Programming with Java for Beginners

Bineet Sharma

Object Oriented Programming (Advanced)

Advanced OOP Series

Assumptions & Expectations

Advanced OOP Series

- **Assumptions**
  - Class and Objects Series
- **Expectations**
  - More in-depth knowledge of OOP

Objectives

Advanced OOP Series

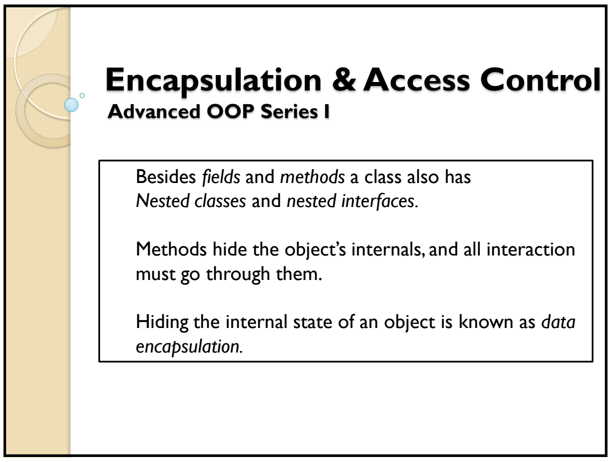
- **Encapsulation & Access Control**
- **Inheritance**
- **Abstract Class**
- **Interface**

Encapsulation & Access Control

Advanced OOP Series I

A *class* refers to a blueprint which defines state (variables) and behavior (methods) of a problem domain.

An *object* in the other hand is an instance of a class (a snapshot with its own concrete data in memory) and has behavior defined in the class (a blue print)

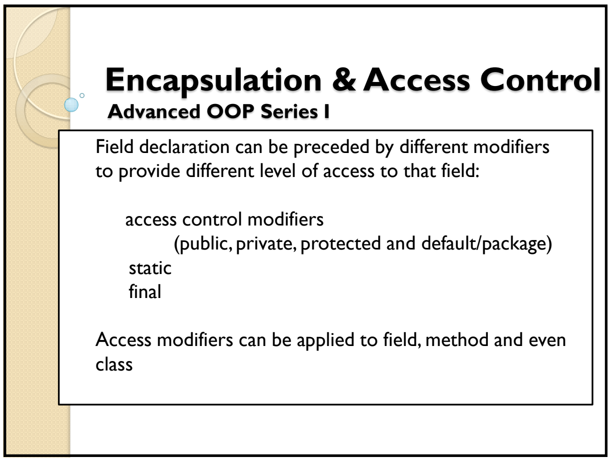


Encapsulation & Access Control
Advanced OOP Series I

Besides *fields* and *methods* a class also has *Nested classes* and *nested interfaces*.

Methods hide the object's internals, and all interaction must go through them.

Hiding the internal state of an object is known as *data encapsulation*.

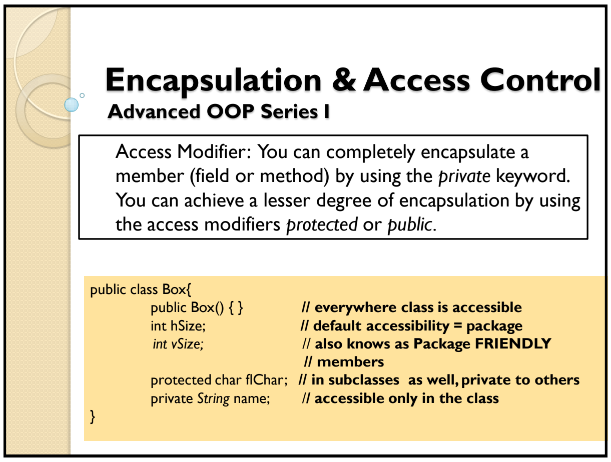


Encapsulation & Access Control
Advanced OOP Series I

Field declaration can be preceded by different modifiers to provide different level of access to that field:

access control modifiers
(public, private, protected and default/package)
static
final

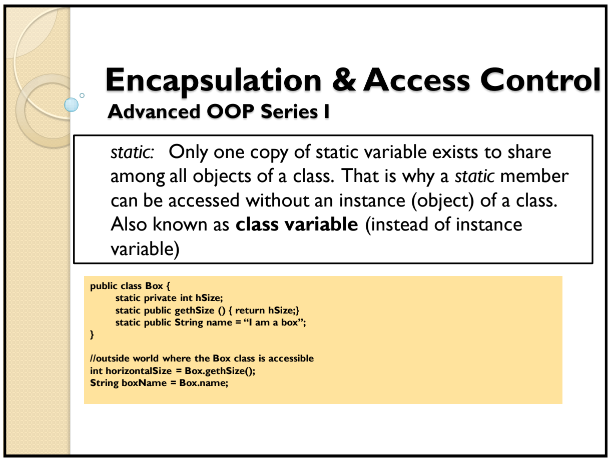
Access modifiers can be applied to field, method and even class



