Avishek (8210928619)

2.

* Access Modifiers

#30. Declarations & Access Modifiers

10 | Apr | 23

1. Java Source file structure
2. Class level modifiers
3. Member Level modifiers
4. Interfaces

* Java Source file structure :-

→ A java program can contain any no. of classes but atmost one class can be declared as public.

→ File name & Public class name should mustmat

> class A { }
> class B { }
> class C { }

C-1 → any name, no restriction
C-2 → class B is Public, name ⇒ B.java
C-3 → class B & C is Public, C.E:-

> Any name Durg.java

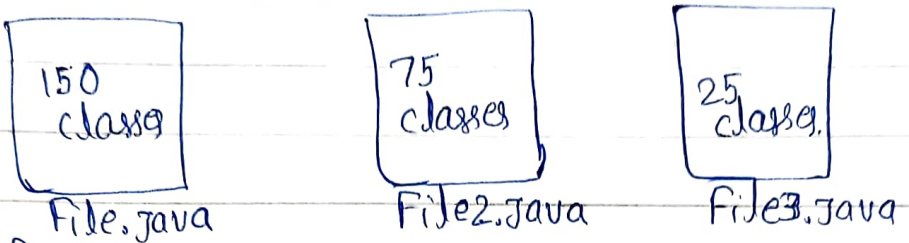
```
class A {
    PSV m(S[] args){
        Sopln("A class");}
}
class B{
    PSVm(S[] args){
        Sopln("B class");}
}
class C{
    PSVm(S[] args){
        Sopln("C class");}
}
class D{}
```

A.class, B.class, C.class
D.class ⟶ 4-file gen.

Java A↵ o/p : A class
Java B↵ o/p : B class
Java C↵ o/p : C class
Java D↵ RE:

Java Durga↵ RE:

→ 290 classes



→ Not recommended at all.

⇒ Per source file one class ⇒ Recommended.

→ ↳ Readability & maintainability improved.

> class Test {
    p s v m (String [] args) {
        ArrayList l = new AL();
    }
    ↳ CE:

> world.asia.india.tg.hydrabad.SRnagar.durgasoft;

> java.util.Arraylist; → fully classified name.
    ↳ Readeblity drop is difficult.

> import java.util.Arraylist.

* import statement

→ import statement acts as typing shortcut.

#31. Declarations & Access Modifiers

10/Apr/23

C-1 :- Type of import statement.

- Explicit class Import. : import java.util.AL;
- Implicit class Import. : import java.util.*;

→ It is highly recommended to use explicit class import bcz it improves readibility of the code.

→ Write is one time action but reading is required on regular basis.

C-2 :-

> ~~import~~ java.util.AL;
> ~~import~~ java.util.AL.*;
> ~~import~~ java.util.*;
> ~~import~~ java.util;

C-3 :-

> class MyObject extends java.rmi.UnicastRemoteObj{ }
fully qualified name

C-4 :-

> import java.util.*; Date; → compile fine
> import java.sql.*;
> class Test{
    p s v main(String[]args){
        Date d = new Date();}
} CE: reference to Date is ambeguous

util . awt
List

C-5 :-

→ Precidence order

① Explicit ⇒ ② classes present in CWD ⇒ ③ Implicit.

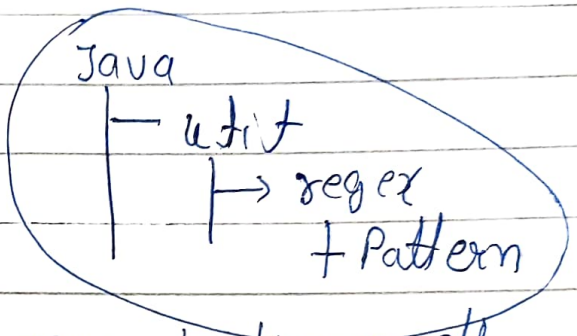
C-6 :-

① import java.*;

② import java.util.*;

③ import java.util.regex.*;

④ No import required



→ When we are importing java package all classes & Interfaces present in package by default avaible but not sub-pkg classes.

→ need to write import until sub-pkg level.

C-7 :-

① Java.lang ② default pkg (CWD)

⇒ These two ⇒ no need to import.

C-8 :-

→ import stat. is totally compile time related. more import, more time but same Run time.

C-9 :-

⇒ diff. b/w 'C' # include & Java import.

→ in 'C' all input-output headers files will be loaded at the begning only (static include)

→ In Java, no .class file will be loaded at the begning, whenever we are using particular class then only .class file will be loaded.

↳ Dynamic include (or) Load on Demand.

* static import :- (1.5 v)

→ If no specific requirement the not recommend to use static import.

Without static import

```
class Test{
  psvm(S[] args){
    sopln(Math.sqrt(4));
    sopln(Math.max(10,20);
    sopln(Math.random());}
}
```

With static import

```
import static java.lang.
  Math.sart;
class Test{
  P s v m (S[] args){
    sopln(sart(4));
    sopln(max(10,20));}
}
```

⇒ With static import we can access static methods directly.

#32. Declarations & Access Modifiers

10/Apr/23

* Explain about System.out.println():

> class Test {
    static String s = "java";
    :
  }

Test.s.length()

length() is a method present in String class.

'Test' is class name

's' is a static var. present in test class of the type String.

> class System {
    static PrintStream out;
    :
  }

System.out.println()

println() is a method present in PrintStream class.

class in java.lang

'out' is a static var. present in System class of the PrintStream.

> import static java.lang.System.out;
  class Test {
    p s v m(S[] args){
      out.println("Hello"); }
  }

=> Precidence for static.

① Current class Static ~~import~~ member
② Explicit static import
③ Implicit static import

* Normal Import

① Explicit import

import packagename.classname;

② Implicit import

import packagename.*; eg. import java.util.*;

* Static Import

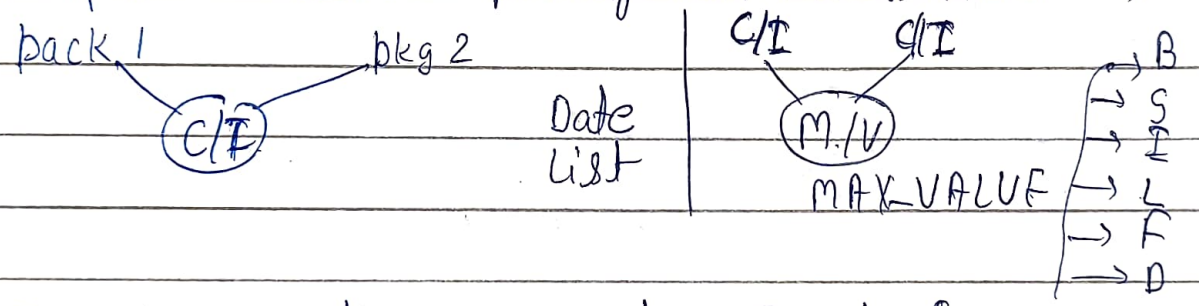
① Explicit static import

import static packagename.classname.staticmember;

② Implicit import

import static packagename.classname.*;

*



→ Two pkg with same class or Interface is rare. So, ambiguity is rare in normal import.

