Unit 3

INFORMATION HIDING & REUSABILITY

-Inheritance basics

-Using super

-Method Overriding

-Constructor call

-Dynamic method

Inheritance

- 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.

Inheritance Basics

- To inherit a class, you simply incorporate the definition of one class into another by using the **extends** keyword. To see how, let's begin with a short example.
- The following program creates a superclass called **A** and a subclass called **B**.
- Notice how the keyword **extends** is used to create a subclass of **A**.

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));
}
}
```

A simple example of inheritance

```
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;
```

A simple example of inheritance

```
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();
}
}
```

OUTPUT

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

As you can see, the subclass **B** includes all of the members of its superclass, **A**. This is why **subOb** can access **i** and **j** and call **showij( )**. Also, inside **sum( )**, **i** and **j** can be referred to directly, as if they were part of **B**.

Even though **A** is a superclass for **B**, it is also a completely independent, stand-alone class. Being a superclass for a subclass does not mean that the superclass cannot be used by itself. Further, a subclass can be a superclass for another subclass.

OUTPUT

SYNTAX:

The general form of a **class** declaration that inherits a superclass is shown here:

class *subclass-name* **extends** *superclass-name*

{

// body of class

}

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**. For example, consider the

following simple class hierarchy:

- In a class hierarchy, private members remain private to their class.
- This program will not compile because the use of **j** inside the **sum( )** method of **B** causes an access violation. Since **j** is declared as **private**, it is only accessible by other members of its own class. Subclasses have no access to it.

EXAMPLE

```
// 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;
}
}
// 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);

}
}
```

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 super class.

```
import java.io.*;
class X{
int ax;
}
class Y extends X{
int ay;
}
```

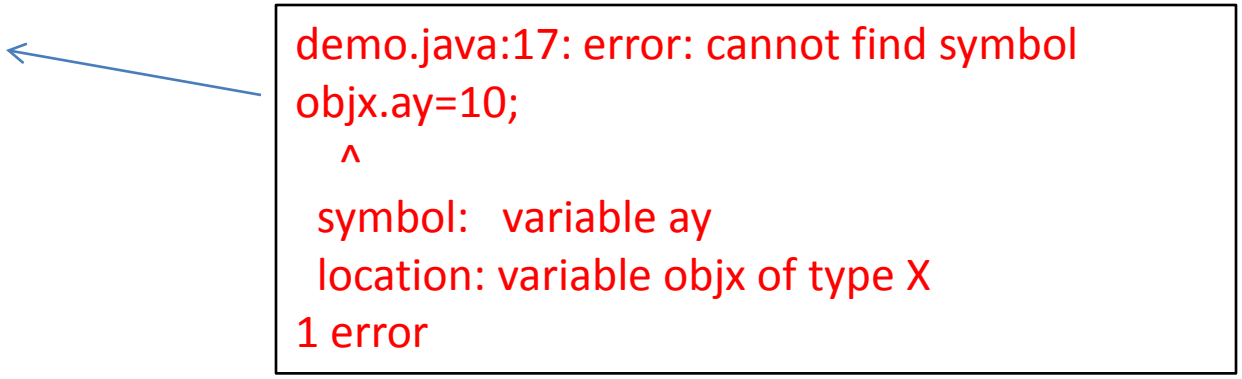
A Superclass Variable Can Reference a Subclass Object

A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass.

```
class demo{
public static void main(String args[])
{
X objx=new X();
Y objy=new Y();
objy.ax=5;
objy.ay=6;
System.out.println("ax="+objy.ax+" ay="+objy.ay);
objx=objy; // It is permissible to assign the reference
objx.ay=10;
}
}
```

A Superclass Variable Can Reference a Subclass Object

```
class demo{
public static void main(String args[])
{
X objx=new X();
Y objy=new Y();
objy.ax=5;
objy.ay=6;
System.out.println("ax="+objy.ax+" ay="+objy.ay);
objx=objy;
objx.ay=10;
}
}
```



USING SUPER

- Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super**.

super has two general forms.

- The first calls the superclass' constructor.
- The second is used to access a member of the superclass that has been hidden by a member of a subclass.

Using super to Call Superclass Constructors

- A subclass can call a constructor defined by its superclass by use of the following form of

super:

super(*arg-list*);

- 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.
- To see how **super( )** is used, consider this class:

```
// Implementation of BoxWeight.
class Box {
private double width;
private double height;
private 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;
}
}
```