Encapsulation & Access Control
Advanced OOP Series I

Access Modifier: You can completely encapsulate a member (field or method) by using the *private* keyword. You can achieve a lesser degree of encapsulation by using the access modifiers *protected* or *public*.

```
public class Box{
        public Box() { }            // everywhere class is accessible
        int hSize;                  // default accessibility = package
        int vSize;                  // also knows as Package FRIENDLY
                                    // members
        protected char flChar;      // in subclasses  as well, private to others
        private String name;        // accessible only in the class
}
```

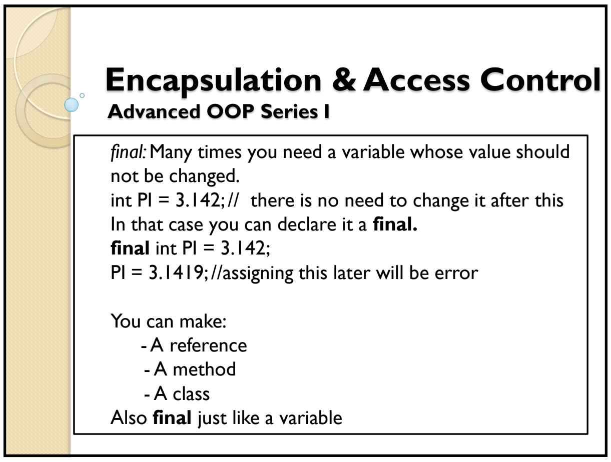


Encapsulation & Access Control
Advanced OOP Series I

static: Only one copy of static variable exists to share among all objects of a class. That is why a *static* member can be accessed without an instance (object) of a class. Also known as **class variable** (instead of instance variable)

```
public class Box {
        static private int hSize;
        static public gethSize () { return hSize;}
        static public String name = "I am a box";
}

//outside world where the Box class is accessible
int horizontalSize = Box.gethSize();
String boxName = Box.name;
```

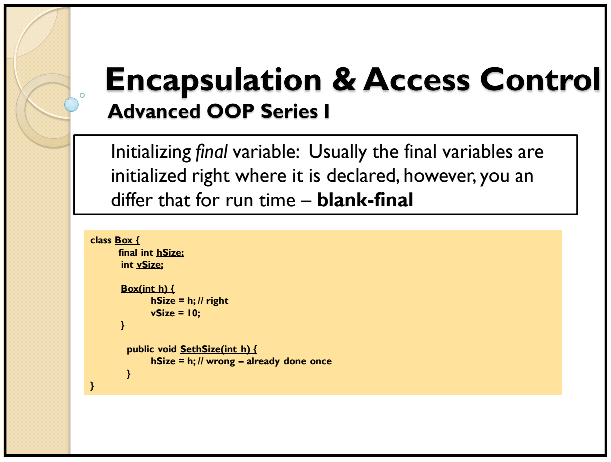


Encapsulation & Access Control
Advanced OOP Series I

final: Many times you need a variable whose value should not be changed.
int PI = 3.142; // there is no need to change it after this
In that case you can declare it a **final.**
final int PI = 3.142;
PI = 3.1419; //assigning this later will be error

You can make:
- A reference
- A method
- A class

Also **final** just like a variable



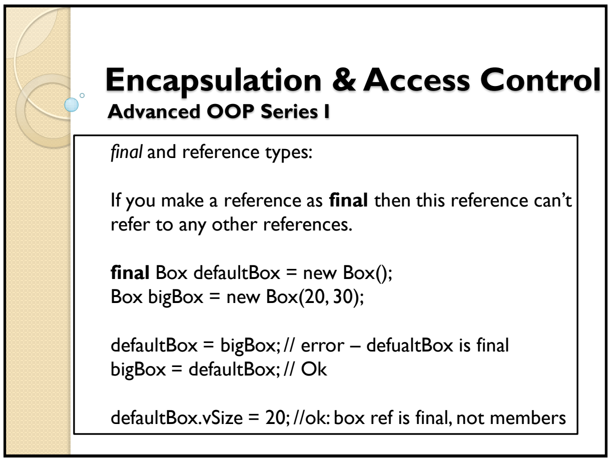
Encapsulation & Access Control
Advanced OOP Series I

Initializing *final* variable: Usually the final variables are initialized right where it is declared, however, you an differ that for run time – **blank-final**

```
class Box {
        final int hSize;
         int vSize;

         Box(int h) {
                 hSize = h; // right
                 vSize = 10;
         }

          public void SethSize(int h) {
                 hSize = h; // wrong – already done once
          }
}
```



Encapsulation & Access Control
Advanced OOP Series I

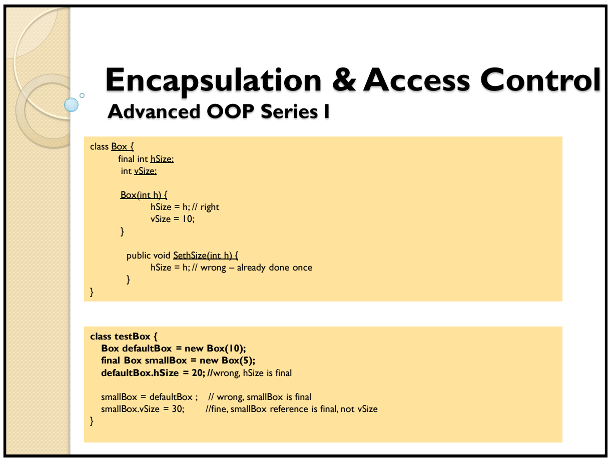
final and reference types:

