

## OOP(Object Oriented Programming) Concept

Created by: **Alan Kay**

First OOP based programming: **Simula**

First famous OOP based programming: **C++**

**Java's OOP comes from C++**

**Java** is more Object Oriented and Secure than C++

**Java** is C++ without the guns, clubs and knives. (by- James Gosling)

# OOP's Concepts



Class

Object

Encapsulation

Abstraction

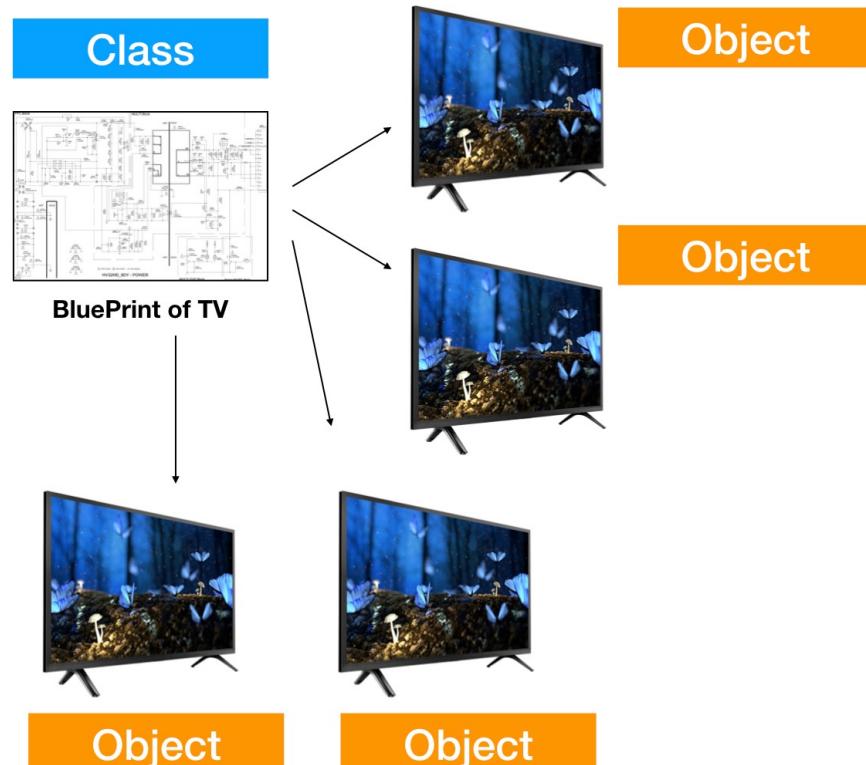
Inheritance

Polymorphism

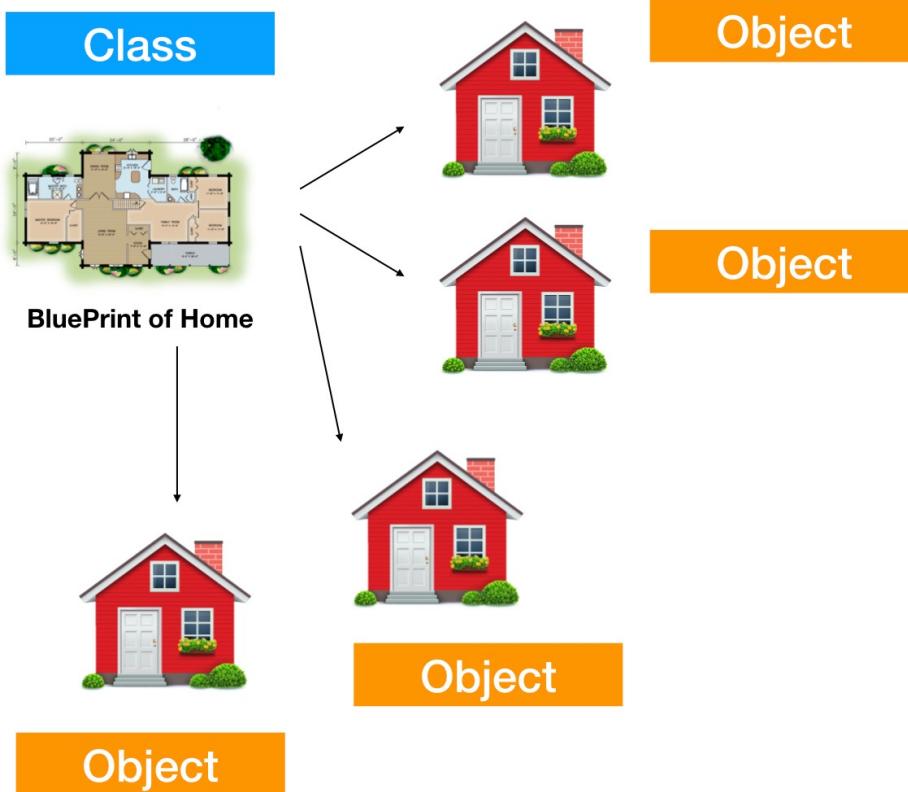
# Class and Object

**Class** is a blueprint of the object.

**Object** is a physical entity.



# Class and Object



# Encapsulation

Binding/Wrapping the component together to make them secure is known as **Encapsulation**.



