Module: Core Java

Session 24: Inheritance and Polymorphism

# Inheritance and Polymorphism

* Introduction
* The Java Platform Class Hierarchy
* What You Can Do in a SubClass
* Private Members in a Superclass
* Casting Objects
* Using Super to Call Superclass Constructors
* A Second Use of Super
* Creating a Multi-level Hierarchy
* When Constructors are Called
* Method Overriding
* Dynamic Method Dispatch
* Why Overridden Methods
* Applying Method Overriding
* Using abstract Classes
* Using Final with Inheritance
* Using Final to Prevent Overriding
* Using Final to Prevent Inheritance

## Objective

At the end of this chapter, you will be able to:

* Define Inheritance
* Know Java Platform Class Hierarchy
* Design Sub Class and their private members
* Cast Objects
* Use Super to Call Super class Constructors
* Create a Multi-level Hierarchy
* Understand
* when to call Constructors
* Prepare yourself for Method Overriding
* Do Dynamic Method Dispatch
* Gather information about abstract Classes
* Know how to use of Final with Inheritance
* Use Final to Prevent Overriding and Inheritance

**Introduction**

This chapter discusses about inheritance. In the Java language, classes can be derived from other classes, thereby inheriting fields and methods from those classes.

Inheritance is one of the basic principles of Objected Oriented Programming. The enabling principle of inheritance is Polymorphism. Inheritance allows you to create hierarchical classifications. You begin the process of creating a hierarchical classification by defining a general class that defines traits that common to a set of related objects. This class can then be inherited by other more specific classes, each adding those things that are unique to it. These classes inherit the general features from the general class. Generalized class definitions are defined and exist at the top of a class hierarchy, while specialized classes appear lower down in the hierarchy. In Java terminology, a class that is inherited is called a 'superclass' and the class that does the inheriting is called the 'subclass'. Inheritance is a concept in object oriented programming languages that allows you to describe more easily real-world “is-a-kind-of” relationships. For example, a car (specialized subclass) is a kind of vehicle (generalized superclass). Another example is that of a manager (specialized subclass) being a kind of employee (generalized superclass).

You therefore encounter more and more generalization as you ascend successive layers up the hierarchy and more and more specialization is evidenced as you descend successive layers down the hierarchy. Hierarchy is ranking or ordering of abstractions. That is why you speak of inheritance as being a generalization/specialization hierarchy. This is an important principle of designing class hierarchies.

**Definitions**

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 (also a base class or a parent class).

Excepting Object, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of Object.

Classes can be derived from classes that are derived from other classes, and so on. These are ultimately to be derived from the topmost class, Object. Such a class is said to have descended from all the classes in the inheritance chain stretching back to Object.

The idea of inheritance is simple but powerful. When you want to create a new class and there is already a class that includes some of the code that you want, you can derive your new class from the existing class. In doing this, you can reuse the fields and methods of the existing class without having to write (and debug) them yourself.

A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

**The Java Platform Class Hierarchy**

The Object class, defined in the java.lang package, defines and implements behavior common to all classes—including the ones that you write. In the Java platform, many classes derive directly from Object, other classes derive from some of those classes, and so on, forming a hierarchy of classes.

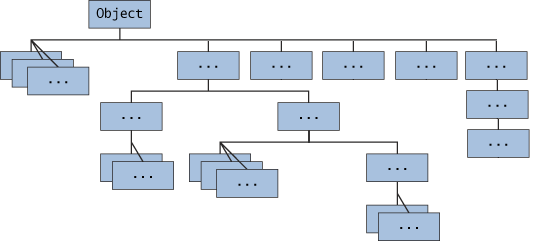


Fig. 1: All Classes in the Java Platform are Descendants of Objects.

At the top of the hierarchy, Object is the most general of all classes. Classes near the bottom of the hierarchy provide more specialized behavior.

Here is the sample code for a possible implementation of a Bicycle class.

public class Bicycle {

// **the Bicycle class has three *fields***

public int cadence;

public int gear;

public int speed;

// **the Bicycle class has one *constructor***

public Bicycle(int startCadence, int startSpeed, int startGear) {

gear = startGear;

cadence = startCadence;

speed = startSpeed;

}

// **the Bicycle class has four *methods***

public void setCadence(int newValue) {

cadence = newValue;

}

public void setGear(int newValue) {

gear = newValue;

}

public void applyBrake(int decrement) {

speed -= decrement;

}

public void speedUp(int increment) {

speed += increment;

}

}

A class declaration for a MountainBike class that is a subclass of Bicycle might look like this:

public class MountainBike extends Bicycle {

// **the MountainBike subclass adds one *field***

public int seatHeight;

// **the MountainBike subclass has one *constructor***

public MountainBike(int startHeight, int startCadence, int startSpeed, int startGear) {

super(startCadence, startSpeed, startGear);

seatHeight = startHeight;

}

// **the MountainBike subclass adds one *method***

public void setHeight(int newValue) {

seatHeight = newValue;

}

}

MountainBike inherits all the fields and methods of Bicycle and adds the field seatHeight and a method to set it. Except for the constructor, it is as if you had written a new MountainBike class entirely from scratch, with four fields and five methods. However, you did not have to do all the work. This would be especially valuable if the methods in the Bicycle class were complex and had taken substantial time to debug.

