21/4/2024

Sealed Classes {added in JDK-17}

```
→ final class A {

  }
                                          Not possible
    Class A extends B {   X

    }
```

```
→ Public sealed class A   permit B, C {

  }
```

Here class A can only be extcended from B & C.

```
Class D extends A                 class B {   {B & C has to extends}
{                                             {A                   }
                                           X
    X ←————————— Error ——————————→
                                         }
}
```

```
① final ————————→ Public final class B extends A {
                                        }
② Sealed ———————→ Public sealed class B extends A permits E {

                  }
③ non-sealed ———→ Public non-sealed class B extends A {

                  }
```

Sealed
Permits
non-sealed
} Context keywords

[Will only be considered as keywords here only, elsewhere we can use these normally.]

Sealed Class → Only permitted class can inherit that class.

Non-Sealed Class → any classes can inherit that class.

Sealed Interface →

```
Public sealed interface MyInt permits A,B {
    void m1();
}
Class A {
    X error.
}
```

```
sealed/non-sealed/final class A implements MyInt {
    --
    --
}
```

Instance Of - Operator

objref instanceOf type

```
Class A {

}
ClassB {

}
Class extends A{

}
Class D extends A{

}
```

```
A a = new A();
B b = new B();
C c = new C();
D d = new D();
```

```
if (a instanceof A){
    True
}
if(b instanceof B){
    True
}
if (c instanceof C){
    True
}
if ( c instanceof A){
    ~~False~~ True
}
if (a instanceof C){
    false.
}
```

```
A ob;
Ob = d;
if (ob instanceof D)
{
    True
}
Ob = C;
if ( ob instanceof D)
{
    False
}
if (ob instanceof A)
{
    True
}
```

```
if (a instanceof Object)
{
    True
}
if (b instanceof Object)
{
    True
}
```

Enumeration

→ List of named constants which works as a datatype.

```
enum Color{ Red, Blue, Yellow, White, Green}
          { 0     1     2       3      4  }
   ↙
[Ordinal
  Values]
```

```
⇒ ~~Ct~~ Color c;
    c= Color.Red;
    Switch(c){
        Case Red:
        ________
        Case Yellow:
        ________
            :
    }
```

⇒ Values() ⇒

will return array of all values of Color.

```
Color allColor[] = Color.values();
```

→ Java enumeration are of Class type.

```
enum Color{

    Red(10), Blue(20), Green(15), Yellow(12);
     int Price;
     Color(int P)
      {
        Price =P;
      }
        int getPrice(){
            return Price;
        }
}

class enumDemo{
    PSVM(__){
        Color c;
        SOP( c.Red.getPrice());
        for(Color c1: Color.values()){
            SOP(c1.getPrice());
        }
    }
}
```

```
                         → Parameterised Constructor call
enum Color{
    Red(10), Blue  , Green(15), Yellow(12), White;
    int Price;                  → default Constructor
    Color(int P){
    Price = P;
    }
    Color(){
      Price =0;
    }
    int getPrice(){
      Return Price;
    }
}
```

→ All enumeration inherits Java.lang.Enum class.

→ Enumeration Cannot inherit any Class.

→ Enumeration Can't be inherited by a class.

Assertions

```
class Test {

    PSVM(_){

        int value = 15;

        assert value >=20;

        SOP(value);
    }
}
```

ignored in normal run

# java Test

~~To~~ Output ⇒ 15

to make assertion work, we have to enable them.

# java -ea Test

→ Exception in thread main java.lang.AssertionError.

```
class Test {

    PSVM(_){

        int value = 15;

        assert value >=20:"Value is less than 20";

        SOP(value);
    }
}
```

