Module : 1

Prepared by Nitesh Karsi

1.1 Programming Approach from Procedural to Object Oriented –

Procedure Oriented Programming / Function Oriented Programming / Structured Programming –

- In procedure oriented programming, a program is a list of instructions. As the length of the program increases, its complexity increases. In structured programming, it is difficult to maintain large programs.
- In structured programming, we can overcome this problem by dividing a large program into different functions or modules, but it results into other problems such as
  a) The functions have unrestricted access to global data
  b) The functions provide poor analogy with real world.
- Typical structure of procedure oriented program can be as follows.

Object Oriented Programming –

The term object oriented means we organize the software as a collection of discrete objects that incorporate both **data structure** and **behaviour.** Object Oriented programs will use collection of classes and their objects. This strategy suits even if the project becomes huge in size. This strategy matches more with real world.

Comparison of Procedure Oriented Programming and Object Oriented Programming –

	Procedure Oriented		Object Oriented
1.	Data and the functions operating on them are kept separated	1.	Data and method are kept in the encapsulated manner.
2.	Handling the large programs i.e. the complexity becomes difficult.	2.	Handling large programs i.e. the complexity of the system is comparatively easy.
3.	Not every suitable for defining abstract data types.	3.	Highly suitable for defining abstract data types.
4.	Debugging is difficult.	4.	Debussing is easy.
5.	Difficult to implement changes and hence less flexible.	5.	Suitable for changes and hence more flexible.
6.	Programming is data based and procedure based.	6.	Programming is object oriented.
7.	Less suitable for reusability.	7.	More suitable for reusability
8.	Uses top down programming approach	8.	Uses bottom up programming approach
9.	Functions can call each other	9.	Object communicate with each other
10.	Not suitable for representing real world objects. This is because the data and the associated functions are loosely coupled.	10.	Suitable for representing the real world objects. This is because, the data and the associated functions are kept in the encapsulated manner.
11.	No data hiding. Hence it can result into the accidental access of the data.	11.	Data hiding is implemented using encapsulation. Hence, accidental access of the data is not possible.
12.	Does not support the concurrency i.e. execution of multiple tasks at a time.	12.	Supports the concurrency using multithreading.

Limitations of Procedure Oriented Programming –

1. Handling the large programs i.e. the complexity becomes difficult.
2. It is not suitable for implementing abstraction.
3. Difficult to implement changes and hence less flexible.
4. Less suitable for reusability.
5. Not very suitable for data hiding. It results into accidental access of data
6. It is not suitable to represent real world objects.
7. It does not support concurrency.
8. With the increase in complexity, debugging becomes easier.

Module : 2

Object Oriented Concepts

Prepared by Nitesh Karsi

2.1 Objects –

- An object is an entity that has **state**, **behaviour** and **identity** that has meaning for an application.
- Some objects are the part of real world and therefore exists in time and space. for example, a mango is an object, a white rook in check is an object, a vending machine that dispenses a soft drink is an object, a hammer is an object, a marker pen is an object, a fan is an object, an elevator is an object, a person is an object and so on.
- Some objects are conceptual entities. For e.g. a formula for solving a quadratic equation is an object, a chemical process in a particular manufacturing plant is an object, saving account of a person is an object.
- The structure and behaviour of similar objects are defined in their common class. The object is also called as an instance (i.e. occurrence) of a particular class.

Example:

An elevator:

State: The **state** of an object covers all of the properties of the object plus the current values of each of these operations

a) Different Properties of an Elevator and their current values –

Property : An elevator can move up or down.
Property : An elevator can stop at a particular floor.
Property : An elevator can carry a particular weight with some predefined maximum capacity

b) Different States of an elevator –

- Elevator halted on a particular floor with the door open either because people are coming out of and / or going into an elevator.
- Elevator moving upward with or without people.
- Elevator moving downwards with or without people.
- Elevator halted in between the two floors with or without people (due to malfunctioning or power failure)
- Elevator out of order.

Behaviour: Behaviour is how an object acts or reacts, in terms of its state changes and message passing.

- For example, if an elevator is halted on a particular floor which is neither the topmost nor the lowermost and if the two requests are made simultaneously, one from some upper floor and the other from some lower floor then an elevator will move to the nearer floor. However if both the requests are done from equidistant floors it may move upwards assuming higher floor number has got higher priority or otherwise it will maintain the previous direction of an elevator before it has halted assuming it is designed in that way. Thus, the behaviour depends on the design.
- Above behaviour was handling a complicated case. Other simple case such as only one request made at a time and the behaviour of an elevator will be obviously, it will reach to the floor from which the request is made.
- Other simple behaviours do exist.

Identity: Identity is that property of an object which distinguishes it from all other objects.

- For example, a complex has got multiple identical towers with identical number of floors. Every tower has got one elevator. All these elevators are identical in terms of dimensions, behaviour and in general all properties. However, elevator-1, elevator-2, elevator-3 and so on will act as an identity of every individual elevator. Using the concept of identity, it is possible to refer a particular elevator. For example, elevator-3 is out of order from today morning.

2.2 Classes –

- An class is a collection / set of objects that share a common structure, common behaviour and common semantics. The concepts of a class and an object are tightly interwoven.
- An object is a concrete entity, that really exist, in time and space whereas a class represents only an abstraction (description of common structure and behaviour).
- The class is nothing but data type that represents set of objects having identical characteristics. For example, brochure of a particular car model is a class that describes properties of that car in terms of structure and behaviour. Every actual car of that model is an object.
- The class definition is often made up of **data members** (also called as instance variables) and **function members** (often called as **members functions** or called as **methods**).
- **Example 1 –**

```
class MusicSystem
{
        private long serialNumber;
        static private float length, breadth, height;
        static private float cost;
        private char door;
        static private int numberOfSpeakers;

        public void playCassette( );
        public void stop( );
        public void pause( );
        public void forward( );
        public void rewind( );
        public void record( );
};
MusicSystem m1=new MusicSystem();
MusicSystem m2=new MusicSystem();
```

- In the above example, class MusicSystem has got **data members** i.e. **instance variable** as length, breadth, height, cost, color and numberOfSpeakers.
- The class MusicSystem has got **member functions (methods)** as playCassette( ), forward( ), rewind( ), record( ).
- The **m1** and **m2** are **objects** of a **class** MusicSystem.

- **Example 2 –**

```
class Rectangle
{
        private float length, breadth;

        public void getSides( );
        public float clacPeri( );
        public float calcArea( );
};
Rectangle r1=new Rectangle( );
Rectangle r2=new Rectangle( );
```

- In the above example, class Rectangle has got **data members** i.e. instance variables as length and breadth.
- The class Rectangle has got **member functions (methods)** as getSides( ), calcPeri( ), calcArea( ).
- The **r1** and **r2** are **objects** of a **class** Rectangle.

- **Example 3 –**

```
class Employee
{
        private String name;
        private long id;
        private String department;
        public long salary;

        public void readData( );
        public void display( );
        public void calcSalary( );
};
Employee e1=new Employee( );
Employee e2=new Employee( );
```

- In the above example, class Employee has got **data members** i.e. **instance variables** as name, id, department and salary.
- The class Employee has got **member functions (method)** as readData( ), display( ), calcSalary( ).
- The **e1** and **e2** are **objects** of a **class** Employee.

2.3 Abstraction –

- An abstraction denotes the essential characteristics of an object that distinguish it form all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.
- An abstraction is a simplified description or specification of a system that emphasizes some of the system's details or properties while suppressing others.
- The purpose of an abstraction is to limit the universe so that we can understand it. All abstractions are always incomplete and inaccurate. The abstraction must be adequate for some purpose.
- A good abstraction is the one that emphasizes details that are significant to the reader or user and suppresses details that are at least for the moment immaterial.
- **Example – What is a car ?**

Abstraction from driver's perspective –

- Driver will look to the car as a system that can be controlled using accelerator, break, clutch, steering wheel and gear changing lever.
- Using accelerator, its speed can be increased or decreased. Using break it can be stopped. Using clutch and break together it can be stopped and kept at such a state that it can be started immediately when required. Using clutch and accelerator it can be started moving again. Using steering wheel it can be turned and using gear shifting lever it can be driven efficiently.

Abstraction form designer's perspective –

- Designer will look to a car as a complex system made up of multiple smaller systems such as engine, gear transmission technique, starter, ignition system, various sensors, battery, microprocessor based controller, charging system i.e. alternator and so many other systems. Designer's perspective will be always to make the performance of these systems better.

Abstraction form mechanic's perspective –

- The car mechanic will also look to all these systems but not to make the design of these systems better but to repair those systems or parts, that has gone faulty.

Abstraction from owner's perspective –

- Owner will look to the car as a style and status icon, a system that will increase the mobility of owner, will bring luxury to the owner, will bring the betterment in the business etc. and it will be at the cost of all the expenses which will be required to keep that car functioning.

Example 2 – What is a Rectangle?

The abstraction of a Rectangle from user's point of view interested in calculations.

Abstraction : Rectangle
Important characteristics: length and breadth
Responsibilities: read the values of length and breadth calculate and display perimeter calculate and display area

The abstraction of a Rectangle form user's point of view interested in drawing rectangle.

Abstraction : Rectangle
Important characteristics: left, top, right, bottom brush size draw colour fill colour
Responsibilities: draw rectangle with the selected colour and brush size fill the rectangle with the selected colour move the rectangle at a particular place within the limits

Above example, describes different abstractions of Rectangle from the perspective of two different users.

2.4 Encapsulation for Information Hiding or Data Hiding –

- Abstraction is the description of a system. This abstraction must be implemented. The implementation of a system can be done in different ways. Once the implementation is finalized, it should act as a secret of the abstraction and should be hidden from most of the clients.
- Abstraction and encapsulation are complementary concepts. Abstraction focuses on the observable behaviour of an object, whereas encapsulation focuses on the implementation that gives rise to the behaviour.
- Encapsulation is achieved using information hiding. Information hiding is not just data hiding. Information hiding means, hiding the data structure as well as the implementation of methods. The interfaces are made available to the clients and their implementation is kept hidden.
- Thus, while designing a class for given abstraction, the interfaces are made available to the client and those will satisfy the abstraction of the system. The implementation will be encapsulated i.e. hidden from the client.

Example 1 – A tape-recorder(audio cassette player)

Abstraction of a tape-recorder.

Abstraction : Tape-recorder (Auto cassette player)
Important characteristics: Cassette and its size AC and or DC operate with voltage details Dimensions Body colour
Responsibilities: play the cassette stop laying / eject the cassette forward rewind record pause playing the cassette

Encapsulation –

Data Structure –

A dc motor, transformer, audio amplifier circuits, motor driving circuits, transducer that converts magnetic signal stored on cassette to electrical signal, resistors, capacitors, Integrated circuits, printed circuit board etc. This is all **hidden** form the client. **Thus the data structure is hidden.**

Interface –

A button for playing a cassette

A common button for stop playing / eject a cassette

A button for forwarding a cassette

A button for rewinding a cassette

A button for recording a cassette

A button for pausing a cassette

All these buttons (interfaces) are made available for the clients.(**Public interfaces**).

Implementation –

Working of audio amplifier with its gain details. Specifications of dc motor with its speed variations for play, forward, rewind. How the motor rotates in the other direction for rewind. The design of the complete circuitry of the tape recorder. This is all kept **hidden.** Thus, the **implementation of the behaviour** of the tape recorder is kept **hidden.**

Example 1 – A tape-recorder(audio cassette player)

Abstraction of a tape-recorder.

Abstraction : Tape-recorder (Auto cassette player)
Important characteristics: Cassette and its size AC and or DC operate with voltage details Dimensions Body colour
Responsibilities: play the cassette stop laying / eject the cassette forward rewind record pause playing the cassette

Encapsulation –

Data Structure –

A dc motor, transformer, audio amplifier circuits, motor driving circuits, transducer that converts magnetic signal stored on cassette to electrical signal, resistors, capacitors, Integrated circuits, printed circuit board etc. This is all **hidden** form the client. **Thus the data structure is hidden.**

Interface –

A button for playing a cassette

A common button for stop playing / eject a cassette

A button for forwarding a cassette

A button for rewinding a cassette

A button for recording a cassette

A button for pausing a cassette

All these buttons (interfaces) are made available for the clients.(**Public interfaces**).

Implementation –

Working of audio amplifier with its gain details. Specifications of dc motor with its speed variations for play, forward, rewind. How the motor rotates in the other direction for rewind. The design of the complete circuitry of the tape recorder. This is all kept **hidden.** Thus, the **implementation of the behaviour** of the tape recorder is kept **hidden.**

Example 2 – A Rectangle

Abstraction of a Rectangle from user's point of view interested in calculations.

Abstraction : Rectangle
Important characteristics: length and breadth
Responsibilities: read the values of length and breadth calculate and display perimeter calculate and display area

Encapsulation –

Data Structure –

Whether the names of data members used are length and breadth or l and b or may be x and y is kept hidden by defining them as **private members.** The client will not be allowed to access these data elements directly.

