Syllabus of module 2

- Inheritance - Super Class, Sub Class, The Keyword super, protected Members, Calling Order of Constructors, Method Overriding, the Object class, Abstract Classes and Methods, using final with Inheritance.

Inheritance

- The mechanism of deriving a new class from existing class is called Inheritance.
- The old/existing class is known as the **base class**(or **super class** or **parent class**)
- The new one is called the **subclass** (or **derived class** or **child class**).
  - A subclass is a specialized version of a superclass.
  - It inherits all of the instance variables and methods defined by the superclass and adds its own, unique elements.

- **extends is the keyword used to inherit properties of one class.**
- Syntax:

```
class subclass_name extends superclass_name
{
        // body of class
}
```

Inheritance

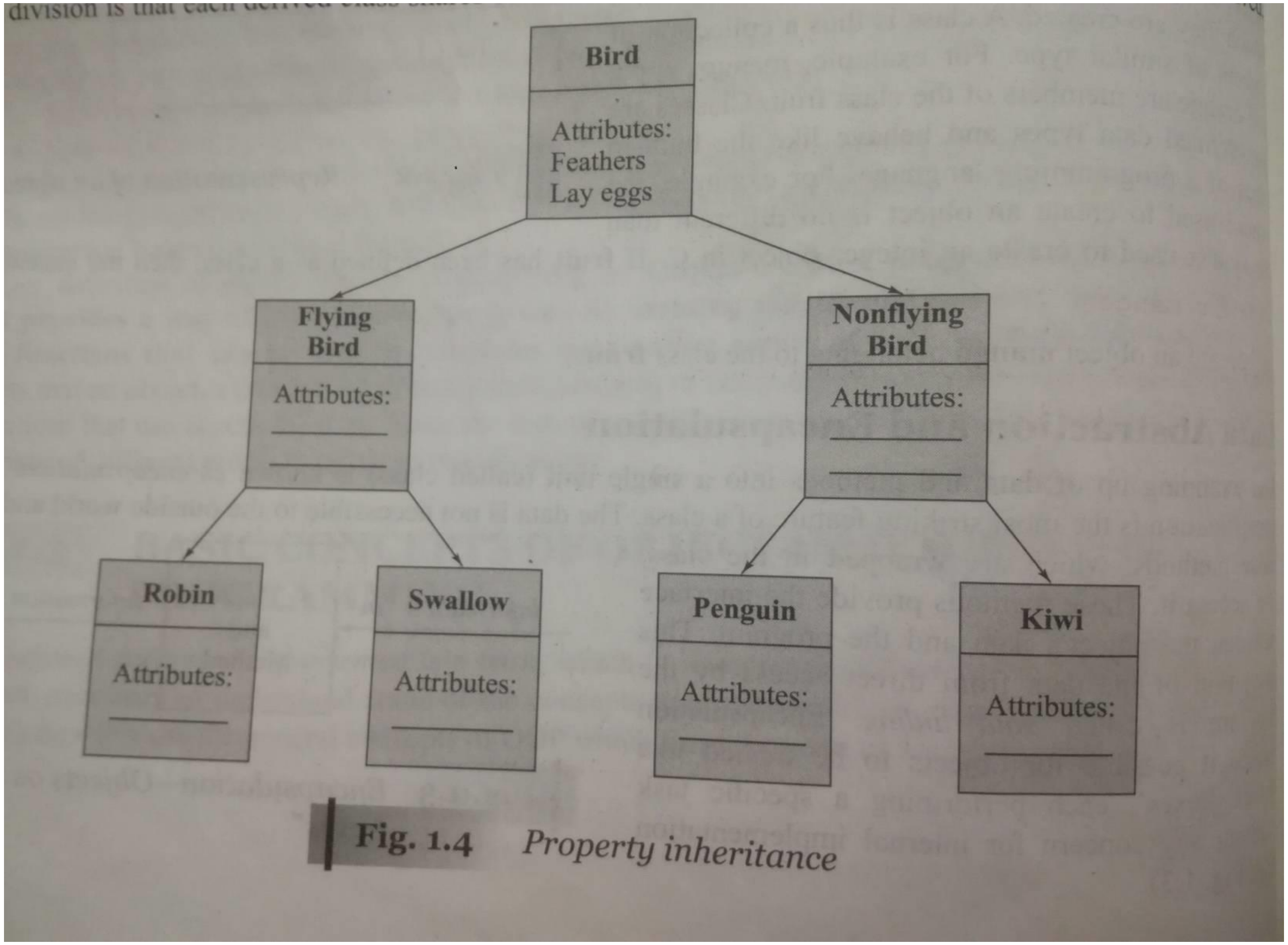
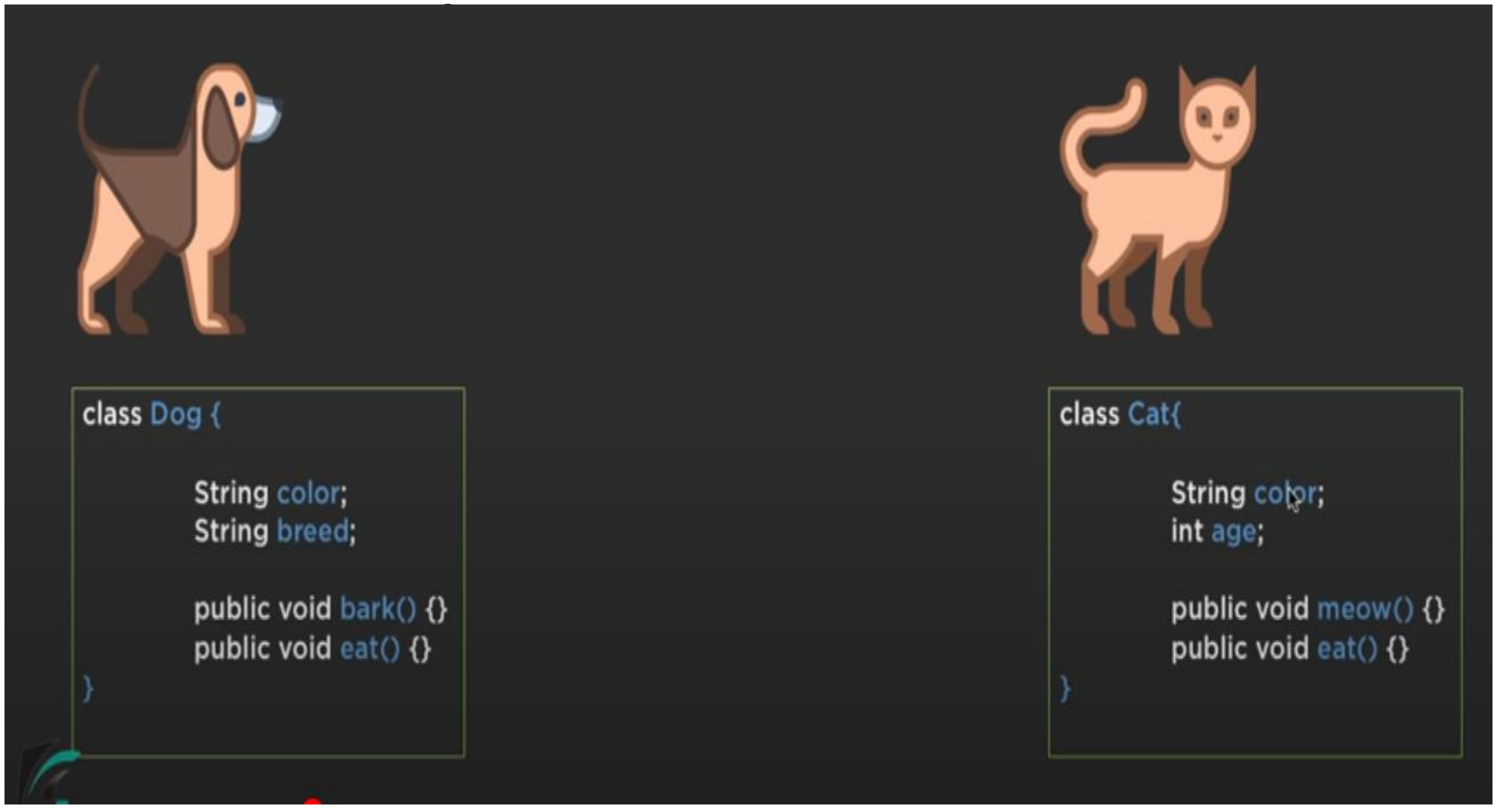


