

Chapter 7:

Concept of Inheritance

Informatics Practices
Class XII

By- Rajesh Kumar Mishra

PGT (Comp.Sc.)

KV No.1, AFS, Suratgarh

e-mail : rkmalld@gmail.com

Objective

In this presentation, you will learn about the following-

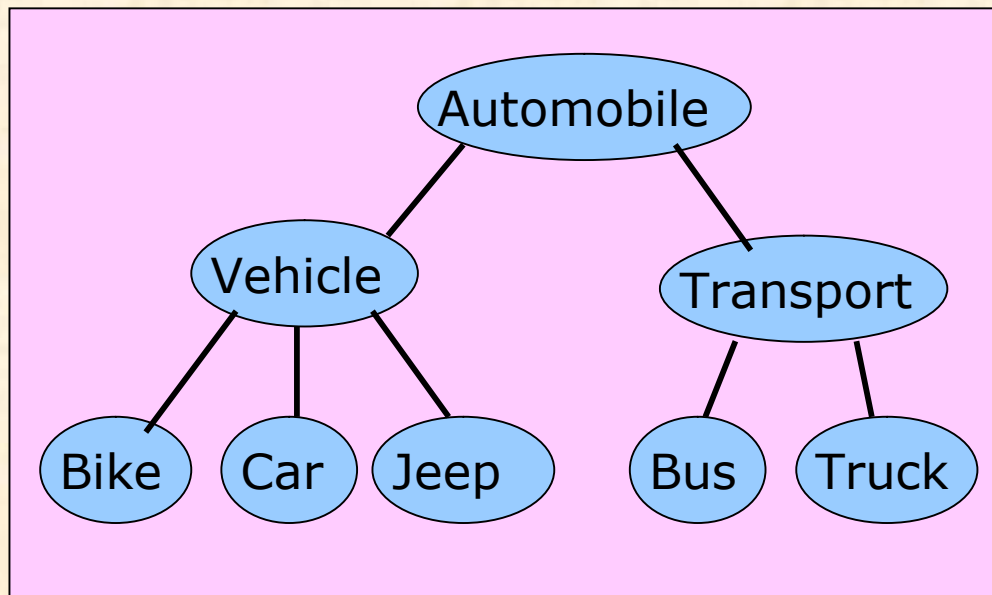
- ❑ **Inheritance:**
Concept & implementation.
 - ❑ **Derived Classes:**
How to derive a new class from existing ones.
 - ❑ **Function Overloading:**
How to overload a function or method.
 - ❑ **Inheritance & Constructors:**
How to call a constructor method of super class in derived class.
 - ❑ **Abstract Class:**
Concept and use Abstract classes and Abstract methods.
 - ❑ **Interfaces:**
How to design an Interfaces.
-

What is Inheritance ?

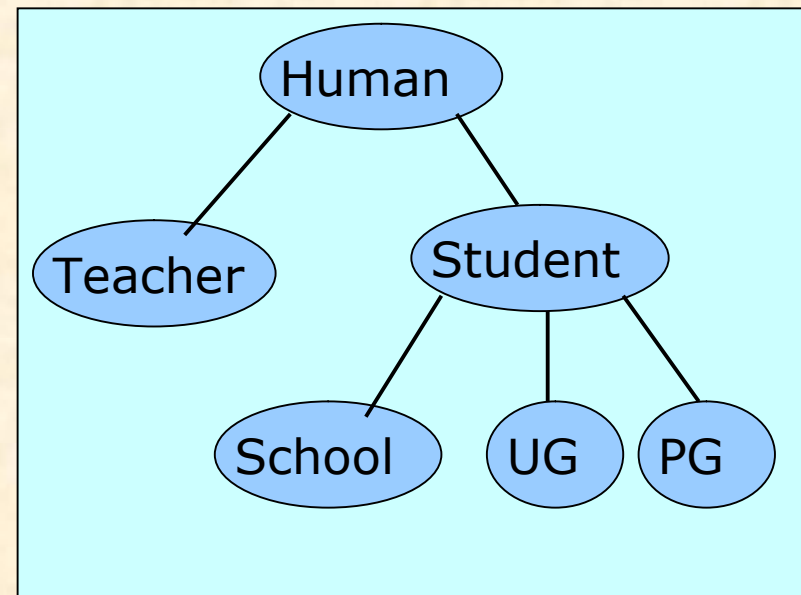
- ❑ Inheritance is one of the most important and powerful feature of OOP.
 - ❑ It is the process of creating new classes called Derived classes (sub-classes) from existing ones (Super class).
 - ❑ A Sub-classes inherit all the properties (data members) and behavior (method members) of Super-class.
 - ❑ The level of Inheritance may be extended i.e. A Sub-class may have their own sub-classes.
 - ❑ Inheritance Hierarchy Graph is used to Inheritance relationship in graphical way.
 - ❑ A Sub-class can not alter the behavior of the super-class.
-

Inheritance in Real life

In Real-life most of the things exhibit Inheritance relationship.