Interface –

A button for reading the values of length and breadth.
A button for calculating and displaying the perimeter.
A button for calculating and displaying the area.
All these buttons (**interfaces**) are made available for the client. (**Public interfaces**)

Implementation –

The algorithm / programming statements to read the values of length and breadth of rectangle are not accessible to client.

The client can not see whether the formula of perimeter is written as length + length + breadth + breadth or 2*length + 2*breadth or 2*(length + breadth).

The client can not also see whether calculation of area is implemented as length*breadth or is it length + length + length + ... + length done breadth times.

All these **implementation** are kept **hidden** from the client. Only the buttons(interfaces) are made available to activate these formulas.

2.5 Inheritance –

- Hierarchy is a ranking or ordering of abstraction.
- The two most important hierarchies in a complex system are its class structure (the **"is a"** hierarchy) and its object structure (the **"part of"** hierarchy). **Inheritance indicates "is a" hierarchy.**
- The inheritance is the mechanism of deriving one class from one or more classes. Inheritance defines a relationship among classes where one class shares the data structure and behaviour defined in one or more classes. (resulting into single inheritance and multiple inheritance respectively.)
- A subclass (also called as derived class) redefines or elaborates the existing structure and behaviour of its superclass (also called as base class).
- **Inheritance implements generalization / specialization hierarchy.**
- **Different types of inheritance are as follows:**
  a) Single inheritance
  b) Multilevel inheritance
  c) Hierarchical Inheritance
  d) Multiple Inheritance

a) Single Inheritance –

b) **Multilevel inheritance –**

c) **Hierarchical Inheritance –**

d) **Multiple Inheritance –**

2.6 Polymorphism –

➢ The terms polymorphism is originated from the words,

poly → **many**

morphos → **forms**

- The **morphos** come from the word **morphus** which is a **Greek God** who appears in **different forms** for different sleeping individuals.

- Thus the term polymorphism denotes **many forms.** In other words, one interface can have multiple behaviours.

- The polymorphism can be implemented using,
  a) operator overloading
  b) function overloading without using inheritance
  c) method overriding within the inheritance

- Java does not support operator overloading. The function overloading without inheritance is of less use in the object oriented environment. The **method overriding** within the interface is the method i.e. preferred to implement polymorphism under the object oriented environment.

Example 1 –

Consider the following class hierarchy.

- While running the software using the above hierarchy, it will be dynamically decided (**dynamic binding**) that whether the data entry operator is going to deal with production engineer, sales engineer or worker. An interface, called as enterData( ) will have different behaviour for production engineer, sales engineer and worker. All these classes will have their own behaviour for enterData( ). Any one behaviour out of many will be selected depending on the data entry operator's current requirement.

- The enterData( ) for production engineer will gather the information of, name, id, basic salary, da and hra.
  The enterData( ) for sales engineer will gather the information of, name, id, basic salary, da and ta.
  The enterData( ) for worker will gather information of, name, id, basic salary, othours and otrate.

- Thus, the single interface enterData( ) result into multiple forms i.e. multiple effects resulting into **polymorphism.** Similar discussion can also be obtained for interface named as calculateSalary( ).

Example 2 –

Consider the following class hierarchy.

- Consider the operation makeAMove( ). Depending on the type of the chess piece being selected, the makeAMove( ) will have different behaviour.
- For a pawn, the behaviour of makeAMove( ) will be either one or two steps forward initially or while cutting the piece of an opponent it will be one step diagonally forward and by considering positions of the other piece on the board.
- For a rook, the behaviour of makeAMove( ) will be up, down, left or right by one or many steps and by considering positions of the other pieces on the board and within the limits of the board.
- For a queen, the behaviour of makeAMove( ) will be the mix behaviour of rook and bishop at a time.
- For a knight, the behaviour of makeAMove( ) will be 8 different possible options by considering positions of the other pieces on the board and within the limits of the board.
- For a king, the behaviour of makeAMove( ) will be one step in 8 different directions by considering positions of the other pieces on the board and within the limits of the board. A king will have a special behaviour along with a rook in case of castle-in.
- Thus, single interface makeAMove( ) will have multiple implementation resulting into **polymorphism.** The one that will be selected will be decided dynamically at run time (**dynamic binding**) depending on which piece is being selected.

2.7 Reuse –

- One of the major advantages of inheritance is **reusability without rewriting.**

Example 1 –

- Suppose in a particular company, the class users are writing the program for their own applications. They all need a class called as countup with three facilities named as constructor, increment and display. This class is being developed by class developer and it is being made available to the class users. Suppose the fourth class user needs a class counter with a new facility called as decrement along with all the existing facilities within the class countup, then the class developer can give solutions in three different ways.

1. Write a new class with all the four facilities named as constructor, increment, display and decrement. This method is disadvantageous as there will be wastage of time and space both.
2. Modify the existing the class by adding a new facility called as decrement. This method is also disadvantageous. This is because the original class users will get one extra facility which was not required to them and then they can make misuse of it or the accidental use of it.
3. Derive a new class called as class coountupdn. This results into inheritance. All the three facilities of class countup will be inherited to the class countupdn without rewriting them in class countupdn and only a new facility is to be developed within the class countupdn called as decrement. This method is advantageous over the above two methods and that is because no time and space wastage as well as no extra facility to the original users and hence no misuse.

- The class countup is a **base class (super class)** and it has got three features named as constructor, increment and display. The class countupdn is **derived class (sub class)** and it is derived from a base class (super class) countup. The features of the base class (super class) countup are inherited to the derived class (sub class) countupdn and the derived class (sub class) has got its own feature named as decrement.

- Inheritance allows us to **reuse** the features of the base class (super class) into the derived class (sub class) **without rewriting** them into the derived class (sub class). The inheritance is the mechanism of providing the **extended version** of the existing classes.

Example 2 –

- The properties named as id, name, basic salary are common to production engineer, sales engineer and worker. Hence, instead of defining them separately in all the three classes, it is better to define these properties only once in a super class named as employee and let it get inherited to all the sub classes resulting into **reusability without rewriting.**

- Similarly, the statements to read values of all these properties i.e. id, name and salary will be required in a class named as production engineer, sales engineer and worker. Instead of writing the same statements at three places, it is better to put them into enterData( ) defined within the super class employee and let it be used by all the sub classes. Thus, achieving **reusability without rewriting.** The sub classes will have their own behaviour to read their own special characteristics such as details of allowances and overtime etc. However, at least for common requirements, the behaviour of enterData() defined within super class employee be **reused.**

Module : 5

Prepared by Nitesh Karsi

5.1 Classes and Members –

- The most important fundamental about the class is that it defines a new data type. Once the class i.e. new type is defined then it can be used to create objects of that type.
- Thus, the class is a **template** for an object and an object is an **instance (i.e. occurrence)** of a class. The words **object** and **instance** are often used interchangeably.
- The class acts as logical framework i.e. logical construct whereas object acts as physical construct i.e. object occupy space in the memory.
- The elements within the class are called as **members.** The class elements are of two types,
  a) attributes (data structure)
  b) operations (behaviour) on the data.
- The general form of a class can be given as follows.

```
class classname
{
    type instance-variable1;
    type instance-variable2;
    .....
    type instance-variableN;

    type methodname1 (parameter list)
    {
        //body
    }
    type methodname2 (parameter list)
    {
        //body
    }
    .....
    type methodnameN (parameter list)
    {
        //body
    }
}
```

- The data variable defined within a **class** are called as **Instance variables.** This is because each object i.e. each instance of the class contains its own copy of variables and will occupy separate memory. Thus, instance variables are allocated separately per every object (instance).

- The code is contained within the function members. The function members are also called as **methods.** The methods must be declared and defined inside the class. The methods allocate the memory only once irrespective of number of objects. The operations are defined within methods. The operations are of different types as follows.

  a) Modifier – An operation that alters the state of an object.

  b) Selector – An operation that access the state of an object but does not alter the state

  c) Iterator – An operation that permits all parts of an object to be accessed in some well defined manner.

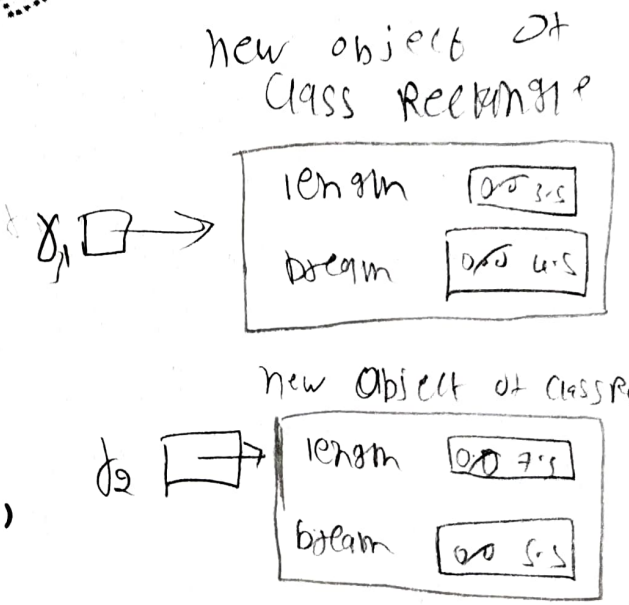
  d) Constructor – An operation that creates an object and/or initializes its state.

  e) Destructor – An operation that frees the state of an object and/or destroy the object itself.

5.2 Object, References and new –

Consider the following program.

```
class Rectangle
{
     private double length, breadth;     // instance variables
     public void setData(double l, double b)
     {
          length=l;
          breadth=b;
     }
     public double area()
     {
          return length * breadth;
     }
}
class RectangleDemo1
{
     public static void main(String args[])
     {
          Rectangle r1 = new Rectangle();
          Rectangle r2 = new Rectangle();
          r1.setData(3.5,4.5);
          System.out.println("Area of first rectangle="+ r1.area());
          r2.setData(7.5,5.5)
          System.out.println("Area of first rectangle="+ r2.area());
     }
}
```



Explanation –

- A **class** defines a new data type. The new data type in the above program is **Rectangle.** The declaration of a class rectangle creates **template** and it does not create objects.

- The **object** of class Rectangle is created as follows,

Rectangle r1 = new Rectangle( );

In the above line, the r1 is actually reference to an object (instance) of type **Rectangle.** The **new** is an operator that allocates the memory dynamically i.e. at run time for the object (instance) of type Rectangle and it also returns reference to this newly allocated object (instance) of type Rectangle. The returned reference is then assigned to a variable r1. Thus, we can conclude that r1 will act as reference to the newly allocated object of class Rectangle. The r1 acts as **reference variable** which is also called as **class variable.**

- Consider the following line,

Rectangle r1 = new Rectangle( );

The above line actually combines two steps which can be given as follows.

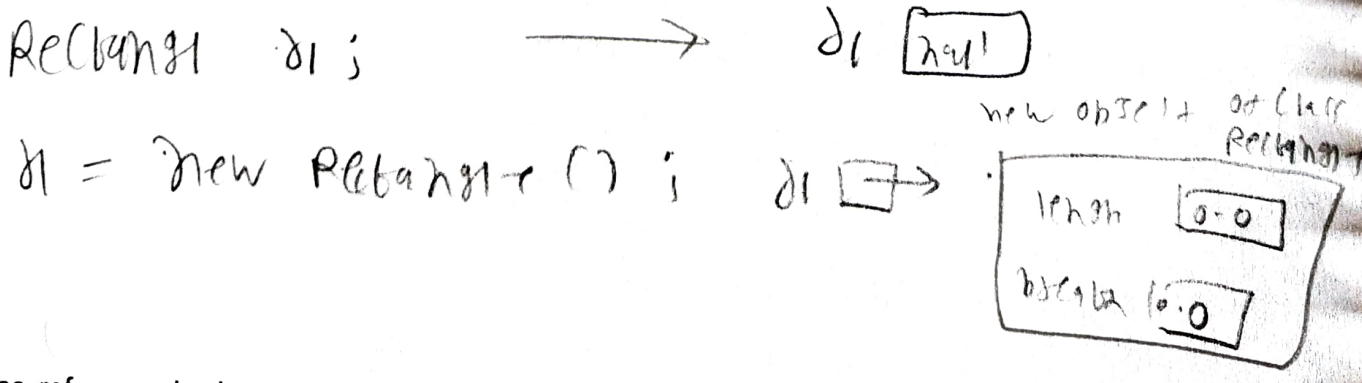
```
Rectangle r1;
r1 = new Rectangle( );
```

The first line declares r1 as a reference variable which has a capability to refer to an object of type Rectangle. When this line gets executed, then the value stored within r1 is **null** which indicates that it does not yet refer any object.

The second line actually allocates the object of type Rectangle using operator new and the reference to that object which is returned by operator new is assigned to reference variable r1.

After the second line, r1 which now acts as a reference to (address of) the newly allocated object can be used to access the allocated object.

The following diagram shows the effect of both the lines.



Note – The reference in Java is similar to a pointer (address) in C++. However, the main difference between the two is that references in Java can not be manipulated whereas the pointers in C++ can be manipulated.