Output

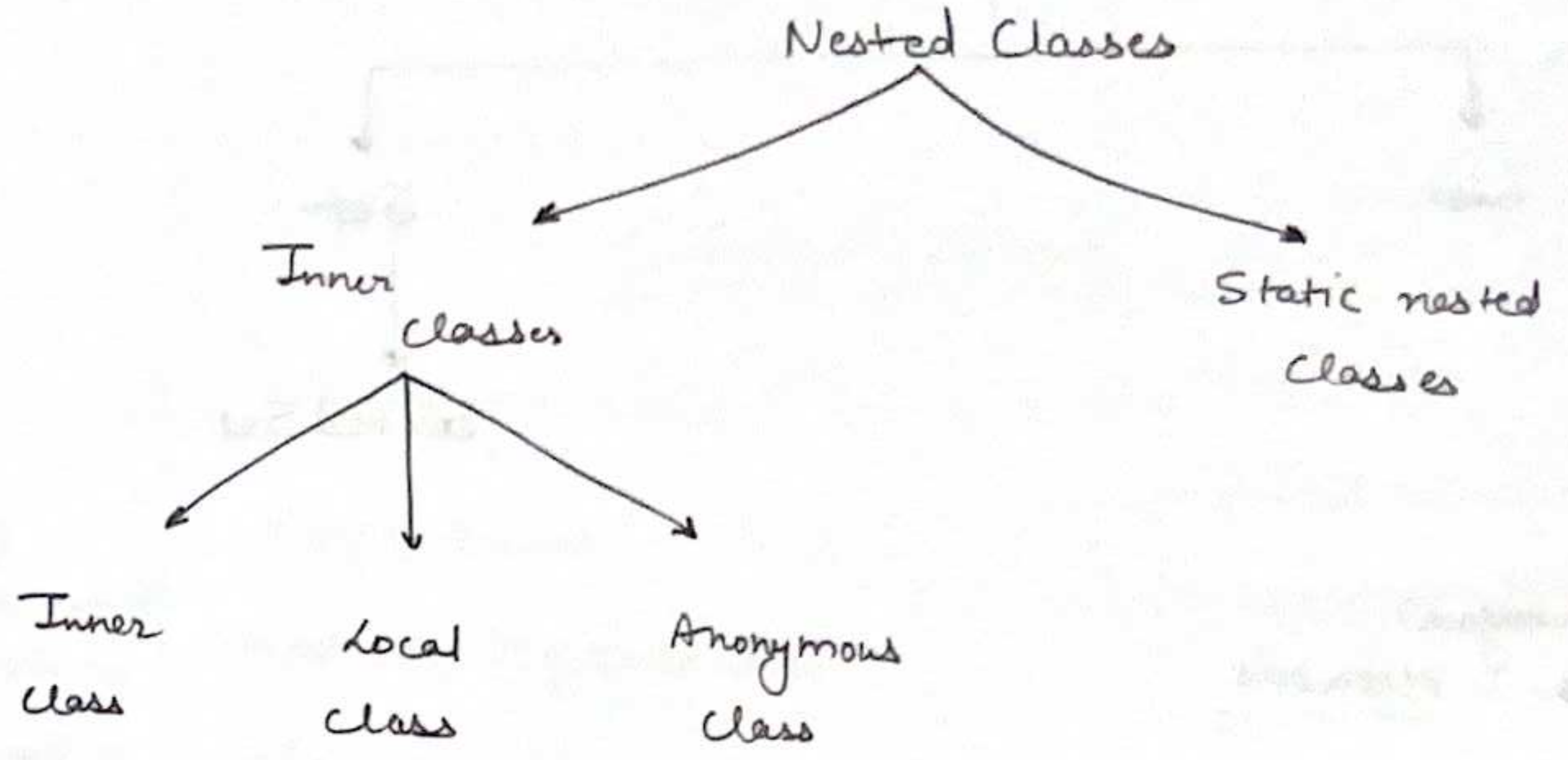
java -ea Test

→ Exception in thread main java.lang.AssertionError : value is less than 20

-ea

- enableassertions
- da]- default.
- disable assertion

Inner Class



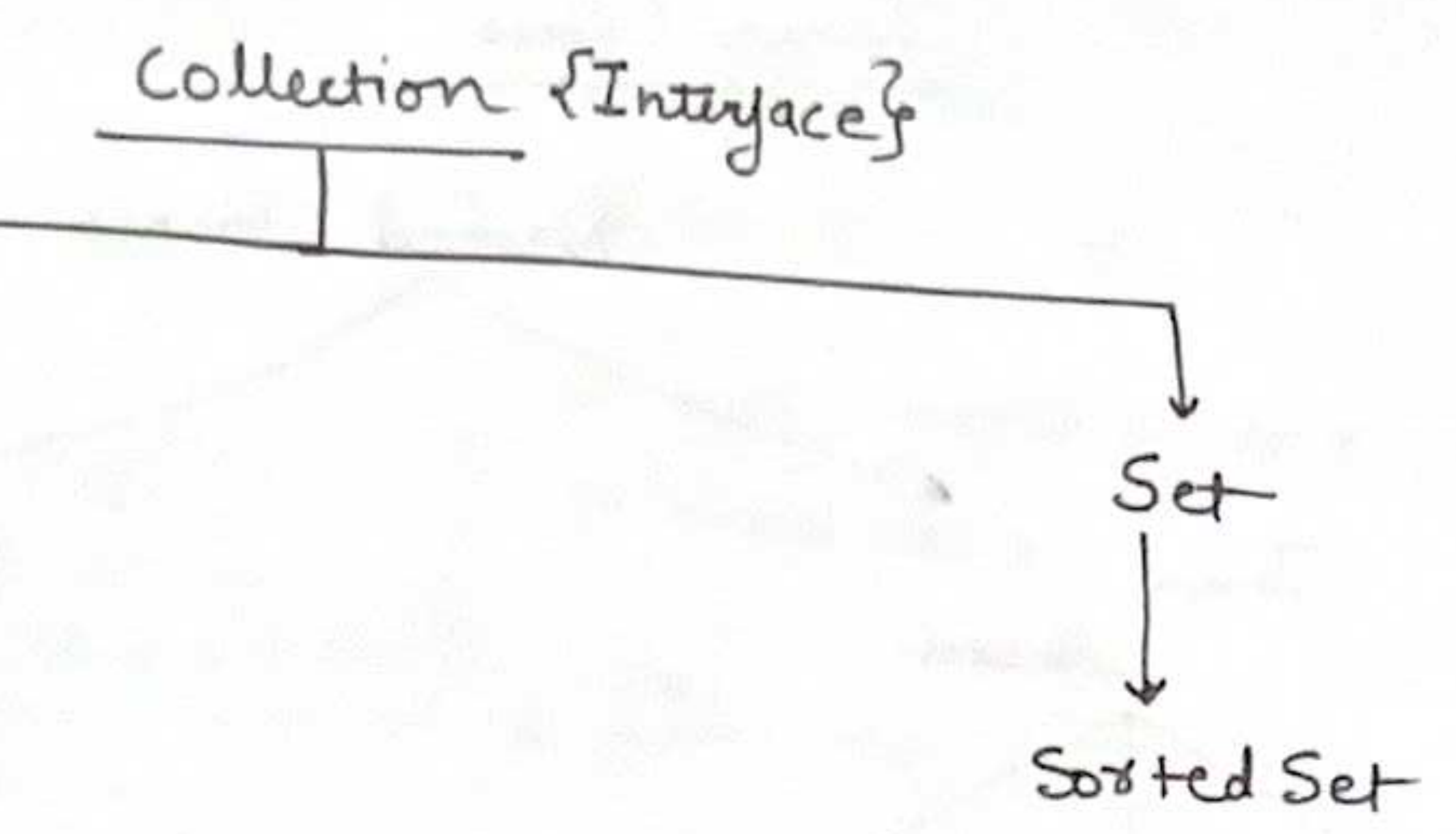
① Inner Calass ⇒ class written inside of another class.

② Local class ⇒ class writen inside a block / Method

③ Anonymous Class ⇒ Class without name and can only be used while its Creation.

④ Static Nested class ⇒ static class written inside another class.

24/4/2021



List (Interface) ⇒ Methods

① – add (int index , Object element)

② – Object get (int index)

③ – int indexOf (Object ob)

④ – int lastIndexOf (Object ob)

~~List Iterator~~

⑤ – ListIterator listIterator ()

⑥ – Object remove (int index)

⑦ – Object set (int index , Object ob)

Set ⇒ SubInterface of Collection

- No duplicates allowed
- No sequence.

Sorted Set ⇒ SubInterface of Set.

