Unit - 2
~~Module 2~~ 1
~~Module 3~~

Inheritance

The mechanism of deriving a new class from old class is called inheritance.

old class is called super class. New class is called sub class.

To inherit a class, you simply incorporate the definition of one class into another class by using the 'extends' keyword.

The different types of inheritance

1) single-level inheritance
2) multilevel inheritance
3) Hierarchical inheritance
4) multilevel inheritance.

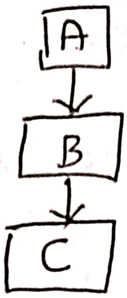
1) single level inheritance

$$\begin{array}{cl} \boxed{A} & \text{— super class} \\ \downarrow & \\ \boxed{B} & \text{— sub class} \end{array}$$

Deriving a new class from single base class is called single inheritance.

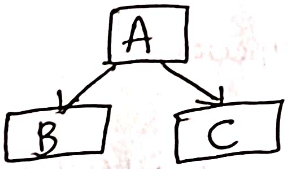
Example

2) <u>multilevel</u> <u>inheritance</u> :-



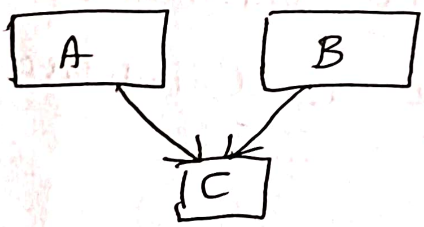
Deriving a class from another derived class is called multi-level inheritance.

3) <u>Hierarchical</u> <u>inheritance</u> :-



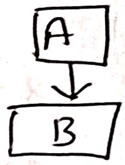
Deriving many classes from the single base class is called hierarchical inheritance.

4) <u>multiple</u> <u>inheritance</u>



Deriving a single ~~base~~ derived class from many base classes is called multiple inheritance.

Example for single inheritance



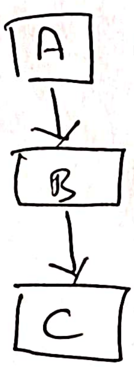

```
import java.util.*;
class A
{
    int m;
    int n;
    void getdata()
    {
        Scanner s = new scanner(system.in);
        System.out.println("enter m, n");
        m = s.nextInt();
        n= s.nextInt();
    }
    void Put data()
    {
        system.out.println(a);
        System.out.println(b);
    }
}

class B extends A
{
    int s;
```

```
void sum()
{
    s = m + n;
    system.out.println(s);
}
}

class single
{
    public static void main(String args[])
    {
        B x = new B();
        x.getdata();
        x.putdata();
        x.sum();
    }
}
```

multilevel inheritance



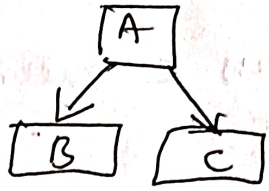
3

```
import java.util.*;
class A
{
    int m;
    int n;
    void getdata()
    {
        Scanner s = new Scanner(sesstem.in);
        system.out.Psientln("enter m, n");
        m= s.nextInt();
        n= s.nextInt();
    }
    void putdata()
    {
        system.out.Psientln(m);
        system.out.Psientln(n);
    }
}
class B extends A
{
    int s;
    void sum()
    {
        s=m+n;
        sesstem.out.Psientln(s);
    }
}
```

```
class c extends B
{
    int p;
    void prod()
    {
        p = m * n;
        system.out.println(p);
    }
}

class multilevel
{
    public static void main(String args[])
    {
        c x = new c();
        x.getdata();
        x.putdata();
        x.sum();
        x.prod();
    }
}
```

Hierarchical inheritance




```
import java.util.*;
class A
{
    int m;
    int n;
    void getdata()
    {
        Scanner s = new Scanner(system.in);
        system.out.println("enter m,n");
        m = s.nextInt();
        n = s.nextInt();
    }
    void putdata()
    {
        system.out.println(a);
        system.out.println(b);
    }
}
class B extends A
{
    int s;
    void sum()
    {
        s = m+n;
        system.out.println(s);
    }
}
```

```
class C extends A
{
    int p;
    void prod()
    {
        p = m * n
        System.out.println(p);
    }
}

class hierarchical
{
    public static void main(String args[])
    {
        B x = new B();
        x.getdata();
        x.putdata();
        x.sum();
        C y = new C();
        y.getdata();
        y.putdata();
        y.prod();
    }
}
```

