###### UNIT-III

**Inheritance** – Inheritance hierarchies- super and subclasses- member access rules- super keyword- preventing inheritance: final classes and methods- the object class and its methods **Polymorphism –**dynamic binding- method overriding- abstract classes and methods

**Interface** – Interfaces VS Abstract classes- defining an interface- implement interfaces- accessing implementations through interface references- extending interface.

**Inheritance** –Hierarchical abstractions, Base class object, subclass, subtype, substitutability, forms of inheritance- specialization, specification, construction, extension, limitation, combination, benefits of inheritance, costs of inheritance. Member access rules, super uses, using final with inheritance, polymorphism- method overriding, abstract classes.

###### Unit – III

###### Inheritance: Inheritance basics, member access and inheritance, constructors, and

###### inheritance, using super, multilevel hierarchy, when are constructors executed, superclass

###### reference and subclass objects, method overriding, polymorphism, using abstract classes.

###### Interfaces: interface fundamentals, creating, implementing, and using interfaces,

###### implementing multiple interfaces.

###### Unit – IV

###### Packages: Package fundamentals, packages and member access, importing packages, static

###### import.

###### Exception handling: the exception hierarchy, exception handling fundamentals, exception

###### types, uncaught exceptions, using try and catch, multiple catch clauses, catching subclass

###### exceptions, nested try, throw, throws, finally, Java’s built-in exceptions, creating your own

###### exception subclasses

###### **Hierarchical abstractions & Concept of Inheritance:**

* In OOP another important feature is Reusability which is very use full in two major issues of development :

Saving Time

Saving Memory

* In C++/Java the mechanism implements this reusability is Inheritance.

General form of a class declaration is:

class subclass-name extends super class-name

{

// body of super class

}

// 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:

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

###### Forms of Inheretence:

Single Inheretence Hierarichal Inherintence Multiple Inherintence Multilevel Inherintence Hybrid Inherintence



Single Inherintence:

Derivation a subclass from only one super class is called Single Inherintence. Hierarchical Inherintence:

Derivation of several classes from a single super class is called Hierarchical Inherintence:

Multilevel Inheritance:

Derivation of a classes from another derived classes called Multilevel Inheritance.

Multiple Inheritance:

Derivation of one class from two or more super classes is called Multiple Inheritance

But java does not support Multiple Inheritance directly. It can be implemented by using interface concept.

###### Defining a Subclass:

A subclass is defined as

Syntax: class subclass-name extends superclass-name

{

Variable declaration;

Method declaration;

}

The keyword extends signifies that the properties of the super class name are extended to the subclass name. The subclass will now contain its own variables and methods as well as those of the super class. But it is not vice-versa.

###### Member access rules

* Even though a subclass includes all of the members of its super class, it cannot access those members who are declared as **Private** in super class.
* We can assign to a reference of super class the object of sub class. In that situation we can access only super class members but not sub class members.

―Super class Reference, Sub class Object‖.

/\* In a class hierarchy, private members remain private to their class.

This program contains an error and will not compile.

\*/

// Create a superclass. class A {

int i;

// public by default

private int j;

// private to A

void setij(int x, int y) {

i = x;

j = y;

}

}

// A's j is not accessible here.

class B extends A {

int total;

void sum() {

total = i + j; // ERROR, j is not accessible here

}

}

class Access {

public static void main(String args[]) {

B subOb = new B();

subOb.setij(10, 12);

subOb.sum();

System.out.println("Total is " + subOb.total);

}

}

Super class variables can refer sub-class object

* + To a reference variable of a super class can be assigned a reference to any subclass derived from that super class.
  + When a reference to a subclass object is assigned to a super class reference variable, we will have to access only to those parts of the object defined by the super class
  + It is because the super class has no knowledge about what a sub class adds to it.

###### Program

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);

}

}

###### Using super keyword

* When ever a sub class needs to refer to its immediate super class, it can do so by use of the key word **super**.
* Super has two general forms:
  + Calling super class constructor
  + Used to access a member of the super class that has been hidden by a member of a sub class

Using super to call super class constructor

* A sub class can call a constructor defined by its super class by use of the following form of super:
  + super (parameter-list);
  + Parameter list specifies parameters needed by the constructor in the super class.

Note: Super ( ) must always by the first statement executed inside a sub-class constructor.