# Abstraction

Hiding the complex working(Complexity) by providing the functionality is known as **Abstraction**.



# Inheritance

Using the old design to creating new design is known as **Inheritance**.

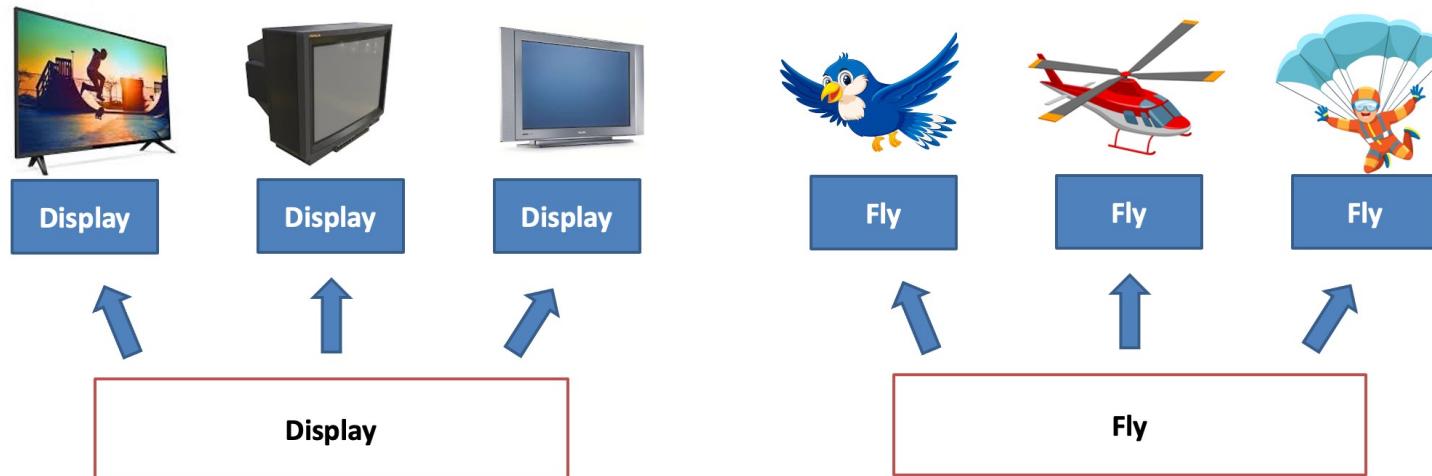


# Polymorphism

Poly means “**Many**” and Morphism means “**Forms**”.

Many forms of doing a job.

Examples- 1) Display has many forms. 2) Fly has many forms.



# Object Anatomy



Class Name/Non-primitive  
DataType

Object/Instance

**Employee** a=**new Employee( );**

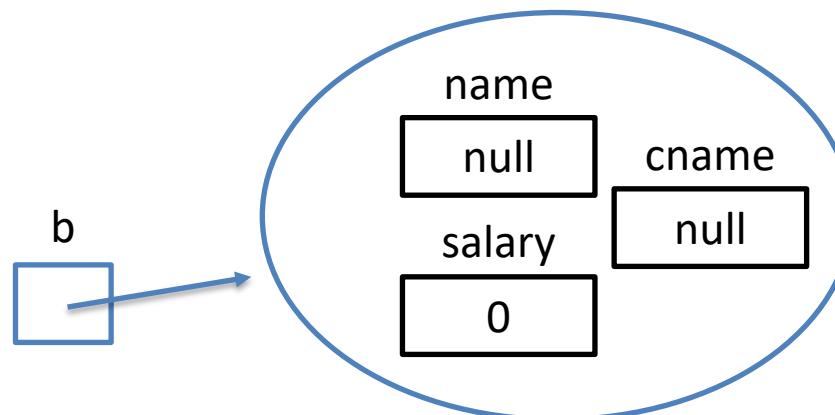
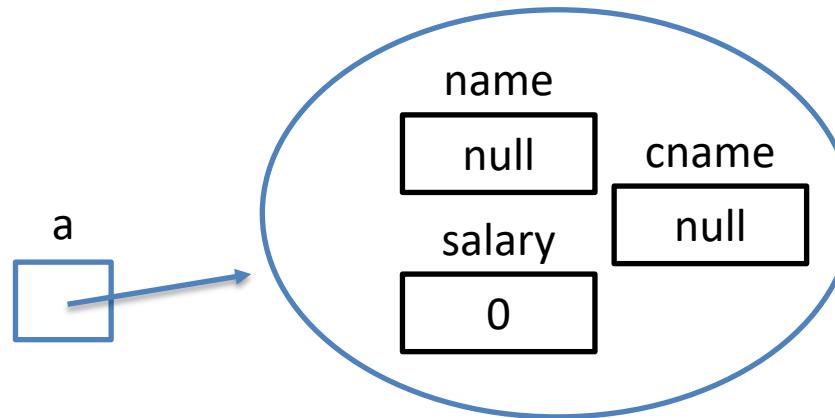
Reference/Non-primitive  
Variable

```
class Employee{  
    String name;  
    int salary;  
    String cname;  
}
```