Constructors in sub classes

If you written a constructor in both super class and sub class then the super class constructor will be executed first and then the sub class, when the object is created.

ex:-

```
class A
{
   A()
   {
     system.out.println("super class constructor
                                   executed");
   }
}
class B extends A
{
   B()
   {
    system.out.println(" sub class constructor
                                   executed");
} }

class subcons
{
   public static void main string(args[])
   {
} }   B x = new B();
```

Constructors in multilevel inheritance

```
class A
{
    A()
    {
        system.out.println("super class constructor
                                    called");
    }
}

class B extends A
{
    B()
    {
        system.out.println("sub class constructor
                                    called");
    }
}

class C extends B
{
    C()
    {
        system.out.println(" last subclass executed");
    }
}

class mulcons
{
    public static void main(String args[])
    {
        C x = new C();
    }
}
```

super

whenever a sub class needs to refer to the immediate super class by the use of keyword "super".

The super keyword has 2 uses

1. it is used in sub class constructor to invoke the immediate super class. The 'super' keyword must always be the first statement executed inside a sub-class constructor.

The general form of super keyword for this purpose is

$\boxed{\text{super(parameter list);}}$

where, parameter list specifies any parameters needed by the super class constructor.

ex:-

```
class A
{
    int m;
    int n;

    A( int P, int Q)
    {
        m = p;
    }   n = q;
```

②

```
class B extends A
{
    int s;

    B( int p, int q, int r)
    {
        super(p, q);        // super always the
                            first statement in
        s = r;              subclass
    }

    void putdata()
    {
        system.out.println(m);
        system.out.println(n);
        system.out.println(s);
    }
}

class supercons
{
  public static void main(String args[])
  {
      B  x = new B();

      x.putdata();
  }
}
```

(7)

(2) The second use of the super is to access a member of the super class that has been hidden by a member of a sub-class.

If the name of the instance variable in super class and sub class are same then, to access the super class data using super keyword and access the sub class data normally.

```
class A
{
    int m;
    A()
    {
        m=10;
    }
}

class B extends A
{
    int n;
    B()
    {
        n=20;
    }
}
```

```
void show()
  {
      System.out.println(" super class m =" +
                                    super.m);
      System.out.println(" sub class m =" + m);
  }
}

class supermember
{
   Public static void main(String args[])
   {
       B x = new B();
       x.show();
   }
}
```

Method overriding (Runtime Polymorphism)

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 override the method in the super class. When an overriden method is called from with in a sub class; then it will always refer to the Version of that method defined by the sub class.

```
class A
{
    void show()
    {
        system.out.println("welcome to C++
                                    world");
    }
}

class B extends A
{
    void show()
    {
        system.out.println("welcome to java
                                    world");
    }
}

class moverride
{
    public static void main(String args[])
    {
        B x= new B();
        x.show();
    }
}
```

output :- welcome to java world.

Dynamic method dispatch (Runtime Polymorphism Dynamic Polymorphism)

1. It is the mechanism by which a call to an overriden method is resolved at runtime, rather than the compile time.

It is used to implement runtime Polymorphism.

A super class reference variable can refer to a sub-class object. Java uses this fact to resolve calls to overridden method at runtime.

When an overridden method is called through a super class reference, Java (JVM) determines which version of that method to execute based on the type of object being referred to at the time call occurs, this determines made at runtime.

ex:

```
class A
{
    void show()
    {
        system.out.println(" ~~Hello~~ welome to c++
                           world");
    }
}
```

```
class B extends A
{
    Void show()
    {
        system.out.println(" welcome to java
                                    world");
    }
}

class superref
{
    Public static Void main (String args[])
    {
        A x;
        A y= new A();
        x=y;
        x. show() ;   // here super class show()
                        is executed
        B z= new B()
        x=z;
        x. show(); // here subclass show() method
                        is executed
    }
}
```

Abstract class

Some times we want to create a super class that only defines a generalized form that will be shared by all of its sub classes. living it to each sub-class to fill in the details. If such situation arises, abstract classes are used.