- The new operator is used to allocate the memory dynamically i.e. at run time.
- The general form of using new operator is given as follows.

class-var = new classname( );

Example –

r1 = new Rectangle();

- The class-vr (i.e. r1) is a reference variable of the class type (i.e. Rectangle) which is being already created. The classname (i.e. Rectangle) is the name of the class whose object is created.
- The classname (i.e. Rectangle) followed by brackets specifies the **constructor** for the class. Normally, classes will have their well defined constructor method(s) but if we do not mention the constructor within a class as that of class Rectangle in the above program then in that case Java will automatically supply the **default constructor.** The default constructor will initialize both the instance variables length and breadth by zero.
- The setData( ) and area( ) are the methods.
- The method setData( ) accepts two parameters of type double. The operation within the method setData( ) is a **modifier**. It modifies the state (i.e. values of length and breadth) of an object.
- The operation within method area( ) is of type **selector.** It only access the state of an object (i.e. values of length and breadth) but does not alter the state.
- The method setData( ) is accessed for objects r1 and r2 using the following syntax.

object reference name . method name(arguments);

For example,

r1.setData(3.5,4.5);

- The method setData( ) is called for the object which is being referred by r1. The values 3.5 and 4.5 act as arguments. These values are copied into corresponding parameters named as l and b. The values of l and b are copied into the instance variables length and breadth respectively of an object which is referred by r1.
- Consider the call,

r1.area( )

The method area( ) is called for the object which is being referred by r1. The values of length and breadth of the object which is being referred by r1 are multiplied and the answer is returned.

- Similar explanation is applicable for the object which is being referred by r2.

5.3 Constructors –

- In an application if there are many objects of a class then initialization of every object becomes tedious even with specific method like setData( ) of the previous program.

- Hence in this kind of situation we will prefer to have a code that will carry out **automatic initialization** of objects. The **constructor method** does exactly the same.

- Following are the properties of the constructor methods.

1) The constructor method has same name as that of the class.

2) The constructor method will not have any return type not even void. This is because the constructor has implicit return type and that is the class type itself.

3) The constructor method is used for automatic initialization of instance variables of objects i.e. instances.

4) The constructor method can be declared without any parameter.

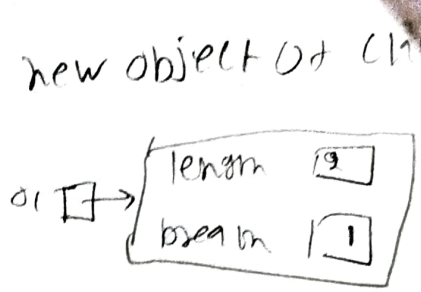
5) The constructor method can bed declared with one or many parameters.

6) It is possible to declare multiple constructors within a single class which are differing from each other in terms of their number of parameters and their types. It means that overloaded constructors can be declared and defined within a class.

7) Constructor can accept parameters of simple data types and it can also accept object of the class as a parameter.

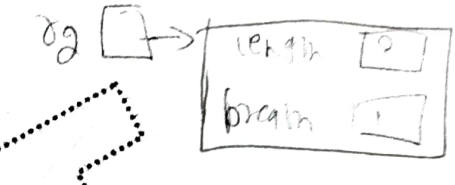
8) When the new object of a particular class is allocated using the new operator i.e. when the new object is created then the appropriate constructor is called automatically and will carry out initialization task of required instance variables.

5.4 Constructor without parameter –



Consider the following program.

```
class Rectangle
{
      double length, breadth;
      double area()
      {
            return length * breadth;
      }
      Rectangle()                    // constructor without parameters
      {
            length=2;
            breadth=1;
      }
}
class RectangleDemo2
{
      public static void main(String args[])
      {
            Rectangle r1 = new Rectangle();
            Rectangle r2 = new Rectangle();
            System.out.println("Area of first rectangle=" + r1.area());
            System.out.println("Area of second rectangle=" + r2.area());
      }
}
```



Explanation –

- Consider the statement;

  Rectangle r1 = new Rectangle( );

- In the above statement, **new** will allocate the object of class Rectangle and the constructor without parameters that we have written in the class Rectangle will be called automatically and it will initialize r1.length and r1.breadth with the values 2.0 and 1.0 respectively. Since we have declared a constructor, the default constructor provided by Java will not come in picture.

- Similarly, r2.length and r2.breadth are also initialized with 2.0 and 1.0 respectively. The initialization is carried out automatically with the help of same constructor.

- The advantage of the constructor in the above program is that initialization of instance variables takes place automatically. Moreover, the values of length and breadth initialized with 2.0 and 1.0 are better than 0.0 and 0.0 which would have happened with Java's default constructor.

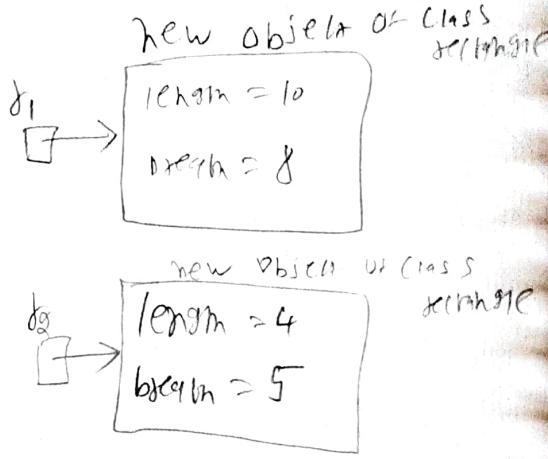
- The disadvantage of the above constructor is that for all the objects of class Rectangle, the length and breadth are compulsorily initialized with specific values i.e. 2.0 and 1.0.

- The above problem can be solved with the help of parameterized constructor.

5.5 Parameterized Constructors (Constructor with parameters) –

Consider the following program.

```
class Rectangle
{
      double length, breadth;
      double area()
      {
            return length * breadth;
      }
      Rectangle(double l, double b)
      {
            length = l;
            breadth = b;
      }
}
class RectangleDemo3
{
     public static void main(String args[])
     {
          Rectangle r1 = new Rectangle(10,8);
          Rectangle r2 = new Rectangle(4,5);

          System.out.println("Area of first rectangle=" + r1.area());
          System.out.println("Area of second rectangle=" + r2.area());
     }
}
```



Explanation –

- *The advantage of the parameterized constructor is that we can initialize length and breadth of the newly created object by any values that want and thus* ***limitation*** *of initializing length and breadth with specific values only* ***is removed.***

Consider the following statements,

Rectangle r1 = new Rectangle(10,8);

Rectangle r2 = new Rectangle(4,5);

When the first object is constructed, then the arguments passed are 10 and 8 which are copied into parameters **l** and **b**. The values of **l** and **b** are then assigned to r1.length and r1.breadth. Thus the first object which is referred by r1 is automatically initialized with the help of parameterized constructor.

When the second object is constructed, then the arguments passed are 4 and 5 which are copied nto parameters **l** and **b**. The values of **l** and **b** are then assigned to r2.length and r2.breadth. Thus he second object which is referred by r2 is automatically initialized with the help of parameterized onstructor.

- **Disadvantages of the above constructor** – If we want to initialize length and breadth of a rectangle by 2 and 1 respectively by default i.e. without specifying the arguments and hence we write the statement as follows.

  Rectangle r3 = new Rectangle();

  then it will cause error in the above program of class RectangleDemo3 where as it will be valid in the program of class RectangleDemo2. This is because the constructor without parameter is not present in the current program and Java will not provide its **own default constructor** in this situation.

  To initialize r3.length and r3.breadth with 2 and 1 respectively in the program of class RectangleDemo3, we will have to specify the arguments in the following form.

  Rectangle r3 = new Rectangle(2,1);

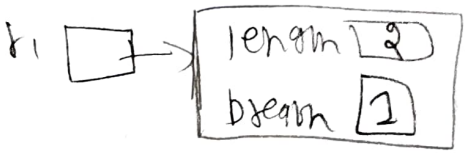
- If we want to have parameterized constructor but we also want object to be initialized with default values (say 2 and 1) if the arguments are not mentioned then we can obtain it with the help of **overloading of constructors.**

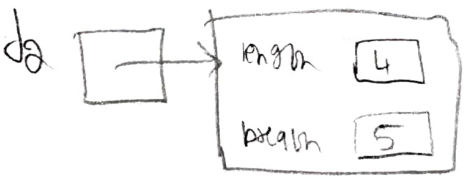
- **Note** – When we define a class Rectangle without any constructor method of our own then and then only, for any newly allocated object of the class Rectangle, Java will supply its own default constructor for that object and it will initialize the length and breadth of that newly allocated with the default values i.e. zeroes.

5.6 Overloaded Constructor / Constructor Overloading / Defining Multiple Constructor within the same class –

consider the following program.

```
class Rectangle
{
      double length, breadth;
      double area()
      {
          return length * breadth;
      }
      Rectangle()                          // constructor without parameter
      {
          length=2;
          breadth=1;
      }
      Rectangle(double l, double b)        // constructor with parameter
      {
           length = l;
           breadth = b;
      }
}
class RectangleDemo4
{
     public static void main(String args[])
     {
          Rectangle r1 = new Rectangle();
          Rectangle r2 = new Rectangle(4,5);

          System.out.println("Area of first rectangle=" + r1.area());
          System.out.println("Area of second rectangle=" + r2.area());
     }
}
```





Explanation –

- In the above program, class rectangle has got two constructors, one without parameters and the other is with parameters. **This result into overloading of constructors.**
- Consider the following statement,

  Rectangle r1 = new Rectangle( );

  In this statement while creating the object, the constructor without parameter is called and hence the values of r1.length and r1.breadth are initialized with 2 and 1 respectively.
- Consider the following statement,

  Rectangle r2 = new Rectangle(4,5);

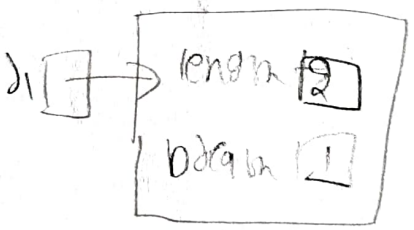
  In this statement while creating the object, the constructor with parameters is called and hence the values of r2.length and r2.breadth are both initialized with 4 and 5 respectively.
- Thus, class user has now got **both** the facilities, **one is**, by default the rectangle will have length and breadth of value 2 and 1 respectively and the **other is** rectangle with the values of length and breadth which are mentioned by the class user as the arguments (4 and 5 in the above example).

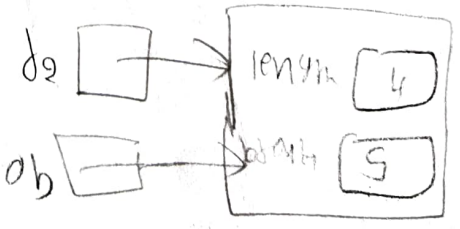
5.7 Overloaded Constructors / Constructors Overloading / Defining Multiple Constructors within the same class along with the additional concept i.e. Constructor accepting Object as Parameter –

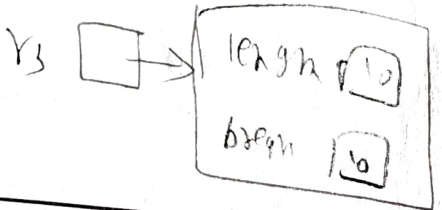
Consider the following program.

- Sometimes when we construct object of a class then we want to initialize its instance variables with the same values as that of the instance variables of some other object of the same class.
- In such a case, we will have to write a constructor that accepts the object of the same class (strictly speaking the constructor that accepts reference of the same class).

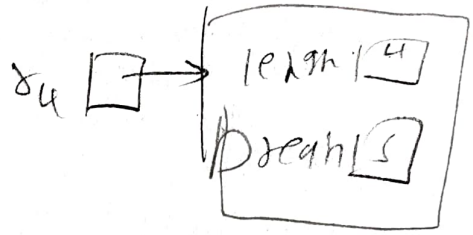
```
class Rectangle
{
      double length, breadth;
      double area()
      {
            return length * breadth;
      }
      Rectangle()
      {
            length=2;
            breadth=1;
      }
      Rectangle(double side)
      {
            length=breadth=side;
      }
      Rectangle(double l, double b)
      {
            length = l;
            breadth = b;
      }
```







```
        Rectangle(Rectangle ob)
        {
              length = ob.length;
              breadth = ob.breadth;
        }
}
class RectangleDemo5
{
      public static void main(String args[])
      {
            Rectangle r1 = new Rectangle();
            Rectangle r2 = new Rectangle(4,5);
            Rectangle r3 = new Rectangle(10);
            Rectangle r4 = new Rectangle(r2);

            System.out.println("Area of first rectangle=" + r1.area());

            System.out.println("Area of second rectangle=" + r2.area());

            System.out.println("Area of third rectangle=" + r3.area());

            System.out.println("Area of fourth rectangle=" + r4.area());

      }
}
```



Explanation –

- In the above program we are writing a class Rectangle and that has four constructor functions.
- The first constructor is not accepting any parameter and it is initializing the length and breadth with values 2 and 1.
- The second constructor, accepted one value type double and initialize both length and breadth with the same value.
- The third constructor, accepted two values of type double and initializes length and breadth with those two accepted values.
- The fourth constructor accepts the object of class Rectangle as parameter (strictly speaking accepts the reference to the object of class Rectangle) and initializes length and breadth of the constructed object with the values of length and breadth of the accepted object.
- Consider the following statements,

  Rectangle r1 = new Rectangle();

  Rectangle r2 = new Rectangle(4,5);

  Rectangle r3 = new Rectangle(10);

  Rectangle r4 = new Rectangle(r2);

- For the first object, the constructor without parameter is called and hence r1.length and r1.breadth are initialized with 2 and 1.
- For the second object, the constructor with two parameters of type double is called and hence r2.length and r2.breadth are initialized with the values 4 and 5 respectively.
- For the third object the constructor with one parameter of type double is called and hence the r3.length and r3.bredth are both initialized by 10.
- For the further object, the constructor accepting object as a parameter is called and hence the r4.length and r4.breadth are initialized with the values of r2.length and r2.breadth respectively.

