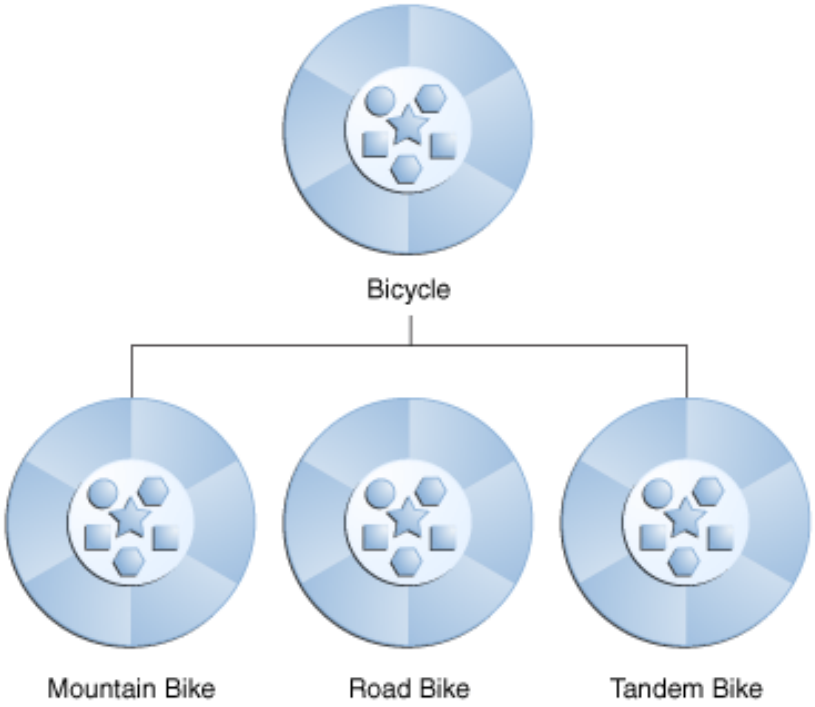


Week 7: Inheritance and Polymorphism

Overview

- Inheritance
- Method Overloading
- Inheritance in Java
- IS-A Relationship
- The *super* keyword
- Overriding Attributes and Operations
- HAS-A Relationship
- Polymorphism

Inheritance

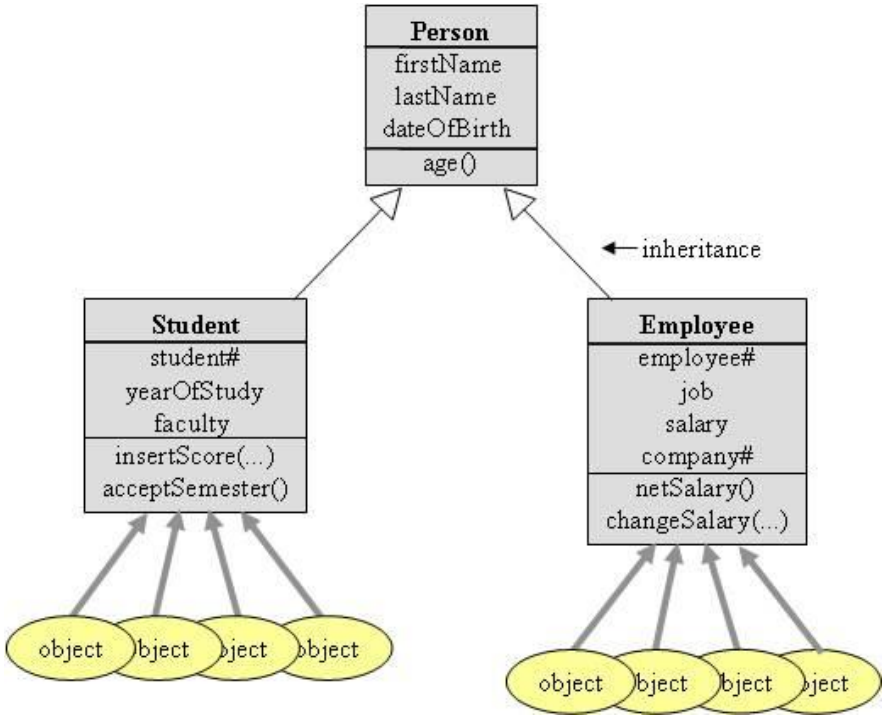
- Inheritance is the mechanism for implementing generalisation and specialization in an object-oriented programming language.