If you make a reference as **final** then this reference can't refer to any other references.

final Box defaultBox = new Box();
Box bigBox = new Box(20, 30);

defaultBox = bigBox; // error – defualtBox is final
bigBox = defaultBox; // Ok

defaultBox.vSize = 20; //ok: box ref is final, not members



Encapsulation & Access Control
Advanced OOP Series I

```
class Box {
        final int hSize;
         int vSize;

         Box(int h) {
                 hSize = h; // right
                 vSize = 10;
         }

          public void SethSize(int h) {
                 hSize = h; // wrong – already done once
          }
}
```

```
class testBox {
   Box defaultBox = new Box(10);
   final Box smallBox = new Box(5);
   defaultBox.hSize = 20; //wrong, hSize is final

   smallBox = defaultBox ;    // wrong, smallBox is final
   smallBox.vSize = 30;       //fine, smallBox reference is final, not vSize
}
```

Encapsulation & Access Control

Advanced OOP Series I

final method arguments: You can have a method arguments which are *final.* It is a same concept, you can use them however, you can't assign another reference to it.

```
public void setvSize(final Box b) {
                b = new Box(5); //wrong b is final
                b.vSize = 20;   //fine, be is final, not members
}
```

Encapsulation & Access Control

Advanced OOP Series I

final methods: A *final* method cannot be overridden in subclasses.

final class: A final class can no longer be inherited to create subclasses. All methods of a final class are implicitly final

Encapsulation & Access Control

Advanced OOP Series I

this keyword: Java has a concept of referring to itself (same object). You use *this* keyword in same object to refer itself. You can use them for:

A) Resolving name conflict where the field name and method arguments are same.

```
public class Box {
        private int hSize;
         public sethSize (int hSize) {
                hSize = hSize;        //which hSize? Error
                this.hSize = hSize; //avoids name conflict
         }
}
```

Encapsulation & Access Control

Advanced OOP Series I

this keyword: Java has a concept of referring to itself (same object). You use *this* keyword in same object to refer to itself. You can use them for:

B) To pass reference to the current object as a parameter to other methods

```
public class Box {
     private int hSize;
     Box (int hSize){
          Box(this); //call another constructor
     }
}
```

Encapsulation & Access Control

Advanced OOP Series I

Initialzer in Java: It is block of instructions which are performed right after the fields are created and just before the constructors are called.

This is confusing though, so, a constructor is better way to initialize fields.

```
public class Box {
            //you can initialize like this
            int height = 10;
            int width = 20;
            String hChar = "-";
            String vChar;
            static String boxName;
            //or this - just before constructors
            {vChar = "*";}
            //static initializer
            static {boxName = "Default Box";}
}
```

Encapsulation & Access Control

Advanced OOP Series I

Method *overloading:* A class can have more than one method with the same name as long as they have different parameter list.

Compiler calls the right method by matching the name and parameter lists.

```
public void changeDimension(char vChar) {

}
public void changeDimension(char vChar, int vSize) {

}
```

Encapsulation & Access Control

Advanced OOP Series I

Constructor *overloading:* You can overload constructors as well.

You can:

• Have constructors w/o parameters(empty constructor)
• Not have a constructor (Java creates a default constructor for you – which does nothing)
• Have one constructor calling another in the class
• Call Super class constructor

Encapsulation & Access Control

Advanced OOP Series I

```
public class Box {   //Example 1
            //you can initialize like this
            int height = 10;
            int width = 20;
            String hChar = "-";
            String vChar;
             //or this - just before constructors
            {vChar = "*";}

            public Box() {};  //empty constructor
            //better way to initialize as the user get more choices
            public Box(int hSize, int vSize)   //overloaded constructor
            {
                        height = hSize;
                        width = vSize;
            }

            public Box(String hChar, String vChar, nt hSize, int vSize)   //overloaded constructor
            {
                        this (hSize, vSize); //call another constructor, must be first call
                        this.hChar = hChar; // use this reference to avoid name conflict
                        this.vChar = vChar;
            }
}
```

Encapsulation & Access Control

Advanced OOP Series I

```
public class Box { //example II
        //you can initialize like this
        int height = 10;
        int width = 20;
        String hChar = "-";
        String vChar;

        public Box(int hSize, int vSize)   //overloaded constructor
        {
                height = hSize;
                width = vSize;
        }
}

Box b; // is ok
//however
b = new Box();          // is not ok as there is no constructor for this
b  = new Box(10,20); // ok
```

Once you provide one constructor, Java does not provide a default (empty constructor)

Encapsulation & Access Control

Advanced OOP Series I

- **Demo**

Encapsulation and Access Control

Inheritance:

Advanced OOP Series II

One of the advantage of OOP is *code re-use*

Java allows multiple ways for you to re-use already developed and time tested class in your own code.

