Unit - 2

Inheritance, Packages, Interfaces

Inheritance: Basics, Using Super, Creating Multilevel hierarchy, Method overriding, Dynamic Method Dispatch, Using Abstract classes, Using final with inheritance, Object class,

Packages: Basics, Finding packages and CLASSPATH, Access Protection, Importing packages.

Interfaces: Definition, Implementing Interfaces, Extending Interfaces, Nested Interfaces, Applying Interfaces, Variables in Interfaces.

Inheritance

- **Definition:** Inheritance is a mechanism in Java by which derived class can borrow the properties of base class and at the same time the derived class may have some additional properties.
- The inheritance can be achieved by incorporating the definition of one class into another using the keyword extends.

Advantages of Inheritance

One of the key benefits of inheritance is to minimize the amount of duplicate code in an application by sharing common code amongst several subclasses.

1. Reusability : The base class code can be used by derived class without any need to rewrite the code.
2. Extensibility : The base class logic can be extended in the derived classes.
3. Data hiding : Base class can decide to keep some data private so that it cannot be altered by the derived class.
4. Overriding : With inheritance, we will be able to override the methods of the base class so that meaningful implementation of the base class method can be designed in the derived class

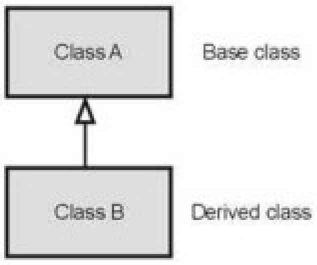
Concept of Super and Sub Classes

- The inheritance is a mechanism in which the child class is derived from a parent class.
- This derivation is using the keyword extends.
- The parent class is called base class and child class is called derived class.

For example

```
        Class A //This is Base class
        {
        ...
        }
        Class B extends A // This is Derived class
        {
        ... // uses properties of A
}
```

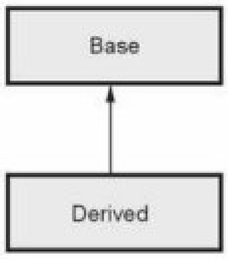
Inheritance is represented diagrammatically as follows



Types of Inheritance

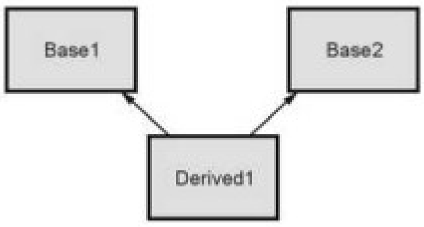
1. **Single inheritance :**

   - In single inheritance there is one parent per derived class. This is the most common form of inheritance.
   - The simple program for such inheritance is -

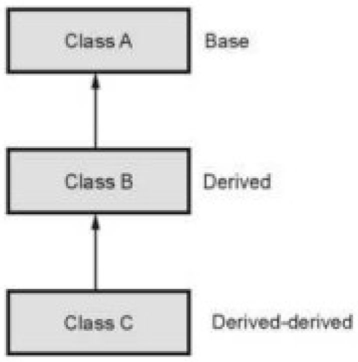


2. **Multiple inheritance :**

   - In multiple inheritance the derived class is derived from more than one base class. Java does not implement multiple inheritance directly but it makes use of the concept called interfaces to implement the multiple inheritance.



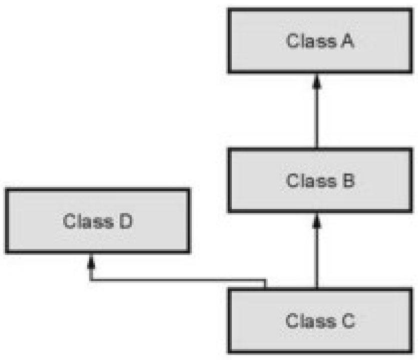
3. Multilevel inheritance :



When a derived class is derived from a base class which itself is a derived class then that type of inheritance is called multilevel inheritance.

For example - If class A is a base class and class B is another class which is derived from A, similarly there is another class C being derived from class B then such a derivation leads to multilevel inheritance.

4. Hybrid inheritance :



When two or more types of inheritances are combined together then it forms the hybrid inheritance. The given Fig. represents the typical scenario of hybrid inheritance.