**Guidelines for Inheritance**

**Inheritance of structure**

When designing a class hierarchy, the following strategies can be used by the development team for inheriting instance variables.

**No Redefinition:**

In this case, the type of the instance variable of the superclass is inherited as it is. No modifications are made to the superclass definition.

**Constrained Redefinition**

In this case, the type of the instance variable in the subclass is a subtype of the type of the variable in the superclass. This is the instance variable in the subclass. This gives the instance variable in the subclass a narrowly scoped definition while still retaining its semantic relation to and broadly remaining within the domain of the superclass instance variable. For example, the class Vehicle might have an instance variable noOfWheels of type Natural, which is inherited and subtyped by the subclass car to the type {3, 4}.

**Arbitrary Redefinition**

In this strategy, the name of the instance variable in the subclass is retained, but the type of the instance variable in the subclass is changed to a type that is unrelated to the type of the instance variable in the superclass.

**Conclusion**

The more conservative strategies (no redefinition and constrained redefinition) result in the most maintainable and reusable inheritance trees.

**Inheritance of Behavior**

The following strategies can be used by the development team for the inheritance of methods.

**Arbitrary Redefinition:** The subclass can provide an entirely different and unrelated implementation of the method.

**Constrained Redefinition:** The methods in the subclass must have signatures that are subtypes of the signatures of the methods in the superclass they override. A signature of a method in a subclass is a subtype of a signature of a method in the superclass if the two methods have the same name and the arguments. You will see more of constrained redefinition in the section on overriding.

**Conclusion**

The more conservative strategy (constrained redefinition) results in the most maintainable and inheritance trees.

**Single Versus Multiple Inheritance**

Single inheritance is a phenomenon wherein a subclass inherits from one and only one superclass. Multiple inheritance on the other hand is characterized by a subclass inheriting from more than one superclass.

Multiple inheritance, therefore provides richer functionality when modeling subclasses for it can inherit the functionality of more than one superclass. This richer functionality of subclasses, does come with a cost, however.

The problem is that multiple superclasses can have instance variables and /or methods with the same name, but with totally unrelated semantics. As a result, the inheritance tree using multiple inheritance will be less understandable and eventually also less maintainable than an eventually also less maintainable than an inheritance tree using single inheritance. The following diagram depicting an inheritance tree points out the ambiguities that can occur in multiple inheritances:

int a

int b

fn1()

{//some code}

Inherited from B

Inherited from A

D

C

B

A

int a

int b

int c

fn1()

{//some code}

int a

int b

int d

fn1()

{//some code}

int a

int b

int d

fn1()

{//some code}

int a

int b

int c

fn1()

{//some code}

Fig. 2: An improbable case of Multiple Inheritance

When an object of class D is created, it would have inherited attributes a and b in the method fn1( ) twice from class A via multiple access paths. Therefore, whenever your code refers to variable a or b in an object of class D, it becomes an ambiguous reference, because we need to further specify whether we are referring to variable a or b inherited from Class A or the variable a and b inherited from class B. This places a considerable overhead on the complier in working around such ambiguities.

Looking at the problems and complexities posed by multiple inheritance, Java chose to implement the single inheritance model.

**Inheritance Basics**

To define a new class from an already existing class, i.e., to inherit a class, you need to incorporate the definition of an already existing class into the class to be defined by using the extends keyword. You will now see an example program that creates a superclass called X and a subclass called Y. Pay attention to the usage of the extends keyword while defining the subclass Y.

//A simple program demonstrating inheritance

//

//Superclass Definition

class X{

int a, b;

void showab( ){

System.out.println (“a and b are : “+a+” “+b);

}

}

//Defining a class by extending class X

class Y extends X {

int c;

void showc( ){

System.out.println(“c :” + c);

}

void sum() {

System.out.println(“a+b+c = “ + (a+b+c));

}

}

class InheritanceDemo {

public static void main (String args[]){

X superob = new X();

Y subob = new Y();

/\* The superclass can be used by itself in a program and need not always be used in the context of a subclass.\*/

superob.a = 10;

superob.b = 20;

System.out.println(“State of object X:”);

superob.showab();

/\* The subclass has access to all public members of its superclass \*/

subob.a = 7;

subob.b = 8;

subob.c = 9;

System.out.println(“State of object Y:”);

subob.showab();

subob.showc();

System.out.println(“sum of a,b and c in object Y is:”);

subob.sum();

}

}

**InheritanceDemo.java** program generates the following output:

State of object X:

a and b: 10 20

State of object Y:

a and b: 7 8

C: 9

Sum of a, b and c in object Y is:

a+b+c: 24

It is obvious that an object of subclass Y included all of the members of its superclass X. This is why subob can access a and b and call showab( ). Also, inside sum( ), a and b can be referred to directly, as if they were part of Y.

