Course: Object Based Modeling
Code: CS-33105
Branch: MCA-3

Lecture 7: Inheritance

Dr. J Sathish Kumar (JSK)
(Faculty & Coordinator)

Department of Computer Science and Engineering

Motilal Nehru National Institute of Technology Allahabad,
Prayagraj-211004

Inheritance Basics

- Inheritance is one of the cornerstones of object-oriented programming because it allows the creation of hierarchical classifications
- In the terminology of Java, a class that is inherited is called a *superclass*.
- The class that does the inheriting is called a *subclass*.
- Therefore, a subclass is a specialized version of a superclass.
- It inherits all of the members defined by the superclass and adds its own, unique elements.
- To inherit a class, you simply incorporate the definition of one class into another by using the **extends** keyword.

Example #1

```
// A simple example of inheritance.

// Create a superclass.
class A {
  int i, j;

  void showij() {
    System.out.println("i and j: " + i + " " + j);
  }
}

// Create a subclass by extending class A.
class B extends A {
  int k;

  void showk() {
    System.out.println("k: " + k);
  }

  void sum() {
    System.out.println("i+j+k: " + (i+j+k));
  }
}
```

The output from this program is shown here:

```
Contents of superOb:
i and j: 10 20

Contents of subOb:
i and j: 7 8
k: 9

Sum of i, j and k in subOb:
i+j+k: 24
```

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

    // The superclass may be used by itself.
    superOb.i = 10;
    superOb.j = 20;
    System.out.println("Contents of superOb: ");
    superOb.showij();
    System.out.println();

    /* The subclass has access to all public members of
       its superclass. */
    subOb.i = 7;
    subOb.j = 8;
    subOb.k = 9;
    System.out.println("Contents of subOb: ");
    subOb.showij();
    subOb.showk();
    System.out.println();

    System.out.println("Sum of i, j and k in subOb:");
    subOb.sum();
  }
}
```

Member Access and Inheritance

- Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as **private**.

```
// Create a superclass.
class A {
  int i; // public by default
  private int j; // private to A

  void setij(int x, int y) {
    i = x;
    j = y;
  }
}
```

In a class hierarchy, private members remain private to their class.

This program contains an error and will not compile.

Example #2

```
// A's j is not accessible here.
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);
  }
}
```

Example #3

```
// This program uses inheritance to extend Box.
class Box {
  double width;
  double height;
  double depth;

  // construct clone of an object
  Box(Box ob) { // pass object to constructor
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
  }

  // constructor used when all dimensions specified
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // constructor used when no dimensions specified
  Box() {
    width = -1;  // use  -1 to indicate
    height = -1; // an uninitialized
    depth = -1;  // box
  }

  // constructor used when cube is created
  Box(double len) {
    width = height = depth = len;
  }

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}
```

Example #3

```
// Here, Box is extended to include weight.
class BoxWeight extends Box {

  double weight; // weight of box

  // constructor for BoxWeight
  BoxWeight(double w, double h, double d, double m) {
    width = w;
    height = h;
    depth = d;
    weight = m;
  }
}
```

The output from this program is shown here:

```
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3

Volume of mybox2 is 24.0
Weight of mybox2 is 0.076
```

```
class DemoBoxWeight {
  public static void main(String args[]) {
    BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
    BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
    double vol;

    vol = mybox1.volume();
    System.out.println("Volume of mybox1 is " + vol);
    System.out.println("Weight of mybox1 is " + mybox1.weight);
    System.out.println();

    vol = mybox2.volume();
    System.out.println("Volume of mybox2 is " + vol);
    System.out.println("Weight of mybox2 is " + mybox2.weight);
  }
}
```

BoxWeight inherits all of the characteristics Of **Box** and adds to them the **weight** component.
It is not necessary for **BoxWeight** to re-create all of the features found in **Box**.
It can simply extend **Box** to meet its own purposes.

Member Access and Inheritance

- A major advantage of inheritance is that once you have created a superclass that defines the attributes common to a set of objects, it can be used to create any number of more specific subclasses.
- Each subclass can precisely tailor its own classification.
- Remember, once you have created a superclass that defines the general aspects of an object, that superclass can be inherited to form specialized classes.
- Each subclass simply adds its own unique attributes. This is the essence of inheritance.

