Chapter 9: Inheritence

TASNIM SHARMIN ALIN

LECTURER

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEEING

LEADING UNIVERSITY,SYLHET

Inharitance

- **Inheritance** is a mechanism in which one object acquires all the properties and behaviours of parent object.

- **Superclass** :A class that is **inherited** is called as **super class** or **parent class**.

- **Subclass:** A class that does the **inheriting** is called **sub class** or **child class.**

- A class that is derived from another class is called **a *subclass*** (also a *derived class*, *extended class*, or *child class*). The class from which the subclass is derived is called **a *superclass***

Inheritance examples

Superclass	Subclasses
Student	GraduateStudent, UndergraduateStudent
Shape	Circle, Triangle, Rectangle
Loan	CarLoan, HomeImprovementLoan, MortgageLoan
Employee	Faculty, Staff
BankAccount	CheckingAccount, SavingsAccount

Fig. 9.1 Inheritance examples.

- **Why inheritance?**

- For Method Overriding (So Runtime Polymorphism).
- In order to implement tree like structure
- To implement parent child relationship
- It makes possible the reusability of already existing class
- It makes possible for a child to inherit all the properties of parents

- If we want to inherit a class, we simply use **extends** keyword.

- **The keyword extends indicates that you are making a new class that derives from an existing class.**
- The general form of inheritance would be:

```
class subclass-name extends superclass-name
 {
   /*body of class*/
 }
```


```
class Employee{
   float salary = 30000f;
}
public class Programmer extends Employee {
    float bonus = 20000f;
   public static void main(String[] args) {
      Programmer p = new Programmer();
      System.out.println("Bonus of programmer is " + p.bonus);
      System.out.println("Salary of programmer is " + p.salary);
   }
}
```

Note: Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

```
class Vehical{
   String color;
   int speed;
   int size;
   void attribute(){
      System.out.println("Color: "+ color);
      System.out.println("Speed: "+ speed);
      System.out.println("Size: "+ size);
   }
}
class Car extends Vehical{
   int cc;
   int gear;
   void attributeCar(){
      System.out.println("Color: "+ color);
      System.out.println("Speed: "+ speed);
      System.out.println("Size: "+ size);
      System.out.println("CC: "+ cc);
      System.out.println("gear: "+ gear);
   }
}
```


```
public class VehTest extends Car{
    public static void main(String[] args) {
      Car ob = new Car();
      ob.color = "Blue";
      ob.size=22;
      ob.speed=200;
      ob.cc=1000;
      ob.gear=5;
      ob.attributeCar();
    }
}
```

Output:

Color: Blue

Speed: 200

Size: 22

CC: 1000

gear: 5

```
class Vehical{
    String color;
    private int speed;
    private int size;
    public int getSize(){
        return size;
    }
     public int getSpeed(){
        return speed;
    }
    public void setSize(int size){
        this.size = size;
    }
    public void setSpeed(int speed){
        this.speed = speed;
    }
}
```


```
class Car extends Vehical{
    int cc,gear;
    String color;
}

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

        Car ob = new Car();
        ob.color = "Blue";
        ob.setSpeed(200) ;
        ob.setSize(22);
        ob.cc = 1000;
        ob.gear = 5;
```


```
    System.out.println("Color of Car : " + ob.color);
     System.out.println("Speed of Car : " + ob.getSpeed());
     System.out.println("Size of Car : " + ob.getSize());
     System.out.println("CC of Car : " + ob.cc);
     System.out.println("No of gears of Car : " + ob.gear);

    }
}
```

Output:

Color of Car : Blue

Speed of Car : 200

Size of Car : 22

CC of Car : 1000

No of gears of Car : 5

Access Modifiers

- There are two types of modifiers in java: **access modifier** and **non-access modifier**.
- **The basic Accessibility Modifiers are of 4 types in Java. They are**

- public
- protected
- package/default
- Private

- **The non-access Modifiers in Java are**

- static
- abstract
- final
- synchronized
- transient
- native
- volatile

`protected` Members

Protected Members:

If members are declared as protected then these are accessible to all classes in the package and to all subclasses of its class in any package where this class is visible.

- `protected` access
  - Intermediate level of protection between `public` and `private`
  - `protected` members accessible to
    - superclass members
    - subclass members
    - Class members in the same package
  - Subclass access superclass member
    - Keyword `super` and a dot (.)

Method Overriding

- Declaring a method in **subclass** which is already present in **parent class** is known as method overriding.

- In other words, If subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as Method Overriding.