# Object Diagram

```
class Employee{  
    String name;  
    int salary;  
    String cname;  
}
```

```
Employee a=new Employee();  
Employee b=new Employee();
```

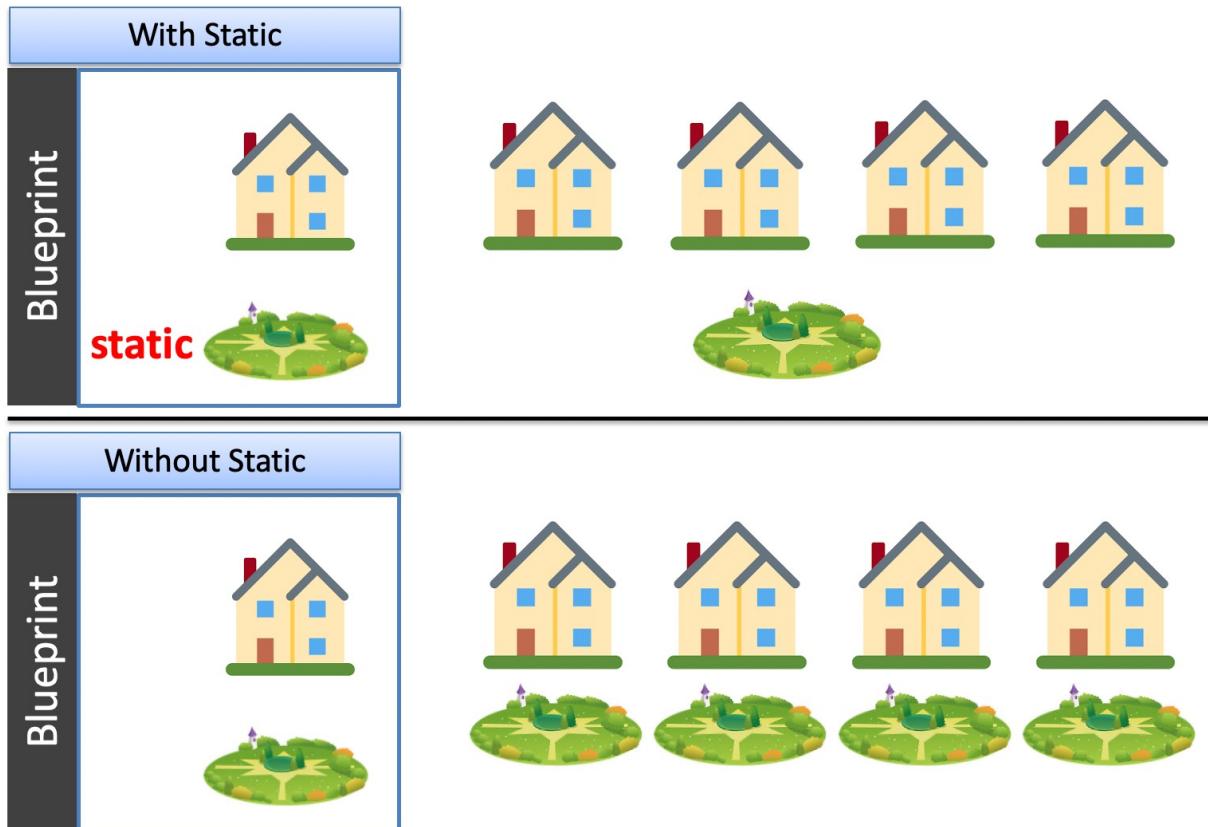


**static** keyword can be used inside the class as:

- Static variable
- Static method
- Static block
- Static nested class

# Static Variable

A common member for every object of that class.



