Object Oriented
Programming with JAVA

Inheritance and Interfaces

Inheritance

- Inheritance is one of the key feature of Object Oriented Programming.
- Inheritance **provided mechanism** that allowed a **class to inherit property** of another class.
- When a Class **extends** another class it inherits all **non-private members** including **fields and methods**.
- Inheritance in Java can be best understood in terms of **Parent** and **Child** relationship, also known as **Super class**(Parent) and **Sub class**(Child).

“IS-A” relationship

- Inheritance defines **IS-A** relationship between a **Super class** and its **Sub class**.
- For Example :
  - Car **IS A** Vehicle
  - Bike **IS A** Vehicle
  - EngineeringCollege **IS A** College
  - MedicalCollege **IS A** College
  - MCACollege **IS A** College

“extends” keyword

- ***extends*** is the keyword used to implement inheritance.

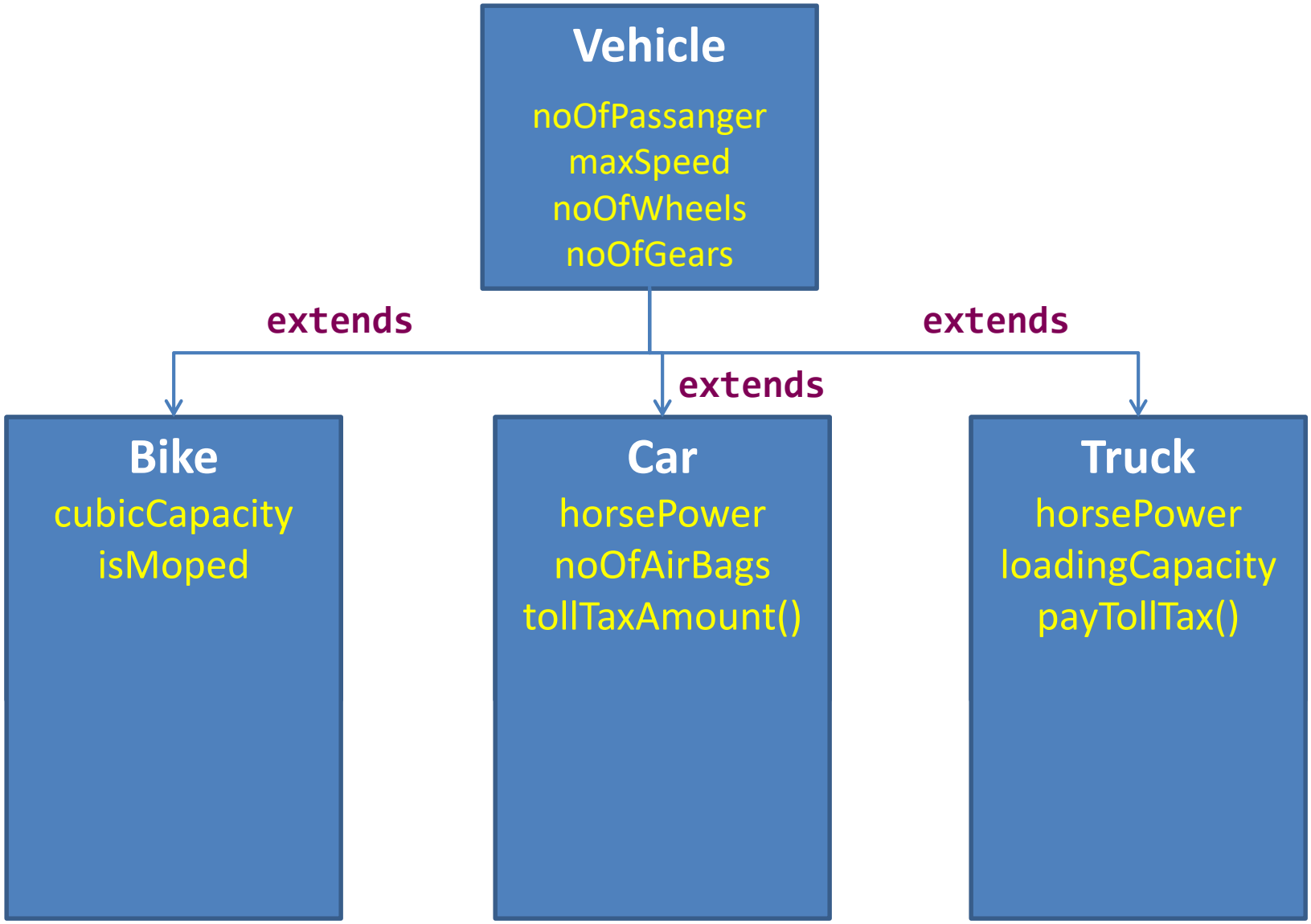
Syntax :

```
class A {

        // code

}

class B extends A{

        // code

}
```