Example 1



Example 2

Why Inheritance?

❑ **Modeling of Real-world:**

By Inheritance, we can model the real-world inheritance relationships in easy way.

❑ **Reusability of codes:**

Inheritance allows the derivation of a new class from existing classes. We can add new features to derived class to make a new one with minimal efforts. A derived class extends the code of base class.

❑ **Transitive nature of Inheritance:**

If we modify the base class then the changes automatically inherit into derived classes. Since inheritance is transitive in nature. This offers faster and efficient way to maintain the large program.

How to derive a sub-class ?

In JAVA a new class (Sub-class) can be derived from an existing class (Super-class) by using **extend** keyword. A derived class may inherit all the data and method members from its parent.

E.g. If **human** class is defined, we can derive **student** class by inheriting all the members of human class, since students are human beings.

```
//e.g. derivation of sub class//
class human
{ String name;
  int age;
  void method1()
  {.....}
}
class student extends human
{int rollno;
  .....
  void method2()
  {.....}
}
```

A sub-class may be derived in the same package or different package also. The access of members of super class depends upon the Access Specifiers.

Student class has name, age, rollno (data) and method1(), method2() as method members

Types of Inheritance

In OOP approach, the Inheritance are many types.

- ❑ **Single Level Inheritance :**

When a sub-class inherits only one base-class.

- ❑ **Multi-level Inheritance:**

When a sub-class is derived from sub-class of another base-class and so on.

- ❑ **Multiple Inheritance:**

When a sub-class inherit from multiple base-classes.

- ❑ **Hierarchical Inheritance:**

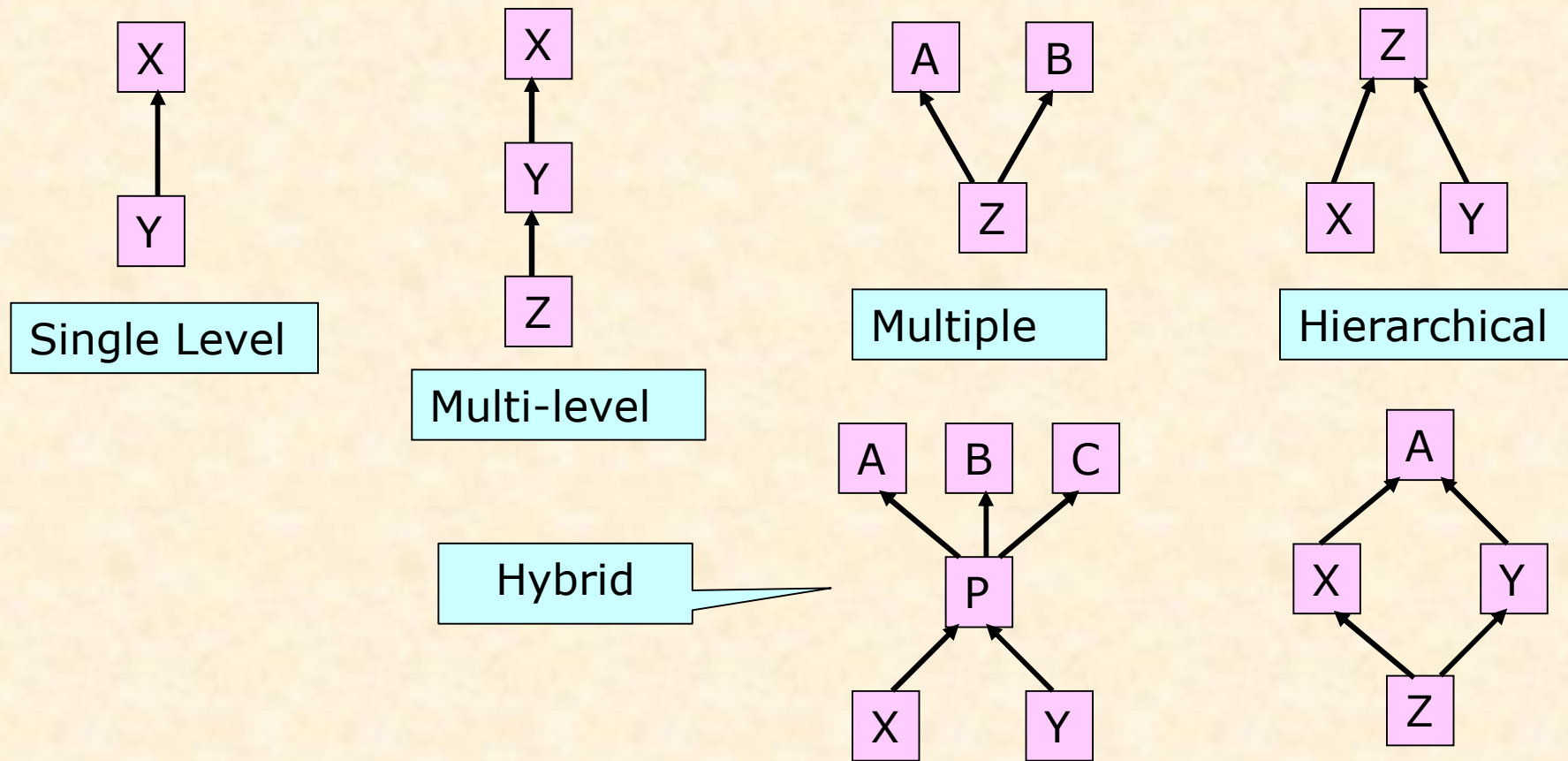
When many sub-classes inherits from a single base class.