Fig. 1.4 *Property inheritance*

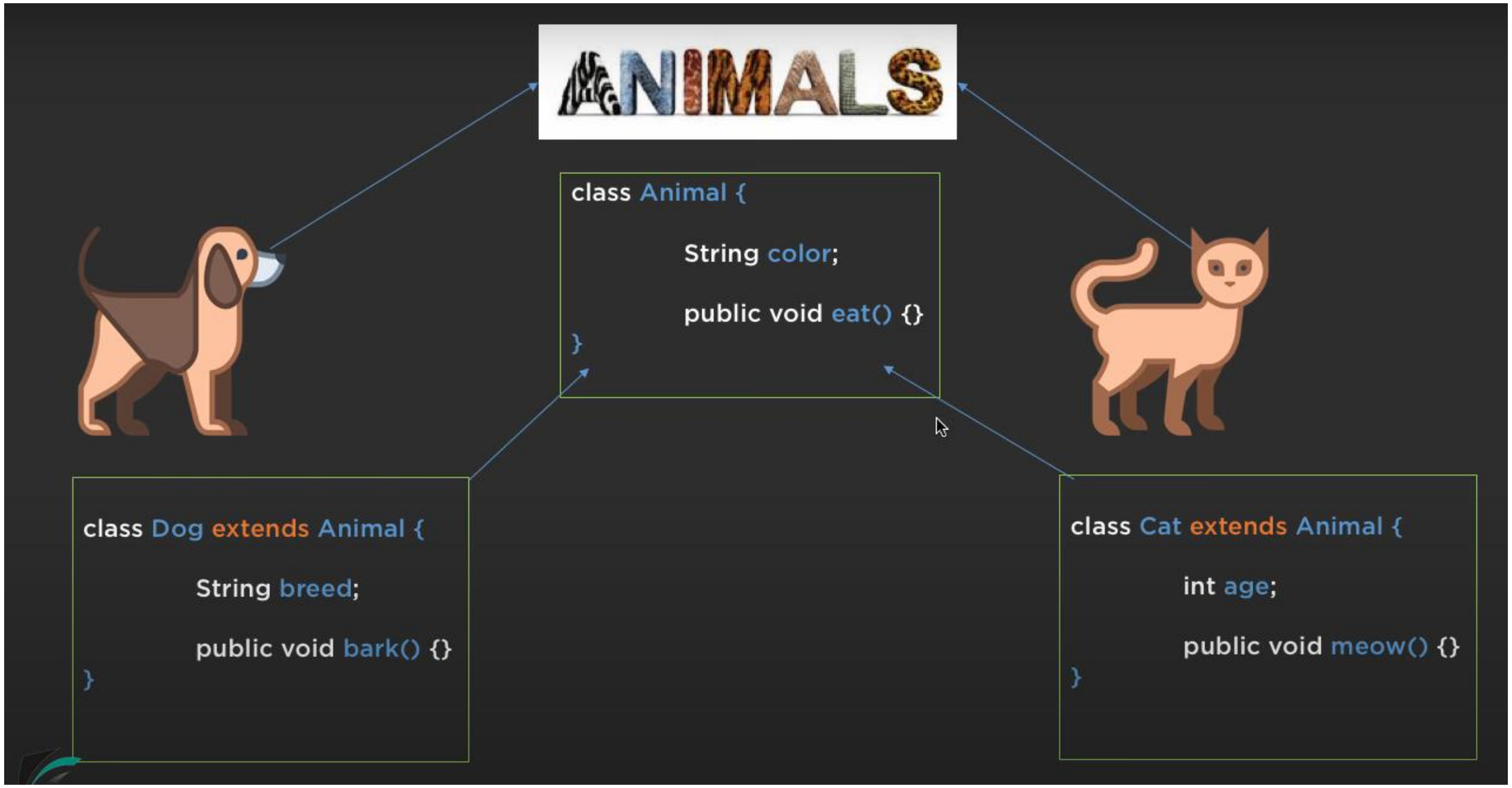
Why use inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.



```
class Dog {

        String color;
        String breed;

        public void bark() {}
        public void eat() {}
}
```

```
class Cat{

        String color;
        int age;

        public void meow() {}
        public void eat() {}
}
```



```
class Animal {
        String color;
        public void eat() {}
}
```

```
class Dog extends Animal {
        String breed;
        public void bark() {}
}
```