Example :

```
class Vehicle {

        . . . . . .

}

class Car extends Vehicle{

        . . . . . .

}
```

Example


Vehicle
noOfPassanger
maxSpeed
noOfWheels
noOfGears
extends
extends
extends
Bike
cubicCapacity
isMoped
Car
horsePower
noOfAirBags
tollTaxAmount()
Truck
horsePower
loadingCapacity
payTollTax()

Example (Cont.)

```java
class Vehicle {
    int noOfPassanger;
    int maxSpeed;

    public void display() {
        System.out.println("Passangers = " + noOfPassanger);
        System.out.println("Max Speed = " + maxSpeed);
    }
}
```

```java
class Car extends Vehicle {
    double horsePower;
    int noOfAirbags;
    public void display() {
        System.out.println("Passangers = " + noOfPassanger);
        System.out.println("Max Speed = " + maxSpeed);
        System.out.println("Hourse Power = " + horsePower);
        System.out.println("Airbags = " + noOfAirbags);
    }
}
```

Example (DemoInheritance.java)

```
public class DemoInheritance {
    public static void main(String ar[]) {
        Vehicle v = new Vehicle();
        v.maxSpeed = 80;
        v.noOfPassanger = 2;
        System.out.println("---- Vehical ----");
        v.display();

        Car c = new
        c.maxSpeed =
        c.noOfPassang
        c.horsePower
        c.noOfAirbags
        System.out.pr
        c.display();
    }
}
```

C:\WINDOWS\system32\cmd.exe

```
D:\DegreeDemo>javac DemoInheritance.java

D:\DegreeDemo>java DemoInheritance
---- Vehical ----
Passangers = 2
Max Speed = 80
---- Car ----
Passangers = 5
Max Speed = 200
Hourse Power = 1.2
Airbags = 2
```

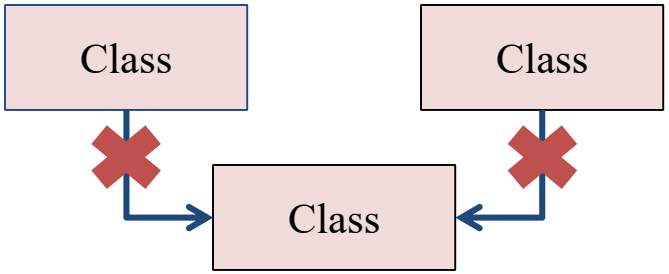
Inheritance (Cont.)

- Each Java class has **one (and only one)** superclass.