<u>Implementation of Different Types of Inheritance</u>

Single Inheritance

- The class which is inherited is called the base class or the superclass and the class that does the inheriting is called the derived class or the subclass.
- The method defined in base class can be used in derived class. There is no need to redefine the method in derived class. Thus inheritance promotes software reuse.
- The subclass can be defined as follows –

  class nameofSubclass **extends** superclass

  {

  variable declarations

  method declarations

```
    }
```

Note that the keyword extends represents that the properties of superclass are extended to the subclass. Thus the subclass will now have both the properties of its own class and the properties that are inherited from superclass.

- Following is a simple Java program that illustrates the concept of single inheritance -

Java Program[InheritDemo1.java]

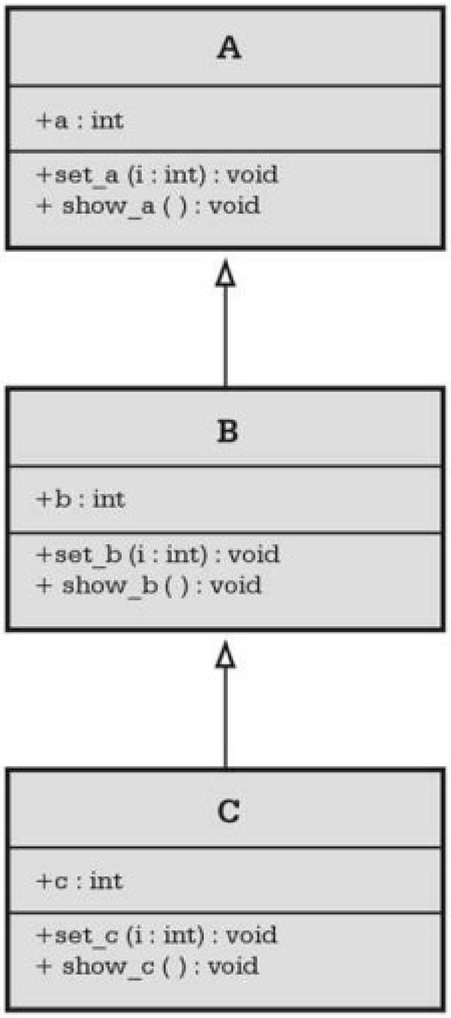
```
class A
{
int a;
void set_a(int i)
{
a=i;
}
void show_a()
{
System.out.println("The value of a= "+a);
}
}
class B  extends  A  //extending the base class A
{
int b;
void set_b(int i)
{
b=i;
}
void show_b()
{
System.out.println("The value of b= "+b);
}
void mul()
{
int c;
c=a*b;
System.out.println(" The value of c= "+c);
}
}
class InheritDemo1
{
public static void main(String args[])
{
A obj_A=new A();
B obj_B=new B();
obj_B.set_a(10);
obj_B.set_b(20);
obj_B.show_a();
obj_B.show_b();
obj_B.mul();
}  }
```

Output

```
F:\test>javac InheritDemo1.java
F:\test>java InheritDemo1
The value of a= 10
The value of b= 20
The value of c= 200
```

Multilevel Inheritance

The multilevel inheritance is a kind of inheritance in which the derived class itself derives the subclasses further.



In the following program, we have created a **base class A** from which the **subclass B is derived.**

There is a **class C** which is **derived from class B.** In the function main we can access any of the field in the class hierarchy by creating the object of class C.

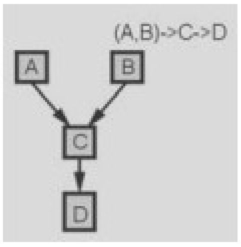
Java Program[MultiLvlInherit.java]

```
class A
{
int a;
void set_a(int i)
{
a=i;
}
void show_a()
{
System.out.println("The value of a= "+a);
}
}
class B extends A
{
int b;
void set_b(int i)
{
b=i;
}
void show_b()
{
System.out.println("The value of b= "+b);
}
}
class C extends B
{
int c;
void set_c(int i)
{
c=i;
}
void show_c()
{
System.out.println("The value of c= "+c);
}
void mul()
{
int ans;
ans=a*b*c;
System.out.println(" The value of ans= "+ans);
}
}
class MultiLvlInherit
{
public static void main(String args[])
{
A obj_A=new A();
B obj_B=new B();
C obj_C=new C();
obj_C.set_a(10);
obj_C.set_b(20);
obj_C.set_c(30);
obj_C.show_a();
obj_C.show_b();
obj_C.show_c();
obj_C.mul();
}
}
```

