OOP Concept

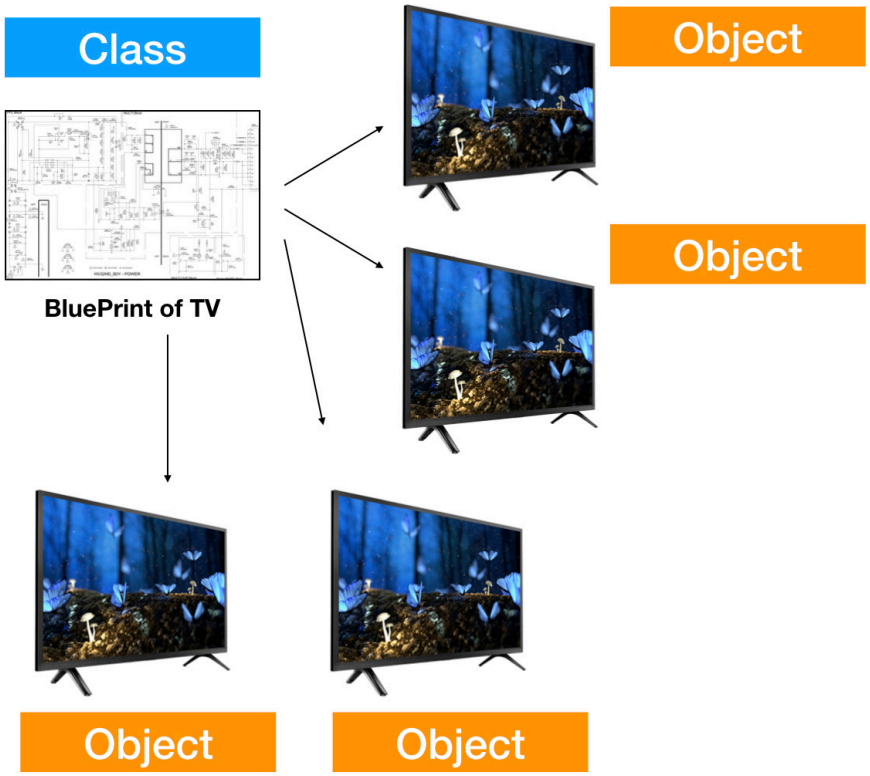
OOP(Object Oriented Programming) Concept
Created by: Alan Kay
First OOP based programming: Simula
First famous OOP based programming: C++
Java's OOP comes from C++
Java is more Object Oriented and Secure than C++
Java is C++ without the guns, clubs and knives. (by- James Gosling)

OOP's Concepts

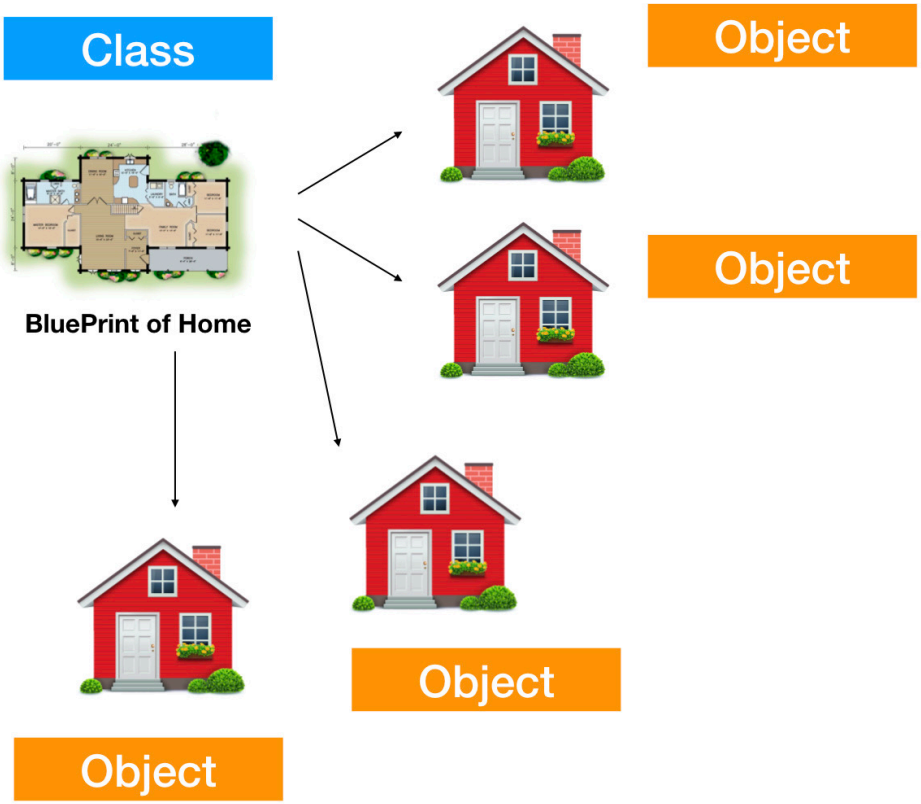
- Class
- Object
- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Class and Object

