# Inheritance & 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.

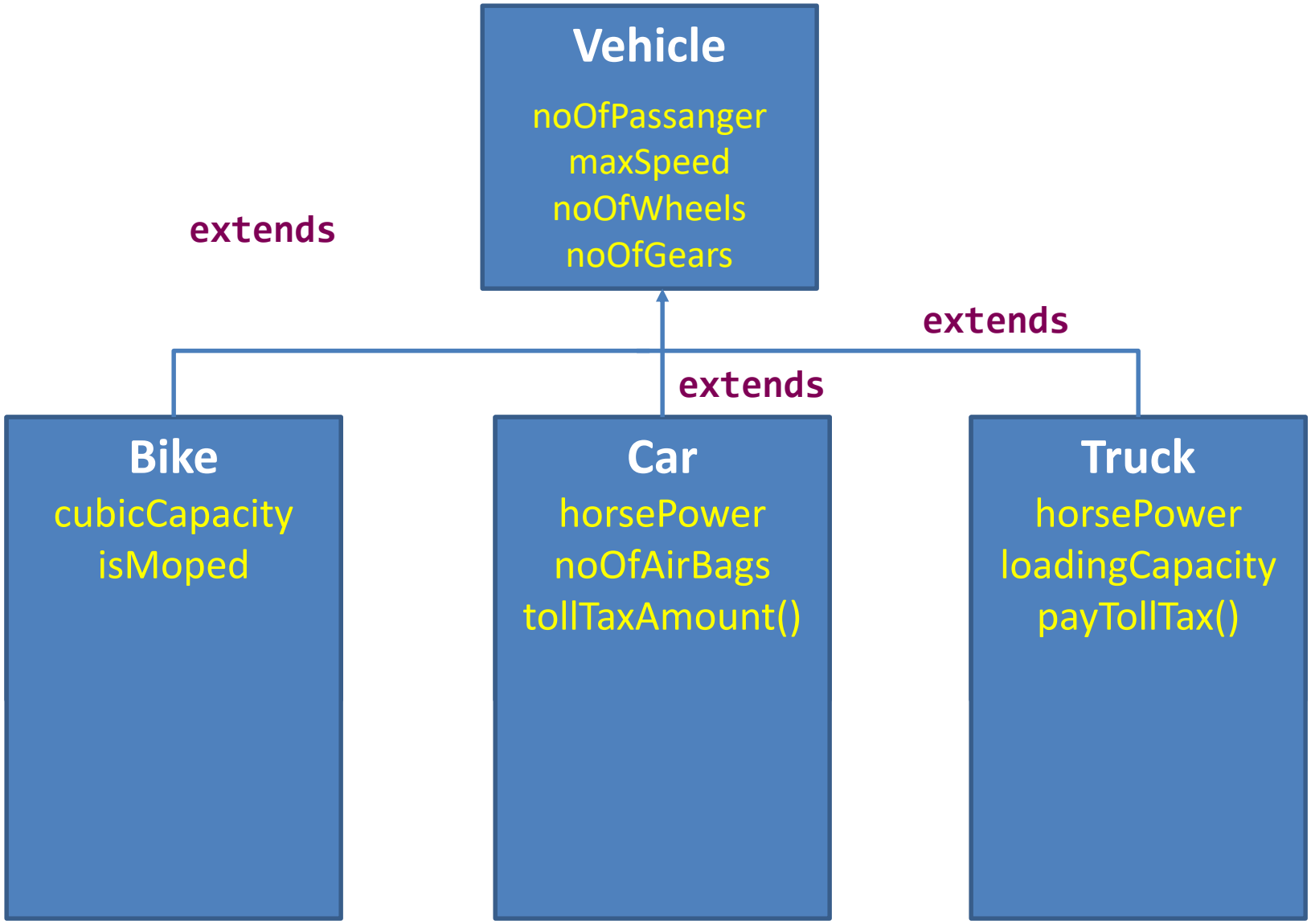
**Syntax :**

```
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)