Ex:

- Abstract class is an incomplete class contain one or more abstract methods.
- An Abstract method is a method doesnot contain the implementation.
- A class that contain one or more abstract methods and that class must be declared as abstract. we can not create or instanciate an object for an abstract class.
- Abstract classes can be inherited.
- The method in the abstract class must be implemented int its sub-class otherwise, the ~~sub~~ subclass will also become an abstract class.

Ex:-

```
abstract class A
{
    abstract void show();

        void disP()
        {
            system.out.println("welcome to
                                  Java world");
        }
}

class B extends A
{
    void show()
    {
        system.out.println(" welcome to C++
                              world");
    }
}

class abstractdemo
{
    public static void main(String args[])
    {
        // A x = new A();   you can not
                            create a object
        B x = new B();
        x.disP();
        x.show();
} }
```

Final keyword

There are 3 uses of the final keyword

1. it is used to declare a variable. when a variable is declare as a final its content cannot be modify.

2. it is used to prevent method overriding.

3. it is used to prevent inheritance.

final variable

```
class A
{
    int a = 10;
    final int b = 20;
    void change ()
    {
        a = a+10
    //  b = b+10      final variable can not be
                             modified.
      system.out.println(a);
      system.out.println(b);
    }
}
```

```
class finalvariable
{
    public static void main (String args[])
    {
        A x = new A();
        x.change();
    }
}
```

Program to demonstrate the final keyword is used to prevent method overriding

```
class A
{
    void disp()
    {
        System.out.println(" welcome to c world");
    }
    final void show()
    {
        System.out.println("welcome to java world");
    }
}
class B extends A
{
    void disp()
    {
    }   System.out.println(" welcome to ~~java~~ C++ world");
```

```
/*      void show()
        {
            system.out.println(" welcome to C++ world");
        }
```

it cannot be overriden because it is used with keyword final. */

```
class finalmethod
 {
    Public static void main(String args[])
    {
        B  x = new  B();
           x. show();
           x. disp();
    }
 }
```

<u>final</u> <u>class</u> :- is used to prevent inheritance.

ex:-

```
class   A
 {
    void  disp()
    {
        system.out.println(" welcome to
                              C++ world");
    }

 }
```

```
final class B extends A
{
    void show()
    {
        system.out.println("welcome to java
                               world");
    }
}

/* class C extends B
{
    void print()
    {
        system.out.println("welcome to C#");
    }
}   This class will not execute because the
    final keyword is used to prevent
    further inheritance. */

class finalclass
{
    public static void main(string args[])
    {
        B x = new B();
        x.disp();
        x.show();
    }
}
```

Differences between Abstract class and final class

Abstract class	final class
1. This class is partially implemented	1. This class is completely implemented
2. it contains abstract methods	2. it does not contain abstract methods.
3. we cannot create a object	3. object can be created
4. it can be inherited	4. it cannot be inherited.
5. we can have abstract class, methods but there is no abstract variables.	5. we can have final variable, final class and final methods.

Interfaces

Interfaces are syntactically similar to classes but they lack of instance variables and their methods are declared without any body.

Once it is defined any number of classes can implement.

To implement an interface, a class must create complete set of methods defined by the interface. However, each class is free to determine the details of its own implementation by providing the interface keyword, java allows you to fully utilize the one interface, multiple method aspects of polymorphism. The general form of defining a interface is

```
access interface name
{
    type final variable 1 = value
    type final variable 2 = value
    =
    return type methodname 1 ( Parameter list);
    return type method name2 ( Parameter list);
    =
}
```

Here, access is either public or not used. when no access specifier is ~~used~~ included then the default access specifier results, and these interface is only available to other members of the Package in which it is declared. when it is declared as public, the interface can be used by any other code.

Name is the name of the interface. The methods which are declared inside the interface have no bodies. They end with semicolon after the parameter list. The methods which are declared inside a interface by default abstract methods.