5.7 Another program using Overload Constructors, the Constructor accepting Object as parameter and another Method accepting Object as parameter –

Consider the following program.

```
class Rectangle
{
      double length, breadth;
      double area()
      {
            return length * breadth;
      }
      Rectangle()
      {
            length=2;
            breadth=1;
      }
      Rectangle(double side)
      {
            length=breadth=side;
      }
      Rectangle(double l, double b)
      {
            length = l;
            breadth = b;
      }
      Rectangle(Rectangle ob)
      {
            length = ob.length;
            breadth = ob.breadth;
      }
      boolean equals(Rectangle ob)
      {
            return(area() = = ob.area());
      }
}
class RectangleDemo6
{
      public static void main(String args[])
      {
            Rectangle r1 = new Rectangle(10,2);
            Rectangle r2 = new Rectangle(4,5);
            Rectangle r3 = new Rectangle(10);
            Rectangle r4 = new Rectangle(r3);

            System.out.println("Area of first rectangle=" + r1.area());

            System.out.println("Area of second rectangle=" + r2.area());

            System.out.println("Area of third rectangle=" + r3.area());

            System.out.println("Area of fourth rectangle=" + r4.area());
```

```
        if(r1.equals(r2))
        System.out.println("Areas of first rectangle and second rectangle are equal");
        else
        System.out.println("Areas of first rectangle and second rectangle are not equal");

        if(r3.equals(r4))
        System.out.println("Areas of third rectangle and fourth rectangle are equal");
        else
        System.out.println("Areas of third rectangle and fourth rectangle are not equal");

        if(r1.equals(r3))
        System.out.println("Areas of first rectangle and third rectangle are equal");
        else
        System.out.println("Areas of first rectangle and third rectangle are not equal");

        if(r2.equls(r4))
        System.out.println("Areas of second rectangle and fourth rectangle are equal");
        else
        System.out.println("Areas of second rectangle and fourth rectangle are not equal");
    }
}
```

Explanation –

- Consider the statement.

  if(r1.equals(r2))
  System.out.println("Areas of first rectangle and second rectangle are equal");
  else
  System.out.println("Areas of first rectangle and second rectangle are not equals");

  - The method equals( ) is called for object referred by r1 and the object referred by r2 is passed as parameter.
  - The parameter ob acts as other reference to the same object which is being referred by r2. Within the method equals( ), we are calling another method called as area( ) of class rectangle. The call area( ) calculates and returns the answer of area of r1.
  - The call ob.area() calculates and returns the answer of area of r2.
  - If these returned values are same then the method equals( ) returns true and otherwise it returns false. Based on the returned value the appropriate message will be displayed.
  - Similar explanation is applicable to other calls.

- The header of the method equals( ) is given as follows.

  boolean equals(Rectangle ob)

It indicates that equals() is a method that accepts reference to the object of class Rectangle into ob and return a value of type boolean.

5.9 keyword this –

- In certain situations, the method, need to refer the object for which the method is called. In Java, it can be done using a keyword **this.**
- The keyword **this** is always a reference to the current object for which the method is invoked i.e. called.

Consider the following example,

```
Rectangle(double l, double b)
{
        this.length = l;
        this.breadth = b;
}
```

When the above constructor is called for object referred by r1, then this.length and this.breadth are nothing but r1.length and r1.bredth. The reason is, since the constructor is called for object referred by r1, then **this** acts as reference to the same object to which the r1 is referring.

However, in the above discussed constructor, the keyword **this** is redundant. The reason is following constructor will also give us the same effect.

```
Rectangle(double l, double b)
{
        length = l;
        breadth = b;
}
```

Hence the keyword this is not really necessary to be used within the constructor.

The significant use of keyword this –

- In the previous constructors the parameter used are having the names **l** and **b** wheres the names of instance variables are length and breadth.
- It is possible that programmer may want to name the parameters **also** as length and breadth however, this will cause the names of instance variables to be **overlapped** by the names of the parameter names.
- Hence, it will not be possible to access the instance variable names within the constructor method (in general in any method).
- This problem is called as hiding of instance variable names due to the local parameters having same names as that of instance variable names.
- The particular problem can be solved with the help of keyword **this.** The solution is given as follows.

```
Rectangle(double length, double breadth)
{
        this.length = length;
        this.breadth = breadth;
}
```

- In the above example, the length and breadth are the names of the local parameters whereas the instance variable names are referred as this.length and this.breadth.
- Consider the following statement,

  Rectangle r1 = new Rectangle(10,8)

- When the object is created using new then the constructor is called and the arguments 10 and 8 are passed. These values will be copied into parameters length and breadth. The value of parameters length and breadth are assigned to this.length and this.breadth (which will be instance variables) i.e. r1.length and r1.breadth.

- **Note –**
  - Some programmers prefer to write the names of parameters same as that of the instance variables for clarity and use the keyword **this** to overcome the hiding of the instance variables.
  - Whereas some programmers think exactly opposite and to avoid confusion they prefer to use the different names for the local parameters (say **l** and **b**) and instance variables (say **length** and **breadth**). In that case the keyword **this** will not be required at all.

5.10 Returning the Object

(also covers another significant use of keyword this) –

Consider the following program.

```
class Rectangle
{
    double length, breadth;
    double area()
    {
        return length * breadth;
    }
    Rectangle()
    {
        length=2;
        breadth=1;
    }
    Rectangle(double side)
    {
```

5.12 Method Overloading –

- Method overloading is one of the ways to achieve polymorphism in Java.
- Each method within a class is unique identified with tits name, number of parameters and their types.
- Hence it is possible to have two ore more methods with the same name but different parameter list. The difference will either in the number of parameters or their types or both.
- This feature of Java is called as method overloading. (we are overloading (i.e. overusing)) the name of the method.
- The method overloading allows us to perform the same action on different types of data inputs.
- In Java, whenever, the method is being called, first of all, the method name is matched with the available methods and then the number of arguments and their types are matched.
- In method overloading, two methods can have same name but different **signatures** i.e. the number of parameters and their types can be different.
- Out of available methods with same name, with whichever signature the exact match is found, that method is called during the execution of a method call.

Program – Consider the following program.

```
class OverLoadDemo1
{
     int add(int x, int y)                    //method1
     {
          return x+y;
     }
     int add(int x, int y, int z)             //method2
     {
          return x+y+z;
     }
     float add(float x, float y)              //method3
     {
          return x+y;
     }
     double add(double x, double y)           //method4
     {
          return x+y;
     }
     long add(long x, long y)                 //method5
     {
          return x+y;
     }
```

```
        public static void main(String args[])
        {
            OverLoadDemo1 obj1=new OverLoadDemo1();
            System.out.println(obj1.add(5,10));                          //Method1
            System.out.println(obj1.add(5.25F,10.5F));                   //Method3
            System.out.println(obj1.add(2.3,4.5));                       //Method4
            System.out.println(obj1.add(2,3,4));                         //Method2
            System.out.println(obj1.add(2147483650L,2147483651L));       //Method5
            System.out.println(obj1.add(23.4,5));                        //Method4
        }
    }
```

Explanation –

- The class OverLoadDemo1 has five methods all with name add( ). All the method have different signatures. This is because, all these methods are having same name but they are different term of the number of parameters and their types.
- These methods are invoked from the main method. Every call is compared with every signature and whenever the perfect match is found the corresponding method is executed.
- The comments in the above program are showing the correct match of every call with the available add methods.
- Consider the last call. The arguments are of type, double and integer. There is no exact match. In that case, the second argument i.e. integer is implicitly converted to double and hence the add with parameters double and double comes in picture.

Program – Modify the above program by defining all the add( ) methods as static methods.

Program – Develop a class to demonstrate the method overloading with method named as max( ) to find out maximum out of two numbers. The two numbers can be of different data types.

Program – Develop a class ThreeDObject. Use this class to calculate and return the volume of Sphere, Cylinder and parallelepiped. Use the concept of method overloading.

Program – Develop a class Triangle. Use this class to calculate the area of triangle either using base and height or using three sides of triangle. Use the concept of method overloading.

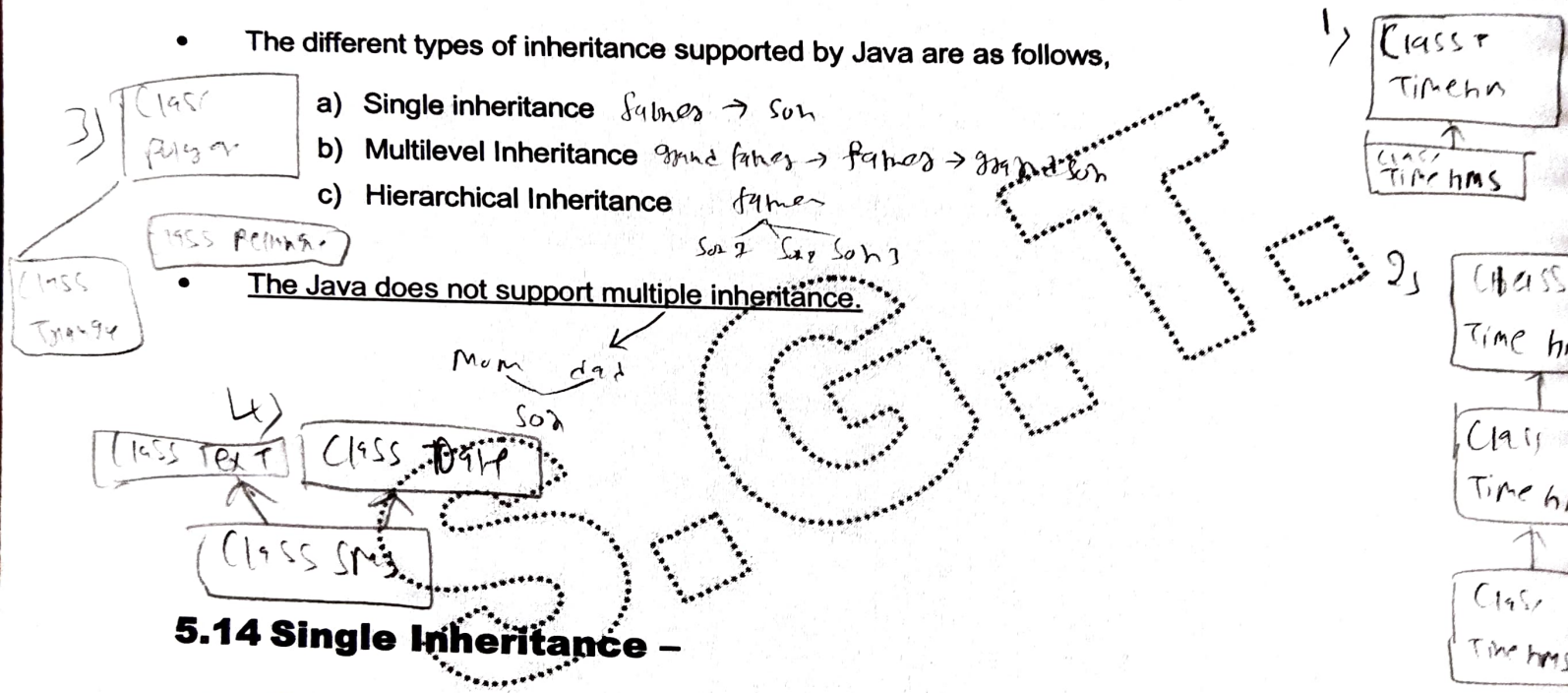
5.13 Basics of inheritance –

- Inheritance is the mechanism of deriving one class from the other class. The class from which the other class is derived is called as **superclass.** The class which is derived from the superclass is also called as **subclass.**

- A subclass is a **specialized version** of the superclass or it can also be said that the subclass is the **extended version** of the superclass.

- The instance variables and the method of the superclass are **inherited** to the subclass or in other words the instance variables and the methods of the superclass are **extended** to the subclass.

