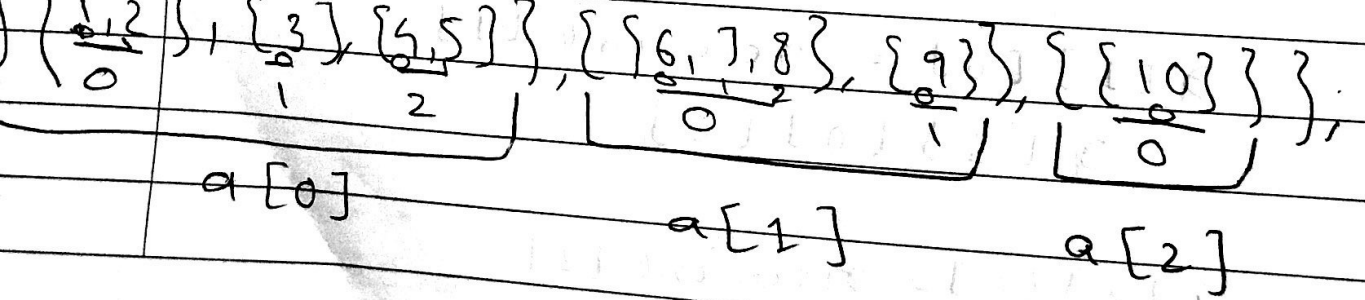




---

int a[][][] = new int [][][]

{ { { 1, 2 }, { 3 }, { 4, 5 } }, { { 6, 7, 8 }, { 9 } },
{ { 10 } } } ;



_____

_____

Date

WildCards in Collection

→ There are three bounds in Wild Card.

1. <?> without the Bound.
2. <? super Integer> → Lower Bound.
3. <? extends Integer> → Upper Bound.

Rules

1→ Wildcards are allowed only on LHS and NOT on RHS of object creation

Ex:

AL<?> list = new AL(); ✓

AL list = new AL<?>(); X

2 → If we use wildcard without the bounds we cannot add anything to the list

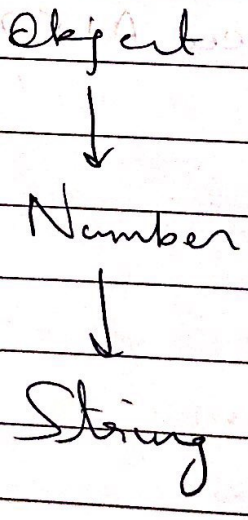
Ex:

```
AL<?> list = new AL();
list.add(1); X
```

3 → If you use wildcard with Super bond we can add the element to the list.

Ex:

```
AL<? super String> list = new AL();
list.add("a");
```



Date

4: → if we use wildcard with extends we cannot add anything to the list

Ex:

```
Al < ? extends Integer > list = new Al ();
list.add (1);   X
```

USING METHODS WITH WildCARDS

```
class App {
psvm (-)
  {
    ArrayList list = new ArrayList ();
    list.add(1);
    list.add(3);
    list.add(4);

    m1 (list);
    m2 (list);
    m3 (list);

  }
```

```
static void m1(List<?> list)
{
  for(Object o: list)
  { sout(o); }
}

static void m2(List<? super Integer> list)
{
  list.add(1); // can add as it is super
  for(Object o: list)
  {
    sout(o);
  }
}

static void m3(List<? extends Integer> list)
{
  for(Integer o: list)
  {
    Sout(o);
  }
}
}
```

Date

Files

```
import java.io.*;

class A
{
  psvm(-)
  {
    File dir = new File("DIR");  // dir handle
    dir.mkdir();                 // dir creation

    File file = new File(dir, "abc.txt");
                                 // file handler
    file.createNewFile();        // file created
```

Date

Assign:

String, StringBuffer, StringBuilder. class

What are the Method & differences

what is String Pool?

Assign

Study Abt foll Classes

FileReader
FileWriter
BufferedReader
BufferedWriter
PrintWriter
Console

Date

Strings

```
String s1 = "abc";

String s2 = "abc";

String s3 = new String("abc");

s2 = "abc.";

s1 = s2;

sop(s1); // abc.
```

Pool

100x : abc

200x : str(obj(abc));

300x : abc.