// A complete implementation of BoxWeight. class Box {

private double width;

private double height;

private double depth;

// construct clone of an object

Box(Box ob) {

// pass object to constructor

width = ob.width;

height = ob.height;

depth = ob.depth;

}

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

width = w;

height = h;

depth = d;

}

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

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

height = -1; // an uninitialized

depth = -1; // box

}

// constructor used when cube is created Box(double len) {

width = height = depth = len;

}

// compute and return volume

double volume() {

return width \* height \* depth;

}

}

// 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

Calling members of super class using super

* The second form of super acts somewhat like **this** keyword, except that it always refers to the super class of the sub class in which it is used.
* The syntax is:

###### Super.member ;

* + Member can either be method or an instance variable

###### Program

// 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 superclass: 1 i in subclass: 2

###### When the constructor called:

**2**

Always the super class constructor will be executed first and sub class constructor will be executed last.

// Demonstrate when constructors are called.

// 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 extendingB.

class C extends B{ C() {

System.out.println("Inside C's constructor.");

}

}

class CallingCons {

public static void main(String args[]) {

C c = new C();

}

}

|  |  |  |
| --- | --- | --- |
| Output: |  | |
| Inside | A’s | constructor |
| Inside | B’s | constructor |
| Inside | C’s | constructor |

###### Using Final keyword:

The final keyword is **a non-access modifier used for classes, attributes and methods, which makes them non-changeable**

We can use final key word in **three** ways:

* Used to create equivalent of a named constant

Final datatype identifier = ;

o Used to prevent inheritance

Final class …………..

* Used to avoid overloading

Final return type ………….

###### Using final to Prevent Overriding:

While method overridingul features,is thereonewill ofbetimesJava’swhenyou most will want to prevent it from occurring. 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!");

}

}

###### Using final to Prevent Inheritance:

Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with **final**. Declaring a class as **final** implicitly declares all of its methods as **final**, too. As you might expect, 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.

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**. **Polymorphism ->Method overriding:**

In a class hierarchy, when a method in a sub class has the same name and type signature as a method in its super class, then the method in the sub class is said to be override the method in the super class.

When an overridden method is called from within a sub class, it will always refers to the version of that method defined by the sub class.

The version of the method defined in the super class is hidden.



In this situation, first it checks the method is existed in super class or not. If it is existed then it executes the version of sub class otherwise it gives no such method found exception.

Note: Methods with different signatures overloading but not overriding.

// 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

###### Dynamic method dispatch

* It is a mechanism by which a call to an overridden method is resolved at run time rather then compile time.
* It is important because this is how java implements runtime polymorphism.
* Before going to that we must know about super class reference sub class object.

// 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

###### **Abstract class:**

* An abstract method is a method that is declared with only its signatures without implementations.
* An abstract class is class that has at least one abstract method.

###### The syntax is:

Abstract class class-name

{

Variables

Abstract methods; Concrete methods;

.

.

.}

* We can’t declare any abstract constructor.
* Abstract class should not include any abstract static method.
* Abstract class can’t be directly instantiated.
* Any sub class of abstract class must be either implement all the abstract methods in the super class or declared itself as abstract.
* Abstract modifier*subclass responsibilities* referred‖.sebecauofasnoimplementation― of methods. Thus, a sub class must overridden them.

// 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();

}

}

// Using abstract methods and classes. 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 right triangle double area() {

System.out.println("Inside Area for Triangle."); return dim1 \* dim2 / 2;

}

}

class AbstractAreas

{

public static void main(String args[])

{

// Figure f = new Figure(10, 10); // illegal now

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());

}

}

**Interfaces** : Defining, Creating and Accessing a Package, Understanding CLASSPATH, importing packages, differences between classes and interfaces, defining an interface, implementing interface, applying interfaces, variables in interface and extending interfaces.

###### Difference between classes and interfaces:

A class is a template for an object. (or)

A class is a way of binding variables and methods in a single unit. With the class it is possible to create object for that object. With the one class we can extend an another class.

A interface is collection of undefined method. Means all the methods are not contain any body. We have to provide the body for that interface. With the interface it is not possible to create object. For the declared interface we have to implement that interface.

###### Defining Interfaces:

Interface is a collection of method declarations and constants that one or more classes of objects will use.