# Static Variable

- Can be accessed via Class-Name and Object reference.
- Allocated when the class loads in memory.

```
class Employee{  
    String name;  
    int salary;  
    static String cname;  
}
```

# Static Method

- Can be accessed via Class-Name and Object reference.
- Allocated when the class loads in memory.
- Can access static member but can not access non-static member of that class.

```
class A{  
    static void methodName(){  
        //body  
    }  
}
```

# Static Block

- Automatically executes when the class loads in memory.
- Only executes once.
- Can access static member but can not access non-static member of that class.
- Mostly used for initializing the static variable.

```
class A{  
    static {  
        //body  
    }  
}
```

# Static vs Non-Static Member

| Static member  | Non-Static member  |
|--|--|
| <ul style="list-style-type: none"><li>• Also known as Class Member.</li><li>• Common for Every Object of the class.</li><li>• Can be accessed via Class name or Object reference.</li><li>• Allocated when class loads in memory.</li><li>• Can be accessed from static and non-static context both.</li></ul> | <ul style="list-style-type: none"><li>• Also known as Instance Member.</li><li>• Individual for Every Object of the class.</li><li>• Can be accessed via Object reference only.</li><li>• Allocated when object created in memory.</li><li>• Can be accessed from a non-static context only.</li></ul> |

# Constructor

**A special method in a class that is executed automatically when an object of the class created.**

```
class A{  
    A(){  
        System.out.println("Hi A");  
    }  
}
```

**A a1=new A();**  
**A a2=new A();**  
**A a3=new A();**

# Constructor vs Method



| Constructor   | Method   |
|---|--|
| <ul style="list-style-type: none"><li>• Special method in a class.</li><li>• Name must be same as class name.</li><li>• Can not declare return type (even, void not allowed).</li><li>• Invoked implicitly with object construction.</li><li>• Mostly used to initialize the object of the class.</li></ul> | <ul style="list-style-type: none"><li>• Normal method in a class.</li><li>• Name is the choice of programmer.</li><li>• Can declare return type or void.</li><li>• Invoked explicitly by programmer's choice.</li><li>• Used to do anything that programmer needs.</li></ul> |

# Types of Constructor



## Non-Parameterized

```
class A{  
    A(){  
        //body  
    }  
}
```

## Parameterized

```
class A{  
    A(int x){  
        //body  
    }  
}
```

# Constructor Overloading



```
class A{  
    A(){  
        //body  
    }  
    A(int x){  
        //body  
    }  
}
```

# Constructor Chaining



```
class A{
    A(){
        //body
    }
    A(int x){
        this();
        //body
    }
}
```

# Constructor Chaining Rules



- Can not chain more than one constructor in a constructor.
- Constructor chaining statement must be the first statement.
- Recursive constructor chaining is not allowed.
- “this” and “super” constructor chaining can not be used together in a constructor.

# Initialize Block

- A **special block** in a class that is executed automatically before constructor when an object of the class created.
- This block having no name that's why also known as **anonymous block** of the class.

```
class A{  
    A(){  
        //body  
    }  
    {  
        //body  
    }  
}
```

# Inheritance

Mechanism of creating new class from old one is called Inheritance.

```
class A{  
    //members  
}
```

Super/Parent/Base class

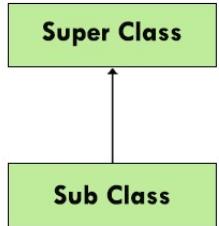


```
class B extends A{  
    //members  
}
```