Even though X is a superclass for y, it is also a completely independent, standalone class. Being a superclass for a subclass does not imply that a superclass cannot be used by itself. A subclass can in turn be a superclass for another subclass.

**What You Can Do in a Subclass?**

A subclass inherits all the public and protected members of its parent, no matter what package the subclass is in. If the subclass is in the same package as its parent, it also inherits the package-private members of the parent. You can use the inherited members as ihey are, replace them, hide them, or supplement them with new members:

The inherited fields can be used directly, just like any other fields.

* You can declare a field in the subclass with the same name as the one in the superclass, thus hiding it (not recommended).
* You can declare new fields in the subclass that are not in the superclass.
* The inherited methods can be used directly as they are.
* You can write a new instance method in the subclass that has the same signature as the one in the superclass, thus overriding it.
* You can write a new static method in the subclass that has the same signature as the one in the superclass, thus hiding it.
* You can declare new methods in the subclass that are not in the superclass.
* You can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword super.

The following sections in this lesson will expand on these topics.

**Private Members in a Superclass**

A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods for accessing its private fields, these can also be used by the subclass.

A nested class has access to all the private members of its enclosing class—both fields and methods. Therefore, a public or protected nested class inherited by a subclass has indirect access to all the private members of the superclass.

/\* An example that demonstrates superclass member access from a subclass object\*/

//Defining a Superclass

class X {

int a; // public by default

private int b;

void setab(int m, int n) {

a = m;

b = n;

}

}

class Y extends x {

int total;

void sum () {

total = a + b;

//b cannot be accessed outside of its

//class; this line will not compile

}

}

class AccessDemo {

public static void main(String args[]) {

Y subob = new Y ();

subob.setab(10,12);

subob.sum();

System.out.println(“Total: “ + subob.total);

}

}

/\* This program will not compile because the reference to b inside sum() method of class Y results in an access violation. The compiler ensures that the wall of encapsulation built using the private keyword is not broken.\*/

**A possible solution to the program is as follow:**

/\* An example that demonstrates superclass member access from a subclass object \*/

//Defining a Superclass

class X

{

int a; //public by default

private int b;

void setab(int m, int n)

{

a = m;

b = n;

}

public int getb()

{

return b;

}

}

class Y extends X {

int total;

void sum ( ) {

total = a + getb( );

//getb( )returns the value of b

}

}

class AccessDemo {

public static void main(String args[]) {

Y subob = new Y ();

subob.setab(10,12);

subob.sum();

System.out.println(“Total: “ + subob.total);

}

}

**Casting Objects**

We have seen that an object is of the type of the class from which it was instantiated. For example, if we write

public MountainBike myBike = new MountainBike();

then myBike is of type MountainBike. MountainBike is descended from Bicycle and Object. Therefore, a MountainBike is a Bicycle and is also an Object, and it can be used wherever Bicycle or Object objects are called for.

The reverse is not necessarily true: a Bicycle may be a MountainBike (is not necessarily). Similarly, an Object may *be* a Bicycle or a MountainBike (is not necessarily).

Casting shows the use of an object of one type in place of another type, among the objects permitted by inheritance and implementations. For example, if we write

Object obj = new MountainBike();

Then obj is both an Object and a Mountainbike (until such time as obj is assigned another object that is *not* a Mountainbike). This is called implicit casting.

If, on the other hand, we write

MountainBike myBike = obj;

we would get a compile-time error because obj is not known to the compiler to be a MountainBike. However, we can tell the compiler that we promise to assign a MountainBike to obj by explicit casting:

MountainBike myBike = (MountainBike) obj;

This cast inserts a runtime check that obj is assigned a MountainBike so that the compiler can safely assume that obj is a MountainBike. If obj is not a Mountainbike at runtime, an exception will be thrown.

**Note:**You can make a logical test to the type of a particular object using the instanceof operator. This can save you from a runtime error owing to an improper cast. For example:

if (obj instanceof MountainBike) {

MountainBike myBike = (MountainBike)obj;

}

Here the instanceof operator verifies that obj refers to a MountainBike so that we can make the cast with the knowledge that no runtime exception will be thrown.

**Using Super to Call Superclass Constructors**

As you know a constructor cannot be inherited by its subclass. A subclass can call a constructor method defined by its immediate super class by using the following form of **super**:

super(parameter-list); //parameter list covers parameters of superclass.

**Note:** super() must be the first statement within the constructor of the sub class. Other statements within the subclass constructor must follow this.

Here, parameter-listspecifies any parameters needed by the constructor in the superclass. **super( )** must always be the first statement executed inside a subclass’ constructor.