- ❑ **Hybrid Inheritance:**

When a sub-class inherits from multiple base-classes and all its base-classes inherits one or more super-classes.

Types of Inheritance

The Inheritance Graph may be used to represent various types of Inheritance in pictorial form.



Access control of Inherited Members

The members (Data and Method) may be defined as Private, Public, Protected, Default or Private protected Access specifiers, which may limit its accessibility in the derived classes.

- ❑ **Private :**

Members are accessible only inside their own class and no where else.

- ❑ **Protected:**

Members are accessible inside their own classes as well as in all Sub-classes in the same or different package.

- ❑ **Public:**

Members are accessible in all the classes whether a sub-class or class in the same or different package.

- ❑ **Package (default):**

Members without any specifier assumed package level scope i.e accessible only inside the package, but not other package even for subclasses.

- ❑ **Private Protected:**

Members are accessible only from sub-classes, whether in same or any other package.

Access control of class members

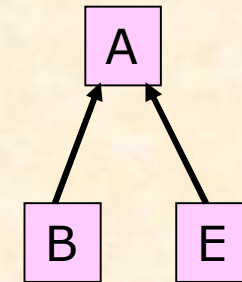
Type	Inside own class	Inside Sub-Class		Outside non-Subclass	
		Same package	Other package	Same package	Other package
Public	✓	✓	✓	✓	✓
Protected	✓	✓	✓	✓	x
Default	✓	✓	x	✓	x
Private Protected	✓	✓	✓	x	x
Private	✓	x	x	x	x



Access specifiers are applicable to both the Data and Method members either Instance or class member (Static).

Example of Access of class members

Suppose a classes B and E are derived from A Base-class.



Package One

class A

```
int i;  
private int j;  
protected int k;  
public int l;  
private protected int m;
```

class B extends A

i	k	l	m	✓
j				x

Class C

i	k	l		✓
j	m			x

Package Two

class D


l				✓
i	j	k	m	x

class E extends A

k	l	m		✓
i	j			x

Class F

l				✓
i	j	k	m	x



Only **public**, **protected** and **private protected** members are accessible in sub-classes of other package

Overriding methods & Hiding Member variables

Some times a subclass uses the same name for variables (data members) and methods as in the super-class. This leads two behaviour.