→ Two classes or Interfaces contain a var. or method with same name is very common & hence ambiguity problem is very common in static import.

→ Usage of static import reduces readibility & create confusion & hence if no specific req. then not rec. to use.

#33. Declarations & Access Modifiers

10|Apr|23

* Packages

→ A group of related classes & interfaces into a single unit.

→ It is Encapsulation mechanism.

eg: lang, util, io, sql etc.

→ used to resolve naming conflict.

→ modularity get improved.

→ Maintainability get improved.

→ Security of our component.

order imp.

Atmost → Pkg

Anyno → import

Anyno → class | I | enum

* nameing convention

→ Use domain name in reverse order.

com.icicibank.loan.housing.Account;

- client's internet domain name in reverse
- module name
- sub-domain name
- class name

> package com.durgasoft.scjp;

```
Public class Test.{
    p s v m(S[] args){
        Sop ln("Pkg demo");}
}
```

① Javac Test.Java ↵
CWD ↦ Test.class

② Javac -d . Test.Java ↵

destination to place generated .class files.

. → cwd

cwd
- com
  - durgasoft
    - scjp
      - Test.java

→ If pkg is not there then cmd. will create pkg.

→ At the time of execution we need to use fully qualified name.

→ At most one package statement.

→ First statement should be pkg. statement.

#34. Declarations & Access Modifiers

11/Apr/23

* Class level Modifiers :-

→ We need to provide some info. about our class to the JVM.

① whether this class accessible from anywhere or not.
② whether child class creation possible or not.
③ w Object creation is possible or not. etc.

public	final	native
private	abstract	strictfp
<default>	static	transient
protected	synchronized	volatile.

→ for top level classes

public
<default>
final
abstract
strictfp

+

Private
protected
static

for inner classes

> private class Test{
p s v m (S[] args){
Sop ln("Hello"); }
}
CE: modifier private not allowed here.

> private class A{ } } → inner classes
static class B{ }
p s v m (S[] args){
Sop ln("Hello"); }
}

* Access specifiers Vs modifiers

public	final
private	abstract
protected	static
default	

only for old languages.

→ In Java all are modifiers.

* public classes :-

→ can be accessed from anywhere.

```
package pack1;
public class A {
    public void m1(){
        Sop1n("Hello");}
}
```

Java -d . A.Java ↵

```
package pack2; import pack1.A;
class B {
  p sv m (S[]args){
  A a = new A();
  a.m1();        }
}
```

Java -d . B.Java

Java pack2.B

o/p : Hello

* default classes :-

→ can be accessed only within the current package.

→ It is also known as package level access.

* final modifier

→ It is applicable for classes, method & variables.

⇒ final method :-

⇒ final method :-

```
class P {
    public void property () {
        Sopln (" cash + land + Gold)"); }
    public final void marry() {
        Sopln (" Suba Laxmi"); }
}
```

CE:

```
class C extends P {
    public void marry() {
        Sopln (" Ishd Nitara"); } (;
}
```

→ If parent class method is defined with final then we can not override that method in child class. bcz its implementation is final.

⇒ final class :-

P → 10 methods

↑

C → 5 methods

→ If a class declared as final then we can't extends functionality of that class i.e. we can't create child class for that class.

→ So, inheritance is not possible for final classes.

```
final class P { }
class C extends P { }
```

CE: can not inherite from final P

#35 Declarations & Access Modifiers

11-Apr-23

Note:- Every method present inside final class is always final. but every variable present in final class need not be final.

→ Advantage :- we can achieve security & can provide unique implementation.

→ Dis Advantage:- we are missing key benifit of oops : Inheritance (final classes) and polymorphism (final methods)

* abstract modifier

→ abstract modifier applicable for classes & method but not for variable.

⇒ abstract method :-

```
abstract class Vehicle {
        abstract public int getNoOfWheels();
}

class Bus extends Vehicle {
        public int getNoOfWheels() {
                return 7;
        }
}
```

→ even though we don't know the implementation still we can declare a method with abstract modifier.

→ For abstract method only declaration is available but not implementation.

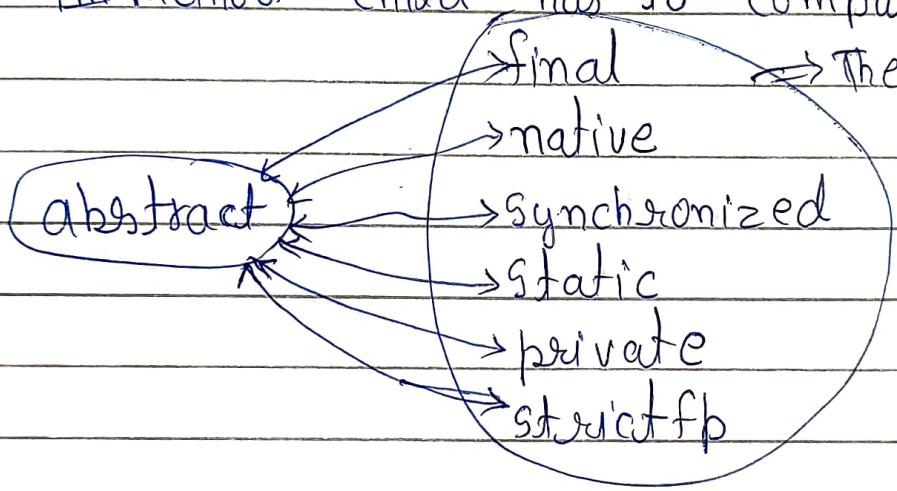
→ It should end with ";"

✓ public abstract void m1();

✗ public abstract void m1() {}

→ child class is responsible to provide impl. for parent class abstract method.

→ By declaring abstract method in parent class we are providing guidlines to child classes which method child has to compulsary implement



✗ abstract final void m1(); CE:

* abstract class

→ The class which has partial implementation.