Stack

s1 :	100x	300x
s2 :	100x	300x
s3 :	200x	

Ex

```
class App {
  psvm(-)
  {
    String s1 = "abc";
    ——— s2 = "abc";
    ——— s3 = new String("abc");

    StringBuffer sb1 = new StringBuffer("abc");
    ——— sb2 = ——— ("abc");

  // sout (s1 == sb1); // compilation fails

    sout (sb2 == sb1); // false.
    ——— (s1 == s3); // false.
    ——— (s1 == s2); // true.

  }
}
```

Note:

== operator needs both the variables of same class and it checks the locations from the stack.

#

equals() mtd.

This mtd belongs to object class and is available to all the classes.

String class and Byte class Override equals mtd and so have their own version of this equals mtd.

Ex:

```
class App {
psvm(-)
  {

Para

  }
  Sout(sb1.equals(sb2)); // false
  Sout(s1 == s3);        // false
  Sout(s1.equals(sb2)); // false
  sout(s1.equals(s3); // true
}
}
```

Date

equals() Mtd

Objects Equal()

~~Same~~

same as double equals (==) operator

String Equal()

① Both s1 & s2 should be string References then ②
else
(==) operator

② check the values of s1 & s2

Date

FILES

```
class App {
 psvm (-) throws IOException
 {
  File file = new File ("abc.txt");
  FileWriter fr = new FileWriter(file);

  BufferedWriter bw = new BufferedWriter(fr);
  bw.write("Hello World");
  bw.close();

BufferedReader br = new BufferedReader(new
                    FileReader(new File
                    ("abc.txt")));

BufferedReader br1 = new BufferedReader(new
                     FileReader("abc.txt"));

String s = br.readLine();

    Sout (s);
br.close();

}

}
```

What makes you happy?

#

#

Date

Dates

1) Date class

This class is used to generate date from system clock.
— The date that is returned from sys. is in form of complete TimeStamp.

Eg.

```
Date date = new Date();

System.out(date);
```

2) Date format class.

This class is used to format the date.

It is an abstract class and so cannot be instansiated hence we use factory ~~ma~~.mtd

1) ~~G~~getInstance()
2) getDateInstance()

to initialize date formate ref.

This class has 2 mtds.

1) ~~parse~~ format

Takes date as argument and returns a string.

2) Parse.

Takes string as argument and returns a date. (Throws Parse Ex)

#

#

Date

Locale Class:

Locale is used for country specific manupiwation of date.

while creating a locale object We can pass:

1) Either language code
2) Country Code
3) Both language & County Code.

Note:

Locale object should be pass as a second argument to date format reference

Number format

```
class App {
psvm (-) throws ParseException
{
    Date date = new Date();
    sop(date);
    Locale loc = new Locale("HI", "IN");
DateFormate df = DateFormate.getDateInstance
                        (DateFormate.FULL, Loc);
DateFormate df1 = DateFormat.getDateInstance
                        (DateFormat.FULL);
DateFormat df2 = DateFormat.getInstance();

String s = df.format(date);
 Sop(s);

Date d2 = df.parse(s);
 sop(d2);
 // Number Format

 float f = 34.5F
```

Date

```
NumberFormat nf = NumberFormat.getCurrencyInstance(loc);

String s1 = nf.format(f);
sop(s1);
double n = (Double) nf.parse(s1);
float f2 = (float) n;
sop(f2);
}
}
```

O/p

Sat Jul 04 18:47:47 IST 2015

शनिवार १ जुलाई २०१५

Sat Jul 04 ~ 00

Date

Regex. (Regular Expression)

Ex:

```
class App{
psvm(-)
{
  String s = "Test A1. TestB2. TestC";

  String [] s1= s. split("\\d\\.\\s");

  for (String s2 : s1)
  {
     sout (s2);
    }
  }
}
```

Space ← \s

One or more char ← \w \w+

One or more digits ← \d+ ← \d

Dot ← \.

Date

Type 2:

Scanner Class

```
import java.util.*;

class App {
  psvm(-) {
    String s = "TestA. TestB. TestC";

    Scanner sc = new Scanner(s);
    sc.useDelimiter("\\.\\s");

    while (sc.hasNext()) {
        sout(sc.next());
    }
  }
}
```

Date

Type 3

Pattern and Matcher

```
import java.util.regex.*;

class App {
  psvm(-)
  {
    String s = "My name is John Oriely";

    Pattern p = Pattern.compile ("\\s\\w[5]\\s
                                  \search 5 letter wo

    Matcher m = p.matcher(s);

    sout (m.find());
  }

}
```

Date

HashCode & equals() method..