Class is a blueprint of the object.
Object is a physical entity.

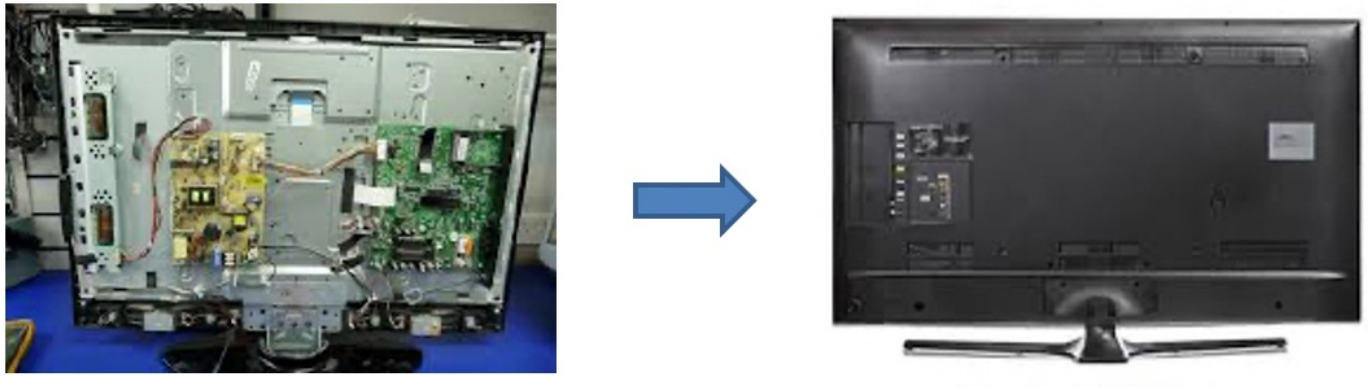


Class and Object



Encapsulation

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



Abstraction

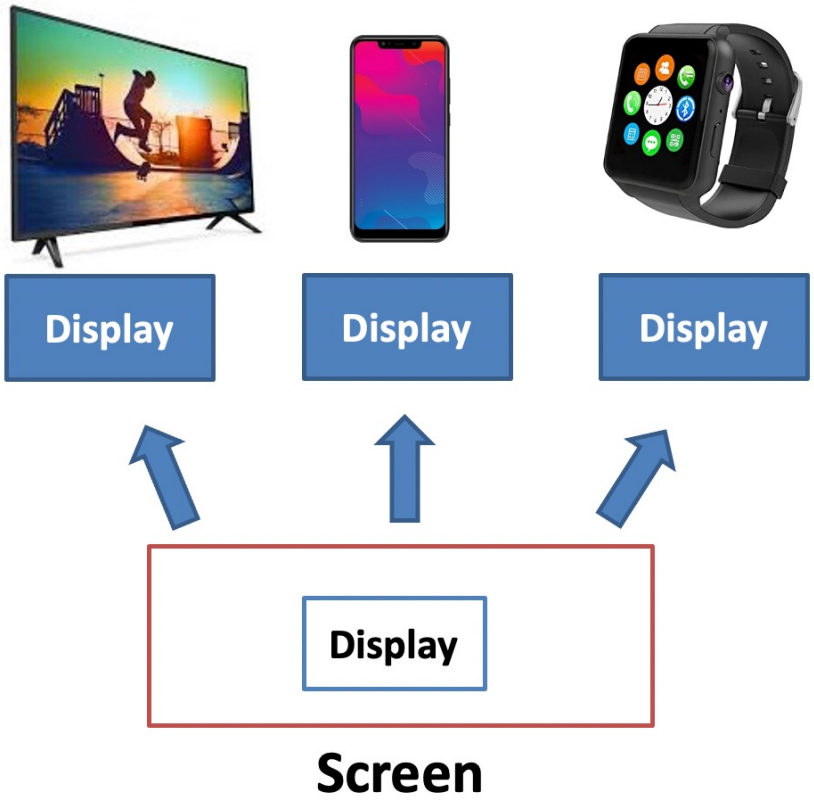
Hiding the complex working(Complexity) by providing the functionality is known as **Abstraction.**

Inheritance