Following example illustrates the usage of **super( )**:

class Cuboid{

double width;

double height;

double depth;

Cuboid(double w, double h, double d)

{

width = w;

height = h;

depth = d;

}

}

// CuboidWeight now uses super to initialize its Cuboid attributes.

class CuboidWeight extends Cuboid {

double weight; // weight of box

// initialize width, height, and depth using super()

CuboidWeight(double w, double h, double d, double m) {

super(w, h, d); // call superclass constructor

weight = m;

}

}

Here, CuboidWeight( ) calls super( ) with the parameters w, h, and d. This causes the Cuboid( ) constructor to be called, which initializes width, height, and depth using these values. CuboidWeight no longer initializes these values itself. It only needs to initialize the value unique to it: weight. This leaves Cuboid free to make these values private if desired. Since constructors can be overloaded not inherited, super( ) can be called using any form defined by the superclass. The constructor executed will be the one that matches the arguments.

For example, here is a complete implementation of CuboidWeight that provides constructors for the various ways that a box can be constructed. In each case, super( ) is called using the appropriate arguments. Notice that width, height, and depth have been made private within Cuboid:

// A complete implementation of CuboidWeight.

class Cuboid {

private double width;

private double height;

private double depth;

// construct clone of an object

Cuboid(Cuboid ob) { // pass object to constructor

width = ob.width;

height = ob.height;

depth = ob.depth;

}

// constructor used when all dimensions specified

Cuboid(double w, double h, double d) {

width = w;

height = h;

depth = d;

}

// constructor used when no dimensions specified

Cuboid() {

width = -1; // use -1 to indicate

height = -1; // an uninitialized

depth = -1; // box

}

// constructor used when cube is created

Cuboid(double len) {

width = height = depth = len;

}

// compute and return volume

double volume() {

return width \* height \* depth;

}

}

// BoxWeight now fully implements all constructors.

class CuboidWeight extends Cuboid {

double weight; // weight of box

// construct clone of an object

CuboidWeight(CuboidWeight ob) { // pass object to constructor

super(ob);

weight = ob.weight;

}

// constructor when all parameters are specified

CuboidWeight(double w, double h, double d, double m) {

super(w, h, d); // call superclass constructor

weight = m;

}

// default constructor

CuboidWeight() {

super();

weight = -1;

}

// constructor used when cube is created

CuboidWeight(double len, double m) {

super(len);

weight = m;

}

}

class TestSuperDemo {

public static void main(String args[]) {

CuboidWeight mybox1 = new CuboidWeight(10, 20, 15, 34.3);

CuboidWeight mybox2 = new CuboidWeight(2, 3, 4, 0.076);

CuboidWeight mybox3 = new CuboidWeight(); // default

CuboidWeight mycube = new CuboidWeight(3, 2);

CuboidWeight myclone = new CuboidWeight(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();

}

}

**TestSuperDemo.java** generates the following 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

Pay special attention to this constructor in **CuboidWeight( )**:

// construct clone of an object

CuboidWeight(CuboidWeight ob) { // pass object to constructor

super(ob);

weight = ob.weight;

}

You can see that super( ) is called with an object of type CuboidWeight—not of type Cuboid. This still invokes the constructor Cuboid(Cuboid 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 CuboidWeight object to the Cuboid constructor. Of course, Cuboid only has knowledge of its own members.

Let us review the key concepts behind super( ). When a subclass calls super( ), it is calling the constructor of its immediate superclass. Thus, super( ) always refers to the superclass immediately above the calling class. This is true even in a multileveled hierarchy. Also, super( ) must always be the first statement executed inside a subclass constructor.

**A Practical Example:**

You will now evolve a class description for a closed rectangular solid, which is referred to as a closed rectangular solid. It has the following attributes- width, height, and depth. Using these attributes, one can compute its volume. The aforesaid points lead you to the definition of a RectangularSolid class. The code for the RectangularSolid class is given below:

class RectangularSolid {

double width;

double height;

double depth;

//constructor used when all dimensions specified RectangularSolid(double w,double h, double d) {

width = w;

height = h;

depth = d;

}

// Constructor used when cube is created RectangularSolid(double length) {

width = height = depth = length;

}

//constructor used when no dimensions specified RectangularSolid() {

width = height = depth = 0;

}

//compute and return volume

double volume() {

return width \* height \* depth;

}

}

//you are going to extend class RectangularSolid to include weight

class RectangularSolidWeight extends RectangularSolid {

double weight;

//Constructor for Rectangular SolidWeight

public RectangularSolidWeight(double w,double h, double d, double wt) {

width = w;

height = h;

depth = d;

weight = wt;

}

}

class RectangularSolidImp1 {

public static void main(String args[]) {

RectangularSolidWeight r1 = new

RectangularSolidWeight (10,20,15,35.0);

RectangularSolidWeight r2 = new

RectangularSolidWeight (1,2,3,5.0);

double volume;

volume = r1.volume();

System.out.println(“volume of rectangular Solid r1 is” + volume);

System.out.println();///print a blank line

volume = r2.volume( );

System.out.println(“Volume of rectangular solid r2 is” + volume);

}

}

The output from the program is as follows:

Volume of rectangular solid r1 is 3000.0

volume of rectangular solid r2 is 6.0

The class RectangularSolidWeight inherits all of the characteristics of the class RectangularSolid and adds its own specific attribute, namely weight. It is not necessary for RectangularSolidWeight class to redefine all of the attributes already defined in the RetangularSolid class. Rather, it can simply extend RectangularSolid class to avail of the attributes width, height and depth that it inherits.