- The different types of inheritance supported by Java are as follows,



a) Single inheritance
b) Multilevel Inheritance
c) Hierarchical Inheritance

- The Java does not support multiple inheritance.

5.14 Single Inheritance –

- The single inheritance is the mechanism of deriving single class from other single class.

General syntax –

The general form of single inheritance is given as follows.

```
class superclassname
{
      //body of the superclass
}

class subclassname extends superclassname
{
      body of the subclass
}
```

5.15 Single Inheritance in details – Constructor Methods with Single Inheritance – Method Overriding –

Consider the following program.

```
Class Timehm
{
        int hour,min;
        void display()
        {
                System.out.println(hour + ":" + min);
        }
        Timehm()
        {
                hour=min=0;
        }
        Timehm(int h)
        {
                hour=h;
                min=0;
        }
        Timehm(int h, int m)
        {
                hour=h;
                min=m;
        }
        Timehm(Timehm ob)
        {
                hour=ob.hour;
                min=ob.min;
        }
}
class Timehms extends Timehm
{
        int sec;
        void display()
        {
                System.out.println(hour + ":" + min + ":" + sec);
        }
        Timehms()
        {
                hour=min=sec=0;
        }
        Timehms(int h)
        {
                hour=h;
                min=sec=0;
        }
        Timehms(int h, int m)
        {
                hour=h;
                min=m;
                sec=0;
        }
        Timehms(int h, int m, int s)
        {
                hour=h;
                min=m;
                sec=s;
        }
        Timehms(Timehms ob)
        {
                hour=ob.hour;
                min=ob.min;
                sec=ob.sec;
        }
}
```

```
class TimeDemo
{
	public static void main(String args[])
	{
		Timehm t1 = new Timehm();
		Timehm t2 = new Timehm(7);
		Timehm t3 = new Timehm(9,30);
		Timehm t4 = new Timehm(t3);
		t1.display();
		t2.display();
		t3.display();
		t4.display();

		Timehms t5 = new Timehms();
		Timehms t6 = new Timehms(7);
		Timehms t7 = new Timehms(9,30);
		Timehms t8 = new Timehms(11,45,27);
		Timehms t9 = new Timehms(t8);

		t5.display();
		t6.display();
		t7.display();
		t8.display();
		t9.display();
	}
}
```

Explanation –

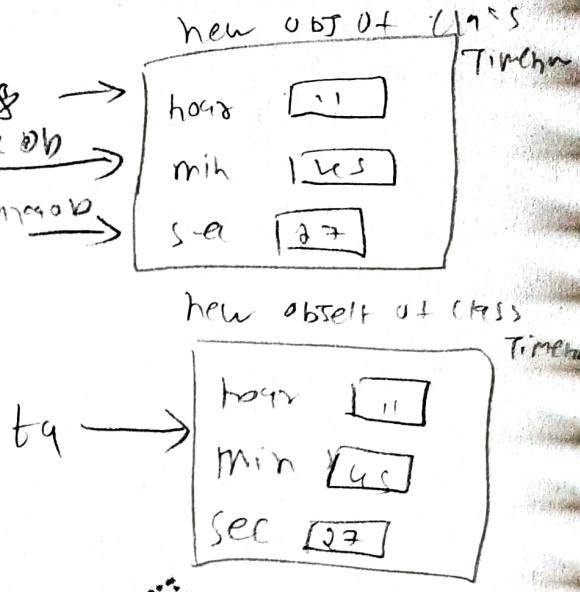
- The class timehm has got four constructor methods and one method called as display( ). It has two instance variables hour and min. All these members are having **default access specifier.**
- The subclass Timehms is **extended** from the superclass Timehm. The class Timehms has got its own five constructor methods and its own method display( ). It has one instance variable called as sec. These members has also got **default access specifier.** The instance variables hour, min and methods display( ) and constructor methods of superclass Timehm **are also available** to the subclass Timehms **due to inheritance.**
- For the objects t1, t2,t3 and t4 of the superclass Timehm, the appropriate constructor methods of the superclass Timehm are called an initialization of each object is carried out. When the method display( ) is called for t1, t2, t3 and t4, then obviously method display( ) belonging to the super class Timehm is called.
- For the objects t5, t6, t7, t8 and t9 of the subclass Timehms, the appropriate constructor methods of the subclass Timehms are called and initialization of each object is carried out.
- In both superclass and subclass **overloading of constructor methods** is implemented. At the time of construction of instances, with whichever **signature** of the constructor method the exact match is found, that particular constructor is called. The **signature** covers method name, number of parameter and their types.
- When the method display( ) is called for t5, t6, t7, t8 and t9, then two display( ) methods are candidates and the display( ) method of the subclass gets more priority and the display( ) method of the subclass Timehms is called for these objects. Thus, it can be said that display( ) method of the superclass Timehm is overridden by display() method of the subclass Timehms or in other words display( ) method of the subclass Timehms is the overriding method. It can also be said that disaply( ) method of the superclass Timehm is hidden by the display( ) of the subclass.
- When the method within the superclass and the method within the subclass has got same names then the method within the subclass overrides the method of the superclass or in other words the method within the superclass is overridden by or hidden due to the method of the subclass.

5.16. First Use of keyword Super –

```
class Timehm
{
	int hour,min;
	void display()
	{
		System.out.println(hour + ":" + min);
	}
	Timehm()
	{
		hour=min=0;
	}
	Timehm(int h)
	{
		hour=h;
		min=0;
	}
	Timehm(int h, int m)
	{
		hour=h;
		min=m;
	}
	Timehm(Timehm ob)
	{
		hour=ob.hour;
		min=ob.min;
	}
}
class Timehms extends Timehm
{
	int sec;
	void display()
	{
		System.out.println(hour + ":" + min + ":" + sec);
	}
	Timehms()
	{
		super();
		sec=0;
	}
	Timehms(int h)
	{
		super(h);
		sec=0;
	}
	Timehms(int h, int m)
	{
		super(h,m);
		sec=0;
	}
	Timehms(int h, int m, int s)
	{
		super(h,m);
		sec=s;
	}
	Timehms(Timehms ob)
	{
		super(ob);
		sec=ob.sec;
	}
}
```

parent चया reference का use child चा object

```
class TimeDemo1
{
        public static void main(String args[])
        {
                Timehm t1 = new Timehm();
                Timehm t2 = new Timehm(7);
                Timehm t3 = new Timehm(9,30);
                Timehm t4 = new Timehm(t3);

                t1.display();
                t2.display();
                t3.display();
                t4.display();

                Timehms t5 = new Timehms();
                Timehms t6 = new Timehms(7);
                Timehms t7 = new Timehms(9,30);
                Timehms t8 = new Timehms(11,45,27);
                Timehms t9 = new Timehms(t8);

                t5.display();
                t6.display();
                t7.display();
                t8.display();
                t9.display();
        }
}
```



Explanation –

- In the above program, using the keyword **super**, the constructor methods of the superclass are called from the constructor methods of the subclass.
- Consider the following statement,

  Timehms t5 = new Timehms();

  The constructor method without parameters of the subclass Timehms is called for the instance referred by t5. The first statement within that constructor is super( ); It means that constructor method without parameter of the superclass Timehm will be called and hence due to super( ) actually, the Timehm() is called and hence t5.hour and t5.min are initialized with zero and then the control comes back to the Timehms() and then the t5.sec is initialized with zero.
- Consider the following statement,

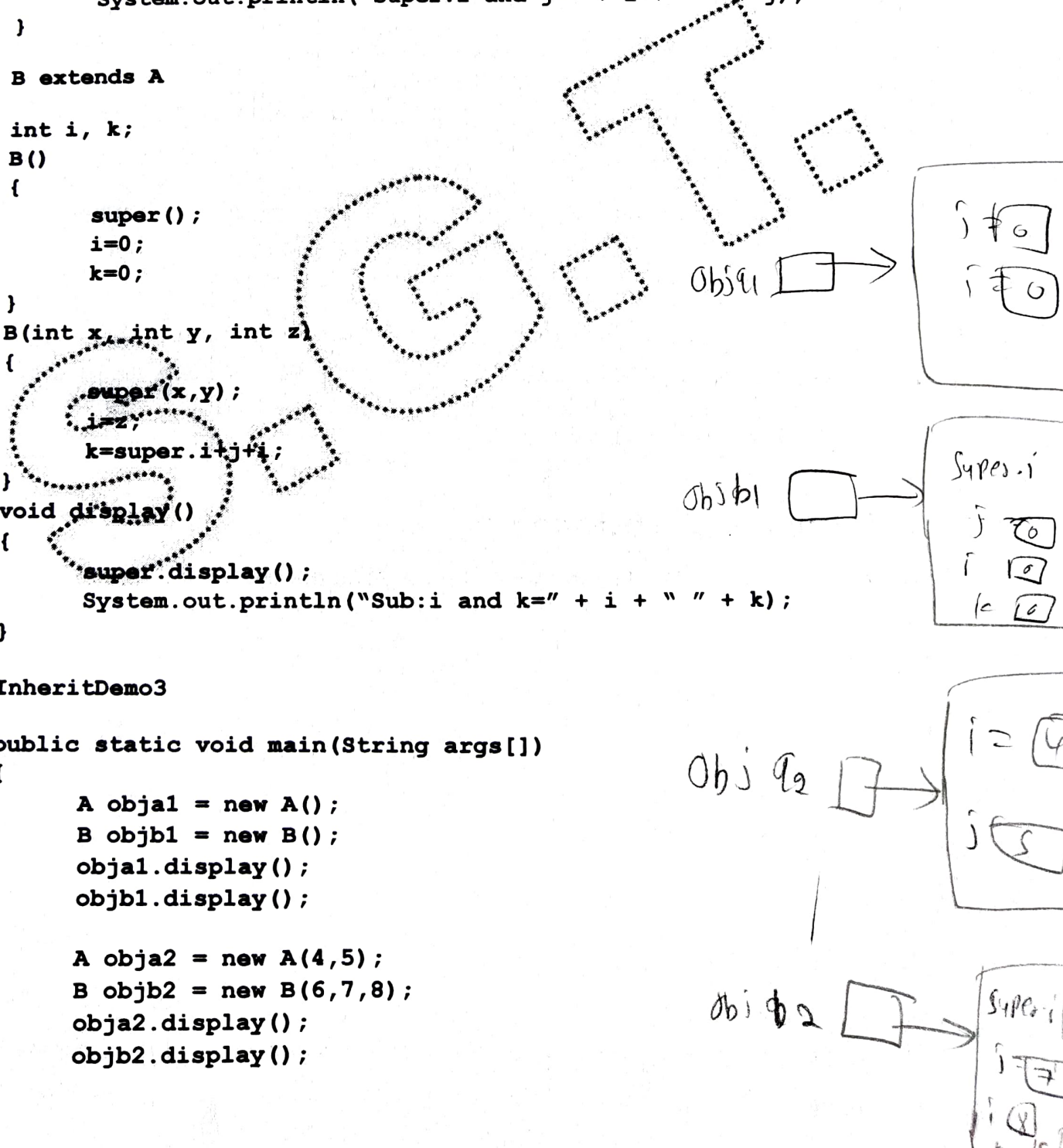
  Timehms t6 = new Timehms(7);

  The constructor method with one parameter of the subclass Timehms is called for the instance referred by t6. The first statement within that constructor is super(h); It means that constructor method with one parameter of the superclass Timehm will be called and hence due to super(h) actually, the Timehm(h) is called and hence t6.hour is initialized with the value of h i.e.7 and t6.min is initialized with zero and then the control comes back to the Timehms(h) and then the t6.sec is initialized with zero.
- Similar explanation is applicable to the other objects of the class Timehms.

5.17 Second use of Super –

Consider the following example.

```
class A
{
        int i, j;
        A()
        {
                i=j=0;
        }
        A(int x, int y)
        {
                i=x;
                j=y;
        }
        void display()
        {
                System.out.println("Super:i and j=" + i + " " + j);
        }
}
class B extends A
{
        int i, k;
        B()
        {
                super();
                i=0;
                k=0;
        }
        B(int x, int y, int z)
        {
                super(x,y);
                i=z;
                k=super.i+j+i;
        }
        void display()
        {
                super.display();
                System.out.println("Sub:i and k=" + i + " " + k);
        }
}
class InheritDemo3
{
        public static void main(String args[])
        {
                A obja1 = new A();
                B objb1 = new B();
                obja1.display();
                objb1.display();

                A obja2 = new A(4,5);
                B objb2 = new B(6,7,8);
                obja2.display();
                objb2.display();
        }
}
```



Explanation –

- In the above program the instance variable i and the method display( ) of the superclass A are overridden by the instance variable i and the display( ) of the derived class B.
- When we access variable i within the class B then the variable i of the derived class comes in picture. This is because it overrides the variable i of the super class. However, in certain situations the subclass B want to access the variable i of the superclass A and not the variable i of its own, then it can be obtained using the keyword super. Consider the following line,

  k = super.i + i + j;

  This line will add the values of the instance variable i of the super class A, the instance variable i of the subclass B and the instance variable j and stores the answer in the instance variable k.
- Thus, even if the instance variable i of the subclass B overrides the instance variable i of the superclass A then also with the help of keyword super it is possible to access the instance variable i of the superclass within the subclass B.
- When we call the method display( ) for objb2 then the display( ) method that belongs to the subclass B is called. This is because the method display( ) of the subclass B overrides the display( ) of the superclass A. Within the method display( ) of the subclass B the first statement is the call i.e.

  super.display( );

  The statement means that call the method display( ) that belongs to the superclass A. The display( ) of the superclass A gets executed and then the control comes back to the display( ) of the subclass B and its remaining logic gets executed.
- The instance variables and methods of the subclass having same names as that of the instance variables and methods of the superclass, overrides (hides) the instance variables and methods of the superclass. The keyword super can be used to make the overridden (hidden) instance variables and methods of the superclass visible.