```
// Here, Box is extended to include color.
class ColorBox extends Box {
  int color; // color of box

  ColorBox(double w, double h, double d, int c) {
    width = w;
    height = h;
    depth = d;
    color = c;
  }
}
```

Example #4

A Superclass Variable Can Reference a Subclass Object

- A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass.
- You will find this aspect of inheritance quite useful in a variety of situations.

Example #5

```
class RefDemo {
  public static void main(String args[]) {
    BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
    Box plainbox = new Box();
    double vol;

    vol = weightbox.volume();
    System.out.println("Volume of weightbox is " + vol);
    System.out.println("Weight of weightbox is " +
                       weightbox.weight);
    System.out.println();

    // assign BoxWeight reference to Box reference
    plainbox = weightbox;

    vol = plainbox.volume(); // OK, volume() defined in Box
    System.out.println("Volume of plainbox is " + vol);

    /* The following statement is invalid because plainbox
       does not define a weight member. */
//  System.out.println("Weight of plainbox is " + plainbox.weight);
  }
}
```

Using super to Call Superclass Constructors

- A subclass can call a constructor defined by its superclass by use of the following form of **super**
- Here, *arg-list* specifies any arguments needed by the constructor in the superclass.
- **super( )** must always be the first statement executed inside a subclass' constructor.

```
// BoxWeight now uses super to initialize its Box attributes.
class BoxWeight extends Box {
  double weight; // weight of box

  // initialize width, height, and depth using super()
  BoxWeight(double w, double h, double d, double m) {
    super(w, h, d); // call superclass constructor
    weight = m;
  }
}
```

Example #6

```
// BoxWeight now fully implements all constructors.
class BoxWeight extends Box {
  double weight; // weight of box

  // construct clone of an object
  BoxWeight(BoxWeight ob) { // pass object to constructor
    super(ob);
    weight = ob.weight;
  }

  // constructor when all parameters are specified
  BoxWeight(double w, double h, double d, double m) {


      super(w, h, d); // call superclass constructor
      weight = m;
    }

    // default constructor
    BoxWeight() {
      super();
      weight = -1;
    }

    // constructor used when cube is created
    BoxWeight(double len, double m) {
      super(len);
      weight = m;
    }
  }
```

Example #7

```
class DemoSuper {
  public static void main(String args[]) {
    BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
    BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
    BoxWeight mybox3 = new BoxWeight(); // default
    BoxWeight mycube = new BoxWeight(3, 2);
    BoxWeight myclone = new BoxWeight(mybox1);
    double vol;

    vol = mybox1.volume();
    System.out.println("Volume of mybox1 is " + vol);
    System.out.println("Weight of mybox1 is " + mybox1.weight);
    System.out.println();

    vol = mybox2.volume();
    System.out.println("Volume of mybox2 is " + vol);
    System.out.println("Weight of mybox2 is " + mybox2.weight);
    System.out.println();

    vol = mybox3.volume();
    System.out.println("Volume of mybox3 is " + vol);
    System.out.println("Weight of mybox3 is " + mybox3.weight);
    System.out.println();

    vol = myclone.volume();
    System.out.println("Volume of myclone is " + vol);
    System.out.println("Weight of myclone is " + myclone.weight);
    System.out.println();

    vol = mycube.volume();
    System.out.println("Volume of mycube is " + vol);
    System.out.println("Weight of mycube is " + mycube.weight);
    System.out.println();
  }
}
```

A Second Use for super

- The second form of **super** acts somewhat like **this**, except that it always refers to the superclass of the subclass in which it is used. This usage has the following general form:

super.member

- Here, *member* can be either a method or an instance variable

```
// Using super to overcome name hiding.
class A {
  int i;
}

// Create a subclass by extending class A.