• By way of **composition**: Embed one class into another.
• Another way is through **inheritance**. Here one class inherits all the properties of another class

Composition defines '**has-a**' relationship, while inheritance defines '**is-a**' relationship.

Inheritance:

Advanced OOP Series II

Composition: When you add a reference to another class you are already using the composition.

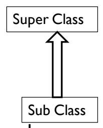
For example:

```
public class Box {
        int height;
         String name;
}
```

Here Box '**has-a**' String, so, we are already using composition relationship.

Inheritance:
Advanced OOP Series II

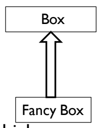
Inheritance: Process of inheriting state and behavior from another class



-Every class in java has one and only one parent
- Only Object class has no parent
- If you don't derive a class from another, it is implicitly derived from java.lang.Object class
- You can have a chain of derivations

Inheritance:
Advanced OOP Series II

What is Fancy Box:? An object of type box, having all members of Box and few of its own.



-It is an object which responds to messages (methods) as though it is a box class.
- If more methods are added in Fancy box class or methods of Box class is over-ridden, then it will also respond differently to these new messages

Inheritance:
Advanced OOP Series II

So what is a Subclass?

Sub-class can be thought of as a class which also contains (like in composition) the superclass.

So, in our example Fancy Box contains the Box as well, as a sub-objet (parts of the whole).

That means, during creation of Fancy Box, a Box object needs to be created as well, and Box needs to be created first.

Call the constructor of Base in the constructor.

Inheritance:
Advanced OOP Series II

How To Implement Inheritance?

public class Box {
    //all the methods
}

public class FancyBo **extends** Box {
    //additional members
}

You use the keyword **extends** to derive one class from another.

Inheritance:
Advanced OOP Series II

How To Implement Inheritance?

-Inheritance is always public, there is no access modifier for it.
- You can access base class functionality by using **super** keyword
- Derived class can call the base class constructors by using **super** keyword (it has to be first instruction.)
-You have to call base constructor if the base class does not have an empty constructor.

Inheritance:
Advanced OOP Series II

How To Implement Inheritance?

```
public class Box {
     public Box() {};
     public Box(int hSize, ….) {
     }
}

public class FancyBox extends Box {
     public FancyBox() {
               super();
     }
```

Inheritance:
Advanced OOP Series II

- **Demo**

Inheritance

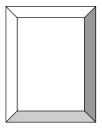
Abstract Class:
Advanced OOP Series III

Abstraction: It is the modeling, representation, of an element of the problem domain in a program.

Example: Representation of a **shape** in a program

Type Name	**Drawable**

Attributes	Draw()
	Expand()
	Shade()



Abstraction is the process of identifying and grouping attributes and behavior of an entity.
Allows architect to focus on important attributes

Abstract Class:
Advanced OOP Series III

Abstract method: Java allows you to declare an object without defining it first:

Box b;

Similarly java allows you to declare a *method* without really defining it – meaning without body

public abstract void draw(); //missing {}

draw is called abstract method.

Why would you need an abstract method?

Abstract Class:
Advanced OOP Series III

Abstract class: A class which has at least one abstract method is called an **abstract class.**

```
abstract class MyBox{
        public void getHeight(){
                return hHeight;
        }
        public void setHeight(int h){
                hHeight=h;
        }
    public abstract void draw(); //we want this
        //method to be implemented by a concrete class
}
```

Abstract class is not complete – can't create new object

Abstract Class:
Advanced OOP Series III

So, why we need abstract class? Examples explains it:

The problem: Students are either undergraduate, graduate or PhD. There is a need for uniform behavior of Student, however, we don't want the user to create student object, as students are one of these three types. Solution is to make Student as abstract class.

Another problem: **Shape** which can be drawn. You need to make sure that this class and all classes which are inherited from it must have certain characteristics, e.g. Draw(), Expand(), Shade(). However, the **Shape** is really not in a concrete shape yet which can be drawn. Unless you know what kind of shape it is you wouldn't know how to draw it, would you? However the subclasses like Box, Rectangle, Triangle, FancyBox which could be inherited from **Shape** class can be *drawn.*

Abstract Class:
Advanced OOP Series III

So, what is the Solution?

Solution is to provide an empty class, with empty behaviors and kind of enforce that this class can only be used for inheritance and not for instantiating any objects as this object can't do anything.

In Java this is achieved by defining an **Abstract Class**.

Abstract methods are needed where two or more subclasses are expected to fulfill a similar role in different ways through different implementation.

These subclasses extend the same Abstract class and provide different implementations for the abstract methods.

Abstract classes define broad types of behaviors at the top of an object-oriented programming class hierarchy, where subclasses provide implementation details of the abstract class.