A major advantage of inheritance is that once you have defined a superclass that defines attributes common to a set of objects, it can be used to create any number of more specific subclasses.

**A Second Use of 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, membercan 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 UseSuperDemo {

public static void main(String args[]) {

B subOb = new B(1, 2);

subOb.show();

}

}

**UseSuperDemo.java** generates the following 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. As you will see, **super** can also be used to call methods that are hidden by a subclass.

**Accessing overridden superclass members from subclass objectives**

The second use of super is applicable in situations in which member names of a subclass hide member by the same name in the superclass and you would want to access an overridden superclass member from a subclass object. Consider the following simple class hierarchy:

class X {

int a ;

}

//create a subclass by extending class X.

class Y extends X {

int a ;// this a hides the a in X

y(int m, int n) {

super.a = m;

/\* initializing the overridden data member in A from the subclass constructor\*/

a = n;//initializing a of the subclass

}

void display() {

System.out.println(“a in superclass:” + super.a);

System.out.println(“a in subclass: “ + a);

}

}

class SuperUsage {

public static void main (String args[]) {

y subOb = new Y(1,2);

subOb.display( );

}

}

This code generates the following output:

a in superclass: 1

a in subclass :2

The keyword super can also be used to call methods that are hidden by a subclass.

**Creating a Multilevel Hierarchy**

Java allows you to define multiple layers in an inheritance hierarchy and not just two layers, as you would have observed till now. You can define a superclass and a subclass where the subclass in turn become a superclass for another subclass. Given three classes are called X, Y and Z, Z can be a subclass of Y, and Y in turn can be a subclass of X. In this scenario, each subclass inherits all attributes from its parent. In the aforesaid example, Z inherits all the attributes and behaviors of X and Y.

You will now extend the RectangularSolidWeight class further to derive a new class called RectangularSolidShipment class. Shipment inherits all of the traits of RectangularSolid and RectangularSolidWeight classes, and adds its own specific attribute called cost, which holds the cost of shipping such a parcel.

Class RectangularSolid {

private double width;

private double height;

private double depth;

//Constructor used when all dimensions specified

RectangularSolid(double w, double h, double d) {

width = w;

height = h;

depth = d;

}

//Constructor used when cube is created RectangularSolid(double length) {

width = height = depth = length;

}

// constructor used when no dimensions specified RectangularSolid ( ) {

width = height = depth = 0;

}

//compute and return volume

double volume ( ) {

return width \* height \* depth;

}

}

/\* you are going to extend class RectangularSolid to include weight;\*/

class RectangularSolidWeight extends RectangularSolid {

double weight;

//constructor for RectangularSolidWeight (double len, double wt) {

super(len);

weight = wt;

}

public RectangularSolidWeight( ) {

super( );

weight = 0;

}

}

class RectangularSolidShipment extends RectangularSolidWeight {

double cost;

//constructor when all parameters are specified RectangularSolidShipment (double w, double h, double d, double wt, double c) {

super (w,h,d,wt);

cost = c;

}

//default constructor

RectangularSolidShipment () {

super ();

cost = 0;

}

//Constructor used when cube is created

RectangularSolidShipment (double len, double wt, double C) {

super (len ,wt);

cost = c;

}

}

class MultilevelDemo {

public static void main(String args[]) {

RectangularSolidShipment s1 = new

RectangularSolidShipment (10,20,15,10,3.41);

RectangularSolidShipment s2 = new

RectangularSolidShipment(2,3,4,0.76,1.28);

double volume;

volume = s1.volume();

System.out.println(“Volume of shipment s2 is : “ + volume);

System.out.println(“weight of shipment s1 is :” + s1.weight);

System.out.println(“shipping cost of shipment s1 is “ + s1.cost);

System.out.println();

volume = s2.volume();

System.out.println(“volume of shipment s2 is : “+ volume);

System.out.println(“weight of shipment s2 is : “ + s2.weight);

System.out.println(“shipping cost of shipment s2 is “ + s2.cost);

System.out.println();

}

}

The output of the program is as follows:

volume of shipment s1 is :3000.0

weight of shipment s1 is :10.0

cost of shipment s1 is 3.41

volume of shipment s2 is : 24.0

weight of shipment s2 is: 0.76

Cost of shipment s2 is: 1.28

**Note:** In the previous example, all the three classes are placed in a single program file. You could chose to place all the three classes in their own files and compile them separately. In fact, using separate files is the generally followed practice.

**Invoke Constructors in a Class Hierarchy**

When a class hierarchy is created, and when an object of the subclass is created, the question that arises is “In what order are the constructors in a class hierarchy invoked?” Recall from your earlier discussion that the creation and initialization of the superclass object is a prerequisite to the creation of the subclass object.