❑ Overridden Methods:

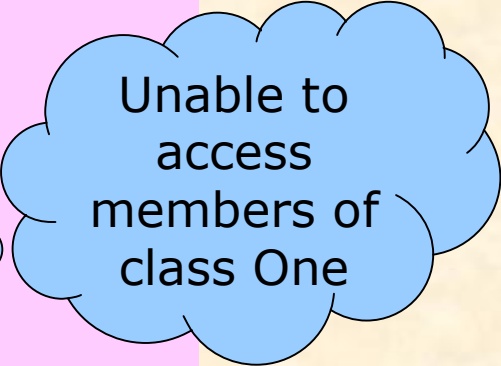
A method in a subclass hides or overshadows a method of super class, if both method have the **same name & signature** i.e. having same name, number and types of argument, and return types. This is called overriding the inherited method.

❑ Hidden Variables:


A variable in a subclass hides or overshadows a variable (data member) of super class, if both have the **same name and data types**. This is called hidden variables.

Ex: Overridden method and hidden variable:

```
class One
{ int x=10;
  void display()
  { system.out.println("I am Class One");
  }
}
class Two extends One
{ int x=15;
  void display()
  { system.out.println("I am Class Two");
  }
  void test()
  { system.out.println("value of x="+x);
    display();
  }
}
```



Unable to
access
members of
class One

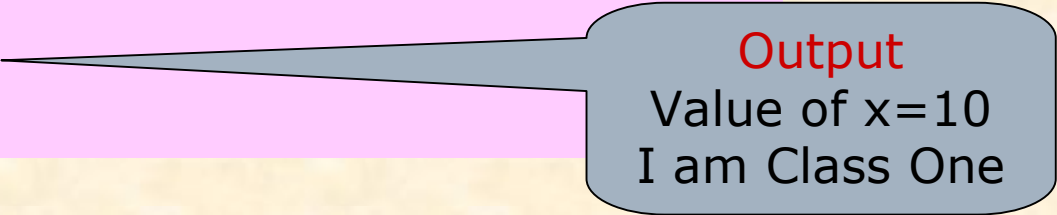


Output
Value of x=15
I am Class Two

Accessing Overridden method and hidden Variable:

We can still refer the Overriding methods and hidden variables of super-class by using **super** keyword.

```
class One
{ int x=10;
  void display()
  { system.out.println("I am Class One");
  }
}
class Two extends One
{ int x=15;
  void display()
  { system.out.println("I am Class Two");
  }
  void test()
  { system.out.println("value of x="+super.x);
    super.display();
  }
}
```



Output
Value of x=10
I am Class One

Role of final keyword in Inheritance

The final keyword can be used with variable, methods and class names. The effect of final keywords is as follows.

- ❑ final variables works as constant i.e. final variables can't be changed.
 - ❑ final methods can't be overridden by sub-class.
 - ❑ final class can't be extended.
-

Inheritance and Constructors

As you may aware that a class may have parameterized or non-parameterized constructor method (must have the same name as class name) to initialize the object.

- ❑ When a super class is extended in a sub-class, constructors are not inherited by the sub-classes automatically.
 - ❑ You may call the constructor of super class explicitly by using **super** keyword.
 - ❑ When a constructor of super class is called by super keyword, it must be the **first statement** in the constructor of the sub class.
-

Accessing constructor of Super class in sub-class

We can still refer the constructor method of super class using **super** keyword.

```
class human
{ String name;
  int age;
  human( String a, int b)
  { name=a;
    age=b;
  }
}
class student extends human
{ int roll;
  student( String p, int q, int r )
  { super (p, q);
    roll=r;
  }
  void display()
  { system.out.println("Name: "+name)
    system.out.println("AgeI:"+age)
    system.out.println("Roll No:"+roll)
  }
}
```

If the following code are executed, then **st** object is initialized and values are displayed.....

```
student st=new student("Amit",20,10);
st.display();
```

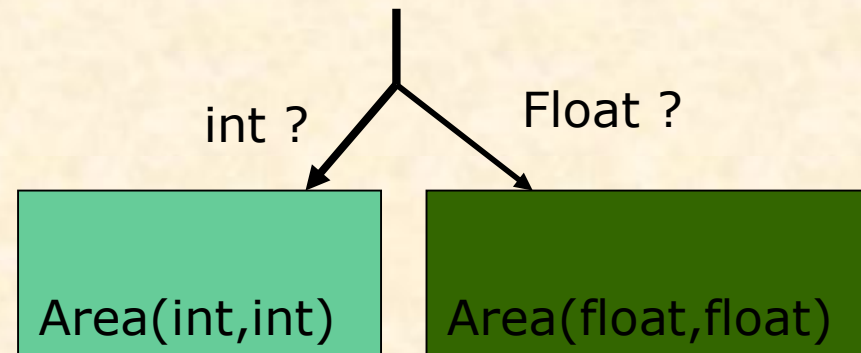
Explicitly call of constructor of super-class by **super()**. It must be the first statement

