Inheritance

- Inheritance Basics
- super keyword
- Creating a Multilevel Hierarchy
- When Constructors Are Called
- Method Overriding

Inheritance basics

- Inheritance is the java's method to reuse the existing code/classes.
- The mechanism of deriving a new class from an existing class is called inheritance.
- The old class is known as base class/super class/parent class and new class is known as sub class/derived class/child class.
- Inheritance allows subclass to inherit all the members(variable and method) of super classes

- *extends* is the keyword for inheritance
- Inheritance may take following forms
- Single (One super class)
- Multiple (Several super classes)
- Hierarchical (One superclass, many subclasses)
- Multilevel (Derived from a derived class)
- Java does not impliment multiple inheritance directly.(Uses interface for it).

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

```
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 of the program is->

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

General format for inheritance:

class *subclass-name extends superclass-name {*

// body of class

}

Member Access and Inheritance

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

[A class member that has been declared as private will remain private to its class. It is not accessible by any code outside its class, including subclasses.]*

A Superclass Variable Can Reference a Subclass Object

- A reference variable of a super class can be assigned a reference to any subclass derived from that super class. This aspect of inheritance quite useful in a variety of situations

```
// This program uses inheritance to extend Box.
class Box {
double width;
double height;
double depth;
// construct clone of an object
Box(Box ob) { // pass object to constructor
width = ob.width;
height = ob.height;
depth = ob.depth; }
// constructor used when all dimensions specified
Box(double w, double h, double d) {
width = w;
height = h;
depth = d; }
// constructor used when no dimensions specified
Box() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box }
```

```
// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
// Here, Box is extended to include weight.
class BoxWeight extends Box {
double weight; // weight of box
// constructor for BoxWeight
BoxWeight(double w, double h, double d, double m) {
width = w;
height = h;
depth = d;
weight = m;
}
}
```

```
class RefDemo {
public static void main(String args[]) {
BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
Box plainbox = new Box();
double vol;
vol = weightbox.volume();
System.out.println("Volume of weightbox is " + vol);
System.out.println("Weight of weightbox is " + weightbox.weight);
System.out.println();
// assign BoxWeight reference to Box reference
plainbox = weightbox;
vol = plainbox.volume(); // OK, volume() defined in Box
System.out.println("Volume of plainbox is " + vol);
/* The following statement is invalid because plainbox
does not define a weight member. */
// System.out.println("Weight of plainbox is " + plainbox.weight); } }
```

- Here, weightbox is a reference to BoxWeight objects, and plainbox is a reference to Box objects. Since BoxWeight is a subclass of Box, it is permissible to assign plainbox a reference to the weightbox object.
- It is important to understand that it is the type of the reference variable—not the type of the object that it refers to—that determines what members can be accessed.
- That is, when a reference to a subclass object is assigned to a super class reference variable, you will have access only to those parts of the object defined by the super class. This is why plainbox can’t access weight even when it refers to a BoxWeight object

Using super

- Whenever a subclass needs to refer to its immediate super class, it can do so by use of the keyword super.
- super has two general forms. The first calls the super class' constructor. The second is used to access a member of the super class that has been hidden by a member of a subclass.

```
class A
{
int a;
int b;
A(int i,int j) {
 a=i;
 b=j;
 System.out.println("HI from A");
}
}
class B extends A{
int c;
B(int i,int j,int k) {
super(i,j);
c=k;
System.out.println("HI from B");
}
public static void main(String args[]){
 B obj_b=new B(1,2,3);
}
}
```

- Since constructors can be overloaded, super( ) can be called using any form defined by the super class. The constructor executed will be the one that matches the arguments.

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

```
class B extends A{
int c;
B(int i,int j,int k){
 super(i,j);
 c=k;
 }
B(B obj_b){
super(obj_b);
c=obj_b.c;
System.out.println("HI from B");
}
public static void main(String args[])
{
 B obj_b1=new B(1,2,3);
 B obj_b2=new B(obj_b1);
}
}
```

[Her the copy constructor of class B pass object of class B('obj_b') to copy constructor of class A which collect the result in 'obj_a'{same as obj_a=obj_b}, this is one case where a super class variable is referring to a subclass object. Another case will come in overriding].

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

  super.*member*

- 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 super class.

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

- Output->
- i in super class: 1
- i in subclass: 2

Creating a Multilevel Hierarchy

- We can build hierarchies that contain as many layers of inheritance as you like. It is perfectly acceptable to use a subclass as a super class of another. For example, given three classes called A, B and C, C can be a subclass of B, which is a subclass of A.
- When this type of situation occurs, each subclass inherits all of the traits found in all of its super classes. In this case, C inherits all aspects of B and A.

Order of constructor calling

- In a class hierarchy, constructors are called in order of derivation, from super class to subclass.
- Further, since super( ) must be the first statement executed in a subclass' constructor, this order is the same whether or not super( ) is used. If super( ) is not used, then the default or parameterless constructor of each super class will be 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();
} }
```

O/p->

Inside A’s constructor

Inside B’s constructor

Inside C’s constructor

Method Overriding

- In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its super class, then the method in the subclass is said to *override the method in the super class.*
- *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 super class 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
}
}
```

output ->k: 3

- If we want to call method of superclass then use super. Like

super.show();//this calls to show in A

```
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);
}  }
```

O/p in this case will be ->i and j: 1 2

k: 3

Inheritance

- Dynamic Method Dispatch
- Why Overridden methods
- Using Abstract Classes
- Using final with Inheritance

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.
- 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 occurs. Thus, this determination is made at run time.

- 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.

** The overridden methods in Java are similar to virtual functions in c++/c# languages.*

```
// 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
}
}
```

Output ->

Inside A’s callme method

Inside B’s callme method

Inside C’s callme method

Why Overridden methods

- Overridden methods allow Java to support run-time polymorphism. Polymorphism is essential to object-oriented programming for one reason: it allows a general class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods. Overridden methods are another way that Java implements the “one interface, multiple methods” aspect of polymorphism.
- By combining inheritance with overridden methods, a superclass can define the general form of the methods that will be used by all of its subclasses.

Using Abstract Classes

- If we want certain methods be overridden by subclasses then it can be done by specifying the abstract type modifier. These methods are sometimes referred to as *subclasser responsibility because they have no implementation specified in the superclass. Thus,* a subclass must override them. To declare an abstract method, use this general form:

  abstract *type name(parameter-list);*
- *The method body is not present, as it will be defined by subclass.*
- Any class that contains one or more abstract methods must also be declared abstract.

- There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the **new** operator. Such objects would be useless, because an abstract class is not fully defined.
- Also, we cannot declare abstract constructors, or abstract static methods. Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared **abstract.**

```
// A Simple demonstration of abstract.
abstract class A {
abstract void callme();
// concrete methods are still allowed in abstract classes
void callmetoo() {
System.out.println("This is a concrete method.");
}
}
class B extends A {
void callme() {
System.out.println("B's implementation of callme.");
}
}
class AbstractDemo {
public static void main(String args[]) {
B b = new B();
b.callme();
b.callmetoo();
}  }
```

O/p-> B's implementation of callm
This is a concrete method.

Using final with Inheritance

- The keyword final has three uses.
- First, it can be used to create the equivalent of a named constant.(final float PI=3.14;)
- Using final to Prevent Overriding
- Using final to Prevent Inheritance
- When we want to prevent overriding from occurring, specify final as a modifier at the start of its declaration. Methods declared as final cannot be overridden. 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 *final method* can be resolved at compile time. This is called ***early binding***.

- Sometimes we want to prevent a class from being inherited. For this, precede 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 provide complete implementations.

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