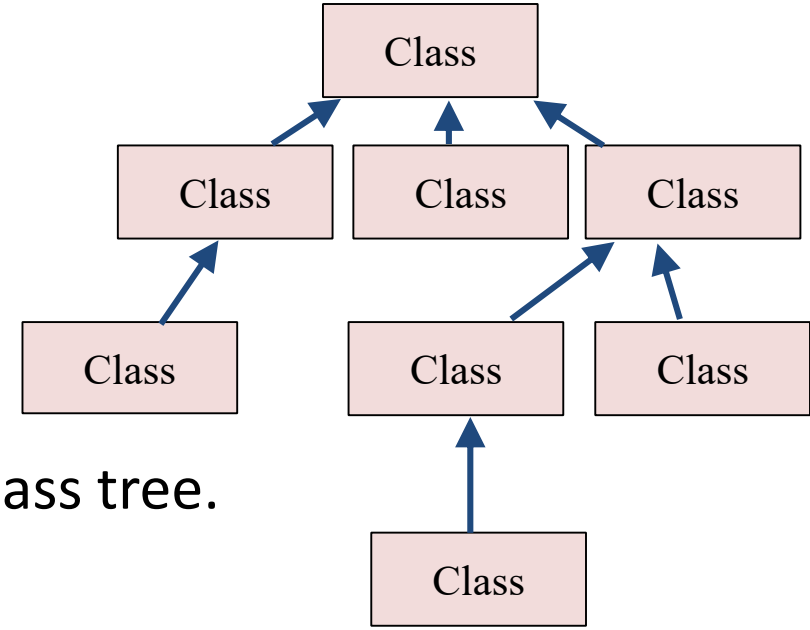
  C++ allows multiple inheritance

  BUT

  Java does **not support** multiple inheritance



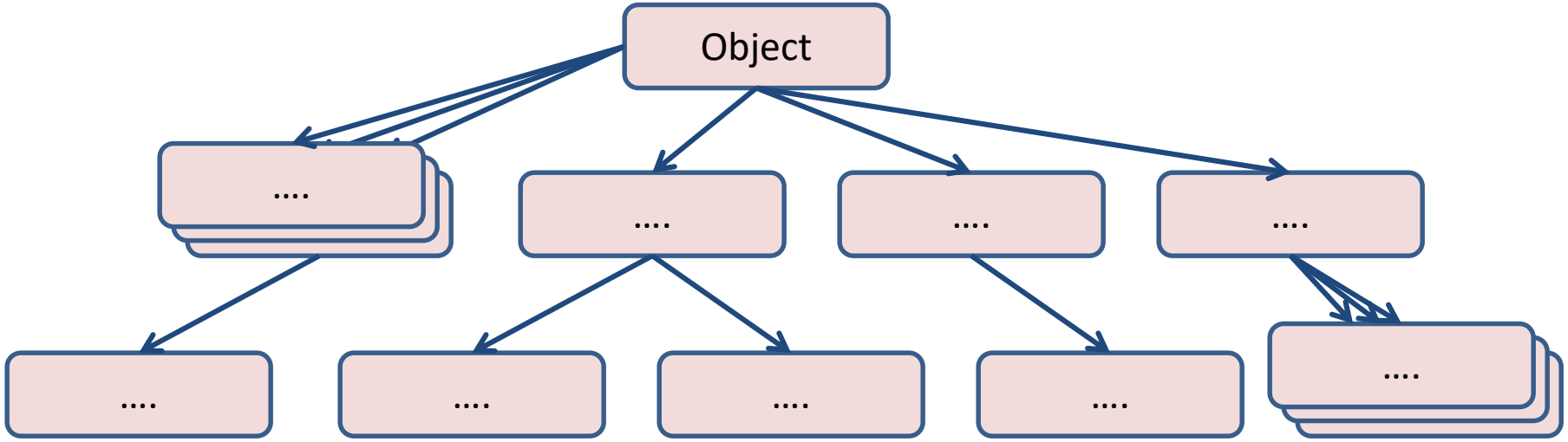
- There is **no limit** to the **number of subclasses** a class can have
- Inheritance creates a **class hierarchy**
  - Classes **higher** in the **hierarchy** are **more general** and **more abstract**
  - Classes **lower** in the **hierarchy** are **more specific** and **concrete**



- There is no limit to the depth of the class tree.

Object class

- **Object class** is **super** class of all the classes.
- The **Object** class is defined in the **java.lang** package



Constructors in Inheritance

- Classes use **constructors** to **initialize instance variables**
  - When a **subclass** object is **created**, its **constructor** is **called**.
  - It is the **responsibility** of the **subclass** constructor to **invoke** the appropriate **superclass constructors** so that the instance variables defined in the superclass are properly initialized
- Superclass constructors can be called using the "**super**" keyword in a manner **similar to "this"**
  - It **must** be the **first line** of code in the **constructor**
- If a call to super is not made, the system will automatically invoke the no-argument constructor of the superclass.