→ Sorting order will be decided by Comparator

→ Comparator Comparator ();

Object first ()

Object last ()

Collection Classes

Only Implemented some of the methods of Interface

- Abstract Collection
- Abstract List
- Abstract Sequenetial list
- Abstract Set
- Linked List
- Array List

- Hash Set
- LinkedHashSet
- TreeSet ← get elements in sorted order

:

ArrayList

Constructors ⇒

- ArrayList () ← Initial default size = 10
- ArrayList (Collection c)
- Arraylist (int initialCapacity)

Method ⇒

- void ensureCapacity (int Capacity) ← does reverse of trim. ⟷
- void trimToSize () ← To bring size according to capacity. →←

```
import java.util.*;
Class ArrayListDemo{
    PSVM(—){
        ArrayList al = new ArrayList();
         al.add ("a");
         al.add ("b");
         al.add (" c");
         al.add ("1,"d");
```

a, b, c

| a d b c |

```
SoP( al.size());
al.add(new Integer(5));          (adbc5)
SoP(al);
al.remove(B);
SoP(al);                         (adc5)
 al.remove("C");
SoP(al);                         (ad5)
  }
}
```

Lets say ~~Ar~~ Instead of : ArrayList al = new ArrayList();

we used : ArrayList<String> al = new ArrayList<String>();

Now we would get ~~a~~ error at (new Integer(5))

```
PSVM(-){
   ArrayList al =new ArrayList();
     al.add("a");
     al.add("b");
     al.add("c");
     al.add(1,"d");

   Iterator  itr = al.iterator();        Output:→
                                           a d b c
     while( itr.hasnext()){
         SoP(itr.next());
     }
  }
}
```

25/4/2024

LinkedList

```
import java.util.*;
Class L {
   PSVM(-){
     LinkedList ll = new LinkList();
       ll.add("A");
       ll.add("B");
       ll.add("C");
       SoP(ll);          A B C
       ll.remove first();
       SoP(ll);          B C
       ll.removeLast();
       SoP(ll);          B
   }
}
```

```
add first (Object)
removefirst()
remove Last().
get first()
get Last()
```

ll. ~~First~~addFirst("F");

```
FABC

ABC

AB
```

HashSet

Constructors ⇒

~~import~~

- HashSet() ← Initial Capacity = 16 , fill ratio. 0.75 (Increases its Capacity when 75% full) 0 to 1
- HashSet(Collection c).
- HashSet(int initial Capacity)
- HashSet(int initial Capacity, float fill ratio)

```
HashSet<String> hs = new HashSet<String>();
  hs.add("A");
  hs.add("B");
  hs.add("C");
  hs.add("D");
  hs.add("E");
   SoP(hs);
```

→ ABCDE can be in any order, it all depends on hashcodes of values stored.

LinkedHashSet <String> ~~H~~ hs = new LinkedHashSet<String>();

↓

If we use Linked HashSet it would store data inorder of their addition

meaning SOP(hs) will give us ABCDE

Tree Set

TreeSet <String> hs = new TreeSet <String>();

hs.add("b");

hs.add("d");

hs.add(" c");

↓

will ~~give~~ Store values in sorted order

SOP(HS) ⟶ bcd

Constructors →

- TreeSet()
- TreeSet (Collection c)
- TreeSet (Comparator c)