Output

```
The value of a= 10
The value of b= 20
The value of c= 30
The value of ans= 6000
```

Explain single level and multiple inheritances in java. Write a program to demonstrate combination of both types of inheritance as shown in given fig. hybrid inheritance.



```
class A
{
int a=10;
}
interface B
{
int b=20;
}
class C extends A implements B
{
int c;
int mul()
{
c=a*b;
return c;
}
}
class D extends C
{
```

```
void sum()
{
System.out.println("Adding all the three variables");
int d=a+b+mul();
System.out.println(d);
}
}
class Demo1
{
public static void main(String[] args)
{
C obj1=new C();
D obj2=new D();
System.out.println("Multiplying two variables");
System.out.println(obj1.mul());
obj2.sum();
}
}
```

Output

```
Multiplying two variables
200
Adding all the three variables
230
```

Method Overloading and Method Overriding

Overriding

Method overriding is a mechanism in which a subclass inherits the methods of superclass and sometimes the subclass modifies the implementation of a method defined in superclass. The method of superclass which gets modified in subclass has the same name and type signature. The overridden method must be called from the subclass. Consider following Java Program, in which the method(named as fun) in which a is assigned with some value is modified in the derived class. When an overridden method is called from within a subclass, it will always refer to the version of that method re-defined by the subclass. The version of the method defined by the superclass will be hidden.

Java Program[OverrideDemo.java]

```
class A
{
int a=0;
void fun(int i)
{
this.a=i;
}
}
class B extends A
{
int b;
void fun(int i)
{
int c;
b=20;
super.fun(i+5);
System.out.println("value of a:"+a);
System.out.println("value of b:"+b);
c=a*b;
System.out.println("The value of c= "+c);
}
}
class OverrideDemo
{
public static void main(String args[])
{
B obj_B=new B();
obj_B.fun(10);//function re-defined in derived class
}
}
```

Output

```
F:\test>javac OverrideDemo.java
F:\test>java OverrideDemo
value of a:15
value of b:20
The value of c= 300
```

Rules to be followed for method overriding

1. The private data fields in superclass are not accessible to the outside class. Hence the method of superclass using the private data field cannot be overridden by the subclass.
2. An instance method can be overridden only if it is accessible. Hence private method cannot be overridden.
3. The static method can be inherited but cannot be overridden.
4. Method overriding occurs only when the name of the two methods and their type signatures is same.

Methods Overloading

Overloading is a mechanism in which we can use many methods having the same function name but can pass different number of parameters or different types of parameter.

For example :

```
int sum(int a,int b);
double sum(double a,double b);
int sum(int a,int b,int c);
```

That means, by overloading mechanism, we can handle different number of parameters or different types of parameter by having the same method name.

Following Java program explains the concept overloading -

Java Program [OverloadingDemo.java]

```
public class OverloadingDemo {
public static void main(String args[]) {
System.out.println("Sum of two integers");
Sum(10,20); <-------------- line A
System.out.println("Sum of two double numbers");
Sum(10.5,20.4); <-------------- line B
System.out.println("Sum of three integers");
Sum(10,20,30); <-------------- line C
}
public static void Sum(int num1,int num2)
{
int ans;
ans=num1+num2;
System.out.println(ans);
}
public static void Sum(double num1,double num2)
{  double ans;
ans=num1+num2;
System.out.println(ans);
}
public static void Sum(int num1,int num2,int num3)
{  int ans;
ans=num1+num2+num3;
System.out.println(ans);
}  }
```

Output

```
F:\test>javac OverloadingDemo.java
F:\test>java OverloadingDemo
Sum of two integers
30
Sum of two double numbers
30.9
Sum of three integers
60
```