→ we are not allowed to create an object

> abstract class Test { }

> Test t = new Test(); CE:

* abstract class vs abstract method

1) If class contain atleast one abstract method the compulsary we should declare as abstract.

Reason: If there is abstract method in class then it's implementation is not complete

So, it is not recom. to create object, to restrict object instanciation compulsary we should declare class as abstract.

2) Abstract class can contain zero abstract method
   ⇒ Httpservlet ⇒ Adapter classes

✗ class P {
    public void m1(); CE:
  }

✗ class P {
    public abstract void m1(){} CE:
  }

✗ class P {
    public abstract void m1(); CE:
  }

> abstract class P{
    public abstract void m1();
    public abstract void m2();
  }

> class C extends P {
    public void m1(){}
  }

→ For each & every abstract method child has to provide implementation otherwise child class has to declare as abstract.

* final Vs abstract.

→ For abstract child has to provide implementation (override par) but we can't override final method.

→ For final classes we can't create child class but for abstract classes we ~~should~~ have to create child class.

→ final & abstract class & method combination is illegal.

> abstract class Test{
    public final void m1(){}
}

X final class Test{
    public abstract void m1();
}

→ Abstract class can contain final method but final class can't contain abstract method.

⇒ It is highly recommended to use abstract modifier bcz it promotes oop feature like inheritence & polymorphism.

#36. Declaration & Access Modifiers

11/Apr/23

* strictfp (strict floating point) (1.2 v)
→ It is used for classes & method but not for variable.
→ result of floating point arthemitic is varied platform to platform, to get platform independent result we use strictfp modifier.

strictfp
-> public void m1() {
    Sop(10.0/3);
}
→ IEEE 754 standard.

→ All calculation ⇒ IEEE 744 standard.
→ It always talk about implementation but abstract not, so, illegal comb. for method.
⇒ class
→ abstract strictfp combination is valid on class level.
✓ abstract strictfp class Test { }
✗ abstract strictfp void m1(); CE:

* Member modifiers (method or variable level)
* public member
→ can be accessed from any where.

-> package pack1;
class A {
    public void m1() {
        Sop("A class"); }
}

-> package Pack2; import pack1.A;
class B {
    p sv m(S[] args) {
    A a = new A();
    a.m1(); } CE:
}

→ It's corresponding class should also be visible.i.e public class.

* default members
→ can be accessed within the current package.
→ also known as package level access.

* private members
→ can be accessed only within the class.
→ private abstract is illegal combination. for meth

* protected member (most misunderstood in java).
→ can be accessed ~~with~~ anywhere in current package ~~and~~ but only in child classes of outside package.

protected = <default> + kids

```
> package pack1;
  public class A {
      protected void m1() {
          sopln("most US m");}
  }
  class B extends A {
      p s v m(S[] args){
  ① A a = new A();
      a.m1() ✓
  ② B b = new B();
      b.m1(); ✓
  ③ A a1 = new B();
      a1.m1(); ✓ }
  }
```

```
> package pac2; import P1.A;
  class C extends A {
      p s v m(S[] args) {
  ① A a = new A();
      a.m1(); ✗
  ② C c = new C();
      c.m1(); ✓
  ③ A a1 = new C();
      a1.m1(); ✗ }
  }
```

→ We can access protected members within the current package either by using parent reference or by using child reference.

but outside package, only in child classes & we should use child reference only.

> package pack2; import pack1.A;

class C extends A { }

class D extends C {

p s v m(S[] args){

① A a = new A();
a.m1(); ✗

② C c = new C();
c.m1(); ✗

③ D d = new D();
d.m1(); ✓

④ A a1 = new C();
a.m1(); ✗

⑤ A a1 = new D();
a1.m1(); ✗

⑥ C c1 = new D();
c1.m1(); ✗

CE:

→ can access outside, only in child classes & we should use that child class reference only. eg :- from D class, we should use D class reference only.

Visibility	private	<default>	protected	public
within same class	✓	✓	✓	✓
child class of same pkg	X	✓	✓	✓
non-child class of same pkg	X	✓	✓	✓
child class of outside pkg	X	X	(child ref. only)	✓
Non-child class of outside pkg	X	X	X	✓

private < default < protected < public

→ recommended modifier for data member (var.) is private.

→ recommended modifier for method is public.

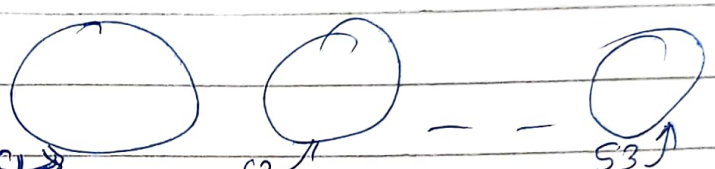
#37. Declarations & Access Modifiers

11/Apr/23

* final variables :-

⇒ final instance variable :-

> class student {
    string name;
    int rollno;
}



> class Test {
    final int x; } CE:

→ If the instance var. declared as final then compulsary we have to perform initilazation explicitly wether we are using or not.

Rule: for final instance var. we should perform init. before constructor completion.

① At the time of declaration
class Test {
    final int x = 10; }

② Inside instance block
class Test {
    final int x;
    { x = 30; }
}

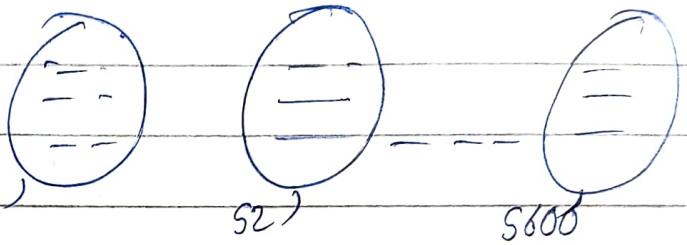
③ Inside constructor
class Test {
    final int x;
    Test () { x = 10; } ;
}

> class Test {
    final int x;
    public void m1() {
        x = 20;
    }
} CE:

> class

* final static variable:-

```
> class Student {
      String name;
      int rollno;
  Static String cname;
  }
```




```
> class Test {
     static int x;
     psvm(S[] args) {
        sopln(x); }   0 ← default value.
  }
```

```
> class Test {
     final static int x; }   CE:
```

→ Compulsary initilization required.

Rule:- Before class loading completion.

① At the time of declaration

```
class Test {
     final static int x = 10;
  }
```

✓

② Inside static Block:

```
class Test {
     final static int x;
     static {
        x = 10; }
}
```

✓

> class Test {
    final static int x;
    public void m1() {
        x = 10; } CE:
}

* final local variables :-

> class Test {
    p s v m(S[] args) {
        final int x;
        Sop1n ("Hello"); }
} o/p: Hello.