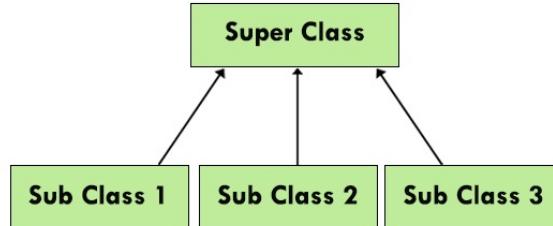
Sub/Child/Derived class

# Types of Inheritance

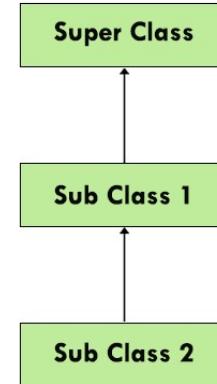
**Single Inheritance**



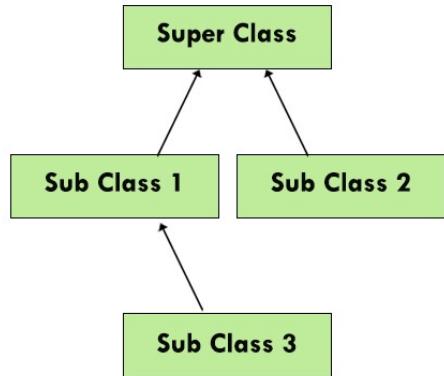
**Hierarchical Inheritance**



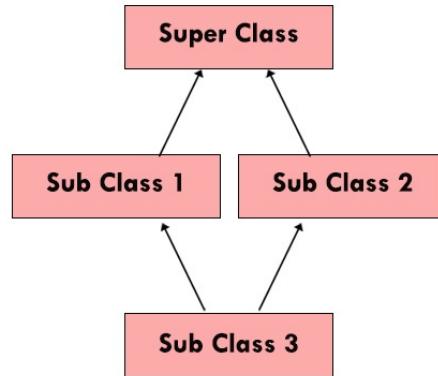
**MultiLevel Inheritance**



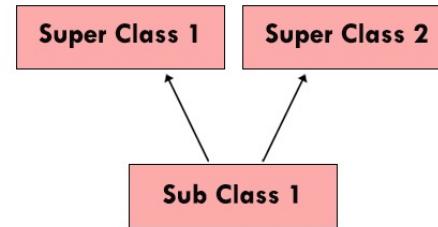
**Hybrid Inheritance 1**



**Hybrid Inheritance 2**



**Multiple Inheritance**



In Java classes **Multiple Inheritance** is **NOT supported**. But in Java Interfaces it is supported.

# Polymorphism

Poly means “**Many**” and Morphism means “**Forms**”.

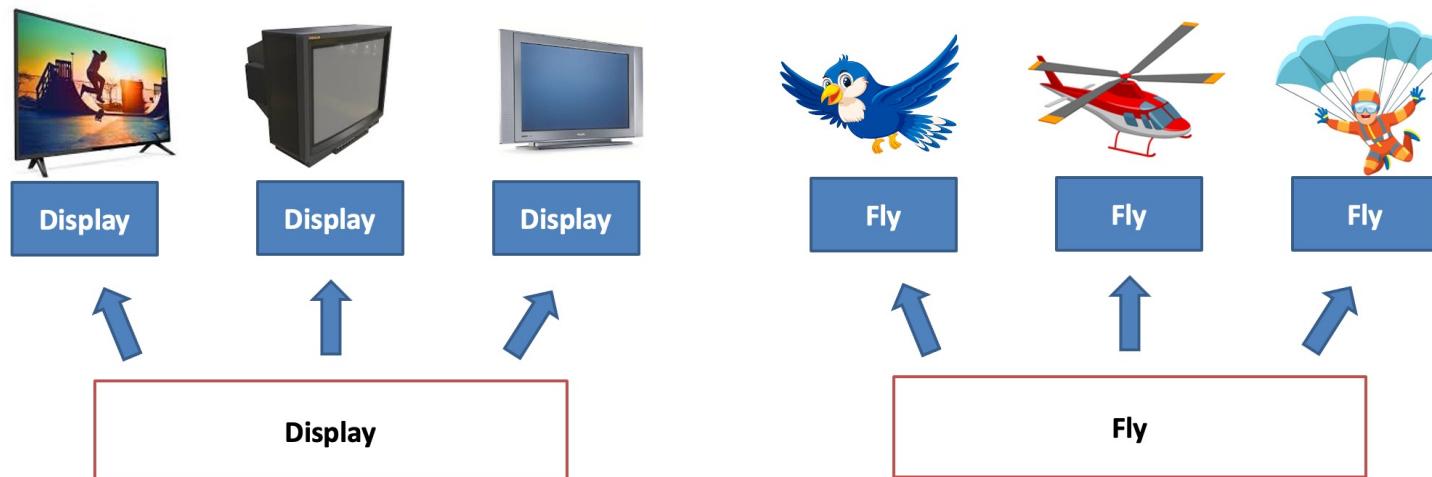
## In Real World:

Many forms of doing a job.

Examples- 1) Display has many forms. 2) Fly has many forms.

## In Programming:

Many methods having same name.