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

Polymorphism

Having multiple functionalities with same name but different way of working is known as **Polymorphism.**



Object Anatomy


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

Static Keyword

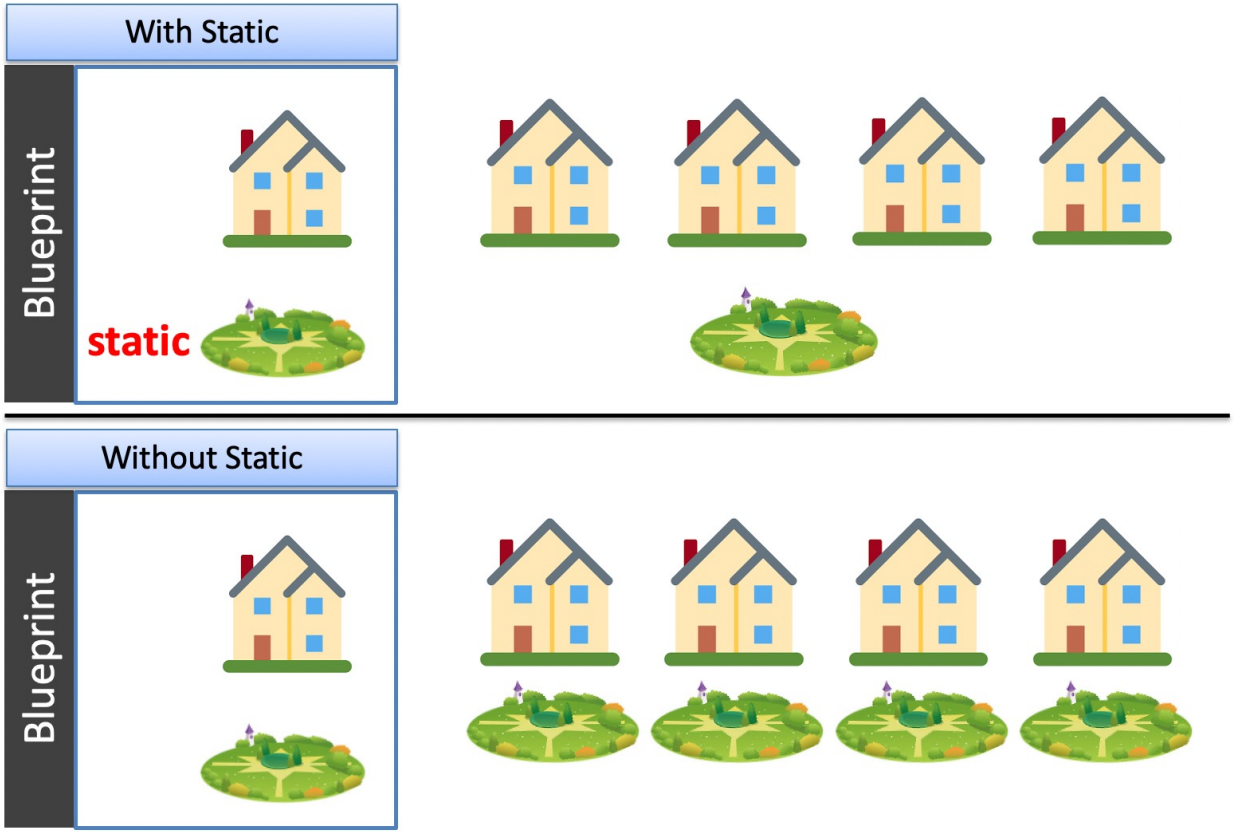
static keyword can be used inside the class as:

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

Static Variable

incapp

A common member for every object of a class.



Static Variable

- **Can be accessed via Class-Name and Reference-name.**
- **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 Reference-name.**
- **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

incapp

Static member	Non-Static member
• Also known as Class Member. • Common for Every Object of the class. • Can be accessed via Class name and Reference variable-name /instance. • Allocated when class loads in memory. • Can be accessed from static and non-static context both.	• Also known as Instance Member. • Individual for Every Object of the class. • Can be accessed via Reference variable-name /instance only. • Allocated when object created in memory. • Can be accessed from a non-static context only.

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

incapp

Constructor	Method
• Special method in a class. • Name must be same as class name. • Can not declare return type (even, void not allowed). • Invoked implicitly with object construction. • Mostly used to initialize the object of the class.	• Normal method in a class. • Name is the choice of programmer. • Can declare return type or void. • Invoked explicitly by programmer's choice. • Used to do anything that programmer needs.

Types of Constructor

incapp

Non-Parameterized	Parameterized

class **A**{
A(){
//body
}
}
	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.**

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

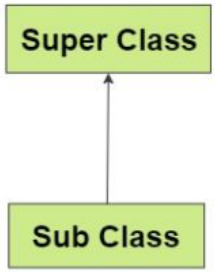
```
class B extends A{
        //members
}
```