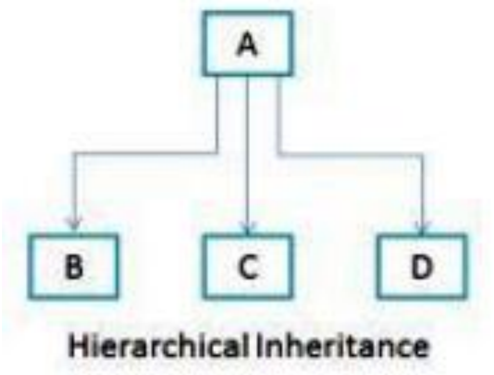
```
class Cat extends Animal {
        int age;
        public void meow() {}
}
```

```
class A
{     int i, j;
      void showij()
     {
      System.out.println("i and j: " +i+ " " +j);
     }
}
class B extends A
{     int k;
     void showk()
     {     System.out.println("k: " + k);
     }
     void sum()
     {
      System.out.println("i+j+k: " + (i+j+k));
     }
}
```

```
class SimpleInheritance
{  public static void main(String args[])
   {
      A superOb = new A();
      B subOb = new B();

      superOb.i = 10; superOb.j = 20;
      superOb.showij();   i and j:  10  20
      System.out.println("------");  -------

      subOb.i = 7; subOb.j = 8;
      subOb.k = 9;
      subOb.showij();     i  and j:  7  8
      subOb.showk();      k:  9
      subOb.sum();        i+j+k:   24
   }
}
```

```
class A
{      int i;                   // public by default
       private int j;       // private to A
       void setij(int x, int y)
       {     i = x;     j = y;       }
}
class B extends A
{      int total;
       void sum()
       {      total = i + j;        }   // ERROR, j is not accessible here
}
class Access
{      public static void main(String args[])
       {      B subOb = new B();
              subOb.setij(10, 12);
              subOb.sum();
              System.out.println("Total is " + subOb.total);
       }
}
```

- Different types of Inheritance :
  - **Single Inheritance**:
    - Only one super class
  - **Multiple Inheritance** :
    - Several super classes for a single sub class.
    - Java does not directly support this type of inheritance. This inheritance is implemented by using the concept of interfaces.
  - **Hierarchical Inheritance** :
    - One super class, many subclasses

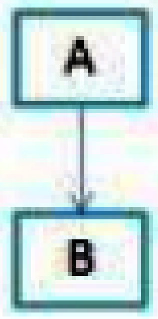


  - **Multilevel Inheritance** :
    - A subclass becomes a superclass of another subclass

Single Inheritance

- When a class extends another one class only then we call it a single inheritance.

- The given diagram shows that class B extends only one class which is A. Here A is a **parent class** of B and B would be a **child class** of A.

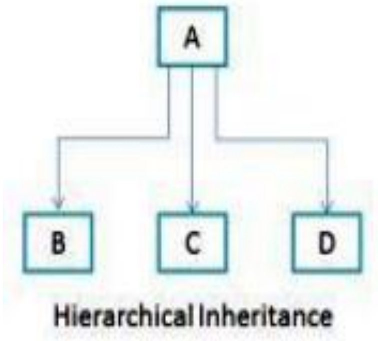


(a) Single Inheritance

```
Class A
{
   public void methodA()
   {
     System.out.println("Base class method");
   }
}

Class B extends A
{
   public void methodB()
   {
     System.out.println("Child class method");
   }
   public static void main(String args[])
   {
     B obj = new B();
     obj.methodA(); //calling super class method
     obj.methodB(); //calling local method
  }
}
```

Hierarchical Inheritance :one class is inherited by several subclasses:

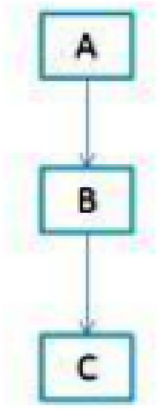
```
class A {
    public void methodA(){
        System.out.println("method of Class A");
    }
}
class B extends A{
    public void methodB(){
        System.out.println("method of Class B");
    }
}
class C extends A{
    public void methodC(){
         System.out.println("method of Class C"),
    }
}
```



```
public class Hello {
    public static void main(String[] args) {
        B obj1 = new B();
        C obj2 = new C();
        obj1.methodA();
        obj2.methodA();
    }
 }
```