- It refers to the mechanism of sharing attributes and operations using the generalisation relationship.

- **Inheritance is a relationship among classes wherein one class shares the structure or behavior defined in one (single inheritance) or more (multiple inheritance) other classes.** [Booch]

Inheritance

- A subclass inherits all the characteristics of all its ancestors not only its immediate superclass.
- The definition of a subclass always includes at least one detail not derived from its superclass.
- Inheritance defines an is-a hierarchy among classes, in which a subclass inherits from one or more superclasses.
- A subclass typically augments (inheritance for extension) or restricts (inheritance for restriction) the existing structure and behaviour of its superclasses.



Inheritance

- We use ***access specifiers*** to define an interface for subclasses. I.e. Use public, private, protected permissions in the class definition to determine which features can be inherited or not.

- We use ***modifiers*** to define an interface for instances. E.g. static variables are not replicated for each instance.

- **Benefits of Inheritance:**

  - Facilitates modeling by capturing what is similar and what is different about classes.

  - Helpful during implementation as a vehicle for reusing code. –After modeling a system, a developer looks at the resulting classes, tries to group similar classes together and reuse common code (past class libraries).

Methods Overloading

- In Java, it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different.
- When this is the case, the methods are said to be *overloaded* and the process is referred to as *method overloading*.
- Method overloading is one of the ways that Java implements polymorphism.
- Two rules apply to overload method:
  - Arguments lists must differ
  - Return types can be different

Methods Overloading

- When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Therefore, the overloaded methods must differ in the type and/or number of arguments.

```
class Shape {
double area;
static final double PI=3.14;

double calArea(double radius) {
            area = PI*radius*radius;
            return area;
}

double calArea(double length, double breath) {
            area = length * breath;
            return area;
} }

public class ShapeArea {
public static void main(String args[]) {
Shape shape1 = new Shape();
Shape shape2 = new Shape();

System.out.println(shape1.calArea(10.0));
System.out.println(shape2.calArea(10.0, 5.2 ));
 } }
```

Inheritance in Java

- Sub-classing is a way to create a new class from an existing class. A subclass inherits all methods and variables from the superclass (parent class).
- A subclass does not inherit the constructor from the superclass. The only way for a class to gain a constructor is by either writing it explicitly or using the default constructor.
- To inherit a class, use the *extends* keyword.

```
public class Employee {
String name, id, jobtitle;
double salary;
.
.
}

public class Manager extends Employee {
String dept;
.
.
}
```

IS-A Relationship

```
public class Animal{
}

public class Mammal
extends Animal{
}

public class Reptile
extends Animal{
}
```

```
public class TestAnimal {
   public static void main(String args[]){

      Animal a = new Animal();
      Mammal m = new Mammal();
      Reptile r = new Reptile();

      System.out.println(m instanceof Animal);
      System.out.println(r instanceof Animal);
      System.out.println(r instanceof Object);
      System.out.println(a instanceof Mammal);
   }
}
```

This would produce the following result:

```
true
true
true
false
```

The *super* keyword

- *super* is used in a class to refer to the member variables of its superclasses.
- Superclass behaviour is invoked as if the object was part of the superclass. The behaviour invoked does not have to be in the superclass, it can be further up in the hierarchy.
- *super* has two general forms:
  - using *super* to call superclass constructors
  - using *super* to access a member (either a method or an instance variable) of the superclass that has been hidden by a member of a subclass.

```
class A {
        int x;
}

class B extends A {
        int x;

        B(int a, int b) {
           super.x = a;     // x in A
           x = b;           // x in B
}

void show() {
   System,out.println("x in superclass:" + super.x);
   System,out.println("x in subclass:" + x);
} }

class C {
   public static void main (String args[]) {
      B subObject = new B(1,5);
      subObject.show();
} }
```