5.18 Multilevel Inheritance –

The general form of the multilevel inheritance can be given as follows.

```
class topclassname
{
      //body of the topclass
}
class middleclassname extends topclassname
{
      //body of the middleclass
}
class lowerclass extends middleclass
{
      //body of the lowerclass
}
```

- The middle class acts as the extended version of the top class and the lower class is the extended version of the middle class. The above example shows the three layer multilevel inheritance and it can be further extended if required according to the application.

Consider the following example.

```
class Timehm
{
	int hour,min;
	void display()
	{
		System.out.println(hour + ":" + min);
	}
	Timehm()
	{
		hour=min=0;
	}
	Timehm(int h, int m)
	{
		hour=h;
		min=m;
	}
}
class Timehms extends Timehm
{
	int sec;
	void display()
	{
		System.out.println(hour + ":" + min + ":" + sec);
	}
	Timehms()
	{
		super();
		sec=0;
	}
	Timehms(int h, int m, int s)
	{
		super(h,m);
		sec=s;
	}
}
class Timehmsms extends Timehms
{
	int ms;
	void display()
	{
		System.out.println(hour + ":" + min + ":" sec + ":" + ms);
	}
	Timehmsms()
	{
		super();
		ms=0;
	}
	Timehmsms(int h, int m, int s, int x)
	{
		super(h,m,s);
		ms=x;
	}
}
```

```
class TimeDemo2
{
      public static void main(String args[])
      {
            Timehm t1 = new Timehm();
            Timehm t2 = new Timehm(9,30);

            t1.display();
            t2.display();

            Timehms t3 = new Timehms();
            Timehms t4 = new Timehms(13,30,45);

            t3.display();
            t4.display();

            Timehmsms t5 = new Timehmsms();
            Timehmsms t6 = new Timehmsms(10,30,25,345);

            t5.display();
            t6.display();
      }
}
```

Explanation –

- In the above program we declare three classes Timehm, Timehms and Timehmsms. These three classes together implements multilevel inheritance.
- Consider the following statement:

  Timehmsms t6 = new Timehmsms(10, 30, 25, 345);

- Due to the above statement the instance of type Timehmsms is created and the constructor method with parameters i.e. Timehmsms(int h, int m, int s, int x) of the class Timehmsms is called. The values 10, 30, 25 and 345 are accepted into h, m, s and x respectively.
- The first line within the constructor method of Timehmsms (int h, int m, int s, int x) is super(h,m,s). The line causes a call to the constructor method Timehms (int h, int m, int s) and the h, m and s of this method accepts the values 10, 30 and 25 into h, m and s respectively.
- The first line within the constructor method Timehms (int h, int m, int s) is super(h,m). This line cause a call to the constructor method Timehm (int h, int m) and the h and m of this method accepts values 10 and 30 into h and m respectively.
- The constructor method Timehm (int h, int m) will initialize t6.hour and t6.min with 10 and 30 respectively. THe control then returns within the constructor method Timehms(int h, int m, int s) and it will initialize t6.sec with 25. The control then returns within the constructor method Timehmsms(int h, int m, int s, int x) and it will initialize t6.ms with 345.

Note1 – If the call to the super is present within the constructor method then the call to the super must be the first statement in the constructor method. Hence, the order in which the execution of the constructor methods take place is same as that of the order of derivation. In the above program the execution of the constructor method topclass i.e. Timehm takes place first then the execution of the constructor method middleclass i.e. Timehms takes place and then the execution of the constructor method of the lowerclass i.e. Timehmsms get completed.

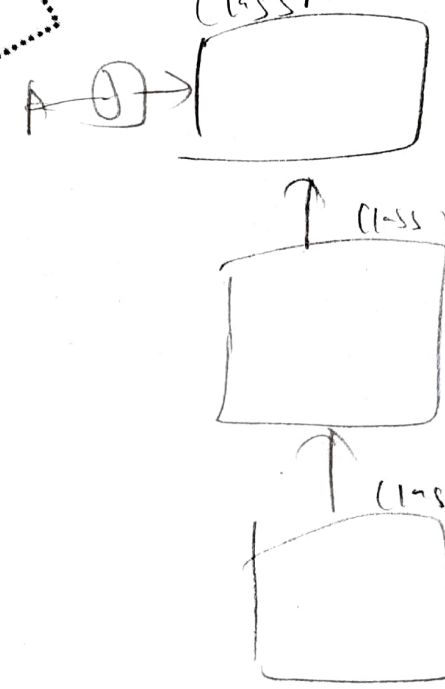
Note2 – If the first statement within the constructor method of a subclass is not a call to super then the constructor without parameters of the superclass will be called.

For example, if the constructor method Timehmsms (int h, int m, int s, int x) is not having a super( ) call then the constructor method without parameters of superclass Timehms is called and which will in turn call the constructor method of the superclass Timehm. The net effect is t6.hour, t6.min and t6.sec will all be initialized with zero and t6.ms will be initialized with 345.

5.19 Multilevel Inheritance to indicate the order in which the constructors are executed –

Consider the following example.

```
class A
{
      A()
      {
            System.out.println("Upper class");
      }
}
class B extends A
{
      B()
      {
            System.out.println("Middle class");
      }
}
class C extends B
{
      C()
      {
            System.out.println("Lower class");
      }
}
class Demo
{
      public static void main(String args[])
      {
            A  obja = new A ( );
            B  objb = new B ( );
            C  objc = new C ( );
      }
}
```



Explanation –

- When instance of class A is constructed which is referred by reference obja then the constructor method of class A is called and it gives the output,

  Upper class

- When instance of class B is constructed which is referred by reference objb then the constructor method of class B is called and since it is not having the super call, the constructor method of class A is called automatically and the constructor method of class A gets executed and then the control comes back to the constructor method of class B and it gets executed. Hence the output is given as,

  Upper class
  Middle class

- When instance of class C is constructed which is referred by reference objc then the constructor method of class C is called and since it is not having the super call, the constructor method of class B is called automatically and since it is also not having the call to the super, the constructor method of class A gets called automatically and executed and then the control comes back to the constructor method of class B and it gets executed and then the control comes back to the constructor method of class C and then it gets executed. Hence the output is given as,

  Upper class
  Middle class
  Lower class

5.20 Hierarchical Inheritance –

Consider the following example.

```
class A
{
      int data;
      A()
      {
            data=10;
      }
      void display()
      {
            System.out.println("Super data:" + data);
      }
}
class B extends A
{
      int data;
      B()
      {
            data = super.data * 5;
      }
      void display()
      {
            super.display();
            System.out.println("Sub(B) data:" + data);
      }
}
```

```
class C extends A
{
        int data;
        C()
        {
                data = super.data * 7;
        }
        void display()
        {
                super.display();
                System.out.println("Sub(C) data:" + data);
        }
}
class Demo1
{
        public static void main(String args[])
        {
                A obja = new A();
                obja.display();
                B objb = new B();
                objb.display();
                C objc = new C();
                objc.display();
        }
}
```

Explanation –

- In the above program the class A acts as super class for the two classes B and C. In other words classes B and C are subclasses of class A.
- The instance variable data and the method display( ) of class A are inherited to both the subclasses B and C. Both the sub classes B and C are having their own instance variables named as data and methods named s display().
- The sub classes B and C can access the instance variable data and the method display( ) of super class A using the keyword super. The data of super class A is accessed within classes B and C as super.data and the method display( ) of the class A is accessed within the classes B and C as super.display().
- The classes A, B and C together forms **hierarchical inheritance.** It is possible to expand this hierarchical further by deriving the classes from B and C.

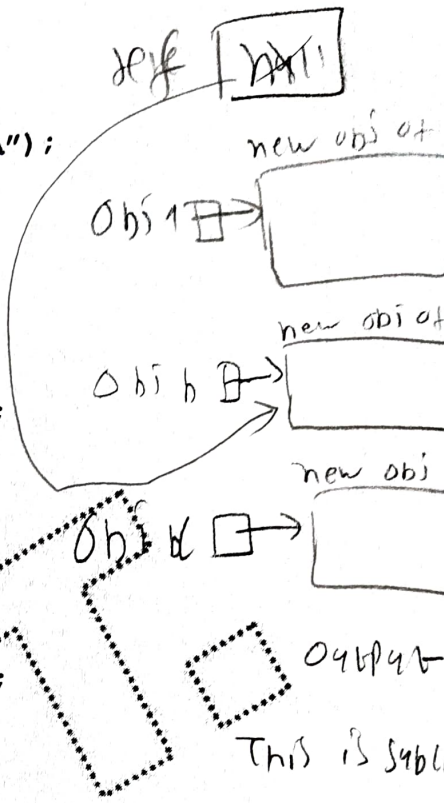
5.21 Dynamic Method Dispatch to implement Run Time Polymorphism –

- **Method overriding** forms the basis of one of the most powerful concepts of Java and that is **Dynamic Method Dispatch.** The dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time rater than at the compilation time. With the help of **Dynamic Method Dispatch** Java implements **run time polymorphism.**
- **The superclass reference variable can refer to a subclass object.** When the overridden method is called using the superclass reference then Java determines which version of that method to execute, based upon the type of the object being referred by the superclass reference at that time. Thus, this determination is done at **run time.** Thus, the type of the object which is being referred is more important and not the type of the reference variable to determine which version of the overridden method will be executed.

Consider the following example.

```
import java.io.*;
class A
{
      void display()
      {
            System.out.println("This is super class A");
      }
}
class B extends A
{
      void display()
      {
            System.out.println("This is subclass B");
      }
}
class C extends A
{
      void display()
      {
            System.out.println("This is subclass C");
      }
}
class Demo2
{
      public static void main(String args[]) throws IOException
      {
            A obja = new A();
            B objb = new B();
            C objc = new C();
            A ref;
            BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
            System.out.print("Enter one integer=");
            int i=Integer.parseInt(br.readLine());
            switch(i)
            {
                  case 1:     ref=obja;
                              break;
                  case 2:     ref=objb;
                              break;
                  case 3:     ref=objc;
                              break;
                  default:    ref=null;
                              break;
            }
            ref.display();
      }
}
```

i = 2



Note:

It is not runtime Polymorphism because of switch case

It is Run Time polymorphism because Object is created at run time

Explanation –

- The **ref** is the reference variable of type class A. Therefore ref can refer to the instance class A. Moreover it can also refer to the instances of subclass B and subclass C.
- The superclass A and the two subclasses B and C has got their own display( ) methods. Thus, the display() of B and C overrides the display( ) of class A.
- The value of the variable i is accepted from user. Depending on the value of i which case will come in picture and ref will refer instance of which class is all decided at run time and not at the compilation time. Therefore, due to the statement ref.display(), out of the three different display( ) methods, exactly which display( ) will be called will be decided at run time.
- If the value of i is 1, then due to switch statement, the ref will refer the instance to which the obja is referring. Hence, due to ref.display( ), the display() of the classes A will be called and the output will be,

  This is super class A.
- If the value of i is 2, then due to switch statement, the ref will refer the instance to which the objb is referring. Hence, due to ref.display( ). The display( ) of the class B will be called and the output will be,

  This is subclass B.
- If the value of i is 3. Then due to switch statement, the ref will refer the instance to which the objc is referring. Hence, due to ref.display( ), the display( ) of the class C will be called and the output will be,

  This is subclass C.
- Thus, the single interface i.e. **ref.display()** gives multiple effects. **This results into the run time polymorphism.**

5.22 Dynamic Method Dispatch to implement Run Time Polymorphism –

Consider the following example.

```
import java.io.*;
class Shape
{
      double x, y;
      Shape(double x1, double y1)
      {
            x=x1;
            y=y1;
      }
      double area()
      {
            System.out.println("Area undefined");
            return 0;
      }
}
```