Constructor Example

```
import java.util.Date;

class Person
{
    public String name;
    public Date dateOfBirth;
    public Person()
    {
        this.name = "Not Set";
        this.dateOfBirth = new Date();
    }
    public Person(String name,Date dateOfBirth)
    {
        this.name = name;
        this.dateOfBirth = dateOfBirth;
    }
}
```

Constructor Example (Cont.)

```
import java.util.Date;

class Employe extends Person{
    public int employeID;
    public double salary;
    public Date dateOfJoining;

    public Employe(){
        this.employeID = 0;
        this.salary = 0;
        this.dateOfJoining = new Date();
    }
    public Employe(String name,Date dateOfBirth,double salary,Date
    dateOfJoining,int employeID){
        super(name,dateOfBirth);
        this.employeID = employeID;
        this.salary = salary;
        this.dateOfJoining = dateOfJoining;
    }
}
```

Constructor Example (Cont.)

```java
import java.util.Date;

public class CallEmploye {
    public static void main(String[] ar) {
        Employe e1 = new Employe();
        System.out.println("Name = " + e1.name);

        Employe e2 =
        new Employe("DIET",new Date(1988,10,20),1000.0,new Date(),1);
        System.out.println("Name = " + e2.name);
    }
}
```

C:\WINDOWS\system32\cmd.exe

```
D:\DegreeDemo\PPTDemo>javac CallEmploye.java
D:\DegreeDemo\PPTDemo>java CallEmploye
Name = Not Set
Name = DIET
```

Method Overriding

- Subclasses **inherit** all **methods** from their **superclass**
  - Sometimes, the implementation of the method in the superclass **does not** provide the **functionality** required by the **subclass**.
  - In these cases, the method must be **overridden**.
- Rules for Method overriding
  - **Method signature** must be same as of Super Class method.
  - The **return type** should be the **same**.
  - The **access level** cannot be more **restrictive** than the **overridden** method's access level.
    - Example :
      - protected -> public // is allowed
      - protected -> private // is not allowed

Overriding (Example)

```java
class SmartPhone{
    public void setAlarm(){
        System.out.println
        ("Goto Apps\n
        Open Clock\n
        Set Alarm");
    }
}
```

```java
class IPhone extends SmartPhone
{
    public void setAlarm()
    {
        System.out.println
        ("Tell Siri to Set Alarm");
    }
}
```

```java
public class OverrideDemo {
    public static void main(String[] ar) {
        SmartPhone s = new SmartPhone();
        System.out.println(
        s.setAlarm();

        IPhone i = new IPho
        System.out.println(
        i.setAlarm();
    }
}
```

C:\WINDOWS\system32\cmd.exe

```
D:\DegreeDemo\PPTDemo>javac OverrideDemo.java
D:\DegreeDemo\PPTDemo>java OverrideDemo
--- SmartPhone ---
Goto Apps
 Open Clock
 Set Alarm
--- IPhone ---
Tell Siri to Set Alarm
```

“final” keyword

- The final keyword is used for **restriction**.
- final keyword can be used in many context
- Final can be:

1. Variable

   If you make any variable as final, you **cannot change the value** of final variable(It will be constant).

2. Method

   If you make any method as final, you **cannot override** it.

3. Class

   If you make any class as final, you **cannot extend** it.

1) “final” as a variable

- Can **not change** the **value** of final **variable.**

```
public class FinalDemo {

 final int speedlimit=90;//final variable
 void run(){
        speedlimit=400;
 }
 public static void main(String args[]){
        FinalDemo obj=new  FinalDemo();
        obj.run();
 }
}
```

2) “final” as a method

- If you make any **method** as **final**, you **cannot override** it.

```
class BikeClass{
  final void run(){System.out.println("Running Bike");}
}

class Pulsar extends BikeClass{
   void run(){System.out.println("Running Pulsar");}

   public static void main(String args[]){
   Pulsar p= new Pulsar();
   p.run();
   }
}
```