Overriding Attributes and Operations

- A subclass can modify behaviour inherited from a parent class.
- A subclass may override a superclass feature by defining a feature with the same name. The overriding feature (the subclass feature) refines and replaces the overridden feature (the superclass feature).
- Reasons for overriding:
  - To specify behaviour that depends on the subclass
  - To improve performance by taking advantage of information specific to the subclass but not changing the semantics of the operation overridden
- Subclass can create a method in a subclass with a different functionality than the parent's method but with the same
  - Name
  - Return type
  - Argument list

Overriding Attributes and Operations

- **There are four types of methods that you cannot override in a subclass:**
  - private methods
  - static methods
  - final methods
  - methods within final classes

- **Rules about Overridden methods**
  - Must have a return type that is identical to the method it overrides
  - Cannot be less accessible than the method it overrides
  - Cannot throw more exceptions than the method it overrides

- **Why overridden methods**
  - Overridden methods allow Java to support run-time polymorphism. Dynamic, run-time polymorphism is one of the most powerful mechanisms that object-oriented design brings to bear on code reuse and robustness.

Overriding Attributes and Operations

```
public class Employee {
String name;
int salary;

public Employee(String name, int
salary){
  this.name=name;
  this.salary=salary; }

public String printDetails() {
return name + “is paid “+ salary;
}
}
```

```
public class Manager extends
Employee {
String dept;
public Manager(String name, int salary,
String dept){
  super(name,salary);
  this.dept=dept; }

public String printDetails() {
 return name + “is the manager of “ +
dept;
}
}
```

Overriding Attributes and Operations

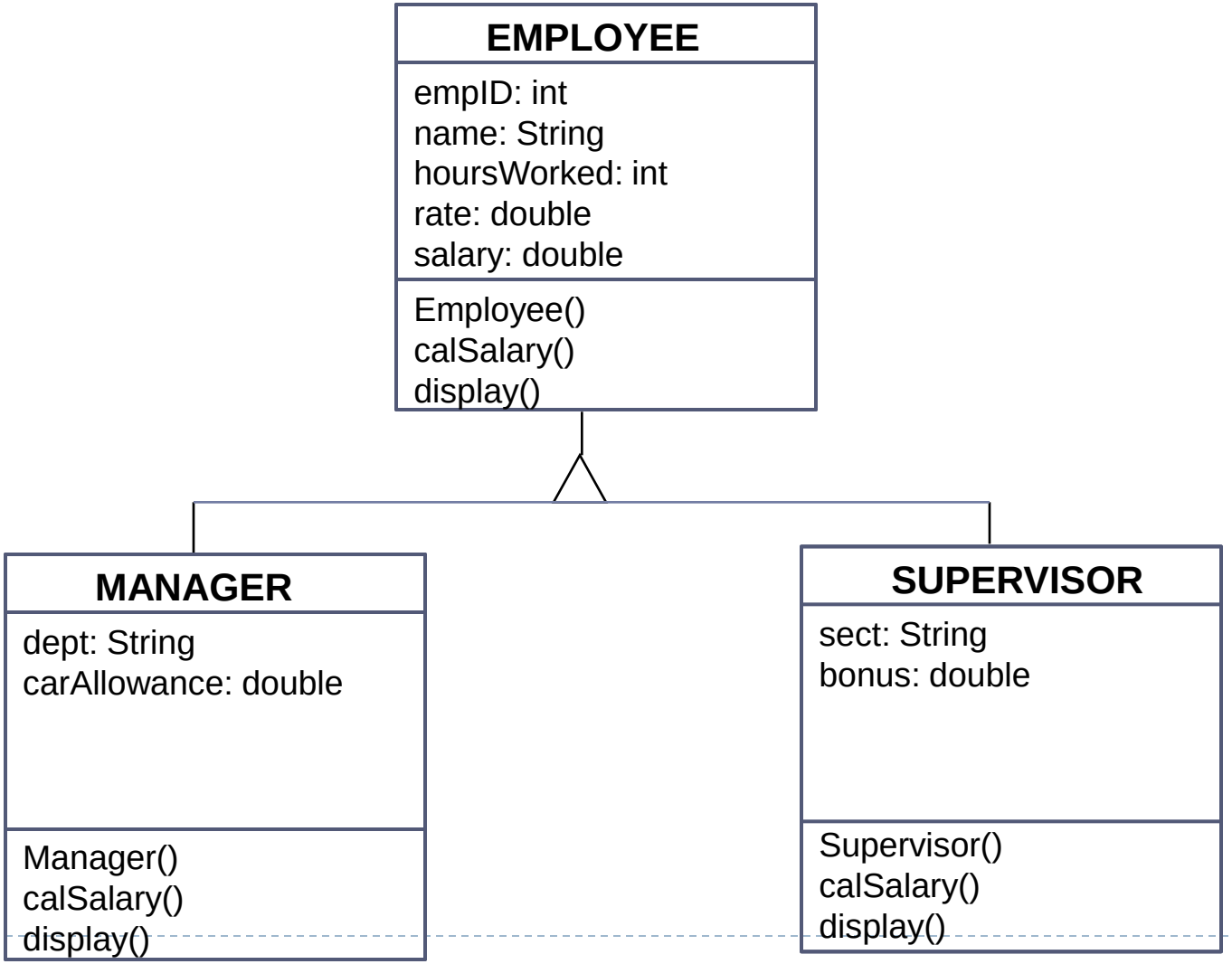
```
public class TestEmployee {
public static void main(String args[]) {

   Employee emp = new Employee("John", 10000);
   Manager man = new Manager("Jim", 20000, "Marketing");

   System.out.println(emp.printDetails());
   System.out.println(man.printDetails());
} }
```