Class MyComp implements Comparator{

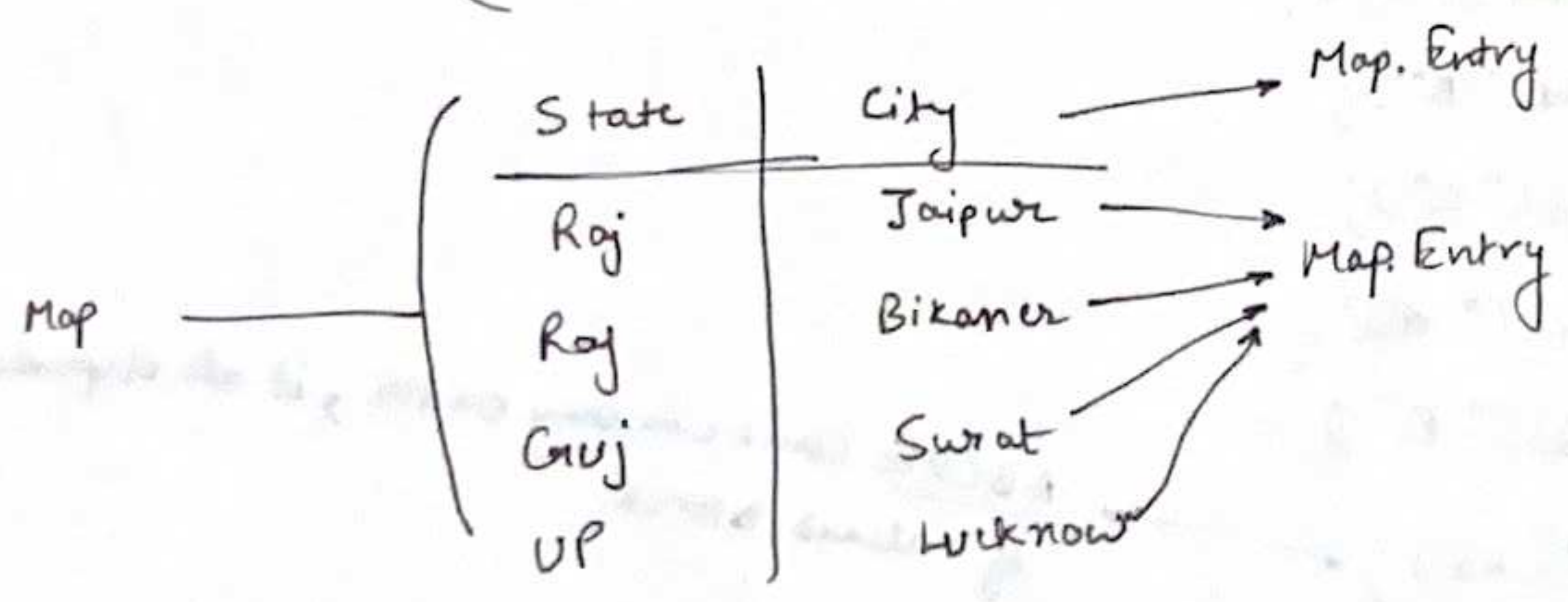
P int Compare (Object O1, Object O2){

① If return < 0,
O1 < O2,
O1 ⟶ O2

② If returns > 0,
O1 > O2,
O2 ⟶ O1

Map

(Key : Value)



State	City
Raj	Jaipur
Raj	Bikaner
Guj	Surat
UP	Lucknow

If Map gets in sorted order of its key then it would be represented as "Sorted Map."

Map
Map.Entry
Sorted Map
} all three are Interface

Map Interface

- Clear()
- Contains Key (Object o)
- Contains Value (Object o)
- Set entrySet()
- Object get (Object)
- boolean is Empty()
- Set KeySet()
- Void put (Object O1 , Object O2)

Sorted Map — Sub-Interface of Map

- Comparator Comparator()
- Object firstKey()
- Object Last Key()

Map.Entry() - Sub-Interface

- getKey()
- getValue()
- setValue(Object ob)

Map Classes

- Abstract Map
- HashMap
- Linked listHashMap
- TreeMap

```
HashMap   hm = new HashMap();
 hm.put( "abc", 5000);
 hm.put("aaa", 6000);
 hm.put(" bbb", 5500);
 Set set= hm.EntrySet();
 Iterator itr= set.iterator();
 while( itr.hasNext() ){
      Map.Entry me= (Map.Entry) itr.next();
      SoP( me.getKey + " " + me.getValue()); <--  Sequence will be random
   }
}
```

→ If we want data in sequence manner then we have to use

→ LinkedHashMap

Output will => In same sequence of addition.

```
abc   5000
aaa   6000
bbb   5500
```

→ If we want output in Sorted order. then we have to use TreeMap.

```
TreeMap hm = new TreeMap();
```

Output =>

```
aaa   6000
abc   5000
bbb   5500
```

{according to the order of Key}

26/4/2024

Collections were introduced in JDK 2 before that we used classes which now known as *Legacy classes*

→ Vector

→ Stack

→ Dictionary

→ Hashtable

→ Properties.

Vector → Vector is just like ArrayList.