→ before using only we have to perform initialized if not using then it's not required.

→ The only applicable modifier for local var. is final

class Test {
    p s v m (S[] args) {
CE: { public int x = 10
      private int x
      protected int x
      static int x
      transient int x = 10;
      volatile int x = 10;
      final int x = 10; }
}

#38.

```
Class Test {
    p s v m (S[] args) {
        m1(10,20);          → Actual Parameters
    }
    p s v m1(int x, int y) {
        x = 100; y = 200;   → formal parameters.
        Sopln("x = " + x);
    }
```

CE:

→ Formal parameters of a method acts as local variable of that method.

→

#38. Declarations & Access Modifiers 13/Apr/23

* static modifier

→ It is applicable for method & variable but not for classes.

→ We can't declare top level class with static modifier but we can declare inner class as static (inner class called static nested classes)

```
> class Test {
    int x = 10;    static int y = 20;
    public void m1() {
        Sopln x ();  Sopln(y);  ✓ }
    public static void m1() {
        Sopln(x);  X
        Sopln(y);  ✓
    }
```

→ We can't access instance member directly from static area but we can access from instance area directly.

→ We can access static member both instance & static area directly.

> I. int x = 10;
II static int x = 10;
III public void m1() {
Sopln(x); }
IV public static void m1() {
Sopln(x); }

valid combination?
(A) I & III
(B) I & IV
(C) II & III
(D) II & IV
(E) I & II
(F) III & IV {Same Signature}

C-I → Method overloading is possible for main method but JVM will only call String one.

C-II
```
class P{
    p s v main(S[] args){
        Sopln("parent main");}
}
Class C extends P{ }
```

→ Inheritance is applicable for static method including main method.

C-III → Method hiding is applicable but not method overriding.

⇒ Inside method, if we using at least one instance variable then that method talks about a particular object hence we should declare as instance method.

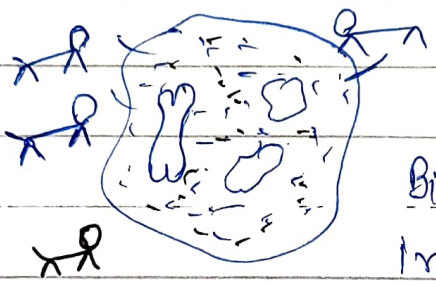
⇒ If we are not using any instance variable then this method we can declare as static method. irrespective of whether we are using static var. or not.

```
> class Student {
      String name;
      int rollno;
      int marks;
      static String cname;
      getStudentInfo() {                    => Instance method
          return name+"---"+marks; }
      getCollegeInfo() {
          return cname; }  -> static method
      getAverage(int x, int y) {
          return x+y/2; }  -> static method
      getCompleteInfo() {
          return name+"---"+rollno+"---"+marks+
              "---"+cname; }  => Instance method.
  }
```

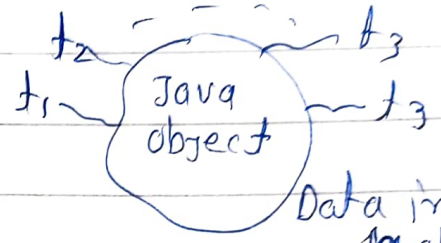
→ For static method implementation should be available where as for abstract imple. is not available hence abstract static comb. is illegal for method.

* Synchronized modifier

→ If is applicable for methods & blocks but not for classes & variables.



Biryani



Data inscons. problem

Insconsistency problem

→ If multiple threads are trying to operate simultaneously on the same java object then there may be a chance of data insconsitency problem. this is called race condition.

→ We can overcome this problem with using synchronized keyword.

→ After declaring, at a time only one thread is allowed to execute that method or block.

→ The main disadv. is it increases waiting time of threads & creates performance prob.

→ If no specific req. the it is not rec. to use it.

→ Synchronized abstract is illegal combination.

#39. Declaration & Access Modifiers

13-Apr-23

* native modifier

→ It is applicable only for methods & can't apply anywhere else.

⇒ Native :- The methods which are implemented in non-java (Mostly C/C++) are called native or foreign methods.

⇒ Main Objective :-

1) To improve performance of the system.
2) To achieve machine & memory level communication
3) To use already existing non-java code.

→ Pheudo code :-

```
> class Native {
        static {
①       System.loadLibrary("native lib. path");   // load native lib.
②       public native void m1();                  // Declare a native method
```

```
> class Client {
        p s v m(S[] args) {
            Native n = new Native();
            n.m1();                               // ③ Invoke a native method
        }
```

① Load
② Declare
③ Invoke

→ for native method implementation is already available in old lang. & we no need to provide implementation hence it should ends with ;

→ public native void m1();

✗ public native void m1(){} CE:

→ Abstract native combination is illegal.

→ strictfp native combination is illegal.

↳ we don't know other lang. will follow IEEE 754 standard.

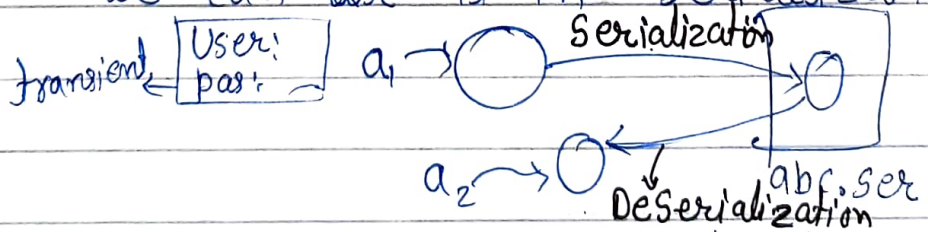
→ Dis.adv :-

It breaks platform independence nature of java.

✗ transient keyword :-

→ It is applicable only for variables.

→ We can use it in serialization context.



→ At the time of serialization if we don't want to save a value of a particular variable to meet security constraint then we should declar that variable as transient.

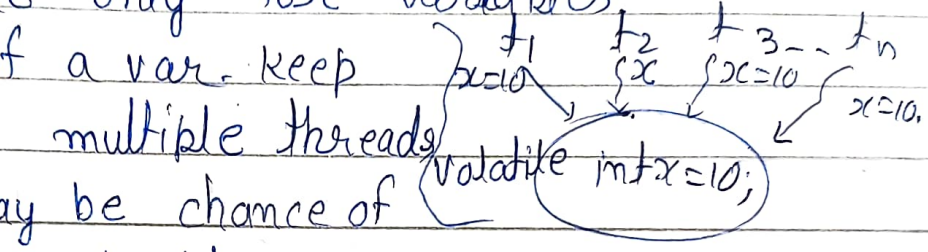
→ At the time of serialization JVM ignores orignal value & save defaut value to file

hence transient means not to serialize.