# Types of Polymorphism

## Polymorphism

Compile-Time(Static)  
Polymorphism

Achieved by  
**Method Overloading**

Run-Time(Dynamic)  
Polymorphism

Achieved by  
**Method Overriding**

# Compile-Time Polymorphism



Having more than one method with same name in a class is known as **Compile-Time polymorphism**.

```
class Addition{  
    void add(int x,int y){  
        //body  
    }  
    void add(String x,String y){  
        //body  
    }  
}
```

Compile-Time Polymorphism = Method Overloading

# Method Overloading



Having same name with different argument methods in a class is known as **Method Overloading**.

## Rules for method overloading:

- Method name must be same.
- Method argument must be different.
- Method return type can be same or different.
- Inheritance is optional.
- Method can be static or non-static.

# Run-Time Polymorphism



Having more than one method with same signature in different classes that has inheritance relation, and calling the sub class method from the super-class reference, then we achieve **Run-Time polymorphism**.

Run-Time Polymorphism = Method Overriding + Non-primitive Upcasting

# Method Overriding



Having same signature non-static methods in super and sub classes is known as **Method Overriding**.

## Rules for method overriding:

- Method name must be same.
- Method argument must be same.
- Method return type must be same.
- Inheritance is compulsory.
- Access modifier must be same or strong.
- Method must be **non-static**.

Having same signature static method in super and sub classes is known as **Method Hiding**.

## Rules for method hiding:

- Method name must be same.
- Method argument must be same.
- Method return type must be same.
- Inheritance is compulsory.
- Access modifier must be same or strong.
- Method must be **static**.

# Method Overriding & Hiding Table



| Defining a Method in Sub-class with the Same Signature as in Super-class | Super-class Instance (Non-Static) Method | Super-class Static Method |
|--|--|---------------------------|
| Sub-class Instance (Non-Static) Method                                   | Overrides                                | error                     |
| Sub-class Static Method  | error                                    | Hides                     |

Having same name variable in super and sub classes is known as **Data Hiding**.

## Rules for data hiding:

- **Variable name must be same.**
- **Variable type can be same or different.**
- **Inheritance is compulsory.**
- **Access modifier can be same or different.**
- **Variable can be static or non-static.**

# Type Casting

## Non-Primitive Type-Casting

```
class A{  
    void m1(){ }  
}
```

```
class B extends A{  
    void m2(){ }  
}
```

B x=new B();  
**A y=x; [ upcasting ]**  
~~B z=y ; [ compile error ]~~  
**B z=(B)y; [ downcasting ]**

**A y=new B() ; [ upcasting ]**  
~~B x=new A(); [ compile error ]~~  
**B x=(B)new A() ; [ ClassCastException ]**

## Primitive Type-Casting

```
int x=10;  
double y=x; [ upcasting ]  
int z=y ; [ compile error ]  
int z=(int)y; [ downcasting ]
```

# Without Run-Time Polymorphism



```
public class Circle{  
    private double a;  
    public void area(){  
        a=3.14 X 5 X 5 ;  
    }  
    public void show(){  
        SOP("Circle's Area: "+a);  
    }  
}
```



```
public class Rectangle{  
    private double v;  
    public void m1(){  
        v=10 X 7 ;  
    }  
    public void m2(){  
        SOP("Rectangle's Area: "+v);  
    }  
}
```



```
public class Triangle{  
    private double ar;  
    public void calculate(){  
        a= 15 X 8 / 2;  
    }  
    public void display(){  
        SOP("Triangle's Area: "+ar);  
    }  
}
```

```
public class ShapeApp {  
    public static void main(String [] s){  
        Circle c=new Circle();  
        c.area();  
        c.show();  
        Rectangle r=new Rectangle();  
        r.m1();  
        r.m2();  
        Triangle t=new Triangle();  
        t.calculate();  
        t.display();  
    }  
}
```



# With Run-Time Polymorphism



```
public class Circle extends Shape{  
    private double a;  
    public void findArea(){  
        a=3.14 X 5 X 5 ;  
    }  
    public void printArea(){  
        SOP("Circle's Area: "+a);  
    }  
}
```



```
public class Rectangle extends Shape{  
    private double v;  
    public void findArea(){  
        v=10 X 7 ;  
    }  
    public void printArea(){  
        SOP("Rectangle's Area: "+v);  
    }  
}
```



```
public class Triangle extends Shape{  
    private double ar;  
    public void findArea(){  
        a= 15 X 8 / 2;  
    }  
    public void printArea(){  
        SOP("Triangle's Area: "+ar);  
    }  
}
```

```
public class Shape{  
    public void findArea(){  
    }  
    public void printArea(){  
    }  
}
```

```
public class ShapeApp {  
    public static void main(String [] s){  
        Shape s;  
        s=new Circle();  
        s. findArea();  
        s. printArea();  
        s=new Rectangle();  
        s. findArea();  
        s. printArea();  
        s=new Triangle();  
        s. findArea();  
        s. printArea();  
    }  
}
```