For option = 1
Output
Area is undefined.
Area is 0

```
class Rectangle extends Shape
{
        Rectangle(double x1, double y1)
        {
                super(x1, y1);
        }
        double area()
        {
                return x*y;
        }
}
class Triangle extends Shape
{
        Triangle(double x1, double y1)
        {
                super(x1,y1);
        }
        double area()
        {
                return 0.5*x*y;
        }
}
class Demo3
{
      public static void main(String args[]) throws IOException
      {
            int option;
            Shape ref;
            BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
            System.out.print("Enter your option(1:any shape,2:Rectangle,3:Triangle)=");
            option=Integer.parseInt(br.readLine());
            switch(option)
            {
                case 1:   Shape s = new Shape(10,10);
                          ref=s;
                          break;
                case 2:   double l,b;
                          System.out.print("Enter length=");
                          l=Double.parseDouble(br.readLine());
                          System.out.print("Enter breadth=");
                          b=Double.parseDouble(br.readLine());
                          Rectangle r = new Rectangle(l,b);
                          ref=r;
                          break;
                case 3:   double p,q;
                          System.out.print("Enter side1 of right angle triangle=");
                          p=Double.parseDouble(br.readLine());
                          System.out.print("Enter side2 of right angle triangle=");
                          q=Double.parseDouble(br.readLine());
                          Triangle t = new Triangle(p,q);
                          ref=t;
                          break;
                default:ref=null;
                          break;
            }
            System.out.println("Area is " + ref.area());
      }
}
```

Explanation –

- In the above program the superclass shape and the subclasses rectangle and triangle implements hierarchical inheritance.
- The display( ) methods in all the three classes results into overridden methods.
- With the help of ref which is reference of type superclass shape, the run time polymorphism is implemented.
- Depending on the value of option entered by user, a particular case of the switch statement gets selected.
- If the value of option is 1, then the ref will refer the instance to which the reference variable s is referring. If the value of option is 2, then the ref will refer the instance to which the reference variable r is referring. If the value of option is 3, then the ref will refer the instance to which the reference variable t is referring.
- The call ref.area( ), will actually call which area( ) method out of the three available area( ) methods is dependent on the value of the option. Thus, which area( ) method will be dispatched with the call ref.area() is decided at run time i.e. it is decided dynamically depending on the option entered by the user.
- Thus, the single interface ref.area( ) gives, different effects i.e. area of shape, area of rectangle and are of triangle resulting into polymorphism. Moreover, which method is to be dispatched is decided at run time i.e. dynamically resulting into **Run Time Polymorphism.**

5.23 Abstract Classes and Abstract Methods –

- There are situations where we want to define a superclass that declares a structure of a given abstraction without providing the complete implementation of every method.
- It means that we want to create superclass that defines generalized form that will be shared by all the subclasses. The specific implementation of the generalized form will be managed by every subclass.
- This type of the superclass declares the nature of methods and the subclasses implement their details according to their own requirements.
- This situation can occur when the superclasses are unable to provide meaningful implementation for these methods as that of the previous program.
- The class shape of the previous program can not give any meaningful formula for the method area( ) within it. In such situations, the superclass can define **abstract method** and the subclass will override the abstract method. It is said that the abstract method is the responsibility of the subclass.

The general form of declaring the abstract method is as follows.

abstract type name(parameter list);

The abstract method is not having its own body.

When the class has got one or more abstract methods then that class must be declared as abstract class with the help of keyword abstract.

There can not be any object(instance) of abstract class. We can not use new operator with the abstract classes to construct object(instance) of that type.

It is possible to create reference variables of abstract classes so that they can be used to refer instances of the subclasses and thus the run time polymorphism can be implemented.

Consider the following example.

```
import java.io.*
abstract class Shape
{
      double x,y;
      Shape(double x1, double y1)
      {
            x=x1;
            y=y1;
      }
      abstract double area();
}
class Rectangle extends Shape
{
     Rectangle(double x1, double y1)
     {
           super(x1,y1);
     }
     double area()
     {
          return x*y;
     }
}
class Triangle extends Shape
{
     Triangle(double x1, double y1)
     {
          super(x1,y1);
     }
     double area()
     {
          return 0.5*x*y;
     }
}
```

```
class Demo4
{
      public static void main(String args[]) throws IOException
      {
            int option;
            Shape ref;
            BufferedReader br=new BufferedReader(new InputStreamReader(System.in));

            System.out.print("Enter your option(1:Rectangle,2:Triangle)=");
            option=Integer.parseInt(br.readLine());

            switch(option)
            {
                  case 1:  double l,b;
                           System.out.print("Enter length=");
                           l=Double.parseDouble(br.readLine());
                           System.out.print("Enter breadth=");
                           b=Double.parseDouble(br.readLine());
                           Rectangle r = new Rectangle(l,b);
                           ref=r;
                           break;

                  case 2:  double p,q;
                           System.out.print("Enter side1 of right angle triangle=");
                           p=Double.parseDouble(br.readLine());
                           System.out.print("Enter side2 of right angle triangle=");
                           q=Double.parseDouble(br.readLine());
                           Triangle t = new Triangle(p,q);
                           ref=t;
                           break;
            }
            System.out.println("Area is " + ref.area());
      }
}
```

Explanation –

- The shape is declared as abstract class and it has one abstract method in t called as area( ) that accept nothing and its return type is double.
- The abstract method area( ) is implemented in the subclasses triangle and rectangle and the run-time polymorphism is obtained with the help of the reference variable ref.
- There can not be any object i.e. instance of the abstract class shape.
- In the above program the abstract superclass shape and the subclasses rectangle and triangle implements hierarchical inheritance

- The method display( ) in the abstract class shape is an abstract method. The method display( ) is implemented in the sub classes Rectangle and Triangle.
- With the help of ref which is reference to the instance of type superclass shape, the run time polymorphism is implemented.
- Depending on the value of option entered by user, a particular case of the switch statement gets selected.
- If the value of option is 1, then the ref will refer the instance to which the reference variable r is referring. If the value of option is 2, then the ref will refer the instance to which the reference variable t is referring.
- The call ref.area( ), will actually call which area( ) method out of the two available area( ) methods is dependent on the value of the option. Thus, which area( ) method will be dispatched with the call ref.area( ) is decided at run time i.e. it is decided dynamically depending on the option entered by the user.
- Thus, the single interface ref.area( ) gives, different effects i.e. area of rectangle and area of triangle resulting into polymorphism. Moreover, which method is to be dispatched is decided at run time i.e. dynamically resulting into Run Time Polymorphism.

5.24 Use of final –

The keyword final has three uses.

1) To create constant (covered in Module – 2)
2) To prevent overriding
3) To prevent inheritance

Preventing overriding –

- Though the method overriding is a powerful feature of Java, sometimes there are situations where we want to prevent the overriding.
- In this situation, the java has a solution to prevent the overriding. Consider the following example.

```
class A
{
      final void display()
      {
            System.out.println("this is final method");
      }
}
class B extends A
{
      void display( )
      {
            System.out.println("This is illegal");
      }
}
```

- In the above example, the display( ) within the class B can not be defined. It will result into the compilation error. This is because the superclass A has already declared the method display( ) as a final method which means that subclasses of class A can not declare the method with name display( ).
- The final methods can be useful for performance enhancement. This is because if the final methods are smaller then those can be implemented as inline methods and hence their execution will be faster.

- The compiler can convert these methods to inline methods at the time of compilation only. This is because there is no question of subclasses overriding the final methods with the methods having larger codes. The compiler can replace method calls by the corresponding bytecode and thus calling of the method will be saved while running the program.

Preventing Inheritance – (final class)

- Sometimes we might be interested to stop further inheritance from a particular class. In that situation we can declare a class as final class.
- All the methods within the final class will automatically become final methods. It will be illegal to declare a class as abstract as well as final. This is because abstract class defines the details in generalized form which are to be specifically implemented in the subclasses. Consider the following example,

```
final class A
{
        //body
}
class B extends A
{
        //body
}
```

- The declaration of class B will cause compilation error. As the class A is declared as final, no further inheritance from class A is possible.

Consider the following programming example, which covers both the above discussed uses of the keyword final.

```
import java.io.*;
abstract class Employee
{
        String name;
        float salary;
        float tax;
        float total;
        final void readData() throws IOException
        {
                BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
                System.out.print("Name=");
                name = br.readLine();
                System.out.print("Basic salary=");
                salary=Float.parseFloat(br.readLine());

        }
        void display()
        {
                System.out.println("Name=" + name);
                System.out.println("Salary=" + salary);

        }
        abstract void payCalc();
}
```

```
class Engineer extends Employee
{
      float da;
      void payCalc()
      {
            da=1.1f*salary;
      }
      void display()
      {
            super.display();
            System.out.println("DA=" + da);
      }
}

class SalesEngineer extends Engineer
{
      float ta;
      void payCalc()
      {
            super.payCalc();
            ta=0.2f*salary;
            tax=0.1f*salary;
            total=salary+da+ta-tax;
      }
      void display()
      {
            super.display();
            System.out.println("TA=" + ta);
            System.out.println("Tax=" + tax);
            System.out.println("Total=" + total);
      }
}

class ProductionEngineer extends Engineer
{
      float hra;
      void payCalc()
      {
            super.payCalc();
            hra=0.25f*salary;
            tax=0.12f*salary;
            total=salary+da+hra-tax;
      }
      void display()
      {
            super.display()
            System.out.println("HRA=" + hra);
            System.out.println("Tax=" + tax);
            System.out.println(:"Total=" + total);
      }
}
```

```
class Manager extends Employee
{
	float da;
	void payCalc()
	{
		da=1.25f*salary;
	}
	void display()
	{
		super.display();
		System.out.println("DA=" + da);
	}
}

class SalesManager extends Manager
{
	float ta;
	void payCalc()
	{
		super.payCalc();
		ta=0.4f*salary;
		tax=0.15f*salary;
		total=salary+da+ta-tax;
	}
	void display()
	{
		super.display();
		System.out.println("TA=" + ta);
		System.out.println("Tax=" + tax);
		System.out.println("Total=" + total);
	}
}

class ProductionManager extends Manager
{
	float hra;
	void payCalc()
	{
		super.payCalc();
		hra=0.5f*salary;
		tax=0.18f*salary;
		total=salary+da+hra-tax;
	}
	void display()
	{
		super.display();
		System.out.println("HRA=" + hra);
		System.out.println("Tax=" + tax);
		System.out.println("Total=" + total);
	}
}
```

```
final class Worker extends Employee
{
      float othours, otrate;
      void readOvertime() throws IOException
      {
            BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
            System.out.print("Over time hours=");
            othours=Float.parseFloat(br.readLine());
            System.out.print("Over time rate=");
            otrate=Float.parseFloat(br.readLine());
      }
      void payCalc()
      {
            tax=0.05f*salary;
            total=salary + othours*otrate - tax;
      }
      void display()
      {
            super.display();
            System.out.println("Overtime=" + othours*otrate);
            System.out.println("Tax=" + tax);
            System.out.println("Total=" + total);
      }
}
```

```
class EmployeeDemo
{
	public static void main(String args[]) throws IOException
	{
		SalesEngineer e1 = new SalesEngineer();
		SalesManager m1 = new SalesManager();

		ProductionEngineer e2 = new ProductionEngineer();
		ProductionManager m2 = new ProductionManager();

		Worker w1 = new Worker();

		BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
		int option;

		do
		{
			System.out.println("1:Sales Engineer");
			System.out.println("2:Sales Manager");
			System.out.println("3:Production Engineer");
			System.out.println("4:Production Manager");
			System.out.println("5:Worker");
			System.out.println("6:Quit");
			System.out.println("Enter your option=");
			option=Integer.parseInt(br.readLine());

			switch(option)
			{
				case 1:	e1.readData();
					e1.payCalc();
					e1.display();
					break;

				case 2:	m1.readData();
					m1.payCalc();
					m1.display();
					break;

				case 3:	e2.readData();
					e2.payCalc();
					e2.display();
					break;

				case 4:	m2.readData();
					m2.payCalc();
					m2.display();
					break;

				case 5:	w1.readData();
					w1.readOvertime();
					w1.payCalc();
					w1.display();
					break;

			}
		}
		while(option!=6);

	}
}
```

Module : 7

Prepared by Nitesh Karsi

7.1 Interfaces –

- Using the keyword **interface** you can fully abstract interface of a class from its implementation. Thus, using **interface** you can specify what a class must do, but not how it does it.

- Interfaces are syntactically similar to classes. However, they lack instace variables. The methods are declared within interfaces without any body.

- One **interface** can be implemented by many classes. Moreover, one class can implement any number of interfaces.

- To implement an interface, a class must create the complete set of methods defined by that interface. Each class is free to determine details of its own implementation for all the methods declared within interface. Using the keyword **interface** the Java allows you to fully utilize the concept of, **"one interface multiple methods"** aspect of polymorphism.

- Interfaces are designed to support **dynamic method resolution** at run time. The classes from their own hierarchy with the help of inheritance whereas interfaces are in differnet hierarchy than that of the clases. Hence, it is possible for classes that are unrelated in terms of class hierarchy to implement the same interface. This is where the real power of interfaces is realized.

- Interfaces add most of the functionality that is required for many applications that would normally require use of multiple inheritance. Thus, interfaces can act as an alternative to the multiple inheritance.