Method overriding


```
class Human{
      public void eat() {
            System.out.println("Human is eating");
     }
 }
 class Boy extends Human{
   public void eat() {
       System.out.println("Boy is eating");
  }
public static void main( String args[]) {
     Boy obj = new Boy();
     obj.eat();
   }
}
```

Output: Boy is eating

Method overriding

```
class Animal{
   public void move(){
      System.out.println("Animals can move"); }
}
class Dog extends Animal{
   public void move(){
      System.out.println("Dogs can walk and run");    }
}
public class TestDog {
    public static void main(String[] args) {
      Animal a = new Animal();      / / Animal reference and object
      Animal b = new Dog();         // Animal reference but Dog object
      a.move();                     / / runs the method in Animal class
      b.move();                     //Runs the method in Dog class
    }
}
```

Output:

Animals can move

Dogs can walk and run

❑ Rules for Method Overriding

- method must have same name as in the parent class
- method must have same parameter as in the parent class.
- must be IS-A relationship (inheritance).

❑ Advantage of Java Method Overriding

- Method Overriding is used to provide specific implementation of a method that is already provided by its super class.
- Method Overriding is used for Runtime Polymorphism

❑ Can we override java main method?

No, because main is a static method.

❑ Why we cannot override static method?

because static method is bound with class whereas instance method is bound with object. Static belongs to class area and instance belongs to heap area.

Difference between method overloading and method overriding

Method overloading	Method Overriding
1) Method overloading is used to increase the readability of the program.	**1)** Method overriding is used to provide the specific implementation of the method that is already provided by its super class.
2) method overloading is performed within a class.	**2)** Method overriding occurs in two classes that have IS-A relationship.
3) In case of method overloading parameter must be different.	**3)** In case of method overriding parameter must be same.

super keyword

- The **super** is a reference variable that is used to refer immediate parent class object.

- **Usage of super Keyword**

- super is used to refer immediate parent class instance variable.
- super() is used to invoke immediate parent class constructor.
- super is used to invoke immediate parent class method.

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

```
class Animal{
  public void move(){
    System.out.println("Animals can move");
  }
}
class Dog extends Animal{
  public void move(){
    super.move();    // invokes the super class method
    System.out.println("Dogs can walk and run");
  }
}
public class DogSuper {
   public static void main(String[] args) {
    Animal b = new Dog(); // Animal reference but Dog object
    b.move(); //Runs the method in Dog class
   }
}
```


super can be used to invoke parent class method.


```
class Person{
   void message(){
      System.out.println("Welcome");  }
}
public class Student extends Person{
     void message(){
        super.message();      //will invoke parent class message() method
        System.out.println("Java programming");
     }
   public static void main(String[] args) {
      Student S = new Student();
      S.message();     //will invoke current class message() method
   }
}
```

Output: **Welcome**
Java programming

super is used to invoke parent class constructor.

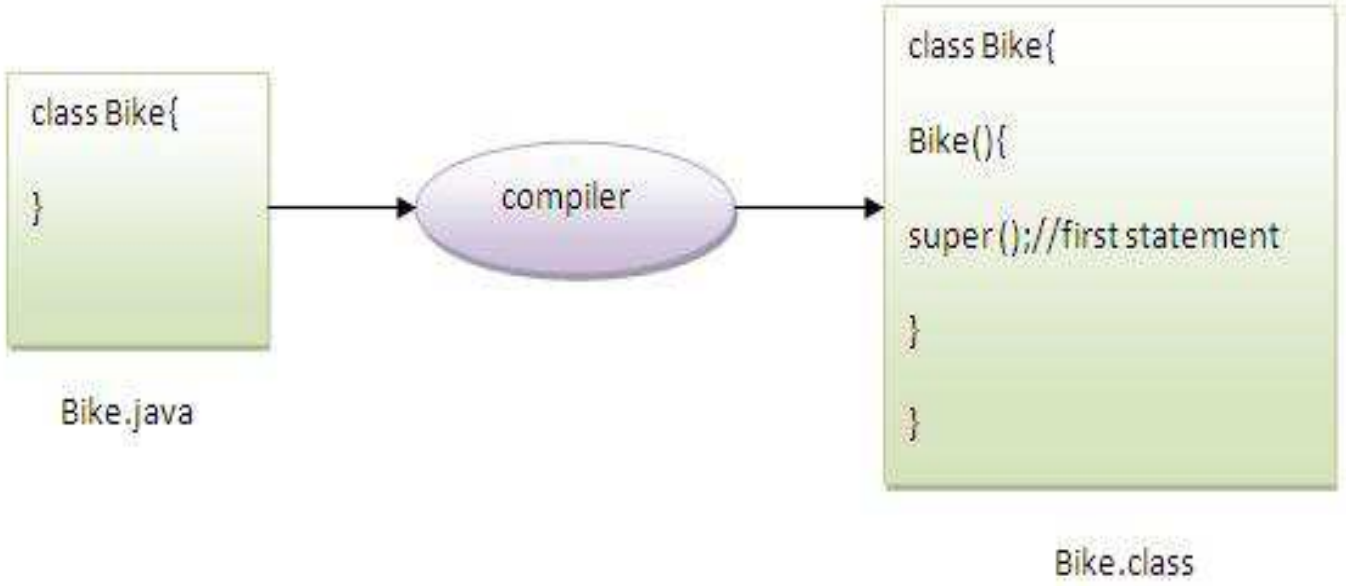

```
class Vehical{
   Vehical(){
      System.out.println("Vehical is created");  }
}
public class Bike extends Vehical {
   Bike(int speed){
      System.out.println(speed);
   }
   public static void main(String[] args) {
      Bike ob =  new Bike(20);
   }
}
```