When a subclass object is directly created without initially creating a superclass object explicitly, it first results in the creation of the superclass object, followed by the initialization of its attributes ensured by the invocation of a relevant superclass constructor. This is finally followed by the creation of the subclass object and the initialization of its own specific attributes through a relevant constructor invocation in the subclass.

Therefore, an important fact that emerges from our discussion is that constructors in a class hierarchy are invoked in the order of their derivation.

Consider the following example:

class X{

X( ) {

System.out.println(“Inside X's Constructor”);

}

}

class Y extends X {

Y()

System.out.println(“Inside Y's Constructor”);

}

}

class Z extends Y {

Z ( ){

System.out.println(“Inside Z's constructor”);

}

}

class OrderOfConstructorCallDemo {

public static void main (String args[]) {

Z z = new Z();

}

}

The output of the program is as follows:

Inside X's constructor

Inside Y's Constructor

Inside Z's Constructor

**Method Overriding**

Consider a class hierarchy in which a method in a subclass has the same name and type signature (the same type, the same sequence and the same number of parameters) as a method in superclass. In these cases the method in the subclass is said to override the method in the superclass. When an overridden method is called from an object of the 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 either be hidden or overridden:

Consider the following example:

class X{

int a,b;

X(int m, int n) {

a = m;

b = n;

}

//display a and b

void display() {

System.out.println(“a and b are :” + a + “ “ + b);

}

}

class Y extends X{

int c;

Y(int m,int n, int o) {

super (m,n);

c = 0;

}

void display() {//this overrides display in X

System.out.println(“c : “ + c);

}

}

class OverriddenDemo {

public static void main (String args[]) {

Y subOb = new Y (4,5,6);

subOb.display();// this calls display in Y

}

}

The output produced by this program is shown below:

c:6

When display() is invoked on an object of class Y, the version of display() defined within Y is invoked. In other words, the version of display() inside Y overrides the version of display declared in its superclass, i.e. Class X.

If you wish to access the superclass version of an overridden method from an object of the subclass, you can do so by using super. Consider the following example where the superclass version of display() is invoked within the subclass version.

class X {

int a,b;

X(int m, int n) {

a = m;

b = n;

}

//display a and b

void display() {

System.out.println)”a and b are :” + a + “ “ + b);

}

}

class Y extends X {

int c;

Y(int m, int n, int o) {

super(m,n);

c = 0;

}

void display () { // this overrides display in X

super.display(); // this calls X's display ()

System.out.println(“c :“ + c);

}

}

class OverrideDemo {

public static void main(String args[]) {

Y subOb = new Y(4,5,6);

subOb.display(); // this calls display in Y

}

}

The following is the output of the aforesaid program:

a and b: 4 5

c: 6

**Note:** Method overriding occurs only when the names and the types signatures of two methods across at least two classes (i.e., a superclass and a subclass) in a class hierarchy are identical. If they are not, the two methods are simply overloaded. Elaborating this theme further, if the method in a subclass has a different signature than the method in the superclass, then the subclass will have two forms of the same method. The subclass object will have one method with a specific signature inherited from the superclass and the other being its own method with the same name as the method in the superclass but with a different argument or arguments. This is nothing but a case of overloading.

The following example should help clarifying this point:

class X {

int a,b;

X(int m, int n) {

a = m;

b = n;

}

//display a and b

void display() {

System.out.println)”a and b are :” + a + “ “ + b);

}

}

class Y extends X {

int c;

Y(int m, int n, int o) {

super(m,n);

c = 0;

}

void display (String msg) {

//this overloads display in X

System.out.println(msg + c);

}

}

class OverrideDemo {

public static void main(String args[]) {

Y subOb = new Y(4,5,6);

subOb.display(“This is c :”); // this calls

//display() in Y

subOb.display( );//this calls display() in X()

//that an object of class Y inherited

}

}

The output of the program is as follows:

This is c: 6

a and b: 4 5

**Dynamic Method Dispatch**

Method overriding forms the basis of one of Java's most powerful concepts: dynamic method dispatch. Dynamic method dispatch is the phenomenon by which the Java language resolves a call to an overridden method at runtime and not compile time. Java uses dynamic method dispatch to implement runtime polymorphism. Java makes runtime polymorphism possible in a class hierarchy with the help of basically two of its features, namely superclass reference variables and overridden methods.

In retrospect, it is important to restate an important feature of Java at this juncture, especially in the matter of comprehending runtime polymorphism. It is the capability of a superclass reference. Java determines which version of the method to call. This decision is based upon the type of the object being referred to at the time the call occurs. This determines which method is to be called for which object/s (subclass objects from the same superclass) are method will be called. It is the type of the object that is 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 at the time when different types of subclass objects are being referred to through a superclass reference variable, different versions of the method are executed.

Here is an example that illustrates dynamic method dispatch:

// 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 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.

If you are familiar with C++ or C# will recognize that overridden methods in Java are similar to virtual functions in those languages.

**Why Overridden Methods? A Design Perspective**

Polymorphism is essential to object-oriented programming for one important reason: it allows a general class to specify methods that will be common to all of its derivatives or subclasses while allowing subclasses to define the specific implementation of some or all of those methods. Overridden methods in a class hierarchy are one of the ways that Java implements the “single interface, multiple methods” aspect of polymorphism.