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[]) {
    B subOb = new B(1, 2);

    subOb.show();
  }
}
```

This program displays the following:

```
i in superclass: 1
i in subclass: 2
```

Example #8

Creating a Multilevel Hierarchy

- Java Supports to build hierarchies that contain as many layers of inheritance as you like.
- It is perfectly acceptable to use a subclass as a superclass of another.
- For example, given three classes called **A**, **B**, and **C**,
  - **C** can be a subclass of **B**, which is a subclass of **A**.
  - When this type of situation occurs, each subclass inherits all of the traits found in all of its superclasses.
  - In this case, **C** inherits all aspects of **B** and **A**.
- In the above example, the subclass **BoxWeight** is used as a superclass to create the subclass called **Shipment**.
- **Shipment** inherits all of the traits of **BoxWeight** and **Box**, and adds a field called **cost**, which holds the cost of shipping such a parcel.

```
// Add shipping costs.
class Shipment extends BoxWeight {
  double cost;

  // construct clone of an object
  Shipment(Shipment ob) { // pass object to constructor
    super(ob);
    cost = ob.cost;
  }

  // constructor when all parameters are specified
  Shipment(double w, double h, double d,
           double m, double c) {
    super(w, h, d, m); // call superclass constructor
    cost = c;
  }

  // default constructor
  Shipment() {
    super();
    cost = -1;
  }

  // constructor used when cube is created
  Shipment(double len, double m, double c) {
    super(len, m);
    cost = c;
  }
}
```

Example #9

Example #9

```
class DemoShipment {
  public static void main(String args[]) {
    Shipment shipment1 =
               new Shipment(10, 20, 15, 10, 3.41);
    Shipment shipment2 =
               new Shipment(2, 3, 4, 0.76, 1.28);

    double vol;

    vol = shipment1.volume();
    System.out.println("Volume of shipment1 is " + vol);
    System.out.println("Weight of shipment1 is "
                       + shipment1.weight);
    System.out.println("Shipping cost: $" + shipment1.cost);
    System.out.println();


     vol = shipment2.volume();
     System.out.println("Volume of shipment2 is " + vol);
     System.out.println("Weight of shipment2 is "
                        + shipment2.weight);
     System.out.println("Shipping cost: $" + shipment2.cost);
   }
 }
```

The output of this program is shown here:

```
Volume of shipment1 is 3000.0
Weight of shipment1 is 10.0
Shipping cost: $3.41

Volume of shipment2 is 24.0
Weight of shipment2 is 0.76
Shipping cost: $1.28
```

Creating a Multilevel Hierarchy

- Because of inheritance, **Shipment** can make use of the previously defined classes of **Box** and **BoxWeight**, adding only the extra information it needs for its own, specific application.
- This is part of the value of inheritance; it allows the reuse of code.
- This example illustrates one other important point: **super( )** always refers to the
constructor in the closest superclass.
- The **super( )** in **Shipment** calls the constructor in **BoxWeight**.
- The **super( )** in **BoxWeight** calls the constructor in **Box**.
- In a class hierarchy, if a superclass constructor requires parameters, then all subclasses must pass those parameters “up the line.”
- This is true whether or not a subclass needs parameters of its own.

When Constructors Are Executed

- When a class hierarchy is created, in what order are the constructors for the classes that make up the hierarchy executed?
- The answer is that in a class hierarchy, constructors complete their execution in order of derivation, from superclass to subclass.
- Further, since **super( )** must be the first statement executed in a subclass'
  constructor, this order is the same whether or not **super( )** is used.
- If **super( )** is not used, then the default or parameterless constructor of each superclass will be executed.

Example #10

```
// Demonstrate when constructors are executed.

// Create a super class.
class A {
  A() {
    System.out.println("Inside A's constructor.");
  }
}

  // Create a subclass by extending class A.
  class B extends A {
    B() {
      System.out.println("Inside B's constructor.");
    }
  }

  // Create another subclass by extending B.
  class C extends B {
    C() {
      System.out.println("Inside C's constructor.");
    }
  }

  class CallingCons {
    public static void main(String args[]) {
      C c = new C();
    }
  }
```

The output from this program is shown here:

```
Inside A's constructor
Inside B's constructor
Inside C's constructor
```

Method Overriding

- 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 its subclass, it will always refer to the version of that method defined by the subclass.
- The version of the method defined by the superclass will be hidden.