3) “final” as a Class

- If you make any **class** as **final**, you **cannot extend** it.

```
final class BikeClass{
        void run(){System.out.println("Running Bike");}
}

class Pulsar
{
   void run(){System.out.println("Running Pulsar");}

   public static void main(String args[]){
   Pulsar p= new Pulsar();
   p.run();
   }
}
```

Interface

- An interface is similar to an abstract class with the following exceptions
  - **All** methods defined in an interface are **abstract**. Interfaces can contain no implementation
  - Interfaces **cannot** contain **instance variables**. However, they can contain **public static final** variables (ie. constant class variables)
- Interfaces are declared using the "interface" keyword
- If an interface is public, it must be contained in a file which has the same name
- Interfaces are more abstract than abstract classes
- Interfaces are implemented by classes using the "implements" keyword

Example (Interface)

```java
interface VehicalInterface {
    int a = 10;
    public void turnLeft();
    public void turnRight();
    public void accelerate();
    public void slowDown();
}
```

```java
class CarClass implements
VehicalInterface
{
    public void turnLeft() {
    System.out.println("Left");
    }
    public void turnRight() {
    System.out.println("Right");
    }
    public void accelerate() {
    System.out.println("Speed");
    }
    public void slowDown() {
    System.out.println("Break");
    }
```

```java
public class DemoInterface{
public static                    a)
{
    CarClass c
    c.turnLeft
}
}
```

We have to provide implementation to all the methods of the interface

C:\WINDOWS\system32\cmd.exe

```
D:\DegreeDemo\PPTDemo>javac DemoInterface.java

D:\DegreeDemo\PPTDemo>java DemoInterface
Left
```

Interface V/S Abstract Class

Interface	Abstract Class

Dynamic Method Dispatch

- Method overriding is one of the ways in which Java supports **Runtime Polymorphism**.
- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved **at run time**, rather than compile time.
- A superclass reference variable can refer to a subclass object, This is also known as **upcasting**.

Example (Dynamic Method Dispatch)

```
class Game {
    public void type() {
        System.out.println("Indoor & outdoor");
    }
}
class Cricket extends Game {
    public void type() {
        System.out.println("outdoor game");
    }
}
class Badminton extends Game {
    public void type() {
        System.out.println("indoor game");
    }
}
class Tennis extends Game {
    public void type() {
        System.out.println("Mix game");
    }
}
```

Example (Cont.) (MyProg.java)

```
public class MyProg {
    public static void main(String[] args) {
        Game g = new Game();
        Cricket c = new Cricket();
        Badminton b = new Badminton();
        Tennis t = new Tennis();
        Scanner s = new Scanner(System.in);
        System.out.print("Please Enter name of the game = ");
        String op = s.nextLine();
        if (op.equals("cricket")) {
                g = c;
        } else if (op.equals("badminton")) {
                g = b;
        } else

        }
        g.type(
    }
}
```

C:\WINDOWS\system32\cmd.exe

```
D:\DegreeDemo\PPTDemo>javac MyProg.java
D:\DegreeDemo\PPTDemo>java MyProg
Please Enter name of the game = tennis
Mix game
```

Dynamic Method Dispatch (Conclusion)

- When an overridden method is called through a superclass reference, Java determines which version(superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs.
- Thus, this determination is made at run time.

Static v/s Dynamic Binding

Static Binding	Dynamic Binding

Understanding System.out.println()

- **System** is a **class** in java which is in the **java.lang** package
- **out** is a **static member** of PrintStream in the System class.
- **println()** is a **method** of PrintStream Class

Programs

- The abstract Vegetable class has three subclasses named Potato, Brinjal and Tomato. Write a java prog. That demonstrates how to establish this class hierarchy. Declare one instance variable of type String that indicates the color of a vegetable. Crete and display instances of these objects. Override the toString() method of object to return a string with the name of vegetable and its color.
- Declare a class called Book having book title & author name as members. Create a sub-class of it, called BookDetails having price & current stock of book as members. Create an array for storing details of n books. Define methods to achieve following: - Initialization of members - To query availability of a book by author name / book title - To update stock of a book on purchase and sell Define method main to show usage of above methods.