Multilevel Inheritance

```
class A {
    public void methodA(){
       System.out.println("method of Class A");
    }
}
class B extends A{
    public void methodB(){
       System.out.println("method of Class B");
    }
}
class C extends B{
    public void methodC(){
        System.out.println("method of Class C");
    }
}
public class Hello {
    public static void main(String[] args) {
        B obj1 = new B();
        C obj2 = new C();
        obj1.methodA();
        obj2.methodA();
        obj2.methodB();
    }
}
```



Multilevel Inheritance

super Keyword

- If a subclass needs to refer to its immediate superclass, use a keyword called **super**.

- **super** is used in the following two situations.

  - To calls the superclass constructor
    - General form: **super(*parameter-list*);**
    - **super()** must always be the first statement executed inside a subclass constructor

  - To access a member of the superclass that has been hidden by a member of a subclass.
    - General form: **super.*member***
    - Here, *member* can be either a method or an instance variable

Example

```
class  A{
A(){
System.out.println("in constructor A");
     } }
class B extends A {
B(){
System.out.println("in constructore B");
    }                 }
class SuperDemo
{
public static void main(String args[])
{
A obj= new A();                    o/p  in constructor A
}
}
```

Example

```
class A{
A(){
System.out.println("in constructor A");
    }  }
class B extends A{
B(){
//super();  by default child class calls the parent class
   constructor
System.out.println("in constructor B");
}   }
class SuperDemo
{
public static void main(String args[])
{
B obj=new B();            o/p in  constructor A
}                              in  constructor B
}
```

Example

```
class A   {
A(){
System.out.println("in constructor A");
     }
A( int i){
System.out.println("in param constructor A");
          }  }
class B extends A{
B(){
System.out.println("in constructor B");
    }
B( int i){
System.out.println("in param constructor B");
        } }
class SuperDemo{
public static void main(String args[]){
B obj=new B(5);
  }}
```

o/p ???

Example

```
class A   {
A(){
System.out.println("in constructor A");
     }
A( int i){
System.out.println("in param constructor A");
          }  }
class B extends A{
B(){
System.out.println("in constructore B");
    }
B( int i){    //super();
System.out.println("in param constructor B");
        } }
class SuperDemo{
public static void main(String args[]){
B obj=new B(5);
  }}
```

o/p in constructor A
in param constructor B

Example

```
class A   {
A()
{    System.out.println("in constructor A");  }
A( int i){
System.out.println("in param constructor A");
         }  }
class B extends A{
B(){
System.out.println("in constructore B");
    }
B( int i){
super(i);
System.out.println("in param constructor B");
        } }
class SuperDemo{
public static void main(String args[]){
B obj=new B(5);
  }}
```

o/p in param constructor A
in param constructor B

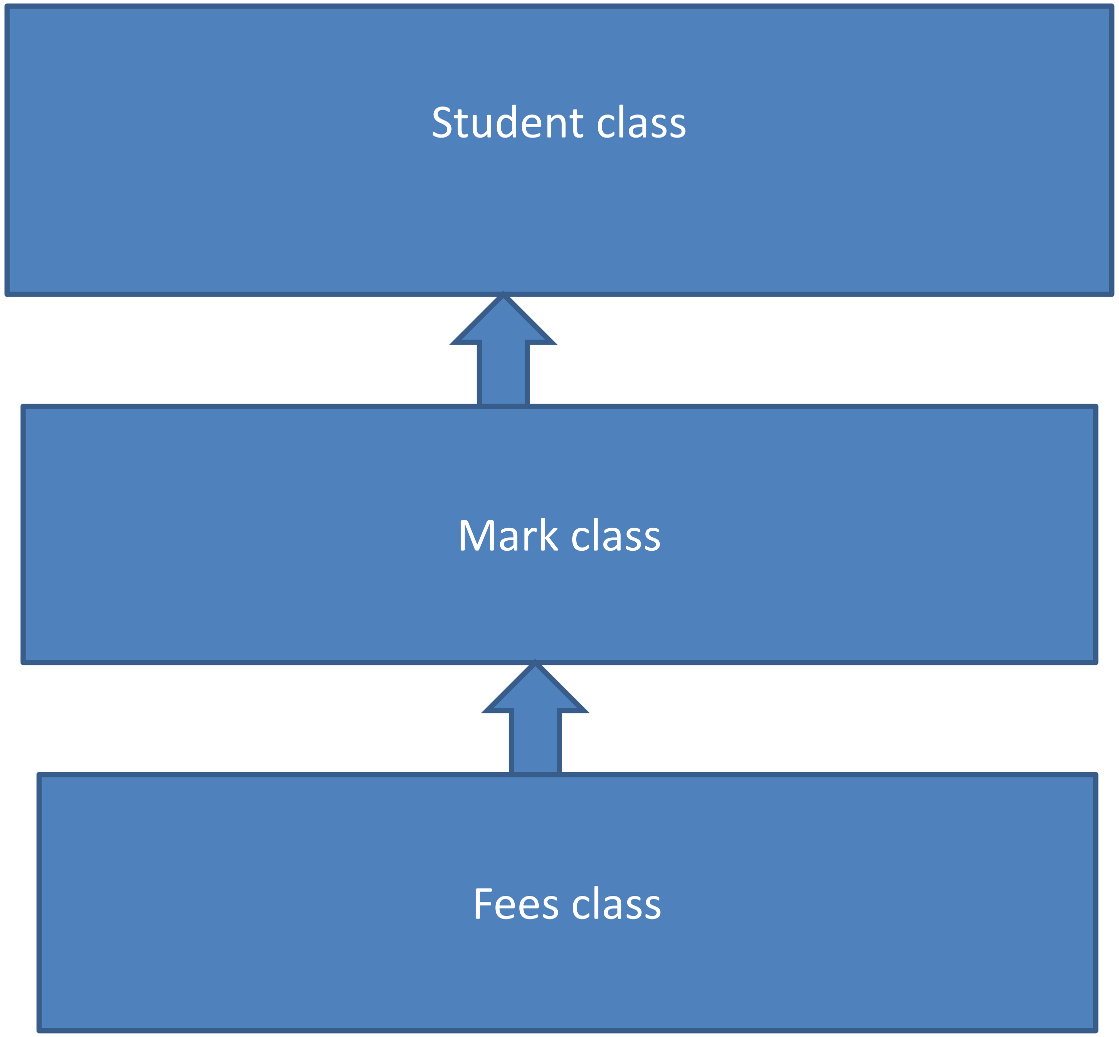
```
class Box
{       double width, height, depth;
     Box(Box ob)
     {       width = ob.width;
             height = ob.height;
             depth = ob.depth;
     }
     Box(double w, double h, double d)
     {    width = w;   height = h;   depth = d;
     }
     Box()
     {      width = height = depth = -1;
     }
     Box(double len)
    {      width = height = depth = len;
    }
    double volume()
    {      return width * height * depth;
    }
}
```

```
class BoxWeight extends Box
{
     double weight;
     BoxWeight(double w, double h, double d, double m)
        {
              super(w, h, d); // call superclass constructor
              weight = m;
        }
}

class DemoBoxWeight
{
   public static void main(String args[])
   {
   BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3)
    double vol  = mybox1.volume();
    System.out.println("Volume of mybox1 is " + vol);
   }
}
```