# Abstract Class



```
public class Circle extends Shape{  
    private double a;  
    public void findArea(){  
        a=3.14 X 5 X 5 ;  
    }  
    public void printArea(){  
        SOP("Circle's Area: "+a);  
    }  
}
```



```
public class Rectangle extends Shape{  
    private double v;  
    public void findArea(){  
        v=10 X 7 ;  
    }  
    public void printArea(){  
        SOP("Rectangle's Area: "+v);  
    }  
}
```



```
public class Triangle extends Shape{  
    private double ar;  
    public void findArea(){  
        a= 15 X 8 / 2;  
    }  
    public void printArea(){  
        SOP("Triangle's Area: "+ar);  
    }  
}
```

```
abstract public class Shape{  
    public void findArea(){  
    }  
    public void printArea(){  
    }  
}
```

```
public class ShapeApp {  
    public static void main(String [] s){  
        Shape s= new Shape();  
        s=new Circle();  
        s. findArea();  
        s. printArea();  
        s=new Rectangle();  
        s. findArea();  
        s. printArea();  
        s=new Triangle();  
        s. findArea();  
        s. printArea();  
    }  
}
```



# Abstract Class



- A class that can not be instantiated.
- A class that is used to access its sub-classes only.
- A class that is used to achieve Run-time polymorphism.

# Abstract Method



```
public class Circle extends Shape{  
    private double a;  
    public void findArea(){  
        a=3.14 X 5 X 5 ;  
    }  
    public void printArea(){  
        SOP("Circle's Area: "+a);  
    }  
}
```

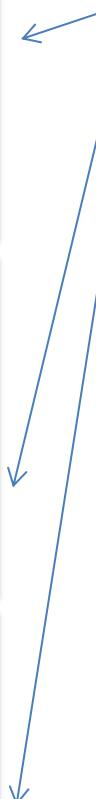


```
public class Rectangle extends Shape{  
    private double v;  
    public void findArea(){  
        v=10 X 7 ;  
    }  
    public void printArea(){  
        SOP("Rectangle's Area: "+v);  
    }  
}
```



```
public class Triangle extends Shape{  
    private double ar;  
    public void findArea(){  
        a= 15 X 8 / 2;  
    }  
    public void printArea(){  
        SOP("Triangle's Area: "+ar);  
    }  
}
```

```
abstract public class Shape{  
    abstract public void findArea();  
    abstract public void printArea();  
}
```



```
public class ShapeApp {  
    public static void main(String [] s){  
        Shape s= new Shape();  
        s=new Circle();  
        s. findArea();  
        s. printArea();  
        s=new Rectangle();  
        s. findArea();  
        s. printArea();  
        s=new Triangle();  
        s. findArea();  
        s. printArea();  
    }  
}
```



# Abstract Method



- A method without body.
- A method that must be overridden in its non-abstract sub-class.

# Interface



```
public class A {  
    public void show(){  
        SOP("Hello A");  
    }  
}
```

```
abstract public interface Shape{  
    abstract public void findArea();  
    abstract public void printArea();  
}
```



```
public class Rectangle extends A implements Shape{  
    private double v;  
    public void findArea(){  
        v=10 X 7;  
    }  
    public void printArea(){  
        SOP("Rectangle's Area: "+v);  
        show();  
    }  
}
```



```
public class Triangle implements Shape{  
    private double ar;  
    public void findArea(){  
        a= 15 X 8 / 2;  
    }  
    public void printArea(){  
        SOP("Triangle's Area: "+ar);  
    }  
}
```



```
public class Test{  
    PSVM(String [] s){  
        Shape s= new Shape();  
        s=new Circle();  
        s. findArea();  
        s. printArea();  
        s=new Rectangle();  
        s. findArea();  
        s. printArea();  
        s=new Triangle();  
        s. findArea();  
        s. printArea();  
    }  
}
```

## Interface

- It is 100% abstract class by default.
- Supports multiple inheritance.
- Does not support constructor.
- Method in interface is by default abstract and public.
- Variable in interface is by default public, static and final.
- To inherit an interface in class, we use **implements** keyword and To inherit an interface in interface, we use **extends** keyword.
- From Java-8, **default method** and **static method** are allowed in the interface.
- From Java-9, **private** method is also allowed in the interface.

