Module 2

Inheritance

Inheritance is a very important feature of an Object oriented programming language. Using inheritance, a general class can be created. It would define traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it. In the terminology of Java, a class that is inherited is called a *superclass.* The class that does the inheriting is called a *subclass.* A subclass is a specialized version of a superclass. It inherits all of the instance variables and methods defined by the superclass and adds its own, unique elements.

To inherit a class extends keyword is used. 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**.

```
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));
      }
}

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();
      }
}
```

Output

```
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
```

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**.

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.

```
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);
      }
}
```

This program will not compile because the reference to **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.

Another Example of Inheritance.

The Base class is Box. Another class named BoxWeight inherits the class Box and defines a variable weight of its own.

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

BoxWeight(double w, double h, double d, double m) {
width = w;
height = h;
depth = d;
weight = m;
      }
}
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);
      }
}
```
Output
```
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3
Volume of mybox2 is 24.0
Weight of mybox2 is 0.076
```

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.

Using super
The constructor for **BoxWeight** explicitly initializes the **width**, **height**, and **depth** fields of **Box( )**. This duplicates the code found in its superclass, also it implies that a subclass must be granted access to these members. If the data members are kept private, then the above way of initialization is not possible.

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

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.

```
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;
      }
}
```

Here, **BoxWeight( )** calls **super( )** with the arguments **w**, **h**, and **d**. This causes the **Box( )** constructor to be called, which initializes **width**, **height**, and **depth** using these values. **BoxWeight** no longer initializes these values itself. It only needs to initialize the value unique to it: **weight**. This leaves **Box** free to make these values **private** if desired.

Super can be with appropriate number of arguments that match the number of arguments in the constructor in the super class.

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

```
}
```

Output

```
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3
Volume of mybox2 is 24.0
Weight of mybox2 is 0.076
Volume of mybox3 is -1.0
Weight of mybox3 is -1.0
Volume of myclone is 3000.0
Weight of myclone is 34.3
Volume of mycube is 27.0
Weight of mycube is 2.0
```

Using Super with members of the class

Super can be used to access a member of the superclass. This usage has the following general form:

super.*member*

```
class A {
int i;
}
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();
      }
}
```

Output

```
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.

Creating a Multilevel Hierarchy

Hierarchies can be built to contain as many layers as a user wants. 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 following example, a new subclass **Shipment** is created along with **BoxWeight. Shipment** inherits all of the traits of **BoxWeight** and **Box**, and adds a field called **cost**, which holds the cost of shipping.

```
class Box {
private double width;
private double height;
private double depth;

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;
}
}
// Add weight.
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;
}
BoxWeight(double len, double m) {
super(len);
weight = m;
}
}
// 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;
}
}
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);
}
}
```

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."

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 a 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.

```
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
      }
}
```

Output
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 the user wishes to access the superclass version of an overridden method, it can be done by using **super**.

```
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);
      }
}
```

In this case the output would be

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

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 implements run-time polymorphism.

The principle “**A superclass reference variable can refer to a subclass object**” is going to be used here. When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call. 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.

```
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, there would been three calls to A's callme( ) method.

Applying Method Overriding.

The following program creates a superclass called **Figure** that stores the dimensions of a Polygon. 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.

```
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;
      }
}
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

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

Here in the above program, a superclass reference variable is used to reference a subclass object.

Abstract classes

There are situations in which one would want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. Sometimes there will be a need 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. In such a condition a method is declared abstract in the base class. This method is overridden in the subclass with the suitable implementation.
Any class that contains one or more abstract methods must also be declared abstract. To declare a class abstract, the **abstract** keyword is used 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. Abstract constructors, or abstract static methods cannot be defined. Any subclass of an abstract class must implement all of the abstract methods in the superclass.

The previous program is rewritten using the abstract keyword.

```
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 triangle
double area() {
System.out.println("Inside Area for Triangle.");
return dim1 * dim2 / 2;
      }
}
class AbstractAreas {
public static void main(String args[]) {
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());
      }
}
```

It is now not possible to create an object of class Figure. But it is still possible to create a reference variable.

Using Final in Inheritance

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. The following fragment illustrates **final**:

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

Because **meth( )** is declared as **final**, it cannot be overridden in **B**.

Inorder to prevent a class from being inherited, final keyword can be used. To do this, precede the class declaration with **final**. Declaring a class as **final** implicitly declares all of its methods as **final**, too.

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

It is illegal for **B** to inherit **A** since **A** is declared as **final**.