```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 Car();
        c.maxSpeed = 200;
        c.noOfPassanger = 5;
        c.horsePower = 1.2;
        c.noOfAirbags = 2;
        System.out.println("---- Car ----");
        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
```

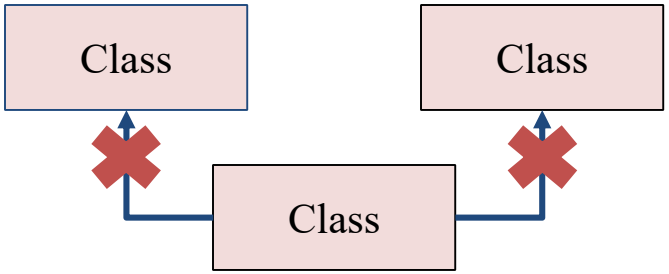
# Inheritance (Cont.)

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

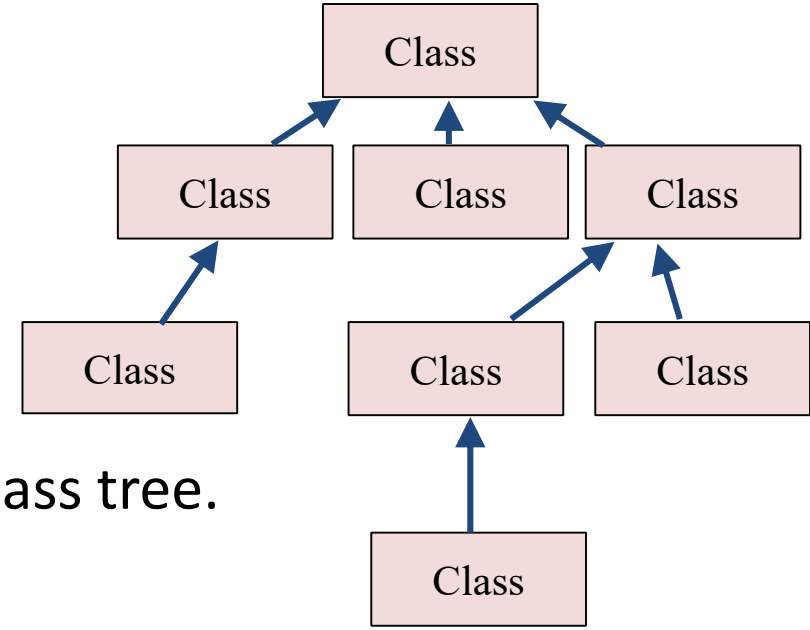
  **C++ allows** multiple inheritance

  **BUT**

  **Java** does **not support** multiple inheritance



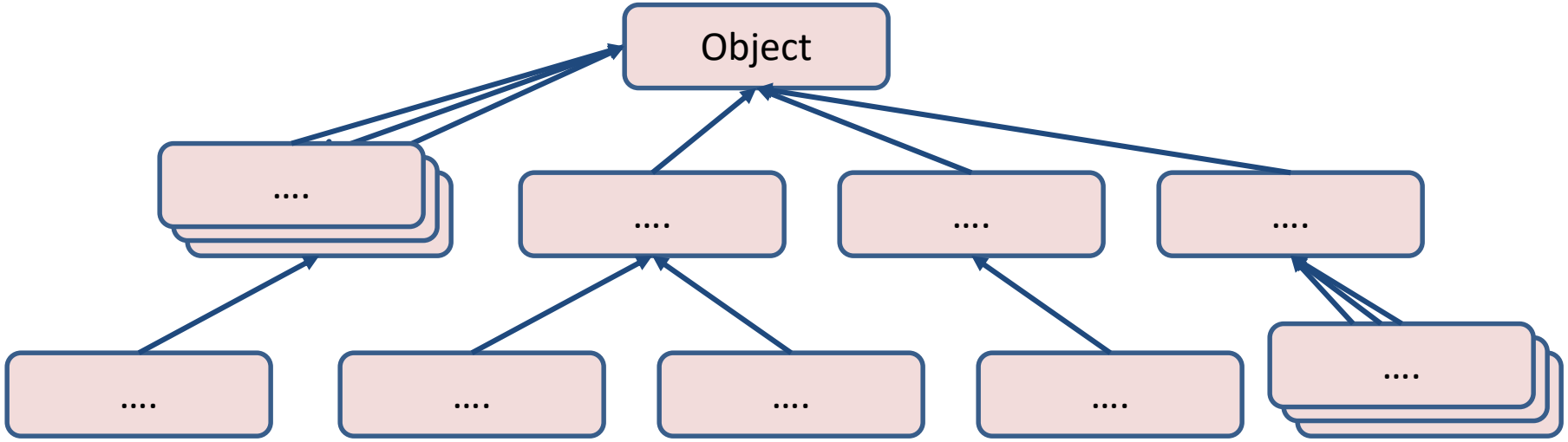

- There is **no limit** to the **number of subclasses** a class can have
- Inheritance creates a **class hierarchy**
  - Classes **higher** in the **hierarchy** are **more general** and **more abstract**
  - Classes **lower** in the **hierarchy** are **more specific** and **concrete**




- 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();
   }
}
```

# Abstraction

- As per dictionary, **abstraction** is the quality of dealing with **ideas** rather than **events**.
- For example, when you consider the case of **WhatsApp**, complex details such as what happens as soon as you send message and the protocol WhatsApp uses are hidden from the user. Therefore, to send message you just need to select the receiver, type the content and click send.

# Abstraction in Java

- Likewise in Object-oriented programming, **abstraction** is a **process of hiding** the **implementation details** from the **user**, only the functionality will be provided to the user.
- In other words, the user will have the information on what the object does instead of how it does it.
- **Abstraction** is achieved using **Abstract classes** and **interfaces**.