→ Now with collections, all legacy classes have two set of methods :→

- All methods of ~~clo~~ collections
- All old methods

Or we can say that now legacy classes also have all methods of collections.

Properties → In properties you can store key, value pairs with key & value both musts be String.

→ All legacy methods are sychronised.

→ Enumeration interface works as iterator for legacy classes. Enumeration have two methods :→

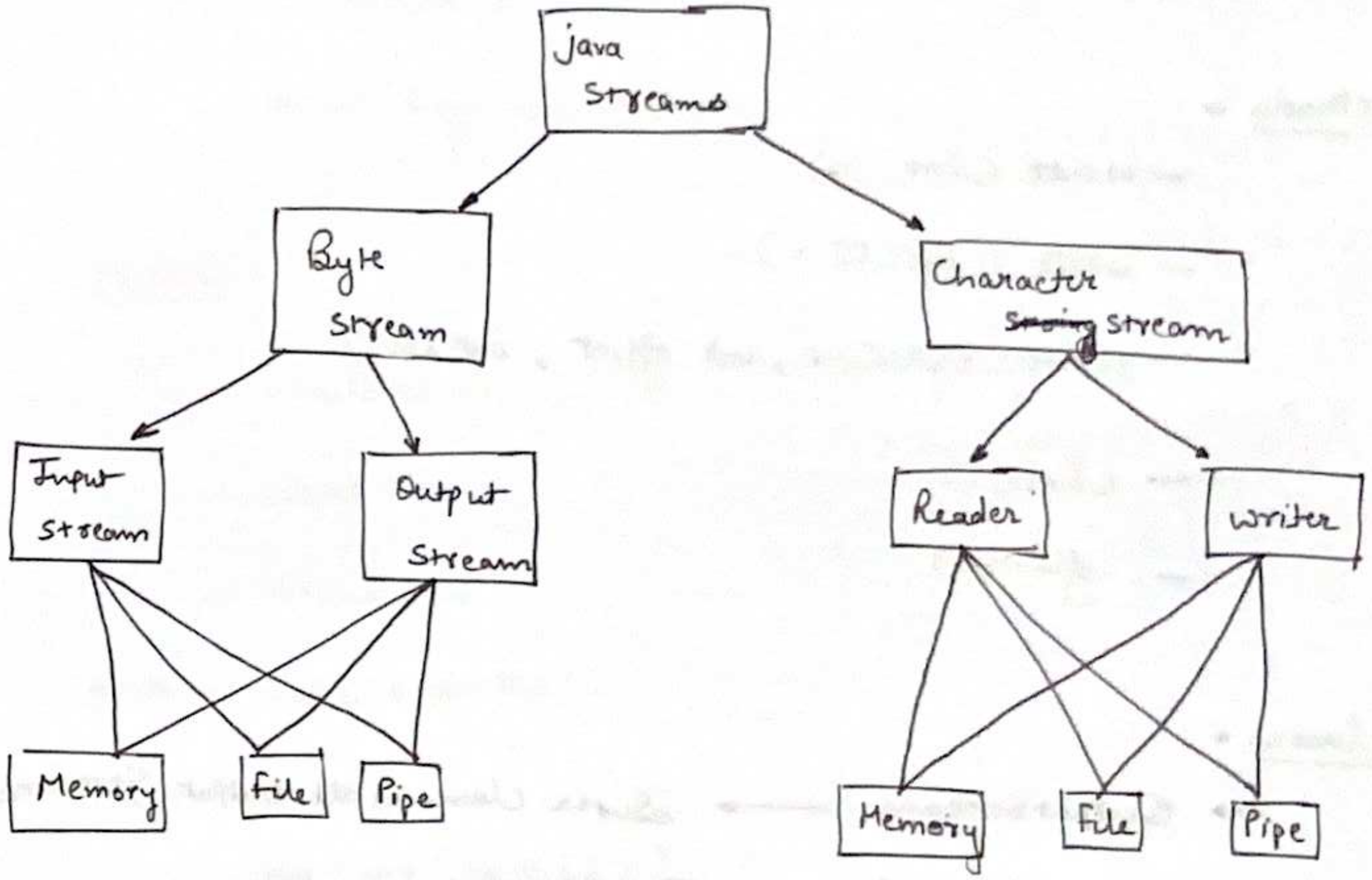
→ Object nextElement();