Function Overloading

Java allows functions to have same name but different in arguments (signature) in the same scope.

```
class Test
{ int a;
  int area( int x, int y)
  { .....
    .....
  }
  int area( float a, float b)
  { .....
    .....
  }
  .....
}
```

Two methods named **area** but different in arguments lists (signature) in the same class.



➔ A function name having several definition but different in their argument (s) is called **Overloaded function**.

Why Function Overloading?

Function overloading is very important feature of OOP and used to implement *polymorphism*.

Polymorphism is the ability of a method to perform different job in different situations.

The advantages of function Overloading are-

- ❑ It is a tool to implement Polymorphism.
 - ❑ It avoids to write multiple functions (methods) for the different work and situations. This makes your code more compact and smarter. Compiler automatically determines which method to be executed and when.
 - ❑ Overloading reduces number of comparison in a program, so that programs run faster.
-

How to implement Function Overloading?

To overload a function, we can write multiple functions having **same name** but **different in their Signature** (i.e. different in number of arguments and their type).

The following points must be considered-

- ❑ Function with same signature and same name but different return type is not allowed in JAVA., then subsequent declaration is treated as erroneous re-declaration of first one and cause of compile time error.

```
float test (int x) {....}
```

```
int test (int x) {....} // error//
```

- ❑ Function may have different return type but only if the signatures are also different.

```
float test (float x) {....}
```

```
int test (int x) {....} // OK//
```

- ❑ Two functions are considered Overloaded only if they are different in either number of arguments or type of arguments.
-

Abstract Classes

Sometimes, we need to define a super-class having general characteristics (data) and behaviour (methods) of its sub-classes. A Sub-class may re-define the methods to perform a task, the super-class contains only the prototype (method with empty body). The **abstract** keyword is used to define a such class.

An Abstract class simply represents a concept for its sub-classes.

An abstract class may have **Abstract Methods** (method with no statements). Abstract methods are always overridden by the sub-class and sub-class provides statements for action.

```
public abstract class Test
{
    int a;
    int b;
    public abstract void method1();
    public abstract void method2();
}
```

Ex: Abstract Classes

```
public abstract class shape
{ String name;
  float area;
  public abstract void display();
}
class circle extends shape
{ float radius;
  float calArea()
  { float t= 3.14*radius*radius;
    return (t);
  }
  public void display()
  { system.out.println(" Area of circle="+calArea());
  }
}
class ractangle extends shape
{ float length, width;
  float calArea()
  { float t= length*width;
    return (t);
  }
  public void display()
  { system.out.println(" Area of ractangle="+calarea());
  }
}
```

Only prototype of display()
method is include in the
Abstract class.

display() method is re-defined

display() method is re-defined

Interfaces

Java does not support Multiple Inheritance (a sun-class having multiple super-classes). To tie elements of different classes together, Java uses an Interface.

Interfaces are declared using **interface** keyword and inherited by **implements** keywords.

An Interface defines a protocol of behaviour. The purpose of interface is to dictate common behaviour among objects from multiple classes.

An interface may declare two things-

1. **Abstract methods** (method with empty body) with public specifier.
2. **Constants** using final keyword and must be declared as public static.

```
interface myinterface  
{ public static final String COLOR="red";  
.....  
public void method1();  
public void method2(int x);  
}  
class myclass implements myinterface  
{.....}
```



A class can implements multiple interfaces but can extend only one subclass.

Interfaces v/s Multiple Inheritance

A class can implement more than one interfaces, which looks like multiple inheritance.

Ex. `class myclass implements interface1, interface2`
`{.....}`

But there are some differences.

- ❑ No variable (except constants) can be inherited from interfaces.
 - ❑ An interface can not implement any method because it contains only prototype of the methods like abstract class and methods (no code for methods).
 - ❑ An interface is not a part of class hierarchy i.e. unrelated classes can implement the same interface.
 - ❑ A class can not inherit more than one base class but it can inherit multiple interfaces to extend multiple methods protocol or behaviours.
-