## Abstract class

- It can be 100% abstract or not.
- Does not supports multiple inheritance.
- Supports constructor.
- Method in abstract class is not by default abstract and public.
- Variable in abstract class is not by default public, static and final.
- To inherit an abstract class, we use **extends** keyword.

# Final keyword

- **Final class** (A class that can not be Inherited)
- **Final method** (A method that can not be overridden or hidden)
- **Final variable** (A variable, who's value can not be changed)

final class

```
final class A{  
    //members  
}
```

final method

```
class A{  
    final void method(){  
        //body  
    }  
}
```

final variable

```
class A{  
    final int x=10;  
    void method(){  
        final int z=20;  
    }  
}
```

# Blank Final Variable

Allowed

```
class Test{  
    final int a;  
    Test(){  
        a=10;  
    }  
}
```

```
class Test{  
    final int a;  
    {  
        a=10;  
    }  
}
```

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

Not Allowed

```
class Test{  
    final int a;  
    Test(){  
        a=10;  
    }  
    {  
        a=10;  
    }  
}
```

```
class Test{  
    static final int a;  
    Test(){  
        a=10;  
    }  
}
```

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

- **Non-static nested class**
- **Static nested class**
- **Local nested class**
- **Anonymous nested class**

# Non-Static Nested Class



```
class A{
    void m1(){
        B b=new B();
        SOP(b.z);
        b.m();
    }
    class B{
        int z;
        void m(){
            SOP("hi B");
        }
    }
}
```

```
class Demo{
    PSVM(-){
        A a=new A();
        A.B b=a.new B();
        SOP(b.z);
        b.m();
    }
}
```

# Static Nested Class

```
class A{  
    void m1(){  
        B b=new B();  
        SOP(b.z);  
        b.m();  
    }  
    static class B{  
        int z;  
        void m(){  
            SOP("hi B");  
        }  
    }  
}
```

```
class Demo{  
    PSVM(-){  
        A.B b=new A.B();  
        SOP(b.z);  
        b.m();  
    }  
}
```

# Local Nested Class

```
class A{
    void m1(){
        class B{
            int z;
            void m(){
                SOP("hi B");
            }
        }
        B b=new B();
        SOP(b.z);
        b.m();
    }
    void m2(){
        // B b=new B(); //error
    }
}
```

```
class Demo{
    PSVM(-){
        A a=new A();
        a.m1();
    }
}
```

# Anonymous Nested Class



```
class A{  
    void m1(){  
        SOP( "Hi A");  
    }  
}
```

```
class Demo{  
    PSVM(-){  
        A a=new A() {  
            void m1(){  
                SOP( "Hi INCAPP");  
            }  
        };  
        a.m1();  
    }  
}
```

# Lambda



- **Introduced in Java 8**

# Package



- Used to avoid the naming confliction of byte codes in a project
- Used to categorized the byte codes.

## Most Accessible



**public**: accessible all over the enclosing project.

**protected**: accessible inside the enclosing package and inside the sub-class of any package of the enclosing project.

**Default(package private)**: accessible only inside the enclosing package .

**private**: accessible inside the enclosing class.

## Less Accessible

# protected keyword

Protected member is accessible inside the enclosing package and inside the sub-class of any package of the enclosing project.

package p2

```
import p1.A;
public class B{
    public void m(){
        A a= new A();
        SOP(a.x); //error
    }
}
```

```
import p1.A;
public class C extends A{
    public void m(){
        SOP(x);
    }
}
```

package p1

```
public class A{
    protected int x=10;
}
```

```
public class B{
    public void m(){
        A a= new A();
        SOP(a.x);
    }
}
```

```
public class C extends A{
    public void m(){
        SOP(x);
    }
}
```

# Sub-Packages

Package inside the package known as Sub-Package or Nested Package.

package p3

```
import p2.B; // Error
import p1.p2.B;
public class D{
    public void m(){
        B b=new B();
        SOP( b.y );
    }
}
```

package p1

```
import p2.B; // Error
import p1.p2.B;
public class A{
    public void m(){
        B b=new B();
        SOP( b.y );
    }
}
```

package p1.p2

```
public class B{
    public int y=20;
}
```

# import static

- **import static** is used to directly import static-member of the class.
- Introduced in java 5.0

package p2

```
import static p1.A.y;
import static p1.A.m2;
public class D{
    PSVM(-){
        m2();
        SOP( y );
    }
}
```

package p1

```
public class A{
    public int x=10;
    static public int y=20;
    public void m1(){
        //body
    }
    static public void m2(){
        //body
    }
}
```

# Rules for class



- Only one class can be public in a java program.
- If a class is public in a java program then Java program name must be same as name of that public class.