* Volatile modifier :-

→ It is applicable only for variables.

→ If the value of a var. keep on changing by multiple threads then there may be chance of data inconstiency problem.



→ To solve it by volatile modifier.

→ It will create a seperate local copy.

→ It creates performance problem (maintaining the copies)

→ It is almost depricated key-word.

→ final volatile is illegal combination.

Modifier for classes

→ The only applicable modifer for local variable is final.

→ The only applicable constroctors are public private, protected & default.

→ only for methods ⇒ native,

→ only for variables ⇒ volatile & transient.

→ classes but not for interface ⇒ final

→ classes but not for enum ⇒ final, abstract.

Modifers.	Classes		Method	variable	blocks	Interface		enum		Constructors
	outer	inner				outer	Inner	outer	inner	
① public	✓	✓	✓	✓		✓	✓	✓	✓	✓
② private		✓	✓	✓			✓		✓	✓
③ protected		✓	✓	✓			✓		✓	✓
④ <default>	✓	✓	✓	✓		✓	✓	✓	✓	✓
⑤ final	✓	✓	✓	✓		X		X	X	
⑥ abstract	✓	✓	✓			✓	✓	X	X	
⑦ static		✓	✓	✓	✓		✓		✓	
⑧ synchronized			✓		✓					
⑨ native			✓							
⑩ strictfp	✓	✓	✓			✓	✓	✓	✓	
⑪ transient				✓						
⑫ volatile				✓						

Empty box = X

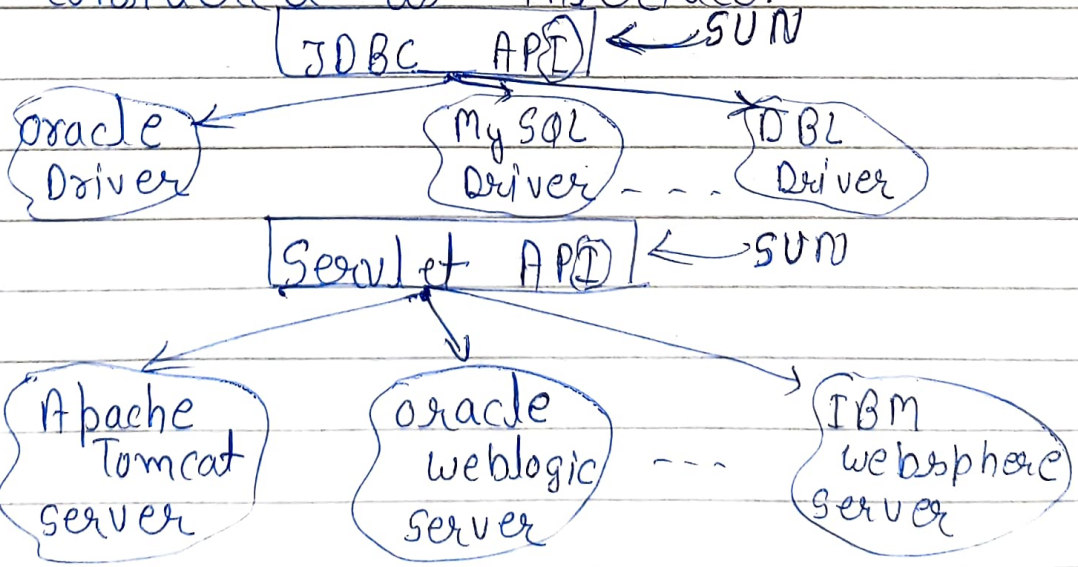
#40. Interfaces

13|Apr|23

① Introduction
② interface declaration & implementation
③ extends vs implements
④ Interface methods
⑤ Interface variables
⑥ Interface Naming conflicts
  (i) method naming conflicts
  (ii) variable naming conflicts
* ⑦ Marker interface
⑧ Adapter classes
⑨ interface vs abstract class vs concrete class.
* ⑩ Difference b/w interface & abstract class.
⑪ Conclusions.

✱ Introduction :-

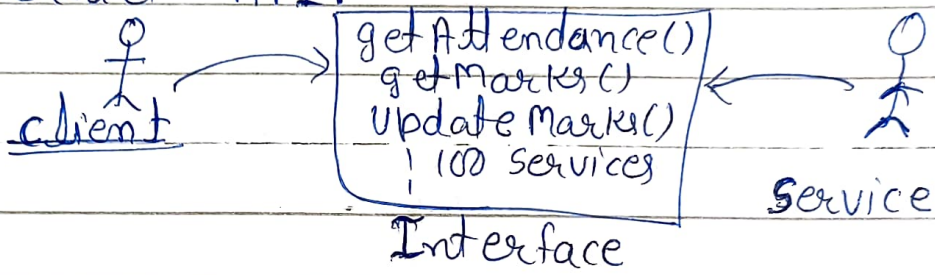
Def 1:- Any service requirement specification (SRS) is considered as interface.



→ Servlet API acts as requirement specification to develope web server,

Webserver vendor is responsible to implement servlet API.

Def 2 :-



→ From client point of view an interface defines the set of services what he is expecting.

from service point of view, the set of services that he is offering hence any contract b/w client & service provider is consider as Interface.

⇒ Bank ATM GUI example.

Def 3 :- Inside interface every method is always abstract whether we are declaring or not hence interface is considered as 100% pure abstract class.

Final Def :- Any service requirement specification or any contract between client & service provider or 100% pure abstract class is interface.

* interface declaration & implementation

> interface Interf {
    void m1();
    void m2(); }

abstract
> class ServiceProvider implements Interf {
    public void m1() { }
  }

> class SubServiceProvider extends ServiceProvider {
    public void m2() { }
  }

→ For each and every method of that interface we have to provide implementation otherwise we have to declare class as abstract. then
→ next level child class is responsible to provide implementation.
→ Every interface method is always public & abstract wether we are declaring or not.
→ whenever we are implementing an interface method compulsory we should declare as public otherwise CE:----.

1→ A class can extend only one class at a time.
2→ A interface can extend any no. of interfaces simaltniously

> interface A { }
> interface B { }
> interface C extends A, B { }

3→ A class can implements any no. of interfaces simultniously.