o/p Volume of mybox1 is 3000.0

```
class A
{       int i;
}
class B extends A
{       int i;                  // this i hides the i in A
B(int a, int b)
{       super.i = a;   // i in A
i = b;             // i in B
}
void show()
{       System.out.println("i in superclass: " + super.i);
System.out.println("i in subclass: " + i);
}
}
class UseSuper
{       public static void main(String args[])                    o/p   i in superclass: 1
{       B subOb = new B(1, 2);                                          i in subclass:   2
subOb.show();
}
}
```

Q1) Create a class hierarchy as shown in below figure.



```
class student
{
 int roll;
 String name;
 student(int r,String n)
 {
 roll=r;
 name=n;
 }

void display()
 {
 System.out.println("Name     :"+name);
 System.out.println("Roll No :"+roll);
 }
}
```

```
class marks extends student
{
 int total;
 marks(int r,String n,int tot)
 {   super(r,n);
  total=tot;
 }
void show()
  {    super.display();
   System.out.println("Total    :"+total);
  }
}
```

```
class fees extends marks
{
 int fee;
 fees(int r,String n,int tot,int f)
 {
  super(r,n,tot);
  fee=f;
 }

 void preview()
 {
  super.show();
  System.out.println("Fees     :"+fee);
 }
}
```

```
class multilevel
{
 public static void main(String args[])
{
  fees f1=new fees(1,"ABC",98,1000);
  fees f2=new fees(2,"DEF",97,1200);
  fees f3=new fees(3,"GHI",92,900);
  f1.preview();
  f2.preview();
  f3.preview();
} }
```

Order of execution of constructors

- In a class hierarchy, constructors are called in order of derivation, from superclass to subclass. If **super( )** is not used, then the default or parameterless constructor of each superclass will be executed.

```
class A
{       A()
        {   System.out.println("Inside A");     }
}
class B extends A
{       B()
       { System.out.println("Inside B");     }
}
class C extends B
{       C()
        {System.out.println("Inside C");     }
}
class CallingCons
{       public static void main(String args[])
        {        C c = new C();     }
}
```

```
Output :    Inside A
            Inside B
            Inside C
```

Polymorphism in Java

- Polymorphism in Java is a concept by which we can perform a single action in different ways.
- There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by **method overloading**(compile time polymorphism) and **method overriding**(run time polymorphism).

Method Overriding

- **Method overriding** is one of the ways in which Java supports Runtime Polymorphism.
- In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass.
- When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass.
- By using **super** we can access the superclass version of an overridden function.
- Method overriding occurs *only* when the names and the type signatures of the two methods are identical. If they are not, then the two methods are simply overloaded.

```
class A
{       int i, j;
        A(int a, int b)
        {       i = a;
                j = b;
        }
        void show()
        {
              System.out.println(i +  " " + j);
        }
}
class B extends A
{       int k;
        B(int a, int b, int c)
        {       super(a, b);
                k = c;
         }
         void show()// this overrides
        {        System.out.println("k: " + k);
        }
}
```

```
class Override
{
        public static void main(String args[])
        {
                B subOb = new B(1, 2, 3);
                subOb.show(); // calls show() in B
        }
}
```

```
Output :   3
```

```
class A
{       int i, j;
        A(int a, int b)
        {       i = a;    j = b;      }
              void show()
             {
               System.out.println("i and j: " + i +
" " + j);
             }
}
class B extends A
{       int k;
         B(int a, int b, int c)
         {      super(a, b);
               k = c;
          }
          void show()
        { super.show(); // this calls A's show()
          System.out.println("k: " + k);
         }
}
```

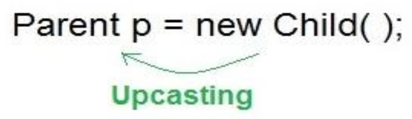
```
class Override
{
        public static void main(String args[])
        {
                B subOb = new B(1, 2, 3);
                subOb.show();// calls show() in B
        }
}
```

```
Output :    i and j : 1 2
            k : 3
```

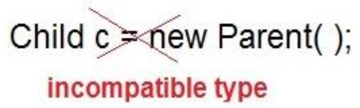
Dynamic Method Dispatch or Runtime polymorphism

- Dynamic method dispatch is a mechanism by which a call to an overridden method is resolved at runtime. This is how java implements runtime polymorphism.