Output:

John is paid 10000

Jim is the manager of Marketing

Inheritance Example



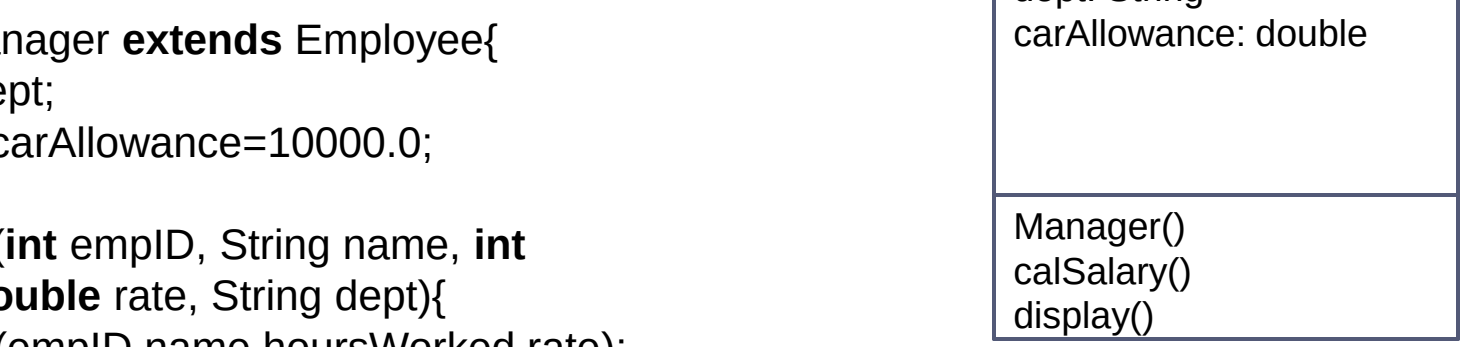
Inheritance Example

EMPLOYEE
empID: int name: String hoursWorked: int rate: double salary: double
Employee() calSalary() display()

```
public class Employee {
private int empID;
private String name;
private int hoursWorked;
private double rate;
protected double salary=0.0;

public Employee(int empID, String name, int hoursWorked,
double rate){        this.empID = empID;
                     this.name = name;
                     this.hoursWorked = hoursWorked;
                     this.rate = rate; }

public double calSalary(){
            salary = hoursWorked*rate;
            return salary; }

public void display(){
            System.out.println("Employee ID:"+empID);
            System.out.println("Employee Name:"+name);
            System.out.println("Employee Salary:"+salary);
} }
```