4→ A class can extends another class & can implement any no. of interfaces simultniously.

→ class A extends B implements C, D, E { }.

① → X extends Y

① Both X & Y should be classes

② Both X & Y should be interfaces

③ Both X & Y can be either classes or interfaces

④ No Restrictions

② X extends Y, Z

X, Y, Z should be interfaces

③ X implements Y, Z

X → class

Y, Z → interfaces

④ X extends Y implements Z

X, Y → classes

Z → interface

⑤ X implements Y extends Z CE:

#42. Interfaces

13/Apr/23

* Interface methods :-

→ Every method present in interface is public & abstract.

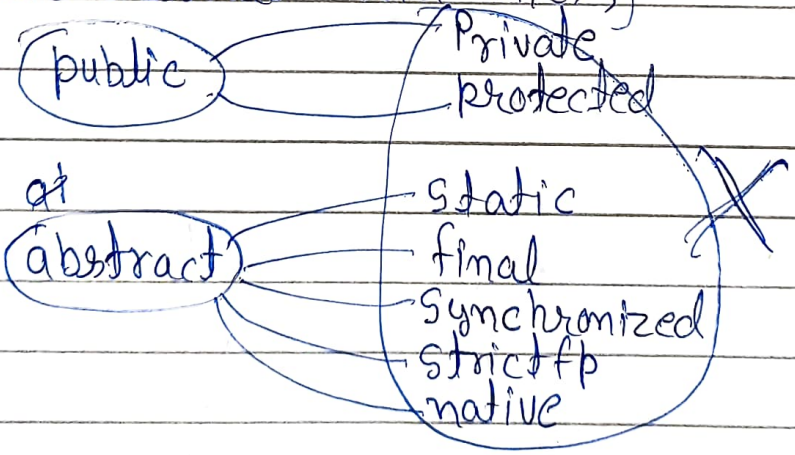
> interface Intf {
    void m1();
}

→ public: To make this method available to every implementation class.

→ abstract: Implementation class is responsible to provide implementation.

⇒ void m1();
> public void m1();
> abstract void m1();
> public abstract void m1();

⇒ all are equal inside interface.



* Interface variables :-

→ An interface can contain variables & the main purpose of it is to define requirement level constants.

> interface Interf {
int x = 10; }

→ public : To make this variable available to every implementation class.

→ static : without existing object, implementation class can access this variable.

→ final : If one implementation class changes value then remaining implementation classes will be effected.

⇒ To restrict this every interface variable is always final.

→ Every interface variable is public, static final whether we are declaring or not.

> int x = 10;
> public int x = 10;
> static int x = 10;
> final int x = 10;
> public static int x = 10;
> static final int x = 10;
> public final int x = 10;
> public static final int x = 10;

⇒ all are equal in interface

public
static
final

private
protected
transient
volatile

> interface Intef {
    int x = 10;
    ~~static { }~~
}

→ For interface var. compulsary we should perform initialization at the time declaration.

> interface Intef {
    int x = 10;
}

> class Test implements Intef {
    p s v m (String[] arg) {
        x = 777;
        Sop(x);
    }

↳ CE:

→ Inside impl. class we can access interface var. but we can't modify values.

> class Test implements Intef {
    p s v m (S[] args) {
        int x = 777;
        Sop(x);
    }

→ not modifing but we are declaring new one.

* Interface naming conflicts
* Method naming conflicts :-

> interface Left {
    public void m1();
}

> interface Right {
    public void m1();
}

> class Test implements Left, Right {
    public void m1() { }
}

C-1 ⇒ If two interfaces contain ~~same~~ a method with same signature & same return type then in the impl. class we have to provide only one method.

C-2 If two interfaces contain a method with
→ Same name but diff argument types
→ We need to provide for both method
→ these methods acts as overloaded method.

```
> interface Left{
    public void m1();
  }
```

```
> interface Right{
    public void m1(int i);
  }
```

```
> class Test implements Left, Right {
overloaded method { public void m1() { }
                    public void m1(int i) { }
  }
```

C-3

```
> interface Left{
    public void m1();
  }
```

```
> interface Right{
    public int m1();
  }
```

```
abstract class Test implements Left, Right {
    public void m1() { }
  }
class SubTest extends Test {
    public int m1(){
        return 10;
    }
}
```

→ If two interfaces contains a method with same signature but different return types then it is impossible to implement both interface simaltniously. (If return type are not co-varient {object & string})

Q:- Is a java class implement any no. of interface simaltinously ?

Ans:- Yes, except case-3

* Variable naming conflicts :-

```
> interface left{                  | > interface Right{
    int x = 777;}                  |   int x = 888;   }
> class Test implements left, Right{
      P S V m( S[] args){
          Sopln(x); CE:
          Sopln(left.x); 777
          Sopln(Right.x); 888   }
  }
```

→ Two interfaces can contain a variable with same name & there may be chance of var. naming conflict.

→ We can solve this problem by using interface names.

#42. Interface

14-Apr-23

* Marker interface :-

→ If an interface doesn't contain any methods & by implementing that interface if our object will get some ablity, called marker int.

→ Serializable (I)
Cloneable (I)
RandomAccess (I)
} These are marked for some ability.

Q:- Without having any method how the object will get some ability in marker interfaces?

Ans:- Internally, JVM is responsible to provide required ability.

Q:- Why JVM is providing required ability in marker interfaces?

Ans:- To reduce complixity of programing & to make Java language as simple.

→ Yes, we can create our own marker interface but customization of JVM is required.

* Adapter classes :-

```
interface X {
    m1();
    m2();
    :
    m1000();
}
```

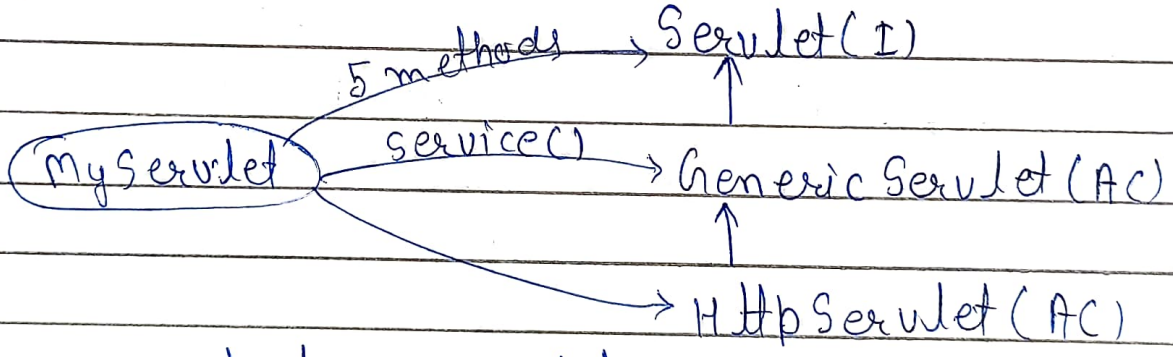
```
class Test implements X {
    m3(){ => 10 lines
    }
    m1(){}
    m2(){}
    :
    m1000(){}
}
```
999 lines.

```
abstract AdapterX
class Test implements X {
    m1(){}
    m2(){}
    :
    m1000(){}
}
```

> class Test extends Adapter X {
    m3() {
        = }
}
> class Sample extends Adapter X {
    m7() {
        = }
}

→ Adapter class is a simple Java class that implements an interface with only empty implementation.