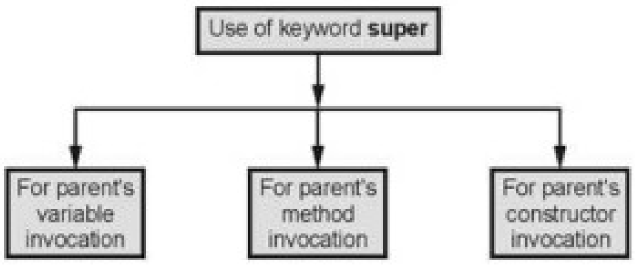
Difference between Method Overloading and Method Overriding

Method Overloading	**Method Overriding**
The method overloading occurs at **compile time.**	The method overriding occurs at the **run time or execution time.**
In case of method overloading **different number of parameters** can be passed to the function.	In function overriding the **number of parameters** that are passed to the function are the **same.**
The overloaded functions may have **different return types.**	In method overriding all the methods will have **the same return type.**
Method overloading is performed within a class.	Method overriding is normally performed between two classes that have inheritance relationship.

Use of keyword Super

Super is a keyword used to access the immediate parent class from subclass.

There are three ways by which the keyword **super** is used.


Use of keyword super
For parent's variable invocation
For parent's method invocation
For parent's constructor invocation

Let us understand these uses of keyword super with illustrative Java programs.

1. **The super() is used to invoke the class method of immediate parent class.**

Java Program[B.java]

```
class A
{
int x=10;
}
class B extends A
{
int x=20;
void display()
{
System.out.println(super.x);
}
public static void main(String args[])
{
B obj=new B();
obj.display();
}
}
```

Output

```
10
```

Program Explanation : In above program class A is a immediate parent class of class B. Both the class A and Class B has variables x. In class A, the value of x variable is 10 and in class B the value of variable x is 20. In display function if we would write

System.out.println(x);

The output will be 20 but if we user super.x then the variable x of class A will be referred. Hence the output is 10.

2. **The super() is used to access the class variable of immediate parent class.**

Java Program[B.java]

```
class A
{
void fun()
{
System.out.println("Method: Class A");
}
}
```

```
class B extends A
{
void fun()
{
System.out.println("Method: Class B");
}
void display()
{
super.fun();
}
public static void main(String args[])
{
B obj=new B();
obj.display();
}
}
```

Output

```
Method: Class A
```

Program Explanation : In above program, the derived class can access the immediate parent's class method using super.fun(). Hence is the output. You can change super.fun() to fun(). Then note that in this case, the output will be invocation of subclass method fun.

3. The super() is used to invoke the immediate parent class constructor.

Java Program[B.java]

```
class A
{
A()
{
System.out.println("Constructor of Class A");
}
}
class B extends A
{
B()
{
super();
System.out.println("Constructor of Class B");
}
public static void main(String args[])
{
B obj=new B();
}
}
```

Output

```
Constructor of Class A
Constructor of Class B
```

Program Explanation : In above program, the constructor in class B makes a call to the constructor of immediate parent class by using the keyword super, hence the print statement in parent class constructor is executed and then the print statement for class B constructor is executed.

Abstract Class

• An abstract class is a class that contains one or more abstract methods.

• An abstract method is a method without method body.

Syntax:

abstract return_type method_name(parameter_list);

• An abstract class can contain instance variables, constructors, concrete methods in addition to abstract methods.

• All the abstract methods of abstract class should be implemented in its sub classes.

• If any abstract method is not implemented in its subclasses, then that sub class must be declared as abstract.

• We cannot create an object to abstract class, but we can create reference of abstract class.

• Also, you cannot declare abstract constructors or abstract static methods.