→ Boolean hasMoreElements();

Java Input / Output

→ Byte Stream

→ Character Stream : {JDK 2}



→ All Input Streams are subclasses of "InputStream"

→ All Output Stream are subclasses of "OutputStream"

→ All Reader Classes are subclasses of "Reader"

→ All writer classes are subclasses of "Writer"

Output Stream

→ Writting Bytes

→ Closing stream

→ flushing

methods -

- write (int a)
- write (byte [] b)
- write (byte [] b, int offset, int len)
- close ()
- flush ()

Classes -

→ OutputStream ——→ Super class of all Output Streams

→ Buffered OutputStream → To write data in buffer.

→ File OutputStream → To write in a file.

→ Data OutputStream → To write data in primitive data format

→ ObjectOutputStream. → write objects specially used in. serifization

→ PrintStream. → General to print the data.

{ System. out. Println()
↓
Out. is object of Print Stream }

```
class System {
    Public static PrintStream Out;
    {
        out = new i - - - -
    }
```

Input Stream

→ Reading of byte

→ close

→ marking

→ skipping

→ finding no. of bytes.

Methods -

- available ()
- close ()
- mark (int a)
- reset ()
- skip (long l)

boolean markSupported ()

int - read ()

- ~~int~~ read (byte [] b)
- read (byte [] b, int off, int len)

Classes →

- InputStream → Super Class
- Buffered InputStream → Read data from buffer.
- Data Input Stream → Read data in form of primitive datatype
- File Input Stream → Read data from file
- Sequence Input stream → Read from more than 1 stream in sequence
- ~~Out~~ ObjectInputStream. → Read data in form of object.

Reader

→ All methods are same as InputStream with slight changes in some :→

- int read ()
- read (char [] c)
- read (char [] c , int off, int len)

Classes →

- Reader
- File Reader
- Buffered Reader
- Input Stream Reader

Writer

→ Same methods but instead of having byte it will have Character.

→ Classes

- writer
- Buffered writer
- File writer
- Output StreamWriter

- Print Stream.

28|4|24 {weekend class}

Inner Classes

→ Class within Scope of Class.

eg→

```
Class Outer {

      Private class Inner {

            P void Print(){

                  SOP(" Inner Print");
            }
      }

      void display(){

            Inner i1 = new Inner();

            i1.print();
      }
}
```

```
class My Class{

      PSVM(—){

            Outer ot = new Outer();

            ot.display();
      }
}
```