→ Using Adapter class we can extends it and implement only our required method.



→ We can develope servlet in 3 ways (above)

① By implementing Servlet(I)
② By extending GenericServlet(AC)
③ By extending HttpServlet(AC)

→ for servlet implementation we need to provide impl. for all 5 methods.

→ If we extend generic servlet then we just need to impl. service() method.

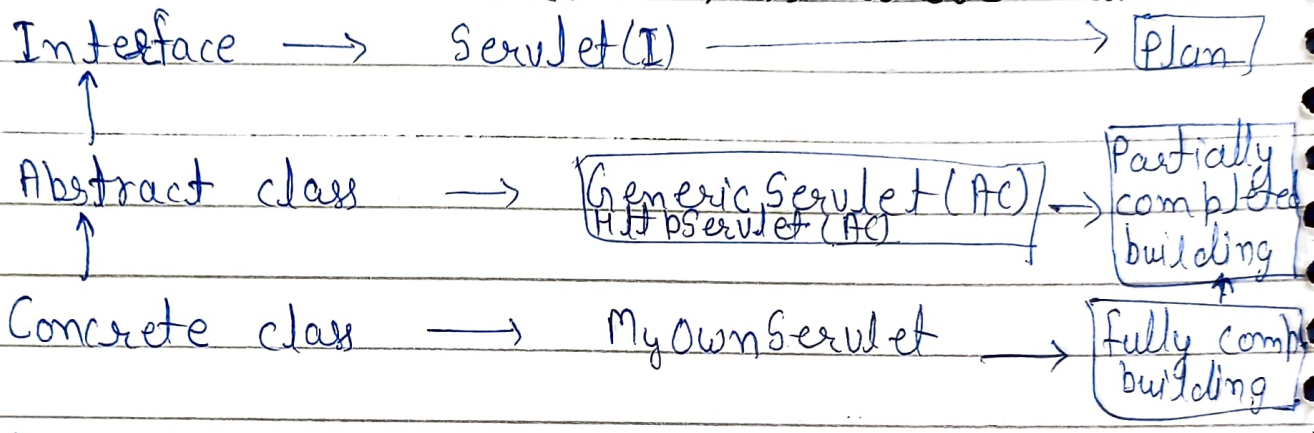
→ Generic Servlet acts as Adaptor class of servlet interface

#43 Interface

14|Apr|23

→ Marker interface & Adapter classes simplify complixity of programming. & these are good utility to us.

* interface Vs Abstract class Vs concrete class



→ If we don't know anything about implementation just we have requirement specifaction ⇒ **Interface**

→ If we are talking about implementation but not completely (Partial impl) ⇒ **Abstract class**

→ If we are talking about complete implementation & ready to provide service ⇒ **concrete class**

* Difference b/w Interface & Abstract class.

Interface	Abstract class
① Don't know about implementation & just we have req. Specification ⇒ Interface	① If are in imple. but not completly (Partial impl). ⇒ Abstract class.
② Every method is always public & abstract. (100% pure abstract class)	② need not be public & abstract & we can take concrete method also.
③ Can't use - private, protected, static, synchronized, native & strictfp	③ No restrictions on modifiers.

Variables

④ Every var. is always public static, final.	④ Every var. need not be public static final.
⑤ Can't declare – private, protected, volatile & transient.	⑤ No restrictions on variable modifiers.
⑥ Compulsary we should perform initilazation at the time of declaration only.	⑥ Not required to perform inti. at the time of declaration.
⑦ Static & instance block not block.	⑦ We can declare static & instance blocks
⑧ We can't declare constructors.	⑧ We can declare constructo

Q:- Anyway we can't create object for abstract class but abstract class can contain constructor what is the need.?

Ans:- Abstract class constructor will be executed whenever we are creating child class object to perform init. of child class object.

⇒ Either directly or indirectly we can't create object for abstract class.

⇒ The main use of constructor is to perform initilized of instance variable.

Abstract class can contain instance var. which are required for child object but every var. present inside interface is always public static final, so there is no chance of instance var. in interface, so, constructor not required.

→ Whenever we are creating child class object parent object won't be created just parent class constructor will be executed for child object purpose only.

```
> class P {
      P() {
        Sopln( this.hashCode()); } 100
  }
> class C extends P {
      C () {
      Sopln (this.hasCode()); } 100
  }
> class Test {
      p s v m (S[] args) {
       C c = new C();
       Sopln(c.hasCode()); } 100
  }
```

Q:- Inside interface every method is always abstract & we can take only abstract method in abstract class also then what is diff blw Interface & Abs. class
⇒ Is is possible to replace interface with Abs. class?

Ans:- We can replace interface with abstract class but it is not a good programming practice. this is something like recruiting IAS as sweeper.

→ If everything is abstract it is highly recommended to go for ~~abstract~~ interface.

abstract class x{ }
class Test extends x{
=
}
	interface x { }
	class Test implements x {
	=
	}

① While extending abs. class it is not possible to extend any other class & hence we are missing inheratence benifit.	① while impl. interface we can extend same other class & hence we won't miss any inheratence benifit
② object creation is costly Test t = new Test(); (2 min)	② object creation is not costly Test t = new Test(); (2 Sec)

#44. Interface & Abstract class loopholes

Q Roles of new keyword & Constructor? 14|Apr|23

* new Vs Constructor :-

> class Student{
    String name;
    int rollno;
    Student(String name, int rollno){
        this.name = name;
        this.rollno = rollno; }
}

Constructor

> Student s = new Student("durga", 101);

Responsible to create object

To initialize object

s → name: ~~null~~ durga
rollno: ~~0~~ 101

→ The main objective of new operator is create an object.