- When Parent class reference variable refers to Child class object, it is known as Upcasting. In Java this can be done and is helpful in scenarios where multiple child classes extends one parent class. In those cases we can create a parent class reference and assign child class objects to it.



```
Parent p = new Child( );
```

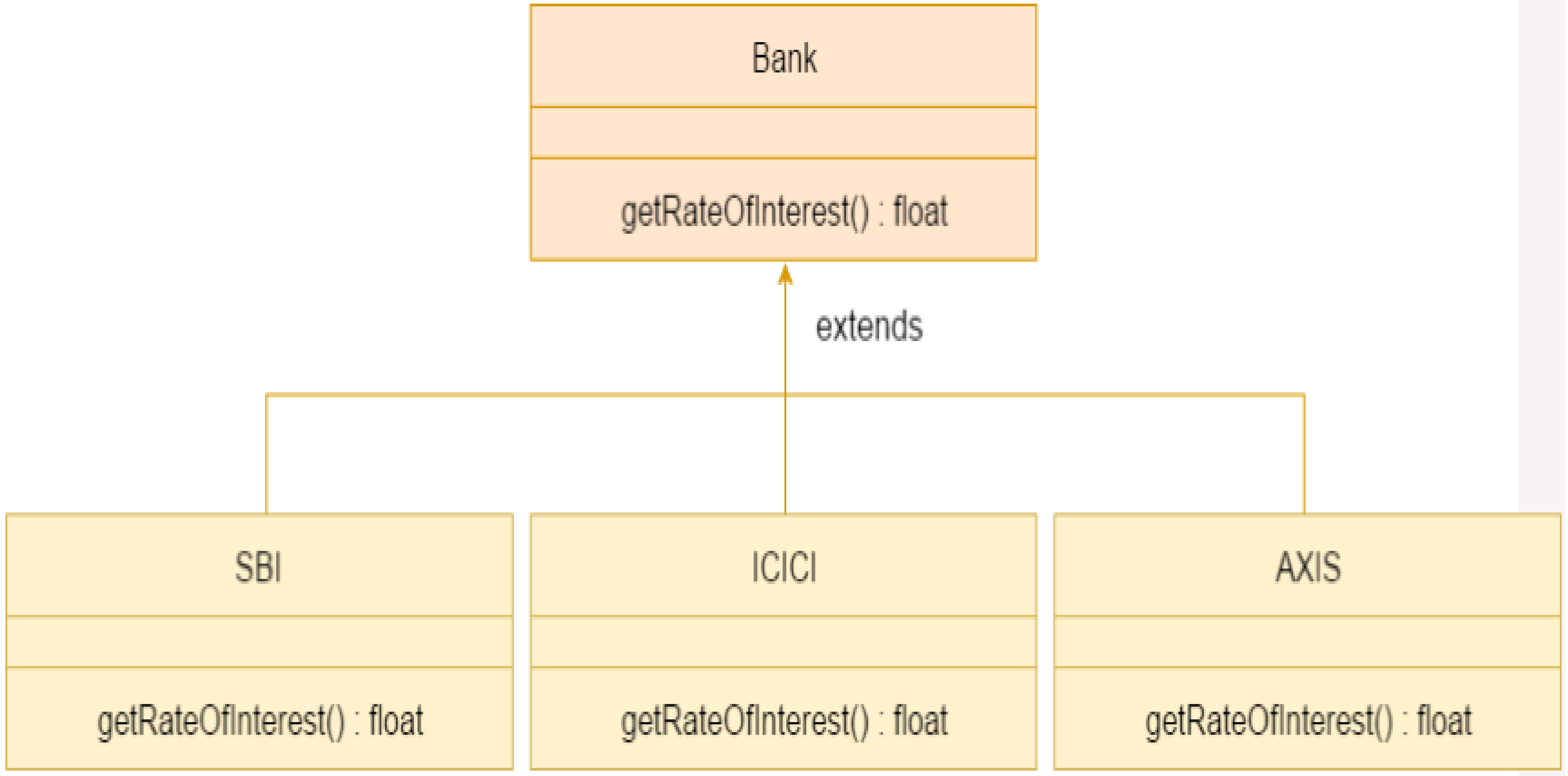


```
Child c = new Parent( );
```

- When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs.
- Method overridding is resolved at run time, rather than compile time. Hence it is an example for run-time polymorphism.

Java Runtime Polymorphism Example: Bank

Consider a scenario where Bank is a class that provides a method to get the rate of interest. However, the rate of interest may differ according to banks. For example, SBI, ICICI, and AXIS banks are providing 8.4%, 7.3%, and 9.7% rate of interest.