Sub/Child/Derived class

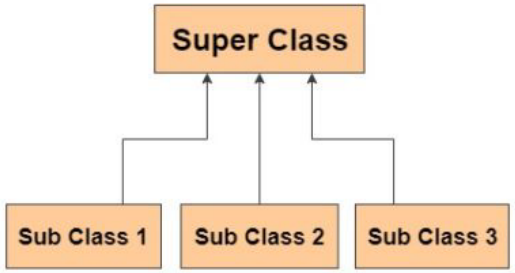
Types of Inheritance

incapp

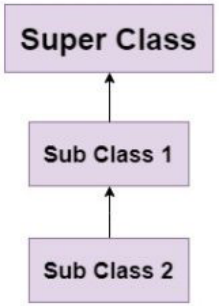
Single Inheritance



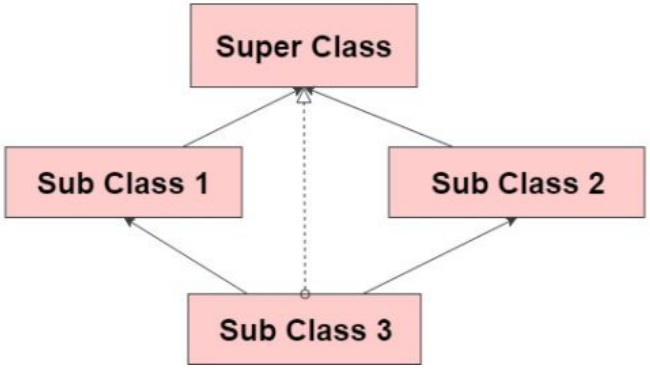
Hierarchial Inheritance



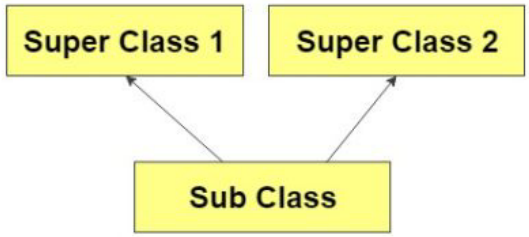
MultiLevel Inheritance



Hybrid Inheritance

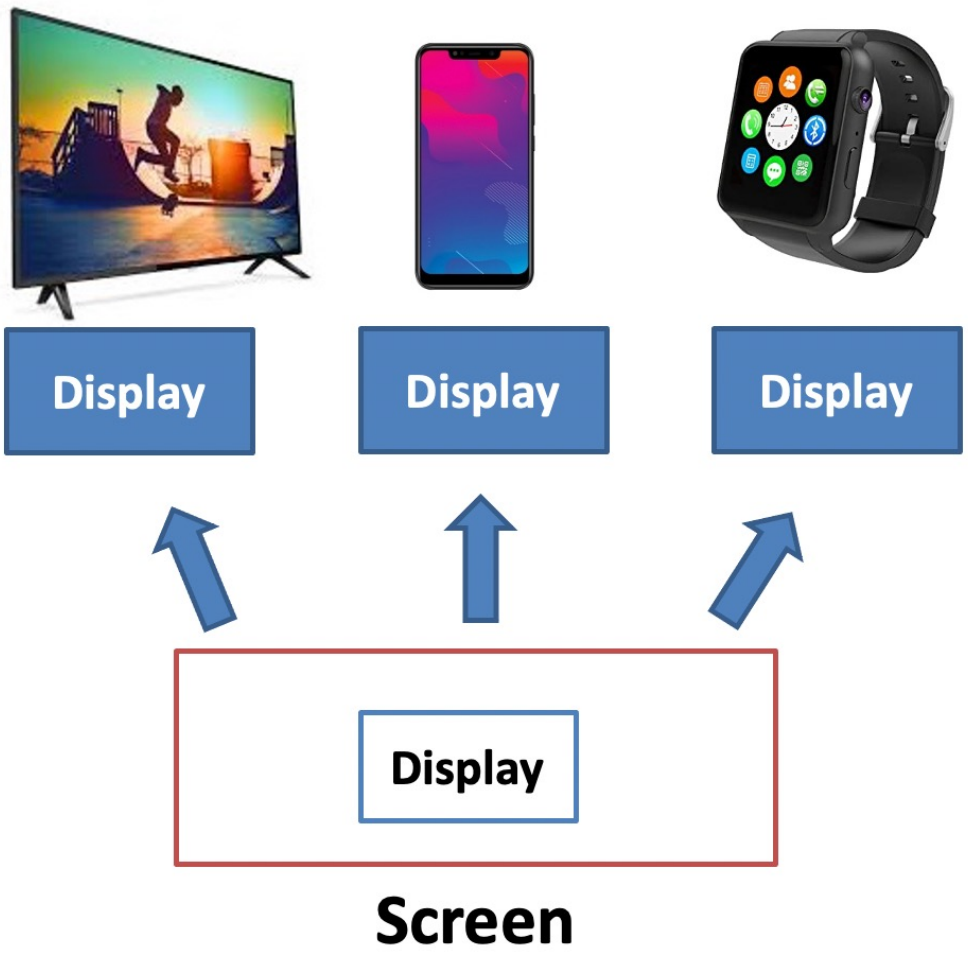


Multiple Inhertance

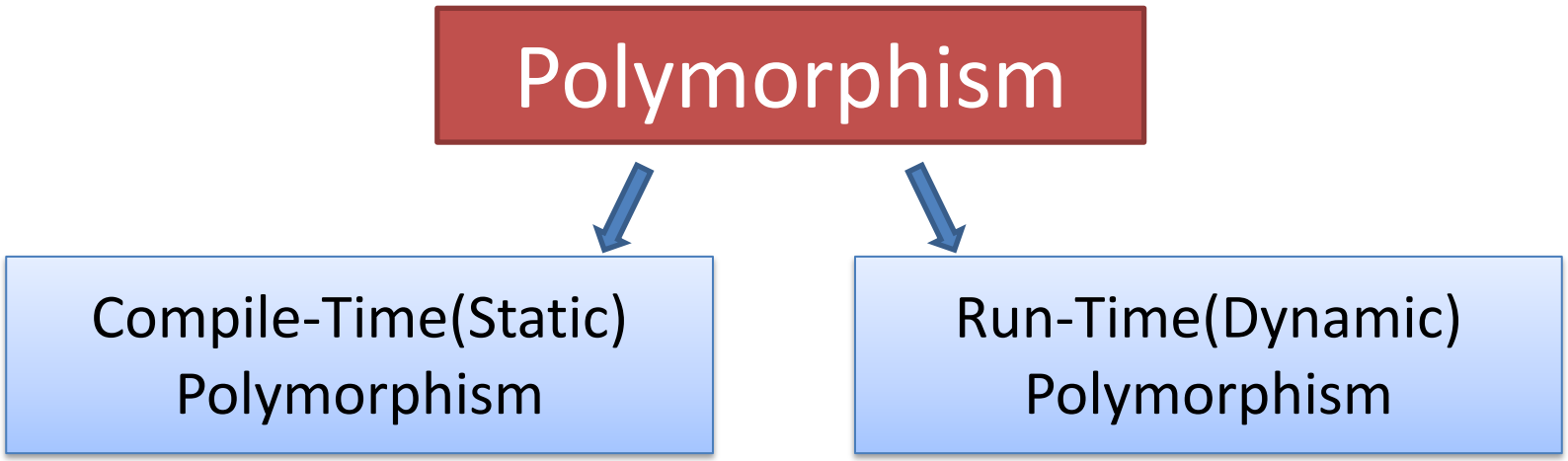


In Java classes Multiple Inheritance is NOT supported.

Polymorphism



Types of Polymorphism



Rules of 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.**

Compile-Time Polymorphism

- **Compile-Time Polymorphism = Method Overloading.**
- **Having multiple methods 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
        }
}
```

Rules of Method Overriding

incapp

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

Rules of Method Hiding

incapp

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

Type Casting

incapp

Non-Primitive TypeCasting

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

```
class Sub extends Super{
    void m2(){ }
}
```

```
Sub x=new Sub();
Super y=x; [ upcasting ]
Sub z=y ; [ compile error ]
Sub z=(Sub)y; [ downcasting ]

Super y=new Sub() ; [ upcasting ]
Super y=(Super)new Sub(); [ upcasting ]
Sub x=new Super() ; [ compile error ]
Sub x=(Sub)new Super() ; [ ClassCastException ]
```

Primitive TypeCasting

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

Run-Time Polymorphism

incapp

- **Run-Time Polymorphism = Method Overriding + Non-primitive Upcasting**
- **Having multiple methods with same signature in different classes and calling them from the super-class reference is known as run-time polymorphism.**

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 / 4;
   }
 public void display(){
   SOP("Triangle's Area: "+ar) ;
   }
}
```

```
public class Test{
   PSVM(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 Test{
   PSVM(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 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();
   }
}
```

Abstract Class

incapp

- **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 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();
   }
}
```

Abstract Method

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

Interface

incapp

```
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 vs Abstract Class

incapp

Interface	Abstract class
• 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 . • From Java-9, **private** method is also allowed in the interface.	• 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)
- **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

incapp

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;
    }
}
```

Nested Class

incapp

- **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();
        }
}
```