Output:→

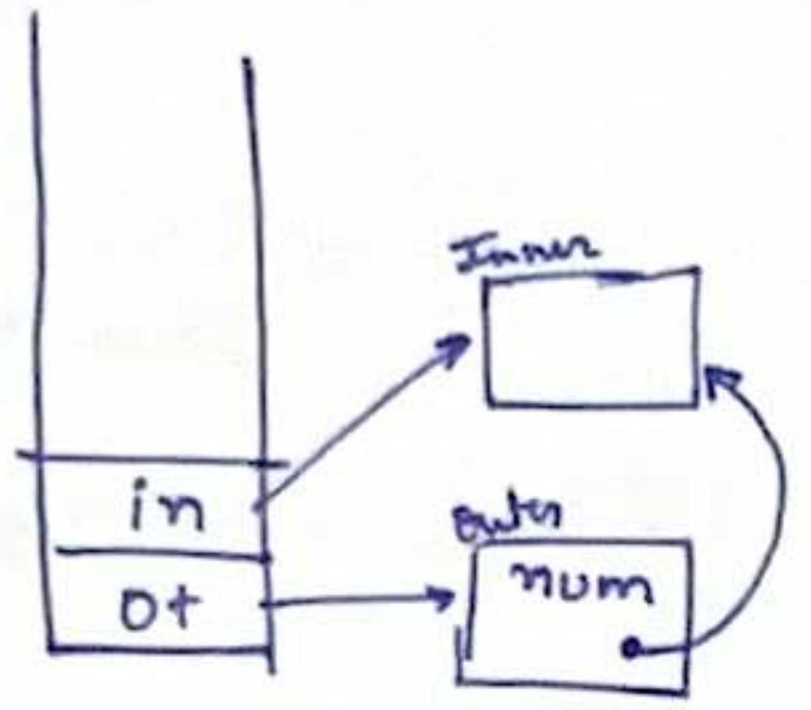
Print of inner

→ We should use the concept of inner classes if incase we need a Class ~~tho~~ whose object is dependent on another class. Meaning in our example existance of Inner depends on Outer. If we don't have Outer, inner will not exists.

```
Class Outer {

      int num = 5;

Pub. Class Inner {

      Public int getNum(){
            SOP("getNum of Inner");

            return num;
      }
  }
}
```

```
class Myclass{
      PSVM(_){

            ~~Obj~~
            Outer ot = new Outer();
            ~~Ob~~
           *{Outer.Inner in= ot.new ~~Outer~~ Inner();}

            SOP( in.getNum());
} }
```

{ → As Inner class is non-static it will only be exisible by object/ Instance of Outer. That's why we have to use Ot. new Inner() }



Output files

→ Outer. Class

→ ~~Inner $ Outer class~~ Outer $ Inner.Class

→ My Class. class

Local Inner Class

{ Class within a method }

* – Object of Local Inner Classes cannot be created outside the method.

* – Till JDK-7 a local class can't access non-final local variables of method but from JDK-8 onwards non-final local members are accessible from local inner class.

```
public Class Outer{
    int a=5;
    void myMethod(){
        int num=10;
        Class Local{
            pv printMethod(){
                SoP(" a= "+ a +" num= "+num);
            }
        }
        Local l1= new Local();
        l1.printMethod();
    }
}
```

```
class MyClass{
    PSVM(--){
        Outer ot= new Outer();
        Ot.myMethod();
    }
}
```

Output :→

→ a = 5 num = 10

{ Accessibility & scope of that class will be within myMethod() }

Anonymous Class

{ – Class without name }

→ Definition of class & creation of objects are done at same time.

→ No Constructor.

→ There can be three types of anonymous class

① – ~~That~~ class that extends a class

②. – Class that implements an Interface

③ – Anonymous class as argument.

(3)

```
AnonymousInner inner = new AnonymousInner(){
        void myMethod(){

        }
    };
```

(1)

```
abstract class MyClass{
        abstract void myMethod();
}

    Class OuterClass{

        PSVM(_){

         MyClass  m1 = new MyClass(){
                void myMethod(){
                    SOP("Method of anon.class");
                }
            };
         m1.myMethod();
        }
    }
```

Extending MyClass ↓ Here anon Class is overriding abstract Class method.

```
abstract class MyClass {
        abstract void myMethod();
}
Class M1 extends MyClass{
        void myMethod(){
            SOP(" — ");
        }
}
```

```
Class M2{
    PSVM(_){
        M1 m1 = new M1();
        m1.myMethod();
    }
}
```

{ In Class M2, we can create anon class that way we can get rid of M1. It will be same as normal. }

```
Class M2{
    PSVM(_){
        MyClass  m1 = new MyClass(){
                void myMethod(){
                    SOP(" — ");
                }
        };
    }
}
```

```
Class MyThread extends Thread{
    MyThread(){
        SOP("my Thread");
    }
    Public void run(){
        SOP(" bar");
    }
    Public void run(String msg){
        SOP(" baz");
    }
}
```

```
Public class TestThread{
    PSVM(_){
        Thread t = new MyThread(){
            PVrun(){
                SOP("foo");
            }
        };
        t.start();
    }
}
```

```
Output => → MyThread
          → foo
```

```
interface MyInt{
    abstract void myMethod();
}
```

Anon class implementing MyInt

```
Class ML{
  PSVM(_){
    MyInt m1 = new MyInt(){
          void myMethod(){
             SOP("My Method");
          }
    };

    m1.myMethod();
  }
}
```

Annonymous class as argument of Method

```
interface I1{
     String greet();
}
P class MyClass{
     Public void displayMessage(I1 i){
          SOP( i.greet());
     }
}
```

```
PSVM(_){
   MyClass m1 = new MyClass();
   m1.displayMessage(
        new I1(){
             String greet(){
                return "Hello";
             }
        });
   }
}
```