**LAB NO: 6 Date:**

# CLASS INHERITANCE

**Objectives:**

1. To understand the basics of inheritance
2. To learn the concept of method overriding
3. To study the dynamic method dispatch
4. To write Java programs using the concepts of inheritance, method overriding and dynamic method dispatch
   1. **Basics of inheritance**

Inheritance allows the creation of hierarchical classifications. Using inheritance, it can create a general class that defines 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.* Therefore, a subclass is a specialized version of a superclass. It inherits all of the instance variables and methods defined by the superclass, and add its own unique elements. A keyword called “***extends***” is used for inheritance.

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

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

// body of class

}

It can only specify one superclass for any subclass because Java does not support the inheritance of multiple super classes into a single subclass. But, it is possible to create a hierarchy of inheritance in which a subclass becomes a superclass of another subclass. However, no class can be a superclass of itself.

NOTE:

* A sub class cannot access those members of the superclass that have been declared as **private**.
* A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass.
* 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 form 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.
* In a class hierarchy, constructors are called 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.

// 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();

}

}

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

* 1. **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

// 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

* 1. **Dynamic method dispatch**

Dynamic method dispatch is the mechanism by which a call to an overridden function 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. 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 superclass reference variable, different versions of the method are executed.

// 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 below:

Inside A's callme method

Inside B's callme method

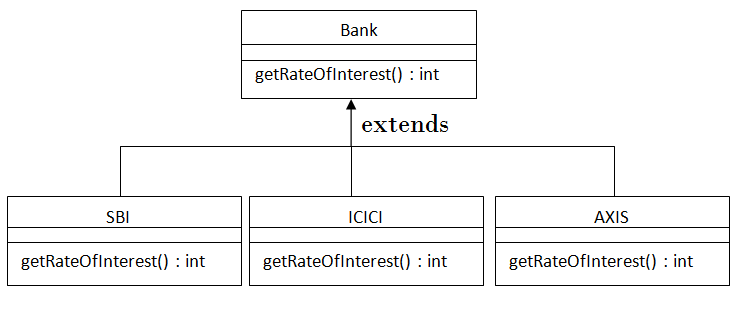
Inside C's callme method

**Lab exercises**

1. Create an Account class that stores customers name, acc-no and type of account. From this derive class current account and savings bank account. Include necessary methods in order to achieve following tasks:-
2. Accept the deposit from a customer and update the balance
3. Display the balance
4. Compute and deposit interest
5. Permit withdraw and update the balance
6. Check for minimum balance impose penalty if necessary and update the balance

For savings bank account the facilities given are computing interest, withdraw facility. No interest on current bank account and should maintain a minimum balance and if balance is less than this level, service tax is imposed.

1. Create a base class for student having registration number, name and age. From this class create two new class UG and PG student with semester and fees as its data members. Use proper member function for demonstrating inheritance. Display the details of students who took admission to UG course and PG course separately, total number of UG admissions and PG admissions.
2. Create a base class called Bank that provides functionality to get rate of interest. But, rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7% and 9% rate of interest. Write a Java program to calculate the interest for SBI, ICICI and AXIS banks, by demonstrating the concept of method overriding and dynamic method dispatch.



**Additional exercises:**

* + - 1. Create a base class “Game” with method called “type” that prints “indoor & outdoor games”. Create a subclass cricket with a method called “type” that prints “cricket is an outdoor game”. Create one more subclass of “Game” called “chess” with a method “type” that prints “chess is an indoor game”. Write a complete Java program for the above, to understand the Dynamic method dispatch concept.
      2. Create two classes Bike and Splendar. Splendar class extends Bike class and overrides its run() method. Write a complete Java program to implement the runtime polymorphism. Include a member called “speedlimit” in both the classes with different values. The run() method should give the information of speed limit. Check whether the runtime polymorphism can be achieved through the data members.