Abstract Class:
Advanced OOP Series III

```
abstract public class Shape {
            public Shape() {};
             abstract public void draw();
             abstract public String getName();
}


public class Box extends Shape {
            public Box() {};
            public void draw() {
                System.out.println("draw is implemented");
            }
            public String getName() {return name};
}
```

Abstract Class:
Advanced OOP Series III

```
abstract public class Shape {
                public Shape() {};
          abstract public void draw();
          abstract public String getName();
}

public class Box extends Shape {
                public Box() {};
          public void draw() {
              System.out.println("draw is implemented");
          }
          public String getName() {return name};
}
```

Shape myShape = new Shape(); //is not allowed, is a syntax error
Shape yourShape = new Box();
String strName = yourShape.getName(); //allowed due to polymorphis**m**
Box defaultBox = new Box();

You can extend an abstract class. If you don't define all the inherited abstract methods, then that class must be abstract to.

Once all the abstract methods are defined, it becomes a concrete (complete) class and can be instantiated.

Interestingly, you can declare a class as abstract class even it does not contain any abstract methods. This is typically done when you want to prevent any instantiation of that class.

Abstract Class:
Advanced OOP Series III

When you extend the Shape class, you must implement the abstract methods, draw and getName;

```
public class Box extends Shape {
            public Box(int hSize, int vSize)    //overloaded customer
            {
                        height = hSize;  width = vSize;
            }
      public void draw(){
           drawHLine();
           drawVLines();
           drawHLine();
      }
      public String getName() {
           return name;
      }
}
```

Abstract Class:
Advanced OOP Series III

When you extend the Shape class, you must implement the abstract methods, draw and getName;

```
public class Box extends Shape {
             public Box(int hSize, int vSize) {};   //overloaded customer
            public void draw(){ }
            public String getName() {}
}

Box myBox = new Box(); //perfect
```

You can extend Box to give it a FancyBox functionality and keep extending if needed. However, a time might come where you don't want any more inheritance of your class. In that case you can use a **final** keyword to prevent your class from inheriting again. This is true for a method or a field as well.

Abstract Class:
Advanced OOP Series III

- **Demo**

Abstract Class

Interface:
Advance OOP Series IV

A Problem: At times during design you will run into a situation where you wish you could provide a uniform functionality across different classes, which may not be directly related by inheritance.

Java does not allow inheritance from multiple classes, so, they provide **Interface** mechanism which comes very close to multiple inheritance.

Interface is really a promise to outside world to guarantee a certain behavior by a class which will implement this **interface,** a contract.

While there are differences, however, **interface** is very much look like **abstract** class.

Interface:
Advance OOP Series IV

How do you define an interface?

You declare an interface using a keyword **interface**, give it a name, and its members. Interface can have three types of members:

- Constant fields
- Methods
- Nested classes and interfaces

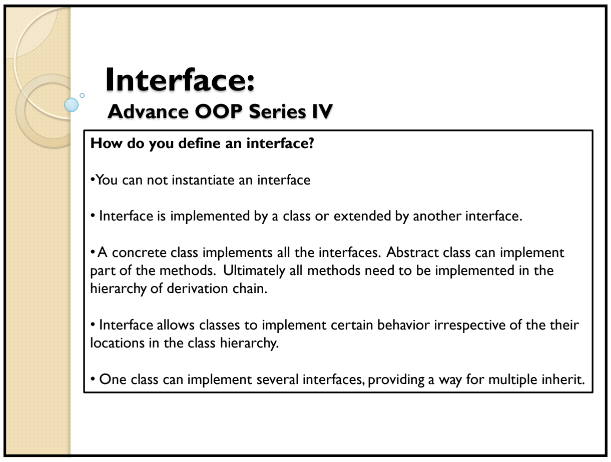
```
public interface IShape {
        public void draw (); //it contains method signatures without body
        public void setFillCharacter(String str);
        // all methods are abstract , so, no need to specify abstract
}
```

Inter face can have only named constants, **public, static and final.** It can have only abstract methods, however all these modifiers are omitted by convention.

Interface:
Advance OOP Series IV

How do you define an interface?

```
public interface IDog { //implicitly abstract.  Can have a public/package
                                //(default)modifier
        int LIFE_EXPECTANCY = 15; //named constants
        int MAX_SPEED = 30;  //implicitly public, void and final
        void fetch (); //methods are implicitly abstract and public
        void wagTail(); //they can't be final or static, why?
}
```



Interface:
Advance OOP Series IV

How do you define an interface?

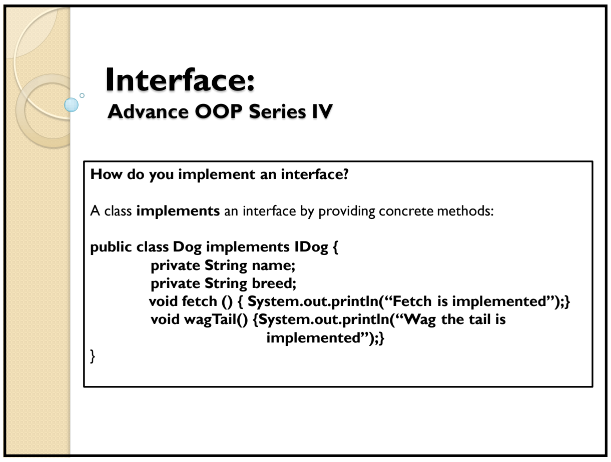
•You can not instantiate an interface

• Interface is implemented by a class or extended by another interface.