Variables can be declared inside a interface. they are implicitly final and static, meaning cannot be changed by the implementing class, they must also be initialized with a constant value.

All methods and variables are implicitly public if an interface is declared as public.

ex:-

```
interface A
{
    void show();
    void disp();
}
```

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 define by the interface.

The general form of implementing class is as follows :-

```
access   class classname [extends superclass name]
             implements interface1, [interface2]...
         {
             class body
         }
```

Access is either public or not used. If a class implements more than one interface, the interfaces are separated with a (,) comma.

The methods that implement an interface must be declared as public.

Also, the type signature of the implementing class, method must match exactly the type signature specified in the interface definition.

Ex:-

```
interface A
{
    void show();
    void disp();
}
class B implements A
{
    public void show()
    {
        System.out.println("welcome to
                              C++ world");
    }
    public void disp()
    {
        System.out.println("welcome to
                              Java world");
    }
}
class interface demo
{
  public static void main(String args[])
  {
      B x = new B();
      x.show();  x.disp();  }  }
```

one interface can be implemented by any number of classes

```
interface A
 {
     void show();
 }

class B implements A
 {
 public void show()
     {
         system.out.println("welcome to C++");
     }
 }

class C implements A
 {
 public void show()
     {
         System.out.println(" welcome to java world");
     }
 }

   class interfacedemo
    {
     Public static void main (string args[])
     {
```

```
{
    B x = new B();
    x. show();
    C y = new C();
    y. show();
  }
}
```

Partial implementation of an interface

If a class implements an interface but does not fully implement the methods defined by that interface, then the class must be declared as abstract.

ex:-

```
interface A
{
    void show();
    void disp();
}

abstract class B implements A
{
public void show()
  {
    system.out.println ("welcome to c++ world");
  }
}
```

```
class C extends B
{
    public void disp()
    {
        System.out.println("welcome to java
                                        world");
    }
}

class Parinterface
{
    public static void main(String args[])
    {
        C x = new C();
        y.show();
        y.disp();
    }
}
```

How to achieve multiple interface in java

multiple inheritance is achieved in java by using the concept of interfaces.

ex:-

```
interface A
{
    void show();
}
```

```
interface B
 {
    void disp();
 }

class C implements A, B
 {
    public void show()
     {
        system.out.println("welcome to c++");
     }
    public void disp()
     {
        system.out.println("welcome to
                            java");
     }
 }

class multiple
 {
    public static void main(String args[])
     {
        C x = new C();
        x.show();
        x.disp();
     }
 }
```

Another method to achive multiple interface (18)

(one class and interface)

```
~~class~~ interface A
{
    void show();
}

class B
{
    void disp()
    {
        system.out.println("welcome to
                                    javaworld");
    }
}

class C extends B implements A
{
    public void show()
    {
        system.out.println(" welcome to C++
                                    world");
    }
}

class multiple
{
    public static void main(string args[])
    {
        C x = new C();
        x.show(); x.disp(); } }
```

interfaces can be extended

one interface can inherits another by use of the keyword extends. when a class implements an interface that inherits another, it must provides implementation for all methods defined with in the interface inheritance chain.

```
interface A
{
    void show();
}

interface B extends A
{
    void disp();
}

class C implents B
{
    public void show()
    {
        system.out.println("welcome to C++ world");
    }

    public void disp()
    {
        system.out.println("welcome to java world");
    }
}
```

```
class interfacedemo
{
    psvm(String args[])
    {
        C x = new C();
        x.show();
        x.disp();
    }
}
```

interfaces can be extended

One interface can inherits another by use of the keyword extends. When a class implements an interface that inherits another, it must provides implementation for all methods defined with in the interface inheritance chain.

```
interface A
{
    void show();
}

interface B extends A
{
    void disp();
}

class C implents B
{
    public void show()
    {
        system.out.println("welcome to C++ world");
    }

    public void disp()
    {
        system.out.println("welcome to java world");
    }
}
```

```
class interfacedemo
{
    psvm(string args[])
    {
        C x = new C();
        x.show();
        x.disp();
    }
}
```