# Abstract Class

- A class which contains the abstract keyword in its declaration is known as abstract class.
  - Abstract classes **may or may not** contain **abstract methods**, i.e., methods without body ( public void get(); )
  - But, if a class has **at least one** abstract method, then the class must be declared **abstract**.
  - If a class is declared abstract, it **cannot** be instantiated.
  - To use an abstract class, you have to inherit it to another class and provide **implementations** of the abstract methods in it.

# Abstract Class (Example)

```java
abstract class Car {
        public abstract double getAverage();
}
class Swift extends Car{
        public double getAverage(){
                return 22.5;
        }
}
class Baleno extends Car{
        public double getAverage(){
                return 23.2;
        }
}

public class MyAbstractDemo{
        public static void main(String ar[]){
            Swift b1 = new Swift();
            Baleno b2 = new Baleno();
            System.out.println(b1.getAverage());
            System.out.println(b2.getAverage());
        }
}
```

File Name
MyAbstractDemo.java

C:\WINDOWS\system32\cmd.exe

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

D:\DegreeDemo\PPTDemo>java MyAbstractDemo
22.5
23.2
```

# 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();
}
```

Variable in interface are by default public, static, final

```java
class CarClass implements
VehicalInterface
{
    public void turnLeft() {
    // Code to turn left
    }
    public void turnRight() {
    // Code to turn right
    }
    public void accelerate() {
    // Code to accelerate
    }
    public void slowDown() {
    // Code for break
    }
}
```

# 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)

```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);
        String op = s.nextLine();
        if (op.equals("cricket")) {
                g = c;
        } else if (op.equals("badminton")) {
                g = b;
        } else if (op.equals("tennis")) {
                g = t;
        }
        g.type();
    }
}
```

# 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 |
| --- | --- |

# Puzzle

```
public class Rowboat _________ _________ {
   public ___________ rowTheBoat() {
      System.out.print("stroke natasha");
   }
}
```

```
public class _________ {
   private int __________ ;
   _______ void _________ ( ______ ) {
      length = len;
   }
   public int getLength() {
      ________ _________ ;
   }
   public ___________ move() {
      System.out.print("___________");
   }
}
```

```
public class TestBoats {
   ______ ______ _______ main(String[] args){
      _________ b1 = new Boat();
      Sailboat b2 = new __________();
      Rowboat ________ = new Rowboat();
      b2.setLength(32);
      b1.__________();
      b3.__________();
      _______.move();
   }
}
```

```
public class __________ ________ Boat {
   public _______ _________() {
      System.out.print("___________");
   }
}
```

**OUTPUT:** drift drift hoist sail

# Package

# Use of Package

- Packages are used in Java in order to prevent **naming conflicts**, to **control access**, to make **searching/locating** and usage of classes, interfaces easier.
- A Package can be defined as a grouping of related types providing access protection and name space management.
- Programmers can define their own packages to bundle group of classes/interfaces, etc.
- It is a good practice to group related classes implemented by you so that a programmer can easily determine that the classes, interfaces are related.

# Creating a package

- When creating a package, you should choose a name for the package and put a package statement with that name at the top of every source file that contains the classes/interfaces that you want to include in the package.
- The package statement should be the first line in the source file.
- There can be only one package statement in each source file, and it applies to all types in the file.
- If a package statement is not used then the class, interfaces, enumerations, and annotation types will be put into an unnamed package.

# Example (Package)

```java
package myPackage;
public class Animal {
    public String name;
    public void eat(){
        System.out.println("Organic Food !!!!");
    }
    public static void main(String[] ar){
        Animal a = new Animal();
        a.eat();
    }
}
```

- To compile

  javac **–d** **.** Animal.java

  .

  Represent the current directory

- To Run the class file

  java **myPackage.Animal**

# Additional points on package

- A package is always defined in a separate folder having the same name as a package name.
- Define all classes in that package folder.
- All classes of the package which we wish to access outside the package must be declared public.
- All classes within the package must have the package statement as its first line.
- All classes of the package must be compiled before use (So that its error free)

# “import” keyword

- import keyword is used to import built-in and user-defined packages into your java source file so that your class can refer to a class that is in another package by directly using its name.
- There are 3 different ways to refer to class that is present in different package
  - Using fully qualified name(But this is not a good practice.)
  - import the only class you want to use(Using packagename.classname)
  - import all the classes from the particular package(Using packagename.*)

# Static Import

- The static import feature of Java 5 facilitate the java programmer to access any static member of a class directly. There is no need to qualify it by the class name.
- Advantage of static import:

  Less coding is required if you have to access any static member of a class more frequently.
- Disadvantage of static import:

  If you overuse the static import feature, it makes the program unreadable and unmaintainable

# Example (static import)

```
import static java.lang.System.out;
// import static java.lang.System.*
public class S2{
  public static void main(String args[]){
        out.println("Hello main");
  }
}
```

We need not to write System.out as we have imported the out statically

# Access Control