```
// Method overriding.
class A {
  int i, j;
  A(int a, int b) {
    i = a;
    j = b;
  }

  // display i and j
  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;
  }

  // display k - this overrides show() in A
  void show() {
    System.out.println("k: " + k);
  }
}
```

```
class Override {
  public static void main(String args[]) {
    B subOb = new B(1, 2, 3);

    subOb.show(); // this calls show() in B
  }
}
```

The output produced by this program is shown here:

```
k: 3
```

Example #11

When **show( )** is invoked on an object of type **B**, the version of **show( )** defined within **B** is used.
That is, the version of **show( )** inside **B** overrides the version declared in **A**.

Method Overriding

- If you wish to access the superclass version of an overridden method, you can do so by using **super**.
- For example, in this version of **B**, the superclass version of **show( )** is invoked within the subclass' version.

```
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);
  }
}
```

Example #12

output:

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

Method Overriding

- 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.

Example #13

```
// Methods with differing type signatures are overloaded - not
// overridden.
class A {
  int i, j;

  A(int a, int b) {
    i = a;
    j = b;
  }

  // display i and j
  void show() {
    System.out.println("i and j: " + i + " " + j);
  }
}
```

Method Overriding

Example #13

```
// Create a subclass by extending class A.
class B extends A {
  int k;

  B(int a, int b, int c) {
    super(a, b);
    k = c;
  }

  // overload show()
  void show(String msg) {
    System.out.println(msg + k);
  }
}

class Override {
  public static void main(String args[]) {
    B subOb = new B(1, 2, 3);

    subOb.show("This is k: "); // this calls show() in B
    subOb.show(); // this calls show() in A
  }
}
```

The output produced by this program is shown here:

```
This is k: 3
i and j: 1 2
```

Why Overridden Methods?

- Overridden methods allow Java to support run-time polymorphism.
- Polymorphism is essential to object-oriented programming for one reason:
  - It allows a general class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods.
  - Overridden methods are another way that Java implements the “one interface, multiple methods” aspect of polymorphism.
- Dynamic, run-time polymorphism is one of the most powerful mechanisms that object oriented design brings to bear on code reuse and robustness.
- The ability of existing code libraries to call methods on instances of new classes without recompiling while maintaining a clean abstract interface is a profoundly powerful tool.

Example #14 : Run Time Polymorphism

```
// Using run-time polymorphism.
class Figure {
  double dim1;
  double dim2;

  Figure(double a, double b) {
    dim1 = a;
    dim2 = b;
  }

  double area() {
    System.out.println("Area for Figure is undefined.");
    return 0;
  }
}
```

```
class Rectangle extends Figure {
  Rectangle(double a, double b) {
    super(a, b);
  }

  // override area for rectangle
  double area() {
    System.out.println("Inside Area for Rectangle.");
    return dim1 * dim2;
  }
}
```

```
class Triangle extends Figure {
  Triangle(double a, double b) {
    super(a, b);
  }

  // override area for right triangle
  double area() {
    System.out.println("Inside Area for Triangle.");
    return dim1 * dim2 / 2;
  }
}
```

Example #14 : Run Time Polymorphism

```
class FindAreas {
  public static void main(String args[]) {
    Figure f = new Figure(10, 10);
    Rectangle r = new Rectangle(9, 5);

    Triangle t = new Triangle(10, 8);
    Figure figref;

    figref = r;
    System.out.println("Area is " + figref.area());

    figref = t;
    System.out.println("Area is " + figref.area());

    figref = f;
    System.out.println("Area is " + figref.area());
  }
}
```

The output from the program is shown here:

```
Inside Area for Rectangle.
Area is 45
Inside Area for Triangle.
Area is 40
Area for Figure is undefined.
Area is 0
```

Using Abstract Classes

- There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method.
- That is, sometimes you will want to create 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.
- Such a class determines the nature of the methods that the subclasses must implement.
- One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method.
- This is the case with the class **Figure** used in the preceding example.
- The definition of **area( )** is simply a placeholder. It will not compute and display the area of any type of object.

Using Abstract Classes

- You can require that certain methods be overridden by subclasses by specifying the **abstract** type modifier.
- To declare an abstract method, use this general form:
  abstract *type name*(*parameter-list*);
- Any class that contains one or more abstract methods must also be declared abstract.
- To declare a class abstract, you simply use the **abstract** keyword in front of the **class** keyword at the beginning of the class declaration.
- There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the **new** operator.
- Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be declared **abstract** itself.