Java program to illustrate abstract class.

```
abstract class MyClass
{
 abstract void calculate(double x);
}
class Sub1 extends MyClass
{
 void calculate(double x)
{
 System.out.println("Square :"+(x*x));
 }
}
```

```
class Sub2 extends MyClass
{
 void calculate(double x)
 {
 System.out.println("Square Root :"+Math.sqrt(x));
 }
}
class Sub3 extends MyClass
{
 void calculate(double x)
 {
 System.out.println("Cube :"+(x*x*x));
 }
}
class AC
{
 public static void main(String arg[])
 {
 Sub1 obj1=new Sub1();
 Sub2 obj2=new Sub2();
 Sub3 obj3=new Sub3();
 obj1.calculate(20);
 obj2.calculate(20);
 obj3.calculate(20);
 } }
```

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
allme(); // 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
allme(); // 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 with out 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 with the new operator.
- Any sub class of abstract class must be either implements all the abstract methods in the super class or declared it self as abstract.
- Abstract modifier referred as ―subclass responsibilities‖ . because of no implementation 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();
a
llme()
;
b.call
metoo
();
}
}
// 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());
}
}
```

Uses of Final:
Final can be used in three ways:
• To prevent modifications to the instance variable
• To Prevent method overriding
• To prevent inheritance

Using final to Prevent Overriding:
While method overriding is one of Java's most powerful features, there will be times when you 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.

Object Class
In Java there is a special class named Object. If no inheritance is specified for the classes then all those classes are subclass of the Object class. In other words, Object is a superclass of all

other classes by default. Hence
public class A { ...} is equal to public class A extends Object {...}
There are two commonly used methods in Object class. Those are **toString()** and **equals().**

Defining Package:

- ➔ Generally, any java source file contains any (or all) of the following internal parts:
  - o A single package statement (optional)
  - o Any number of import statements (optional)
  - o A single public class declaration (required)
  - o Any number of classes private to the package (optional)
  - o Packages and Interfaces are two of the basic components of a java program.
  - o Packages are collection of related classes.
  - o Packages are containers for classes that are used to keep the class name compartmentalized.
  - o Packages are stored in an hierarchical manner and are explicitly imported into newclass defination.

Java packages are classified into two types:
Java API package or pre-defined packages or built – in – packages .
User – defined packages
Java 2 API contains 60 java.* packages.

For Example:

- ➢ java.lang
- ➢ Java.io
- ➢ Java.awt
- ➢ Java.util
- ➢ Java.net
- ➢ Javax.swing

To create a package

Just give package <<packagename>> as a first statement in java program.
Any classes declared within that file will belong to thespecified package.
If we omit package statement, the classes are stored in the default package.

Syntex:

Package packagename

Syntax:

Package packagename.subpackage

Access protection:

➔ Classes and packages both means of encapsulating and containing the name space and scope of variables and methods.

- ➔ Packages acts as a containers for classes and other sub – ordinate packages.
- ➔ Classes act as containers for data and code.
- ➔ Java address four categories of visibility for class members:
  - o Sub – classes in the same package.
  - o Non – sub class in the same package.
  - o Sub – classes in the different package.
  - o Classes that are neither in the same package nor subclasses.

The 3 access specifiers private, public and protected provide a variety of ways to produce the many levels of access required by these categories.

Access specifier	Private	No modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package sub class	No	Yes	Yes	Yes
Same package non – sub class	No	Yes	Yes	Yes
Different package sub class	No	No	Yes	Yes
Different package non sub class	No	No	No	Yes

From the above table,

- ➔ Any thing declared public can be accessed from any where
- ➔ Any thing accessed private cannot be accessed from outside of its class
- ➔ In the default, it is visible to sub-class as well as to other classes in the same package
- ➔ Any thing declared as protected, this is allow an element to be seen outside your current package, but also allow to sub class in other packages access.

UNDERSTANDING CLASSPATH: As just explained, packages are mirrored by directories. This raises an important question: How does the Java run-time system know where to look for packages that you create? The answer has two parts. First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in the current directory, or a subdirectory of the current directory, it will be found. Second, you can specify a directory path or paths by setting the **CLASSPATH** environmental variable. For example, consider the following package specification.

package MyPack;

In order for a program to find **MyPack**, one of two things must be true. Either the program is executed from a directory immediately above **MyPack**, or **CLASSPATH** must be set to include the path to **MyPack**. The first alternative is the easiest (and doesn't require a change to **CLASSPATH**), but the second alternative lets your program find **MyPack** no matter what directory the program is in. Ultimately, the choice is yours.

Defining an Interface

The interface can be defined using following syntax

access_modifier **interface** name_of_interface

{

return_type method_name1(parameter1,parameter2,...parametern);

...