• A concrete class implements all the interfaces. Abstract class can implement part of the methods. Ultimately all methods need to be implemented in the hierarchy of derivation chain.

• Interface allows classes to implement certain behavior irrespective of the their locations in the class hierarchy.

• One class can implement several interfaces, providing a way for multiple inherit.

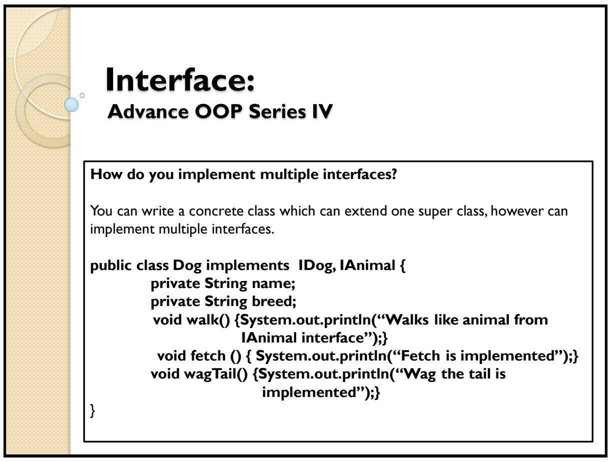


Interface:
Advance OOP Series IV

How do you implement an interface?

A class **implements** an interface by providing concrete methods:

```
public class Dog implements IDog {
            private String name;
            private String breed;
            void fetch () { System.out.println("Fetch is implemented");}
            void wagTail() {System.out.println("Wag the tail is
                                    implemented");}
}
```

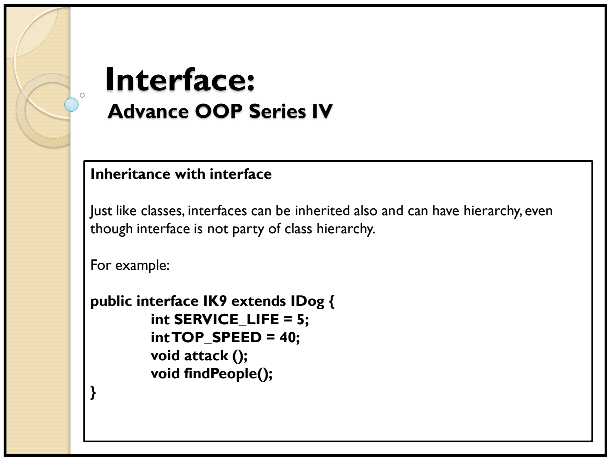


Interface:
Advance OOP Series IV

How do you implement multiple interfaces?

You can write a concrete class which can extend one super class, however can implement multiple interfaces.

```
public class Dog implements  IDog, IAnimal {
            private String name;
            private String breed;
             void walk() {System.out.println("Walks like animal from
                              IAnimal interface");}
              void fetch () { System.out.println("Fetch is implemented");}
            void wagTail() {System.out.println("Wag the tail is
                                    implemented");}
}
```



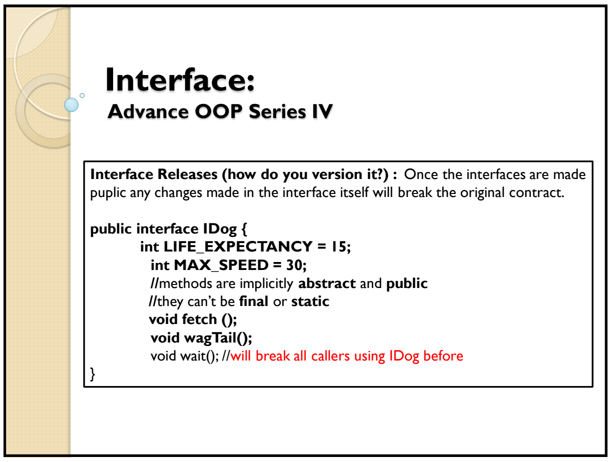
Interface:
Advance OOP Series IV

Inheritance with interface

Just like classes, interfaces can be inherited also and can have hierarchy, even though interface is not party of class hierarchy.