```
// A Simple demonstration of abstract.
abstract class A {
  abstract void callme();

  // concrete methods are still allowed in abstract classes
  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();
  }
}
```

Example #15
and
Class Exercise

Using Abstract Classes

- Notice that no objects of class **A** are declared in the program.
- As mentioned, it is not possible to instantiate an abstract class.
- One other point: class **A** implements a concrete method called **callmetoo( )**. This is perfectly acceptable.
- Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because Java's approach to run-time polymorphism is implemented through the use of superclass references.

Example #16 and Class Exercise

```
// Using abstract methods and classes.
abstract class Figure {
  double dim1;
  double dim2;

  Figure(double a, double b) {
    dim1 = a;
    dim2 = b;
  }

  // area is now an abstract method
  abstract double area();
}


class Rectangle extends Figure {
  Rectangle(double a, double b) {
    super(a, b);
  }

  // override area for rectangle
  double area() {
    System.out.println("Inside Area for Rectangle.");
    return dim1 * dim2;
  }
}
```

```
class Triangle extends Figure {
  Triangle(double a, double b) {
    super(a, b);
  }

  // override area for right triangle
  double area() {
    System.out.println("Inside Area for Triangle.");
    return dim1 * dim2 / 2;
  }
}
```

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

    figref = r;
    System.out.println("Area is " + figref.area());

    figref = t;


      System.out.println("Area is " + figref.area());
    }
  }
```

Example #16 and Class Exercise

Using final with Inheritance

- The keyword **final** has three uses.
  - First, it can be used to create the equivalent of a named constant.
  - The other two uses of **final** apply to inheritance.
    - Using final to Prevent Overriding
- While method overriding is one of Java's most powerful features, there will be times when you will want to prevent it from occurring.
- To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration.
- Methods declared as **final** cannot be overridden.

Using final to Prevent Overriding

```
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!");
  }
}
```

Example #17 and Class Exercise

Using final to Prevent Inheritance

- Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with **final**.
- Declaring a class as **final** implicitly declares all of its methods as **final**, too.
- As you might expect, it is illegal to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

Example #18

```
final class A {
  //...
}

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

Tutorial #4

1. Create a class 'Degree' having a method 'getDegree' that prints "I got a degree". It has two subclasses namely 'Undergraduate' and 'Postgraduate' each having a method with the same name that prints "I am an Undergraduate" and "I am a Postgraduate" respectively. Call the method by creating an object of each of the three classes.
2. A boy has his money deposited $1000, $1500 and $2000 in banks-Bank A, Bank B and Bank C respectively. We have to print the money deposited by him in a particular bank.
   Create a class 'Bank' with a method 'getBalance' which returns 0. Make its three subclasses named 'BankA', 'BankB' and 'BankC' with a method with the same name 'getBalance' which returns the amount deposited in that particular bank. Call the method 'getBalance' by the object of each of the three banks.
3. A class has an integer data member 'i' and a method named 'printNum' to print thevalue of 'i'. Its subclass also has an integer data member 'j' and a method named 'printNum' to print the value of 'j'. Make an object of the subclass and use it to assign a value to 'i' and to 'j'. Now call the method 'printNum' by this object.
4. All the banks operating in India are controlled by RBI. RBI has set a well defined guideline (e.g. minimum interest rate, minimum balance allowed, maximum withdrawal limit etc) which all banks must follow. For example, suppose RBI has set minimum interest rate applicable to a saving bank account to be 4% annually; however, banks are free to use 4% interest rate or to set any rates above it.

   Write a JAVA program to implement bank functionality in the above scenario and demonstrate the dynamic polymorphism concept. Note: Create few classes namely Customer, Account, RBI (Base Class) and few derived classes (SBI, ICICI, PNB etc). Assume and implement required member variables and functions in each class.