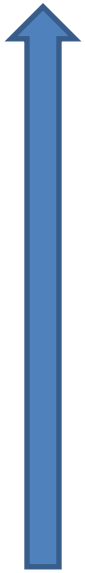
Package

- **Used to avoid the naming confliction of classes in a project**
- **Used to categorized the classes.**

Access Modifiers

incapp

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.

package **p2**

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

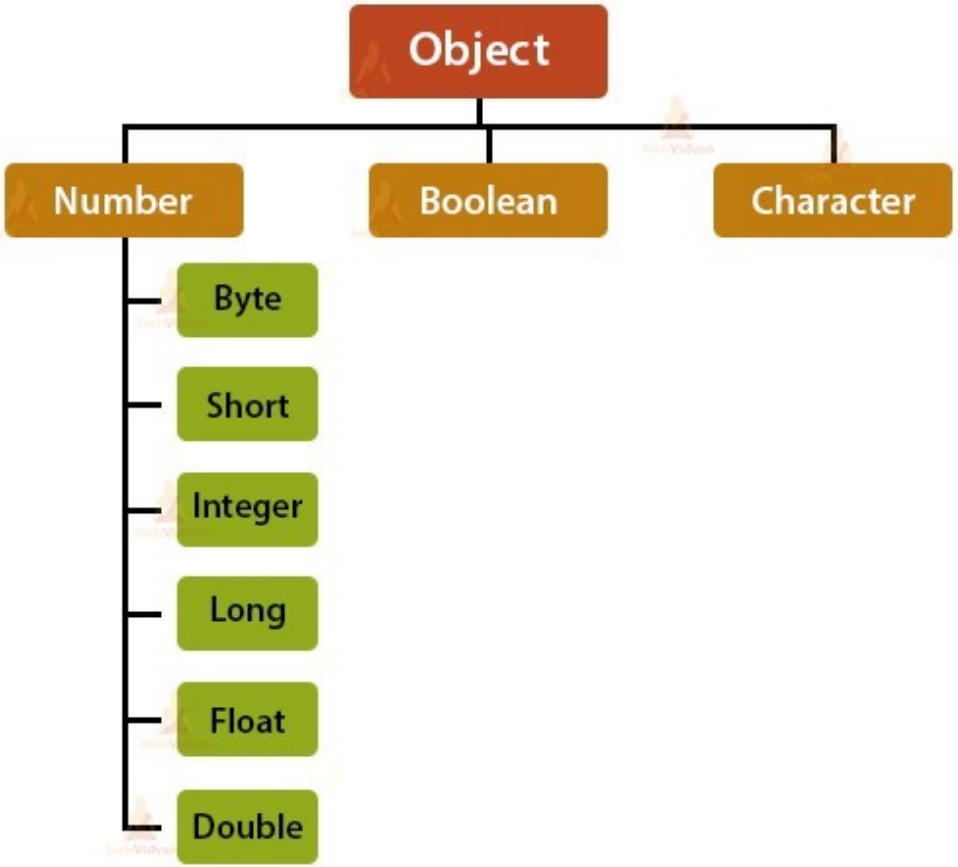
package **p1**

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

}
```

Wrapper Classes

Wrapper classes used to use primitive data types (int, double, char, etc..) as objects.



Boxing/Unboxing/Autoboxing

- **Boxing:**

  We **Box** or **Wrap** an '**int**' value in an Integer object.

```
Integer  obj=new Integer(3);
```

- **Unboxing:**

  We **Unbox** or **Unwrap** an '**int**' value from an Integer object.

```
int  a= obj.intValue( );
```

- Java does **autoboxing/unboxing** as necessary, so we do not have to explicitly do it in our code.

```
Integer  obj=3 ;
```

```
int  a= obj ;
```

Parsing

incapp

```
String x="25";
int y=Integer.parseInt(x);
double z=Double.parseDouble(x);
System.out.println(y);
System.out.println(z);
```