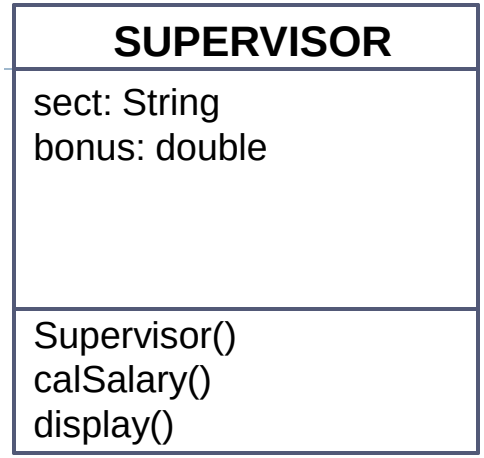
Inheritance Example

```
public class Manager extends Employee{
private String dept;
private double carAllowance=10000.0;

public Manager(int empID, String name, int
hoursWorked, double rate, String dept){
        super(empID,name,hoursWorked,rate);
        this.dept=dept;
}

public double calSalary(){
        salary =    super.calSalary()+carAllowance;
        return salary;
}

public void display(){
        super.display();
        System.out.println("Department : "+dept);
}
}
```



Inheritance Example

```
public class Supervisor extends Employee {
private String sect;
private double bonus = 2000.0;

Supervisor(int empID, String name, int hoursWorked,
double rate, String section){
        super(empID,name,hoursWorked,rate);
        this.sect=section;
}

public double calSalary(){
        salary = super.calSalary()+bonus;
        return salary;
}

public void display(){
        super.display();
        System.out.println("Section: "+sect);
}

}
```



Inheritance Example

```
public class TestEmployee {

        public static void main(String[] args) {
                Employee emp[] = new Employee[5];

                emp[0]= new Employee(100, "John", 38, 300.0);
                emp[1]= new Manager(200, "Paul", 30, 500.0, "Marketing");
                emp[2]= new Manager(300, "Anil", 32, 600.0, "Production");
                emp[3]= new Supervisor(400, "Jenny", 36, 400.0, "Finishing");
                emp[4]= new Supervisor(100, "Jane", 37, 350.0, "Making");

                for(int i=0;i<5;i++){
                        emp[i].calSalary();
                        emp[i].display();
                        System.out.println();
                }
        }
}
```

HAS-A relationship

- Has-A means an instance of one class "has a" reference to an instance of another class or another instance of same class.
- It is also known as "composition" or "aggregation".
- There is no specific keyword to implement HAS-A relationship but mostly we are depended upon "new" keyword.

```
public class Vehicle {  }

public class Speed {  }

public class Van extends Vehicle{
        private Speed sp;
}
```

Exercise

- A company employs two types of worker. One, a salaried worker, gets paid a fixed weekly amount, while the other, an hourly worker, is paid according to the number of hours worked. The system identifies each worker by an identity number, and it also stores their surname. The company wishes to develop a payroll system which can be used to compute the total weekly wages bill.

- Develop a solution to this problem using inheritance. Your solution should contain:

1. Definitions of whatever classes are necessary. Provide a constructor, a method for computing the weekly wage, and a display method in each case. You do not have to provide set and get methods unless they are required to support the payroll computation.

2. A test program which sets up an array of employees and prints out a line for each employee together with the total weekly wages bill. Show what output you expect to obtain from your program.

Polymorphism

- **Polymorphous -** Having, or assuming, various forms, characters, or styles. (From Webster's Collegiate Dictionary, fifth edition).
- In computer science, polymorphism allows us to send identical messages to dissimilar but related objects and achieve identical actions while letting the software system determine how to achieve the required action for the given object.

- **Polymorphism in Programming Languages**
  - A variable that can hold, at different times, values of many different types.
  - A name associated with several different function bodies.
  - A single function that has at least one argument which is a polymorphic variable.

Polymorphism

- **Forms of Polymorphism**
  - Polymorphic variables
  - Pure polymorphism (functions with polymorphic arguments).
  - overloading
  - overriding
  - deferred methods
  - generics or templates

Polymorphism

- **Polymorphic Variables**
  - We have noted already how a variable declared as an instance of one class can, in fact, hold a *value* from a subclass type.
  - Dynamic (run-time) type must be subclass of static (declared) type.
  - In C++ only works with pointers and references.
  - In Java can be formed from class or interface types.
- **Overloading**
  - A single function *name* which is used to denote two or more function *bodies* is said to be **overloaded**.
  - A common example occurs in almost all languages, where + means both integer and floating addition.
  - Determination of which function to execute is made by examining the arguments.
  - Depending upon language and form used, can either be made at compile-time or at run-time.
  - Should not be confused with overriding or refinement - functions need not be related by classes.