We can implement multiple inheritance using interface.



Because interface consists only signatures followed by semi colon and parameter list they are implicitly abstract.

Variables can be declared and initialized inside interface they are implicitly final and static.

An interface method can’t be final or stat An interface can be extended from another interface. Declaration of interface:



Access interface name

{

Return type member-name1(parametelist); Return type member-name2(parametelist);

.

.

.

Type finalvariablename=initialization;

}

* There will be no default implementation for methods specified in an interface.
* Each class that include interface must implements all methods.
* All the methods and variables are implicitly public if interface itself is declared as public.

Implementing Interfaces:

Once an **interface** has been defined, one or more classes can implement that interface. To implement an interface, include the **implements** clause in a class definition, and then create the methods defined by the interface. The general form of a class that

includes the **implements** clause looks like this:

*access* class *classname* [extends *superclass*] [implements *interface* [,*interface...*]] {

// class-body

}

Here, *access* is either **public** or not used. If a class implements more than one interface, the interfaces are separated with a comma. If a class implements two interfaces that declare the same method, then the same method will be used by clients of either

interface. The methods that implement an interface must be declared **public**. Also, the type signature of the implementing method must match exactly the type signature specified in the **interface** definition.

###### Applying Interfaces:

To understand the power of interfaces, let chapters you developed a class called **Stack** that implemented a simple fixed-size stack.

However, there are many ways to implement a stack. For example, the stack can be of a fixed

size or it can be ―growable.‖ The stack can and so on. No matter how the stack is implemented, the interface to the

stack remains the same. That is, the methods **push( )** and **pop( )** define the interface to the stack independently of the details of the implementation. Because the interface to a stack is separate from its implementation, it is easy to define a stack interface, leaving it

to each implementation to define the specific

First, here is the interface that defines an integer stack. Put this in a file called

**IntStack.java**. This interface will be used by both stack implementations.

// Define an integer stack interface. interface IntStack {

void push(int item); // store an item int pop(); // retrieve an item

}

###### Variables in Interfaces:

When you include that interface in a class interface), all of those variable names will be in scope as constants. This is similar to using a header file in C/C++ to create a large number of **#defined** constants or **const**

declarations. If an interface contains no methods, then any class that includes such an interface doesn’tplementanythingactually.Itisasifthatclass wereimimporting the constant variables into the class name space as **final** variables.

import java.util.Random; interface SharedConstants

{ int NO = 0; int YES = 1;

int MAYBE = 2; int LATER = 3; int SOON = 4; int NEVER = 5;

}

class Question implements SharedConstants

{ Random rand = new Random(); int ask() {

int prob = (int) (100 \* rand.nextDouble()); if (prob < 30)

return NO; // 30% else if (prob < 60) return YES; // 30% else if (prob < 75) return LATER; // 15% else if (prob < 98) return SOON; // 13% else

return NEVER; // 2%

}

}

class AskMe implements SharedConstants

{ static void answer(int result) { switch(result) {

case NO: System.out.println("No"); break;

case YES: System.out.println(―yes‖); break;

case MAYBE: System.out.println("Maybe"); break;

case LATER: System.out.println("Later"); break;

case SOON: System.out.println("Soon");

break;

case NEVER: System.out.println("Never"); break;

}

}

public static void main(String args[]) { Question q = new Question(); answer(q.ask());

answer(q.ask());

answer(q.ask());

answer(q.ask());

}

}

###### Interfaces Can Be Extended:

One interface can inherit another by use of the keyword **extends**. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

Following is an example:

// One interface can extend another. interface A {

void meth1(); void meth2();

}

// B now includes meth1() and meth2() -- it adds meth3(). interface B extends A {

void meth3();

}

// This class must implement all of A and B class MyClass implements B {

public void meth1() {

System.out.println("Implement meth1().");

}

public void meth2() { System.out.println("Implement meth2().");

}

public void meth3() { System.out.println("Implement meth3().");

}

}

class IFExtend {

public static void main(String arg[]) {

MyClass ob = new MyClass();

ob.meth1();

ob.meth2();

ob.meth3();

}

}

As an experiment you might want to try removing the implementation for **meth1( )** in **MyClass**. This will cause a compile-time error. As stated earlier, any class that implements an interface must implement all methods defined by that interface, including any that are inherited from other interfaces.