Output:

Vehical is created

20

super is used to invoke parent class constructor



Super Key

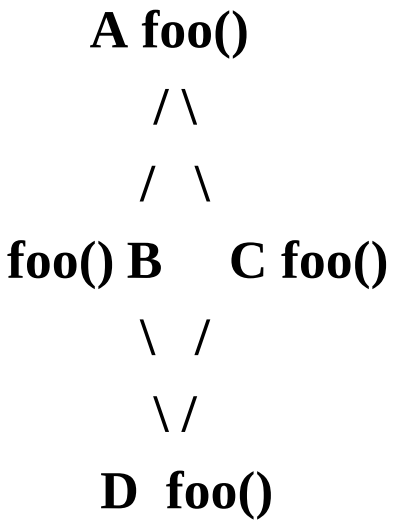
- The default constructor is provided by compiler automatically but it also adds super() for the first statement. If you are creating your own constructor and you don't have either this() or super() as the first statement, compiler will provide super() as the first statement of the constructor.

Types of Inheritence

- “is-a” relationship
- Single inheritance:
  - Subclass is derived from one existing class (superclass)
- Multiple inheritance:
  - Subclass is derived from more than one superclass
  - Not supported by Java
  - A class can only extend the definition of one class

1. **First reason is ambiguity around Diamond problem.**

   Consider a class A has **foo()** method and then B and C derived from A and has there own **foo()** implementation and now class D derive from B and C using multiple inheritance and if we refer just **foo()** compiler will not be able to decide which **foo()** it should invoke. This is also called Diamond problem because structure on this inheritance scenario is similar to 4 edge diamond, see below



2. **Multiple inheritances does complicate the design and creates problem during casting, constructor chaining etc.**

```
class Car{
    public Car(){
        System.out.println("Class Car");
    }
    public void vehicleType(){
        System.out.println("Vehicle Type: Car");
    }
}
class Maruti extends Car{
    public Maruti(){
        System.out.println("Class Maruti");
    }
    public void brand(){
        System.out.println("Brand: Maruti");
    }
    public void speed(){
        System.out.println("Max: 90Kmph");
    }
}
```


Multilevel inheritence


```
public class Maruti800 extends Maruti{
      public Maruti800() {
              System.out.println("Maruti Model: 800"); }
     public void speed(){
             System.out.println("Max: 80Kmph");  }
   public static void main(String[] args) {
      Maruti800 obj=new Maruti800();
              obj.vehicleType();
              obj.brand();
              obj.speed();
   }
}
```

Output:

Class Car

Class Maruti

Maruti Model: 800

Vehicle Type: Car

Brand: Maruti

Max: 80Kmph

Multilevel inheritence

```
class A{
void msg(){
 System.out.println("Hello");}
}
class B{
void msg(){
System.out.println("Welcome"); }
}

class C extends A,B{        //suppose if it were
 Public Static void main(String args[]){
  C obj=new C();
   obj.msg();  //Now which msg() method would be invoked?
}
}
```


The object Class

- The **Object class** is the parent class of all the classes in java by default. In other words, it is the topmost class of java.

- The Object class provides some common behaviors to all the objects such as object can be compared, object can be cloned, object can be notified etc.

Methods of Object class

- **clone**() Creates a new object of the same class as this object.
- **equals**(Object) Compares two Objects for equality.
- **finalize**() Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
- **getClass**() Returns the runtime class of an object.
- **hashCode**() Returns a hash code value for the object.
- **notify**() Wakes up a single thread that is waiting on this object's monitor.
- **notifyAll**() Wakes up all threads that are waiting on this object's monitor.
- **toString**() Returns a string representation of the object.
- **wait**() Waits to be notified by another thread of a change in this object.
- **wait**(long) Waits to be notified by another thread of a change in this object.
- **wait**(long, int) Waits to be notified by another thread of a change in this object.