Hashcode Mtd.

When ever an object is created JVM stores it in the heap on any location computed by hashcode function.

Signature of hashCode.

```
public int HashCode() {

    return <value>;

}
```

If a programmer wants to decide location of the object he has to overide Hashcode mtd and return the location.

Date

Ex:

```
class A {
    int x = 10;

public int hashCode()
{
    return this.x;
}

psvm (-)
{
  A a = new A();

  Sop(a); // 10

}
```

Heap

10 : new A()

Stack

A a : 10

Date

Equals Mtd.

Equals mtd by default is present in object class. String class overides equals mtd and makes jvm compare the values of toString objects.

We can also override equals() mtd and change its behaviour according to our requirement.

Date

Case Study

Consider a banking scenario, where every customer is assign the balance and the name. Every customer is an object. The programmer wants to compare customer according to their balances. If balance of 2 customers are similar, they should be same and equals mtd must return true.

Soln :-

```
public class Customer{

String name;
int balance;

public Customer (String name, int balance)
{
   this.name = name;
   this.balance = balance;

}
```

Date

```
public boolean equals (Object o)
{
  if (o instance of Customer)
  {
    if (this.hashCode() == o.hashCode())
    {
      return true;
    }
    return false;
  }
  return false;
}

public int hashCode() {

  return this.balance;

}
```

Date

~~class~~

class App {

public static void main(0 —)

{

~~C — C1~~

Customer c2 = new Customer ("A", 10000);

Customer c2 = new Customer ("B", 10000);

sout (c1.equals(c2));

}

o/p / True

Date

Coupling And Cohesion

Coupling:

Coupling is a measure of connectivity of 2 classes. If the 2 classes are very closely connected to each other, we consider it as tight coupling.

Eg:

```
class A {

    B b;
}
```

A has a B

A is tightly Coupled to B

Guidelines

The 2 classes should be independent of each other.

If one programmer makes a mistake it should not effect the class of other programmer.

Hence as per the principles of object orientation the 2 class should be loosely couple.

Ex:

```
class A {
    B b = new B();

void Compute() {

int b = b.getBalance() *
                0.40;

sop("loan is" + b);
```

```
class B

int getBalance(){

return 100;

}
```

Guidelines

The 2 classes should be independent of each other.

If one programmer makes a mistake it should not effect the class of other programmer.

Hence as per the principles of object orientation the 2 class should be loosely couple.

Ex: class A {

```
B b = new B();

void Compute() {

int b = b.getBalance() *
        0.40;

Sop("loan is " + b);
```

class B

```
int getBalance(){

return 1000;

}
```

Tight coupling

Ex:. 2

```
class A {

int compute (int balan)
{

return balan * 0.40;
}
}
```

loose Coupling

```
class B
{
int getBalance()
{
return 100;
}
}
```

```
Class App {
psvm()
{
int b = new B().getBal();

new A().
```

Cohesion:

Cohesion is a degree of interconnectivity of various tasks assign to a programmer.

If one task is considered to be one mtd, then co-hesion can also be described as interconnectivity of various mtds of the class.

Guidelines:

A programmer should always keep task independent of each other, this means that the mtd should be independend and should not be called from within each other.

Independ mtd are considered to be highly cohesive and hence as per object oriented design principal, the class should be highly cohesive.

What makes you happy?
#HappyCollegeDays

Scanned with CamScanner

System Properties:

Date

Ex:

```
class A{
psvm(-)
{
String str1 = System.getProperties().
                getProperty("str");

String str2 = System.getProperty("str");

Sout(str1 + "  " + str2);

}
}
```

cmd:

java -Dstr = hello A

o/p hello.

#

#

What makes you happy?

Serialization:

Date

Serializable is an interface which has no mtds.

If a programmer wants to copy its object into a file and retrive it at a later date he needs to implement Serializable interface.

To read and write object from and to the file, we use following classes.

```
class A implements Serializable {
    psvm(-) {
ObjectInputStream is = new ObjectInputStream
                        (file.txt);
A a1 = is.read();
        ↳ Reading from the file
ObjectOutputStream os = new ObjectOutputStream
                        (file.txt);
os.write(new A());
        ↳ writing the object in file.
}
```

ClassPath And Path Variables.

Date

Note :

if a programmer wants to change the program he has to change it from actual location of java file and not from class file.

Sometimes java files are also referred to as class files.

We never follow classpath for changing file. We use it only for executing the file.