Polymorphism

- **Type Signatures**
  - Selection of which procedure to execute is made based on types of arguments.
  - We say the *type signature* determines the procedure to execute.
- **Overriding**
  - As described earlier, overriding occurs when a child class changes the meaning of a function originally defined in the parent class.
  - Different child classes can override in different ways.
  - Parent class can have default behavior, child classes alternatives.
  - Contributes to code sharing and ensures common interfaces.

Polymorphism

- **Deferred Methods**
  - A deferred method is a generalization of overriding.
  - Parent class names method, but provides no implementation.
  - The child class *must* provide an implementation.
  - Usually combined with other techniques.

Polymorphism

- **Binding**
  - Binding is the process of connecting a method call to a method body. It is the association between an operation and an entity. It can happen either at compile time or at run time.
- **Static Binding/Early binding**
  - Binding is performed before the program runs – Compile time.
  - Method calls are bound to specific computer memory locations where methods are placed at compile time.
  - Leads to run-time efficiency since the compiler can optimize the code before executing it.
  - Function overloading implements static binding.
  - Object is going to invoke which overloaded method is decided at compile time by looking at the method signature.

Polymorphism

- **Dynamic Binding**
  - Binding is performed at the time of execution – Run time.
  - The method called depends upon the type of the object and not the type of expression.
  - The type cannot be resolved at compile time – method to be used is resolved at run-time.
  - Method overriding implements dynamic binding.

Polymorphism Example 1

```
class Shape {
void draw() {}
void erase() {}
}

class Circle extends Shape {
void draw() {
System.out.println("Circle.draw()");
}
void erase() {
System.out.println("Circle.erase()");
}
```

```
class Square extends Shape {
void draw() {
System.out.println("Square.draw()");
}
void erase() {
System.out.println("Square.erase()");
}
}

class Triangle extends Shape {
void draw() {
System.out.println("Triangle.draw()");
}
void erase() {
System.out.println("Triangle.erase()");
}
}
```

Polymorphism Example 1

```
public class Shapes {
public static Shape randShape() {
    switch((int)(Math.random() * 3)) {
            default: // To quiet the compiler
            case 0: return new Circle();
            case 1: return new Square();
            case 2: return new Triangle();
}
}
public static void main(String args[]) {
Shape s[] = new Shape[9];          // Fill up the array with shapes:
for(int i = 0; i < s.length; i++)
            s[i] = randShape();          // Make polymorphic method calls:
for(int i = 0; i < s.length; i++)
            s[i].draw();
}
}
```

Polymorphism Example 2

```
public class BaseClass
{
    public void methodToOverride() //Base class method
    {
        System.out.println ("I'm the method of BaseClass");
    }
}
public class DerivedClass extends BaseClass
{
    public void methodToOverride() //Derived Class method
    {
        System.out.println ("I'm the method of DerivedClass");
    }
}
```

Polymorphism Example 2

```
public class TestMethod
{
    public static void main (String args []) {
      // BaseClass reference and object
      BaseClass obj1 = new BaseClass();
      // BaseClass reference but DerivedClass object
      BaseClass obj2 = new DerivedClass();
      // Calls the method from BaseClass class
      obj1.methodToOverride();
      //Calls the method from DerivedClass class
      obj2.methodToOverride();
    }
}
```

Output:

```
I'm the method of BaseClass
I'm the method of DerivedClass
```

Polymorphism Example 3

When invoking a superclass version of an overridden method the super keyword is used.

```
class Vehicle {
    public void move () {
        System.out.println ("Vehicles are used for moving from one place to another ");   } }

class Car extends Vehicle {
    public void move () {
      super. move (); // invokes the super class method
      System.out.println ("Car is a good medium of transport ");   } }

public class TestCar {
    public static void main (String args []){
       Vehicle b = new Car (); // Vehicle reference but Car object
        b.move (); //Calls the method in Car class  } }
```

Output: Vehicles are used for moving from one place to another
Car is a good medium of transport