For example:

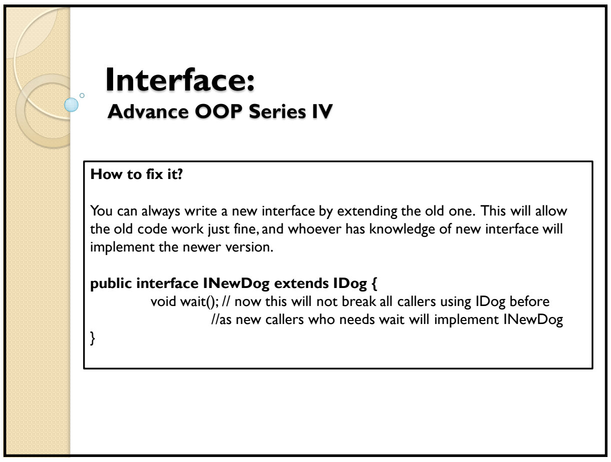
```
public interface IK9 extends IDog {
            int SERVICE_LIFE = 5;
            int TOP_SPEED = 40;
            void attack ();
            void findPeople();
}
```



Interface:
Advance OOP Series IV

Interface Releases (how do you version it?) : Once the interfaces are made puplic any changes made in the interface itself will break the original contract.

```
public interface IDog {
        int LIFE_EXPECTANCY = 15;
         int MAX_SPEED = 30;
         //methods are implicitly abstract and public
         //they can't be final or static
         void fetch ();
         void wagTail();
         void wait(); //will break all callers using IDog before
}
```

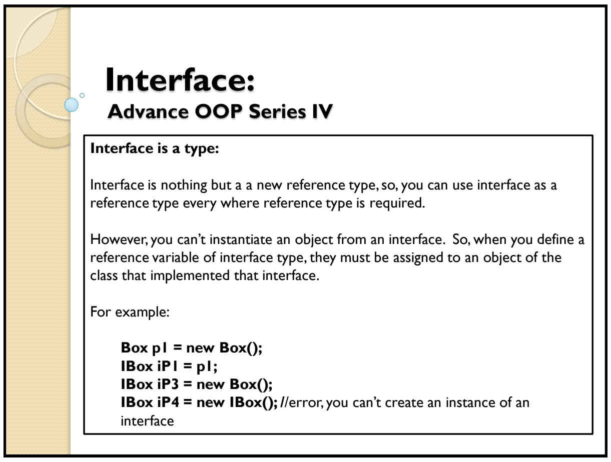


Interface:
Advance OOP Series IV

How to fix it?

You can always write a new interface by extending the old one. This will allow the old code work just fine, and whoever has knowledge of new interface will implement the newer version.

```
public interface INewDog extends IDog {
            void wait(); // now this will not break all callers using IDog before
                        //as new callers who needs wait will implement INewDog
}
```



Interface:
Advance OOP Series IV

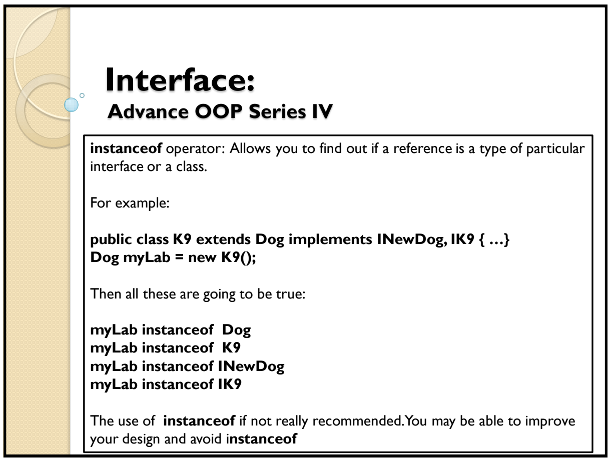
Interface is a type:

Interface is nothing but a a new reference type, so, you can use interface as a reference type every where reference type is required.

However, you can't instantiate an object from an interface. So, when you define a reference variable of interface type, they must be assigned to an object of the class that implemented that interface.

For example:

```
Box p1 = new Box();
IBox iP1 = p1;
IBox iP3 = new Box();
IBox iP4 = new IBox(); //error, you can't create an instance of an
interface
```



Interface:
Advance OOP Series IV

instanceof operator: Allows you to find out if a reference is a type of particular interface or a class.

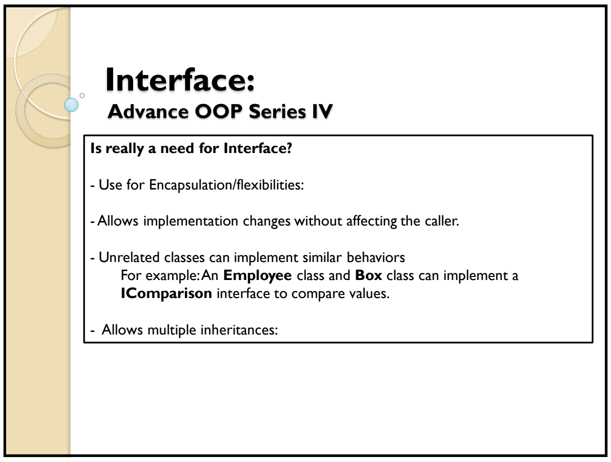
For example:

```
public class K9 extends Dog implements INewDog, IK9 { …}
Dog myLab = new K9();
```

Then all these are going to be true:

```
myLab instanceof  Dog
myLab instanceof  K9
myLab instanceof INewDog
myLab instanceof IK9
```

The use of **instanceof** if not really recommended. You may be able to improve your design and avoid **instanceof**