Part of the key to successfully applying polymorphism is to understand the fact that the superclasses and subclasses form a hierarchy, which then moves from lesser to greater specialization. The superclass provides all elements that a subclass can use directly. It defines those methods that the subclass can use directly. It also defines those methods that the subclass must implement on its own. This allows the subclass the flexibility to define its own method implementations, yet still enforce a consistent interface (keeping the method name and the arguments the same through providing for an overridden function). Thus by combining inheritance with overridden methods, a superclass can define the general form of the methods, a superclass can define the general form of the methods that will be used and implemented by all of its subclasses in their own specific ways.

##### An Example Demonstrating Overridden Methods and Polymorphism

The following program creates a superclass called Figure that stores the dimensions of various two dimensional objects. It also defines a method called area() that computes the area of an object. The program derives two subclasses from Figure. The two subclasses are Rectangle and 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 x, double y) {

dim1 = x;

dim2 = y;

}

double area () {

System.out.println(“area of figure is undefined”);

return 0;

}

}

class Rectangle extends Figure {

Rectangle(double x,double y) {

super (x,y);

}

//override area for a rectangle

double area() {

System.out.println(“Area for rectangle is:”);

return dim1 \* dim2 /2;

}

}

class FindArea {

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 of rectangle is :” + figref.area());

figref = t;

System.out.println(“Area of triangle is : “+ figref.area());

figref =f;

System.out.println(figref.area());

}

}

The following is the output of the aforesaid program:

area of rectangle is 45

area of triangle is 40

area of figure is undefined

Through the dual mechanism of inheritance and runtime polymorphism, it is possible to define a single consistent interface that is used by different, yet related types of subclass objects in a class hierarchy. If an object is derived from Figure, then its area can be computed by calling area(). The interface to this operation is the same no matter what type of object is used.

**Pure Abstraction – Abstract Classes**

Quite often, you would want to define a superclass that declares the structure of a given abstraction without providing the implementation of every method. The objective is 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 provide for its own specific implementations. Such a class determines the nature of the methods that the classes must implement.

Such a superclass is unable to create a meaningful implementation for a method or methods. The class Figure in the previous example is such a superclass. Figure is a pure geometrical abstraction. You have only kinds of figures like Rectangle, Triangle etc. that actually are subclasses of class Figure. The class Figure does not determine the area of a Figure. The Figure class is therefore a partially defined class with no implementation for the area() method. The definition of area() is simply a placeholder.

It makes little sense for a partially defined superclass to be instantiated for there are going to be some methods with no implementations. The question that might arise in your mind is regarding the utility of a partially defined superclass in a class hierarchy that cannot be instantiated. The importance of such classes cannot be discounted because they define a generalized form (possibly some may be shared by all of its subclasses, leaving it to each subclass to provide for its own specific implementations of such methods).

The next design issue that needs to be addressed is the shape that needs to be given to methods with no implementations that you are bound to encounter in such generalized superclasses. One way is to give a relevant message when such a method is invoked indicating that it has no implementation. This was the case with the area() method in the class Figure which when invoked gave the message “ Area for figure is undefined”. While such an approach may be useful in debugging, it is not appropriate when you are creating a generalized superclass as part of reusable class libraries that you maybe creating. The right approach would be to define such methods (i.e., methods with no implementations in a generalized superclass) in such a way that it becomes mandatory for its subclasses to provide for its own version (overridden implementation). This should be done in such a way that it becomes mandatory for its subclasses to provide for its own version (overridden implementation) of such a method if the subclasses are to have a mechanism to ensure that a subclass does, indeed, override, all necessary methods. This mechanism comes in the shape of the abstract method.

You can specify that certain methods in a generalized superclass be compulsorily overridden by its subclasses by defining such methods in a generalized superclass as abstract. These methods are sometimes referred to as responsibile for the subclasses because they have no implementation specified in the superclass. It is mandatory for a subclass to override such method or methods, because it cannot simply use the version of the method defined in the superclass. To use an abstract method, use this general form:

abstract type name(parameter-list);

Abstract methods do not have a body. Abstract methods are therefore characterized by the lack of the opening and closing braces that are customary for any other normal method. This is a crucial benchmark for identifying an abstract class.

Any class that contains one or more abstract method must also be declared abstract. A class is declared as an abstract class by preceding the keyword abstract before the class keyword at the beginning of the class declaration. There can be no objects in an abstract class. An abstract class, cannot, therefore, be instantiated with the new keyword. It is pointless to instantiate an abstract class, because an abstract class is not fully defined. Also, as part of an abstract class declaration, you cannot define abstract constructors, or abstract static methods.

Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or by itself declare abstract. It is perfectly acceptable for a subclass to implement a concrete method. Such concrete methods, though not utilizable from an instance of an abstract class can of course be utilized from concrete instances of any of its subclasses. Abstract classes can include as much implementation as they deem fit.