```
return_type method_name1(parameter1,parameter2,...parametern);

type static final variable_name=value;

...

}
```

- ➔ The access_modifier specifies the whether the interface is public or not. If the access specifier is not specified for an interface then that interface will be accessible to all the classes present in that package only. But if the interface is declared as public then it will be accessible to any of the class.
- ➔ The methods declared within the interface have no body. It is expected that these methods must be defined within the class definition.

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:**

```
class classname [extends superclass] [implements interface1 [,interface2...]]

{

// class-body

}
```

• If a class implements more than one interface, the interfaces are separated with a comma.

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

Example-1: Write a java program to implement interface.

```
interface Moveable
{
int AVG_SPEED=30;
void Move();
}
class Move implements Moveable
{
void Move(){
```

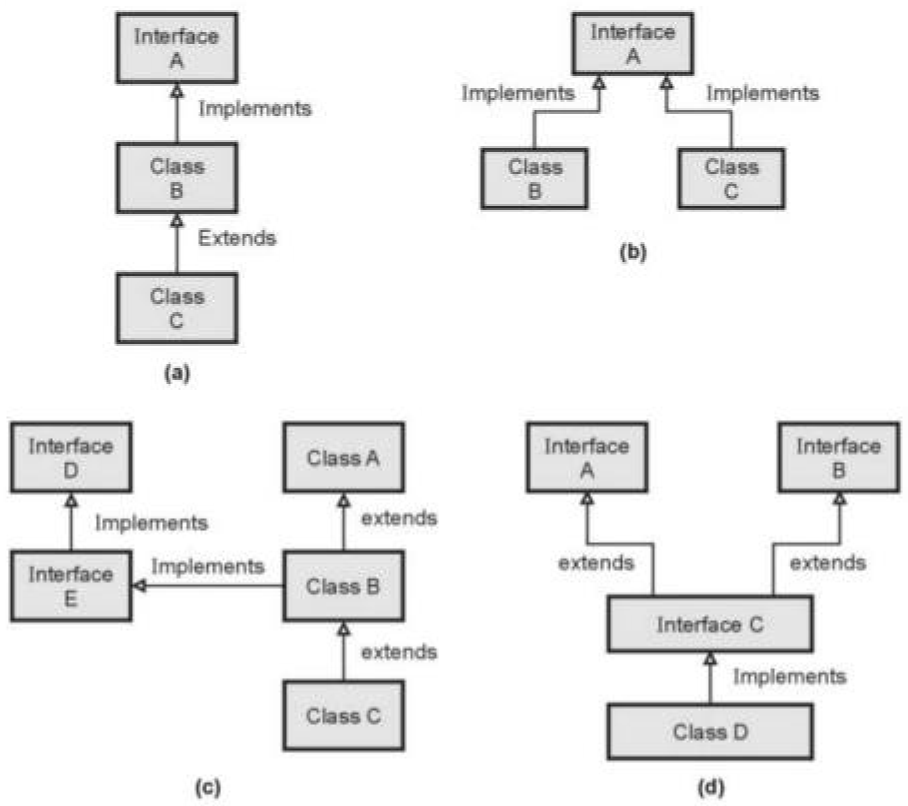
```
System .out. println ("Average speed is: "+AVG_SPEED );
}
}
class Vehicle
{
public static void main (String[] arg)
{
Move m = new Move();
m.Move();
}
}
```

Example 2: Write a java program to implement interface.

```
interface Teacher
{
void display1();
}
interface Student
{
void display2();
}
class College implements Teacher, Student
{
public void display1()
{
System.out.println("Hi I am Teacher");
}
public void display2()
{
System.out.println("Hi I am Student");
}
}
class CollegeData
{
public static void main(String arh[])
{
College c=new College();
c.display1();
c.display2();
}
}
```

Various ways of interface implementation

The design various ways by which the interface can be implemented by the class is represented by following Fig



The

variables can be assigned with some values within the interface. They are implicitly final and static. Even if you do not specify the variables as final and static they are assumed to be final and static.

Interface variables are static because Java interfaces cannot be instantiated. Hence the value of the variable must be assigned in a static context in which no instance exists. The final modifier indicates that the value assigned to the interface variable is a true constant that cannot be re-assigned by program code.

Accessing implementations through interface references:

We can declare variables as object references that use an interface rather than a class type. Any instance of any class that implements the declared interface can be referred to by such a variable. When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to. This is one of the key features of interfaces. The method to be executed is looked up dynamically at run time, allowing classes to be created later than the code which calls methods on them.

Ex:

```
interface Test {
void call();
}
class InterfaceTest implements Test {
```

```
public void call()
{
System.out.println("call method called");
}
}
public class IntefaceReferences {
public static void main(String[] args)
{
Test f ;
InterfaceTest it= new InterfaceTest();
f=it;
f.call();
}
}
```

Applying Interfaces

The interface is a powerful tool. The same interface can be used by different classes for some method. Then this method can be implemented by each class in its own way. Thus same interface can provide variety of implementation. The selection of different implementations is done at the run time. Following is a simple Java program which illustrates this idea.

Step 1 : Write a simple interface as follows

Java Program [interface1.java]

```
interface interface1
{
void MyMsg(String s);
}
```

Step 2 : Write a simple Java Program having two classes having their own implementation of MyMsg method.

Java Program[Test.java]

```
import java.io.*;
import java.util.*;
class Class1 implements interface1
{
private String s;
public void MyMsg(String s)
{
System.out.println("Hello "+s);
}
}
class Class2 implements interface1
{
private String s;
public void MyMsg(String s)
{
System.out.println("Hello "+s);
}
}
```

```
class Test
{
public static void main(String[] args)
{
interface1 inter;
Class1 obj1=new Class1();
Class2 obj2=new Class2();
inter=obj1;
inter.MyMsg("User1");
inter=obj2;
inter.MyMsg("User2");
}
}
```

Program Explanation :

- The selection of method MyMsg is done at the runtime and it depends upon the object which is assigned to the reference of the interface.
- We have created reference inter for the interface in which the method MyMsg is declared.
- Then the objects obj1 and obj2 are created for the classes class1 and class2 respectively. When the obj1 is assigned to the reference inter then the message “Hello User1” will be displayed because the string passed to the method MyMsg is “User1”. Similarly, when the obj2 is assigned to the reference inter then the message “Hello User2” will be displayed because the string passed to the method MyMsg is “User2”.
- Thus using the same interface different implementations can be selected.

Nested Interface

Nested interface is an interface which is declared within another interface.

For example -

```
interface MyInterface1
{
interface MyInterface2 {
void show();
}
}

class NestedInterfaceDemo implements MyInterface1.MyInterface2 {
public void show() {
System.out.println("Inside Nested interface method");
}
public static void main(String args[]) {
MyInterface1.MyInterface2 obj= new NestedInterfaceDemo();
obj.show();
}
}
```

Output

```
Inside Nested interface method
```

Variables in Interface

- The variables can be assigned with some values within the interface. They are implicitly final and static. Even if you do not specify the variables as final and static they are assumed to be final and static.

The members of interface are static and final because –

1) The reason for being static - The members of interface belong to interface only and not object.
2) The reason for being final - Any implementation can change value of fields if they are not defined as final. Then these members would become part of the implementation. An interface is pure specification without any implementation.

Example: Java program to demonstrate variables in interface.

```
interface left
{  int i=10;
}
interface right
{  int i=100;
}
class Test implements left,right
{  Public static void main(String args[])
{
System.out.println(left.i);//10 will be printed
System.out.println(right.i);//100 will be printed*/
} }
```

Extending Interface

Interfaces can be extended similar to the classes. That means we can derive subclasses from the main class using the keyword extend , similarly we can derive the subinterfaces from main interfaces by using the keyword extends.

The syntax is

```
interface Interface_name2 extends interface_name1
{
...
...
...
}
```

For example

```
interface A
{
int val=10;
}
interface B extends A
{
void print_val();
}
```

That means in interface B the display method can access the value of variable val.
Similarly more than one interfaces can be extended.

```
interface A
{
int val=10;
}
interface B
{
void print_val();
}
interface C extends A,B
{
...

...
}
```

Even-though methods are declared inside the interfaces and sub-interfaces, these methods are not allowed to be defined in them. Note that methods are defined only in the classes and not in the interfaces.

****