Interface:
Advance OOP Series IV

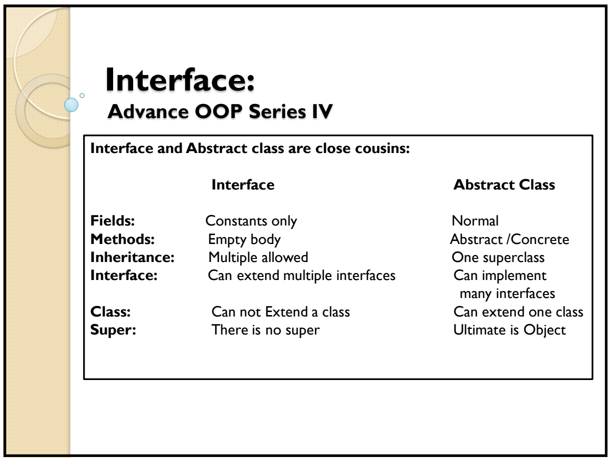
Is really a need for Interface?

- Use for Encapsulation/flexibilities:

- Allows implementation changes without affecting the caller.

- Unrelated classes can implement similar behaviors
  For example: An **Employee** class and **Box** class can implement a **IComparison** interface to compare values.

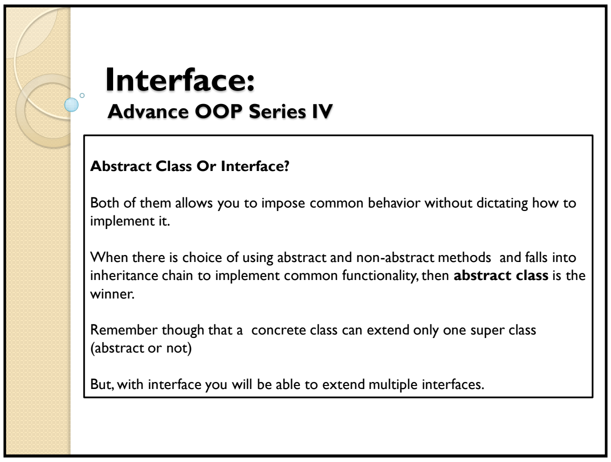
- Allows multiple inheritances:



Interface:
Advance OOP Series IV

Interface and Abstract class are close cousins:

	Interface	**Abstract Class**
Fields:	Constants only	Normal
Methods:	Empty body	Abstract /Concrete
Inheritance:	Multiple allowed	One superclass
Interface:	Can extend multiple interfaces	Can implement many interfaces
Class:	Can not Extend a class	Can extend one class
Super:	There is no super	Ultimate is Object



Interface:
Advance OOP Series IV

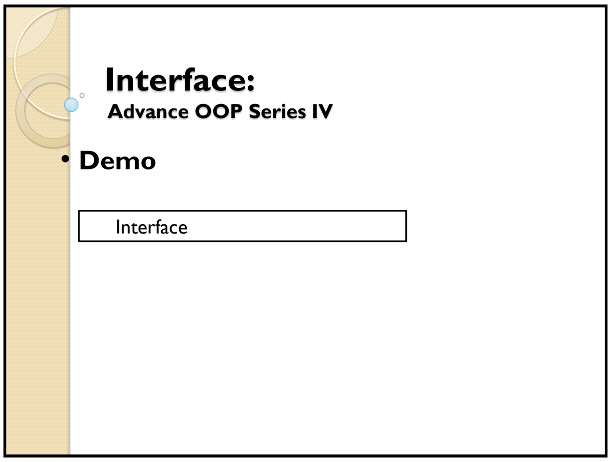
Abstract Class Or Interface?

Both of them allows you to impose common behavior without dictating how to implement it.

When there is choice of using abstract and non-abstract methods and falls into inheritance chain to implement common functionality, then **abstract class** is the winner.

Remember though that a concrete class can extend only one super class (abstract or not)

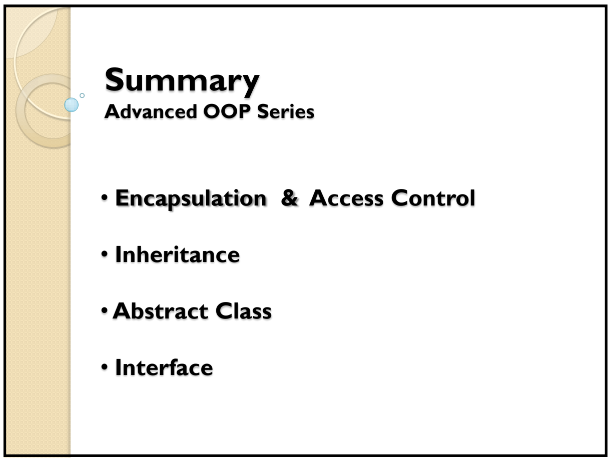
But, with interface you will be able to extend multiple interfaces.



Interface:
Advance OOP Series IV

- **Demo**

Interface



Summary
Advanced OOP Series

- Encapsulation & Access Control
- Inheritance
- Abstract Class
- Interface