→ The main purpose of constructor is to initilize object.

→ First object will be created using new operator & then inilization will be performed by constructor.

→ Before Constructor Only object is ready & hence within the cons. we can access object properties like Hash code.

> class Test{
    Test(){ Sopln(this); // Test@6e3d80
        Sopln(this.hashCode()); 7229672 } }

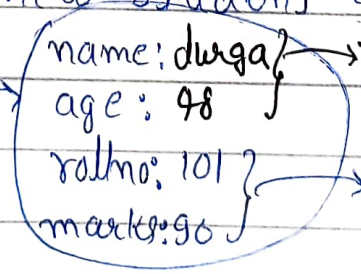
#45. Interface & Abstract class loopholes

* child Object Vs parent constructor. 14|Apr|23

Q:- Whenever we are creating child class object, what is the need of executing Parent class constructor ?

Ans:-
```
class Person {
    String name;
    int age;
    Person(String name, int age) {
        this.name = name;
        this.age  = age;
    }
}
class Student extends Person {
    int rollno;
    int marks;
    Student(String name, int age, int rollno, int marks)
    {   super(name, age);
        this.rollno = rollno;
        this.marks = marks;
    }
}
```

> Student s = new Student ("durga", 48, 101, 90);



→ whenever we are creating child class object automatically parent constructor will be executed

#46. Loopholes

14/Apr/23

to perform initi for the instance variables which are inherating from parent.

→ Above parent & child constructor executed for child object initi. only.

Q:- Whenever we are creating child class object whether Parent object will be created or not?

Ans:-

```
> class P{
      P(){
        Sopln(this.hashCode()); } 100 ~~200~~
  }
> class C extends P {
      C(){
      sopln(this.hashCode(); } 100
  }
> Class Test {
      p s v m(S [] args) {
       ~~sop~~ln(C c = new C();}
        Sopln( C.hashCode()); } 100
  }
}
```

→ Whenever we are creating child class object parent const. will be executed but parent object won't be created.

→ One object created but both const. executed. for child object purpose only.

#47. Loopholes

14/Apr/23

* Need of Abstract class constructor ?

Q:- Anyway we cannot create object for abstract class either directly or indirectly, But abstract class can contain constructor. What is the need ??

Ans:-

* Without constructure in abstract class.

```
> abstract class Person {
      String name;
      int age;          }
> class student extends Person {
      int roll no;
      int marks;
      Student (String name, int age, int rollno, int marks)
      { this.name = name;
        this.age = age;
        this.rollno = rollno;
        this.marks = marks; }

Student S1 = new Student ("durga", 48, 101, 90);
```

S1 → name: durga
age: 48
rollno: 101
marks: 90

```
> class Teacher extends Person{
      double salary;
      string subject;
      Teacher(st na, int age, double sd, String sub){
          this.name = name;
          this.age  = age;
          this.salary = salary;
          this.subject = subject;}
  }
```

t → name: nagor, age: 47, salary: 25, sub: java

```
  Teacher t = new Teacher("nagor", 47, 25, "java");
```

* With constructor in abstract class

```
> abstract class Person{
      String name;
      int age;
      Person(String name, int age){
          this.name = name;
          this.age  = age        }
> class Student extends Person{
      int rollno;
      int marks;
      Student(st name, int age, int rollno, int marks){
          super(name, age);
          this.rollno = rollno;
          this.marks = marks;}
}   Student s = new Student("durga", 48, 101, 90);
```

s → name: age: rollno: marks:

→ Anyway we can't create object for abstract class either directly or indirectly, but abstract class can contain constructor.

Need :- The main objective of abstract class constructor is to perform initilazation for the instance variable which are inherating from abstract class to the child class.

→ whenever we are creating child class object automatically parent constructor will be executed to perform initilization for the instance variable which are inheriting from abstract class. (code re-usability.)

#48. Loopholes

14|Apr|23

Q:- Anyway we cannot create Object for Abstract class and interface. Abstract class can contain constructor but interface does not. Why?

Ans :-

```
> abstract class Person {
      String name;
      int age;
      Person(String name, int age){
          this.name = name;
          this.age = age; }
  }
```

→ child

```
> interface Intef {
      int x = 10;
  }
```

→ public, static, final

⇒ The main purpose of constructor is to perform initilization of an objectie. to perform inti. for instance variables.

⇒ Abstract class can contain instance variable which are required for child class object.

→ To perform initi. for thes instance var. constructor is required for abstract classes.

→ Every var. present inside interface is always public, static, final. hence there is no chance existing instance var. inside interface

#49. Loopholes

14/Apr/23

bcz of this constructor is not required for interfaces.

Q:- Inside Interface we can take only Abstract Methods. But in Abstract class also we can take only Abstract Methods Based on our requirement. Then what is the need of Interface? i.e. Is it Possible to Replace Interface concept with Abstract class?

Ans:-

> interface x{ }	abstract class x {
① class Test extends A implement x { } ✓	① class Test extends x — (A) { } ✗
② class Test implements x { } Test k = new Test(); (2 min)	② class Test extends x { } Test t = new Test(); (20 min)

→ If everything is abstract it is highly recommended to go for interface but not abstract class.

→ We can replace interface with abstract class but it is not a good programming practice. (This somethin recruiting IAS for sweeping purpose)

↳ While implementing interface we can extends any other class & hence we won't miss inheritance benifit.

#50. Loopholes

14|Apr|23

I→ While extending abstract class we can't extend any other class & hence we are missing inheritence benifit.

II→ Object creation is not costly in interface.

II→ In abstract class case object creation is costly as we are calling parent constructor.

* Interface, Abstract class, constructor :~

⇒ which are valid?

1) The purpose of const. is to create an object. ✗

2) The purpose of const. is to initialize an object but not to create object. ✓

3) Once the constructor completes then only object creation completes ✗

4) First object will be created and then const. will be executed ✓

5) The purpose of new keyword is to create object & of constructor is to initilize that object. ✓

6) We can't create object for abstract class directly but wi ndirectly we can create ✗

7) Whenever we are creating child class obj. automatically parent class obj. will be created internally ✗

8) Whenever we are creating child class obj. automatically abstract class const. will be executed. ✓

9) Whenever we are creating child class object automatically parent object will be created. ✗

10) Whenever we are creating child class object automatically parent constructor will be executed but parent object won't be created. ✓

11) Either directly OR indirectly we can't create object for abstract class & hence constructor concept is not applicable for abstract class. ✗

12) Interface can contain constructor. ✗