Bank
getRateOfInterest() : float
extends
SBI
getRateOfInterest() : float
ICICI
getRateOfInterest() : float
AXIS
getRateOfInterest() : float

Sample code

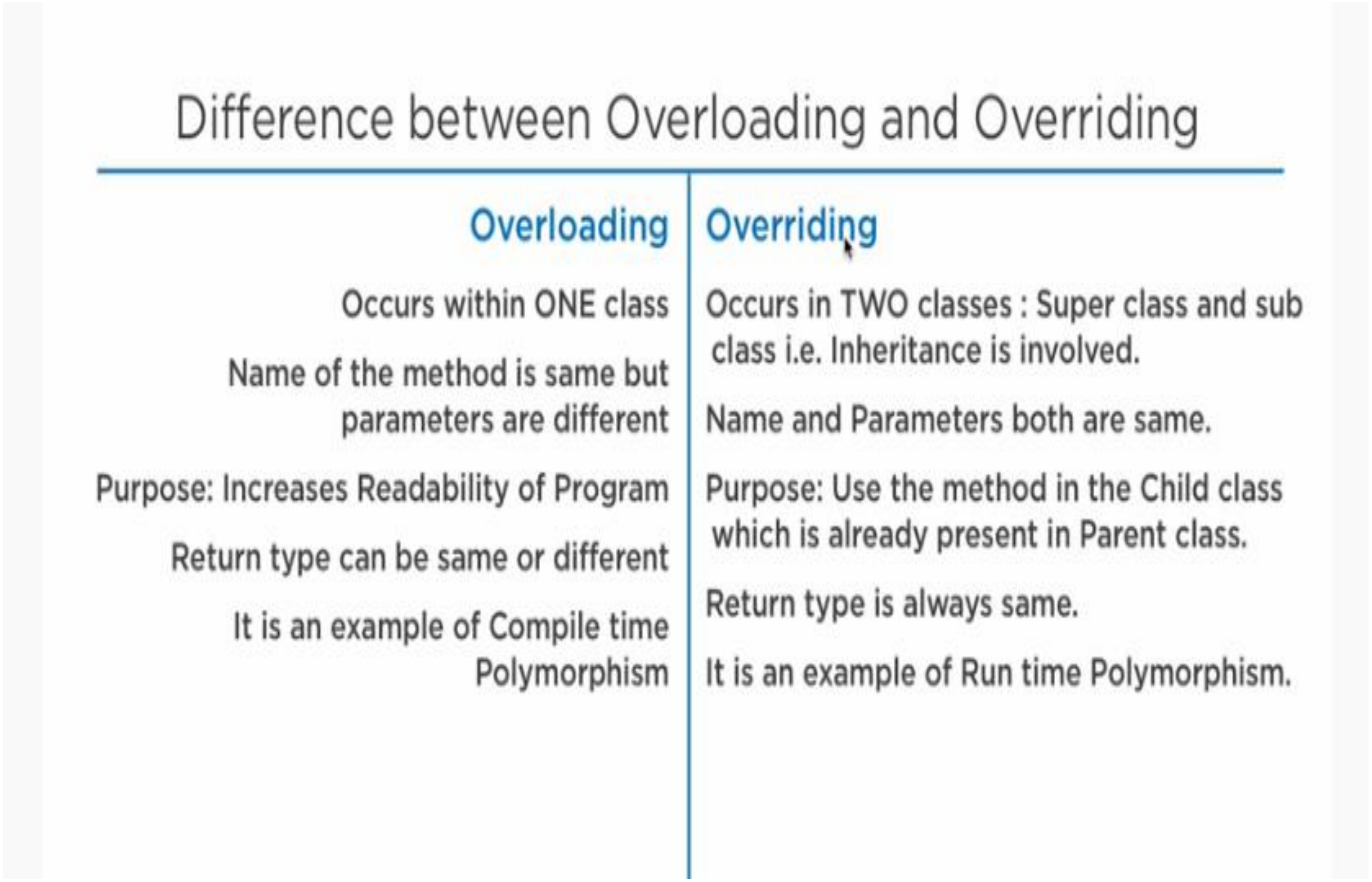
```
class Bank{
float getRateOfInterest(){return 0;}
}
class SBI extends Bank{
float getRateOfInterest(){return 8.4f;}
}
class AXIS extends Bank{
float getRateOfInterest(){return 7.4f;}
}
class TestPolymorphism{
public static void main(String args[])
{
Bank b=new SBI();
 //Upcasting  is typecasting of a child object to a parent object as shown
     above
System.out.println("SBI Rate of Interest: "+b.getRateOfInterest());
b=new AXIS();        //Upcasting
System.out.println("AXIS Rate of Interest: "+b.getRateOfInterest());
}                          o/p   SBI Rate of Interest:  8.4
}                                AXIS Rate of Interest:   7.4
```

```
class A
{       void callme()
        {
                System.out.println("Inside A");
        }
}
class B extends A
{
        void callme()
        {
                System.out.println("Inside B");
        }
}
class C extends A
{
        void callme()
        {
                System.out.println("Inside C");
        }
}
```

```
class Dispatch
{
        public static void main(String args[])
        {
                A a = new A(); // object of type A
                B b = new B(); // object of type B
                C c = new C(); // object of type C
                A r; // obtain a reference of type A
                r = a; // r refers to an A object
                r.callme(); // calls A's callme
                r = b; // r refers to a B object
                r.callme(); // calls B's callme
                r = c; // r refers to a C object
                r.callme(); // calls C's callme
        }
}
```

```
Output :    Inside A
            Inside B
            Inside C
```

```
class Parent{
   int x=10;
   void show(){ System.out.println("parent-show"); }
   void OnlyParentShow(){ System.out.println("OnlyParentShow");
    }
                  }
class Child extends Parent{
   int x=20;
   void show(){  System.out.println("child-show");  }
   void OnlyChildShow(){ System.out.println("OnlyChildShow");  }
                             }
public class ParentChild {

   public static void main(String[] args) {
      Parent p = new Child();
      p.show();
      p.OnlyParentShow();
      System.out.println(p.x);
   }
}
```



Difference between Overloading and Overriding

Overloading	Overriding
Occurs within ONE class	Occurs in TWO classes : Super class and sub class i.e. Inheritance is involved.
Name of the method is same but parameters are different	Name and Parameters both are same.
Purpose: Increases Readability of Program	Purpose: Use the method in the Child class which is already present in Parent class.
Return type can be same or different	Return type is always same.
It is an example of Compile time Polymorphism	It is an example of Run time Polymorphism.