```
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;
}
}
```

```
class DemoSuper {
public static void main(String args[ ])
{
BoxWeight mybox1 = new
BoxWeight(10, 20, 15, 34.3);
BoxWeight mybox2 = 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();
vol = mybox2.volume();
System.out.println("Volume of
mybox2 is " + vol);
System.out.println("Weight of
mybox2 is " + mybox2.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();    }   }
```

SUPER

- Notice that **super( )** is passed an object of type **BoxWeight**—not of type **Box**. This still invokes the constructor **Box(Box ob)**.
- As mentioned earlier, a superclass variable can be used to reference any object derived from that class. Thus, we are able to pass a **BoxWeight** object to the **Box** constructor. Of course, **Box** only has knowledge of its own members.

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.
- This second form of **super** is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.
- Consider this simple class hierarchy:

// 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
```

Although the instance variable **i** in **B** hides the **i** in **A**, **super** allows access to the **i** defined in the superclass. As you will see, **super** can also be used to call methods that are hidden by a subclass.

Creating a Multilevel Hierarchy

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 super classes. In this case, **C** inherits all aspects of **B** and **A**.

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? For example, given a subclass called **B** and a superclass called **A**, is **A**'s constructor executed before **B**'s, or vice versa?
- 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 parameter less constructor of each superclass will be executed

```
// 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( );
}   }
```

OUTPUT

The output from this program is shown here:

Inside A's constructor

Inside B's constructor

Inside C's constructor

As you can see, the constructors are executed in order of derivation.

If you think about it, it makes sense that constructors complete their execution in order of derivation. Because a superclass has no knowledge of any subclass, any initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by the subclass. Therefore, it must complete its execution first.

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

- 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**.
- 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. This allows all instance variables to be displayed.

```
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);
}
}
```

If you substitute this version of **A** into the previous program, you will see the following

output:

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

Here, **super.show( )** calls the superclass version of **show( )**.

- 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. For example, consider this modified version of the preceding example:

```
// 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);
}
}
```

```
// 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);
}
}
```

```
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
```

The version of **show( )** in **B** takes a string parameter. This makes its type signature different from the one in **A**, which takes no parameters. Therefore, no overriding (or name hiding) takes place. Instead, the version of **show( )** in **B** simply overloads the version of **show( )** in **A**.

Dynamic Method Dispatch

- Method overriding forms the basis for one of Java's most powerful concepts: *dynamic method dispatch*.
- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- Dynamic method dispatch is important because this is how Java implement run-time polymorphism.

- When different types of objects are referred to, different versions of an overridden method will be called. In other words, *it is the type of the object being referred to* (not the type of the reference variable) that determines which version of an overridden method will be executed.
- Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

//Here is an example that illustrates dynamic method dispatch:

```
class A {
void callme( ) {
System.out.println("Inside A's callme method");
}  }
class B extends A {       // override callme()
void callme( ) {
System.out.println("Inside B's callme method");
}  }
class C extends A {      // override callme()
void callme( ) {
System.out.println("Inside C's callme method");
}  }
class Dispatch {
public static void main(String args[ ]) {
A a = new A(); // object of type A
B b = new B(); // object of type B
C c = new C(); // object of type C
```

```
A r; // obtain a reference of type A
r = a; // r refers to an A object
r.callme(); // calls A's version of callme
r = b; // r refers to a B object
r.callme(); // calls B's version of callme
r = c; // r refers to a C object
r.callme(); // calls C's version of callme
}  }
```

The output from the program is shown here:

```
Inside A's callme method
Inside B's callme method
Inside C's callme method
```

This program creates one superclass called **A** and two subclasses of it, called **B** and **C**. Subclasses **B** and **C** override **callme( )** declared in **A**. Inside the **main( )** method, objects of type **A**, **B**, and **C** are declared. Also, a reference of type **A**, called **r**, is declared. The program then in turn assigns a reference to each type of object to **r** and uses that reference to invoke **callme( )**. As the output shows, the version of **callme( )** executed is determined by the type of object being referred to at the time of the call. Had it been determined by the type of the reference variable, **r**, you would see three calls to **A**'s **callme( )** method.

Applying Method Overriding

Applying Method Overriding

- Let's look at a more practical example that uses method overriding. The following program creates a superclass called **Figure** that stores the dimensions of a two-dimensional object.
- It also defines a method called **area( )** that computes the area of an object.
- The program derives two subclasses from **Figure**. The first is **Rectangle** and the second is **Triangle**.
- Each of these subclasses overrides **area( )** so that it returns the area of a rectangle and a triangle, respectively.

```
// 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);
}
double area() {     // override area for rectangle
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 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());
}  }
```

Output

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

Through the dual mechanisms of inheritance and run-time polymorphism, it is possible to define one consistent interface that is used by several different, yet related, types of objects. In this case, if an object is derived from **Figure**, then its area can be obtained by calling **area( )**. The interface to this operation is the same no matter what type of figure is being used.