Although abstract classes cannot be used to instantiate objects, you can have a reference variable of an abstract class. Java implements runtime polymorphism through the use of superclass reference variables. Thus, it must be possible to create a reference to an abstract class so that it can be used to reference its subclass object.

##### Improved Version of the Figure Class Hierarchy

The abstract class can be used to improve the declaration of the class Figure in the earlier example. Since there is no meaningful concept of area for an undefined two-dimensional geometrical abstraction such as a Figure, the following version of the program declares the area() abstract inside class Figure. This, of course implies that the class Figure be declared abstract, and all subclasses derived from class Figure must override area():

abstract class Figure {

double dim1;

double dim2;

Figure(double x, double y) {

dim1 = x;

dim2 = y;

}

abstract double area ();

}

class Rectangle extends Figure{

Rectangle(double x,double y) {

super (x,y);

}

//override area for a rectangle

double area() {

System.out.println(“Area for rectangle is:”);

return dim1 \* dim2 /2;

}

}

class Triangle extends Figure{

Triangle(double x, double y) {

super(x,y);

}

//override area for triangle

double area() {

System.out.println(“Area for rectangle is :”);

return dim1 \* dim2 / 2;

}

}

class FindArea {

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; // abstract superclass ref.

figref = r;

System.out.println(“Area of rectangle is :” + figref.area());

figref. = t;

System.out.println(“Area of triangle is : “+ figref.area());

figref =f;

System.out.println(figref.area());

}

}

**The Role of the Keyword Final in Inheritance**

The final keyword has two important uses in the context of class hierarchy. These uses are highlighted as follows:

**Using final with Inheritance to prevent Overriding**

While method overriding is one of the most powerful feature of object oriented design, there may be times when you will want to prevent certain critical methods in a superclass from being overridden by its subclasses. Rather, you would want the subclasses to use the methods as they are defined in the superclass. This can be achieved by declaring such critical methods as final. Final methods cannot be overridden. Any attempt by a subclass to do so will be flagged off as an error by the complier.

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. If you attempt to do so, a compile-time error will result.

Methods declared as final can sometimes provide a performance enhancement. The compiler is free to inlinecalls to them because it “knows” they will not be overridden by a subclass. When a small final method is called, often the Java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus eliminating the costly overhead associated with a method call. Inlining is only an option with final methods. Normally, Java resolves calls to methods dynamically, at run time. This is called late binding*.* However, since final methods cannot be overridden, a call to one can be resolved at compile time. This is called early binding.

**Using final to Prevent Inheritance**

Sometimes you will want to prevent a class from being inherited. This can be achieved by preceding the class declaration with final. Declaring a class as final implicitly declares all of its methods as final too. 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 provided concrete and complete implementations.

Here is an example of a final class:

final class A {

// ...

}

// The following class is illegal.

class B extends A { // ERROR! Can't subclass A

// ...

}

As the comments imply, it is illegal for B to inherit A since A is declared as final.

###### The Cosmic Class – The Object Classification

Java defines a special class called Object. All other classes are subclasses of Object. Object is a superclass of all other classes; i.e Java's own classes as well as user-defined classes. This means that a reference variable of type Object can refer to an object of any other class.

Object defines the following methods, which means that they are available in every object.

| Method | Explanation |
| --- | --- |
| Object clone() | Create a new object that is the same as the object being cloned. |
| Boolean equals(Object object) | Determines whether one object is equal to another. |
| Void finalize() | Called before an unused object is reclaimed from the heap by the garbage collector. |
| Final Class getClass() | Obtains the class of an object at runtime. |
| Int hashCode | Returns the hash code associated with the invoking object. |
| Final void notify() | Resumes execution of all waiting threads on the invoking object. |
| Fina void notifyAll() | Resumes execution of all waiting threads on the invoking object. |
| String toString() | Returns a string that describes the object. |
| Final void wait  final void wait(long milliseconds)  final void wait(long milliseconds,long nanoseconds) | Waits on another thread of execution. |

The toString() method returns a string that contains a description of the object on which it is called. Also, this method is automatically called when an object is output (passed as ab argument ) using println(). Many classes override this method. Doing so allows them to tailor a version specifically suited for the types of objects that they create.

#### Summary

In this chapter you have learnt about one of the main concept of OOPs- *inheritance*. You learnt that:

* Inheritance is a mechanism where one class will get the properties and behavior of another class.
* In Java in order to implement the concept of inheritance we use the keyword extends
* The main class is called parent class or super class, the derived class is called child class or sub class
* A subclass inherits all of the public and *protected* members of its parent, no matter what package the subclass is in. A subclass does not inherit the private members of its parent class.
* Casting shows the use of an object of one type in place of another type, among the objects permitted by inheritance and implementations.
* A subclass can call a constructor method defined by its superclass by use of super(). Even the methods of parent class can be called using super.
* 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.
* Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time
* Superclass 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 super class is called as Abstract Class.
* Finalkeyword applied to variables makes them constant. Final methods cannot be overridden by the child, final classes cannot be inherited.