Abstract Methods and Classes

- Data **abstraction** is the process of hiding certain details and showing only essential information to the user.
- Abstraction can be achieved with either **abstract classes** or **interfaces**
- The abstract keyword is used for classes and methods:
- **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
- **Abstract method:** can only be used in an **abstract class**, and it does not have a body. The body is provided by the subclass (inherited from).
- An abstract class can have both abstract and regular methods:

Abstract Methods and Classes

- Abstract method are those methods that lack definition(i.e. body). It must always be redefined in the subclass.
- Overriding is compulsory for abstract methods.
- General form: **abstract *type name(parameter-list)*;**

- An abstract class is a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.
- Example:

```
abstract class Shape
{         ……………..
          abstract void draw();
          ………………..
}
```

- Abstract classes must satisfy the following conditions:
    1. Abstract classes are not used to instantiate objects directly
    2. The abstract method of the abstract class must be defined in its subclass.
    3. We can not declare abstract constructors or abstract static methods

- Abstract classes can be used to create object references. This reference can be used to point to a subclass object.

```
abstract class Bank {
 abstract int getRateOfInterest();
}
class ICICI extends Bank
{
 int getRateOfInterest() { return 5; }
}
class HDFC extends Bank {
 int getRateOfInterest() { return 6; }
 }
class TestBank {
public static void main(String args[]) {
Bank b; b=new ICICI();
System.out.println("Rate of Interest ICICI: "+b.getRateOfInterest());
    b=new HDFC();
System.out.println("Rate of Interest HDFC :
    "+b.getRateOfInterest()); } }
```

Output:

```
Rate of Interest ICICI: 5
Rate of Interest HDFC: 6
```

```
abstract class A
{               abstract void callme();
                void callmetoo()
                {System.out.println("This is a concrete method.");
                }
}
class B extends A
{                void callme()
                {System.out.println("B's implementation of callme.");
                }
}
class AbstractDemo
{
                public static void main(String args[])
                {
                                B b = new B();
                                b.callme();
                                b.callmetoo();
                }
}
```

Output :

```
B's implementation of
callme
This is a concrete method.
```

```
abstract class Figure
{       double dim1, dim2;
        Figure(double a, double b)
        {   dim1 = a;    dim2 = b;     }
        abstract double area();
}
class Rectangle extends Figure
{       Rectangle(double a, double b)
       {     super(a, b);       }
      double area()
      {  System.out.println("Inside Rect");
          return dim1 * dim2;
      }
}
class Triangle extends Figure
{       Triangle(double a, double b)
       {    super(a, b);     }
       double area()
       {System.out.println("Inside Tri“);
         return dim1 * dim2 / 2;
       }
}
```

```
class AbstractAreas
{
       public static void main(String args[])
       {
       // Figure f = new Figure(10, 10); // illegal
       Rectangle r = new Rectangle(9, 5);
       Triangle t = new Triangle(10, 8);
       Figure figref; // OK, no object is created
       figref = r;
       System.out.println( figref.area());
       figref = t;
       System.out.println( figref.area());
       }
}
```

Final Variables, Methods and Classes

- The keyword **final** has three uses.
  - **Equivalent to named constants** :
    - The value of the final variable can never be changed.
    - Final variables behave like class variables and they do not take any space on individual objects of the class.
    - Example : **final int size=100;**
  - **To Prevent Overriding** :
    - To prevent the subclasses from overriding the members of the super class, we can declare them as final in its super class using the keyword **final**.
  - **To Prevent Inheritance :**
    - Precede the class declaration with **final**.
    - Declaring a class as **final** implicitly declares all of its methods as **final**, too.
    - It is illegal to declare a class as both **abstract** and **final**

```
class A
{       final void meth()
        {System.out.println("This is a final method.");
        }
}
class B extends A
{       void meth()
        { // ERROR! Can't override.
                System.out.println("Illegal!");
        }
}
```

```
final class A
{
      // ...
}
// The following class is illegal.
class B extends A
{    // ERROR! Can't subclass A
      // ...
}
```

Object class in Java

- **Object** class is present in **java.lang** package.
- Every class in Java is directly or indirectly derived from the **Object** class.
- If a Class does not extend any other class then it is direct child class of **Object** and if extends other class then it is indirectly derived.
- Therefore the Object class methods are available to all Java classes and it acts as a

Object defines the following methods

- Object clone() – It returns a new object that is exactly the same as this object.
- boolean equals(Object object)- Determines whether one object is equal to another.
- void finalize()- This method is called just before an object is garbage collected.
- Class getClass()- Obtains the class of an object at run time.
- int hashCode()- Returns the hash code associated with the invoking object.
- void notify()- Resumes execution of a thread waiting on the invoking object.
- void notifyAll()- Resumes execution of all threads waiting on the invoking object.
- String toString()- Returns a stirng that describes the object.
- Void wait()- Waits on another thread of execution.

The methods **getClass()**, **notify()**, **notifyAll()** and **wait()** are declared as **final**.