7.2 Defining Interfaces –

An interface is defined much like a class. The general form of declaring an interface is as follows.

```
access interface name
{
        return-type method-name1(parameter list);
        return-type method-name2(parameter list);
        .....
        .....
        return-type method-nameN(parameter list);
        type final-varname1 = value;
        type final-varname2 = value;
        .....
        .....
        type final-varnameN = value;
}
```

- When the acess specifier is not mentioned then the **default** access is resulted which means that the interface is available only within the package in which it is declared.
- When the access specifier is **public** then the interface can be used by any other code from any package. In this case the interface must be the only public interface declared in the file and the file must have the same name as that of the interface name.
- The declared methods are not having any body. These methods are essentially abstract methods.
- Each class that implements the interface must implement **all the methods** declared within an interface.
- Variables can be declared inside the interface declarations. These variables are implicitly, final, static and public. These variables must be initialized. These variables can not be changed in the classes which are implementing interfaces. It means that interface variables are nothing but constants.

7.3 Implementation of Interfaces –

- Once an interface has been defined then one or more classes can implement that interface. To implement the interfaces, the general form of the class definition is given as follows.

```
access class classname [extends superclass] [implements interface[,interface....]]
{
        body of class
}
```

- If the class implements more than one interface, then the interface names are separated with commas. If a class implements two interfaces that declare method with same name then class will have only one implementation of that method name and that itself will be used by clients of both interfaces.
- The methods that implement an interface must be public. The signature of the implementing method and the signature of the interface definition must match exactly.

Consider the following diagram. Give the declaration of each interface and each class in the diagram.

Consider the following example.

```
interface show
{
      void display();
}
class Date implements show
{
      int dd,mm,yy;
      Date(int d,int m,int y)
      {
            dd=d;
            mm=m;
            yy=y;
      }
      public void display()
      {
            System.out.println("Date=" + dd + '/' + mm + '/' + yy);
      }
}
class Time implements show
{
      int hour,min,sec;
      Time(int h,int m,int s)
      {
            hour=h;
            min=m;
            sec=s;
      }
      public void display()
      {
            System.out.println("Time=" + hour + ':' + min + ':' + sec);
      }
}
class Identity implements show
{
      String name;
      Identity(String s)
      {
            name=s;
      }
      public void display()
      {
            System.out.println("Name=" + name);
      }
}
```

```
class NDTDemo
{
      public static void main(String args[])
      {
            show ref=null;
            Date d1=new Date(10,3,2009);
            Time t1=new Time(11,45,30);
            Identity i1=new Identity("Amit");

            for(int i=1;i<=3;i++)
            {
                  switch(i)
                  {
                        case 1:ref=i1;break;
                        case 2:ref=d1;break;
                        case 3:ref=t1;break;
                  }
                  ref.display();
            }
      }
}
```

Explanation –

- In the above program, we have defined an interface called as show with declaration of one method called as display( ). The class Date, Time and Identity implements this interface show and all the three classes are having their own implementation for display().
- The ref is the reference variable of types inteface show. It can refer to objects of type Date, Time and Identity using statements,

  ref = d1;
  ref = t1;
  ref = i1;

  This is possible because the classes Date, Time and Identity implements the interface called as show.

- The call ref.display( ) is determined at run time depending on the type of object to which ref is referring. The single call ref.display( ) will give us different outputs depending on, the **ref** refers to which type of object and thus the **run time polymorphism** is implemented.
- Thus, using the interface names as show, we have implemented run time polymorphism for the method display( ) for the three unrelated classes called as Identity, Date and Time.

Consider the following example.

```
import java.io.*;

interface methods
{
      double PI=3.14159;
      double area();
}

class Rectangle implements methods
{
      double length,breadth;

      Rectangle(double l,doubel b)
      {
            length=l;
            breadth=b;
      }

      public double area()
      {
            return length*breadth;
      }
}

class Ellipse implements methods
{
      double rx,ry;

      Ellipse(double a,double b)
      {
            rx=a;
            ry=b;
      }
      public double area()
      {
            return PI*rx*ry;
      }
}
```

```
class InterfaceDemo
{
      public static void main(String args[]) throws IOException
      {
            methods ref=null;
            int option;
            BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
            System.out.println("Enter one integer=");
            option=Integer.parseInt(br.readLine());
            switch(option)
            {
                  case 1 :   ref=new Rectangle(4,5);
                             System.out.println("Rectangle");
                             break;
                  case 2 :   ref=new Ellipse(2,2.5);
                             System.out.println("Ellipse");
                             break;
            }
            System.out.println("Area=" + ref.area());
      }
}
```

Explanation –

- The above program defines interface named as methods. The interface declares one method area( ). It declares variable PI and initialize with value 3.14159 (interface variable is nothing but a constant). This two classes named as **Rectangle** and **Ellipse** implements the method called as area( ) as public method. This method is having its own implementation within both the classes.
- The ref is the reference variable of type interface methods. It can refer the objects of type Rectangle and Ellipse which are implementing the **interface methods.** Hence the following statement are valid.

  ref=new Rectangle(4,5);
  ref=new Ellipse(2,2.5);

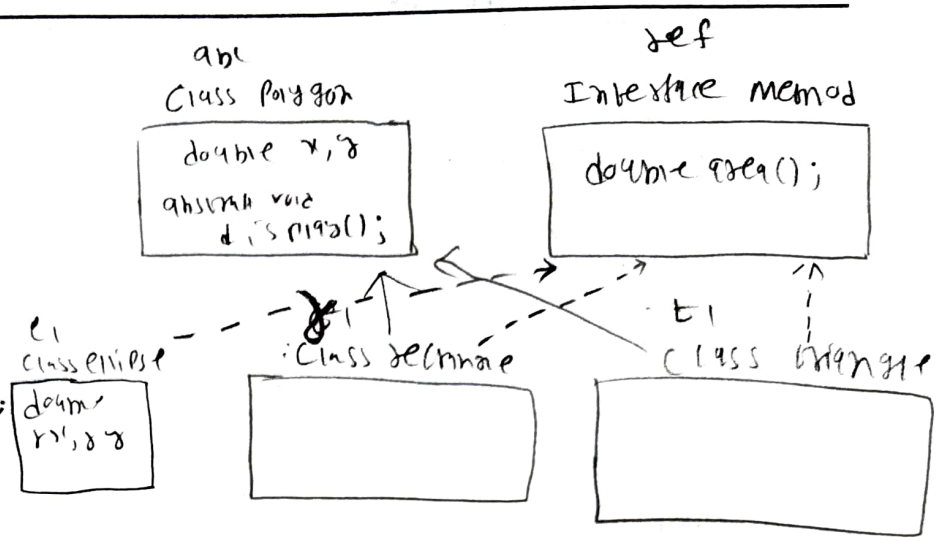
- For the statement, ref.area( ), which implementation of area( ) will be executed is decided at run time depending on the value of option. The value of the option decides that interfce reference variable **ref** refers to which object i.e. object of class Rectangle or object of type Ellipse.
- Thus, using the interface methods, we have implemented runt itme polymorphism for the method area( ) for the two unrelated classes called as Rectangle or Ellipse.

7.4 Combination of Hierarchical Inheritance and Interfaces –

The hierarchy of classes and the hierarchy of interfaces are independent of each other and these hierarchies can be made associated with each other according to the need of the application. Consider the following example.

```
import java.io.*;
interface methods
{
      double area();
}
abstract class Polygon
{
      double x,y;
      abstract void display();
      polygon(double x1,double y1)
      {
            x=x1;
            y=y1;
      }
}
class Rectangle extends Polygon implements methods
{
      Rectangle(double l,double b)
      {
            super(l,b);
      }
      public double area()
      {
            return x*y;
      }
      void display()
      {
            System.out.println("Rectangle has 4 sides");
      }
}
class Triangle extends Polygon implements methods
{
      Triangle(double b,double h)
      {
            super(b,h);
      }
      public double area()
      {
            return 0.5*x*y;
      }
      void display()
      {
            System.out.println("Triangle has 3 sides");
      }
}
```

```
class Ellipse implements methods
{
        double rx,ry;
        Ellipse(double a,double b)
        {
                rx=a;
                ry=b;
        }
        public double area()
        {
                return 3.14159*rx*ry;
        }
}
class InterfaceDemo1
{
       public static void main(String arsg[]) throws IOException
       {
              Polygon abc=null;
              methods ref=null;
              Rectangle r1=null;
              Triangle t1=null;
              Ellipse e1=null;
              int option;
              BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
              System.out.print("Enter one integer=");
              option=Integer.parseInt(br.readLine());
              switch(option)
              {
                   case 1 :  r1=new rectangle(4,5);
                             abc=r1;
                             ref=r1;
                             abc.display();
                             System.out.println("Rectangle");
                             break;
                  case 2 :   t1=new triangle(4,5);
                             abc=t1;
                             ref=t1;
                             abc.display();
                             System.out.println("Triangle");
                             break;
                  case 3 :   e1=new ellipse(2,2.5);
                             ref=e1;
                             System.out.println("Ellipse");
                             break;
              }
              System.out.println("Area=" + ref.area());
       }
}
```



Explanation –

- In the above program, we define an interface named as methods. The interface dclares the method named as area( ).
- The program also defines hierarchical inheritance defined with abstract super class Polygon and two sub classes named as Rectangle and Triangle and one independent class named as Ellipse.
- The run time polymorphism is obtained with the method display( ) with the help of abstract method display( ) declared within the super abstract class Polygon. The display( ) is implemented diffrently within the class Rectangle and class Triangle. The reference variable **abc** of type polygon is used to implement the run time polymorphism with the method display( ).
- The run time polymorphism is obtained with area( ) with the help of the interface called as **methods.** The implementation of area( ) is different in three classes named as Rectangle, Triangle and Ellipse. Thus, even if the class Ellipse is **unrelated** to hierarchy defined with Polygon, Rectangle and Triangle still the run time Polymorphism can be obtained with the method area( ) with the help of interface named as methods and the method area( ) declared withint it. This is the real power of interfaces.

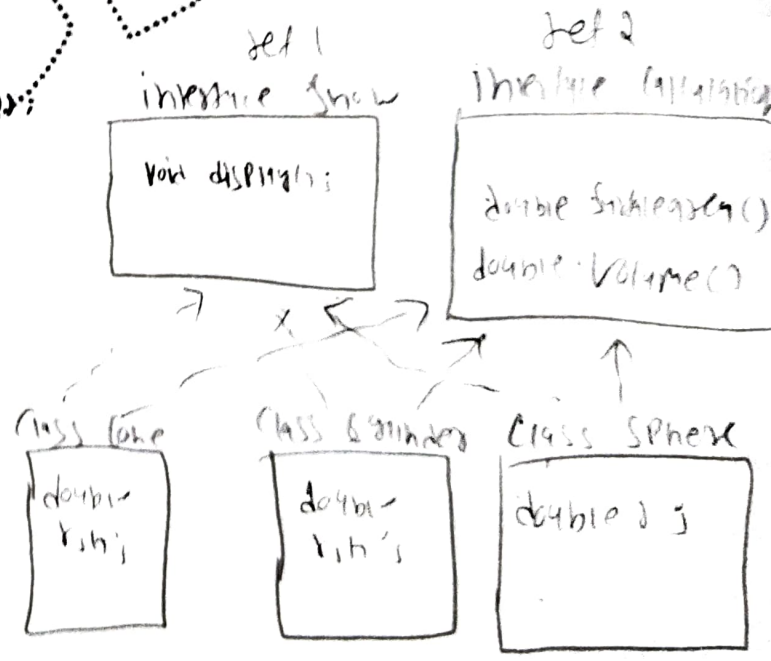
7.5 Classes Implementing Multiple Inherfaces –

- When the classes implement multiple interfaces then the concept similar to multiple inheritance of C++ can be implemented.
- One class can implement more than one interface. Consider the following example.

```
interface show
{
    void display();
}
interface calculations
{
    double PI = 3.14159;
    double surfaceArea();
    double volume();
}
```

```
class Cylinder implements show,calculations
{
    double r,h;
    Cylinder(double x,double y)
    {
        r=x;
        h=y;
    }
    public double surfaceArea()
    {
        return (2*PI*r*h + 2*PI*r*r);
    }
    public double volume()
    {
        return PI*r*r*h;
    }
    public void display()
    {
        System.out.println("Cylinder");
        System.out.println("Radius=" + r);
        System.out.println("Height=" + h);
    }
}
class Sphere implements show,calculations
{
    double r;
    Sphere(double x)
    {
        r=x;
    }
    public double surfaceArea()
    {
        return (4*PI*r*r);
    }
    public double volume()
    {
        return 4.0/3*PI*r*r*r;
    }
    public void display()
    {
        System.out.println("Sphere");
        System.out.println("Radius=" + r);
    }
}
```

```
class Cone implements show,calculations
{
    doubel r,h;
    Cone(double x,double y)
    {
        r=x;
        h=y;
    }
    public double surfaceArea()
    {
        return (PI*r*r + PI*r*Math.sqrt(r*r+h*h));
    }
    public double volume()
    {
        return 1.0/3*PI*r*r*h;
    }
    public void display()
    {
        System.out.println("Cone");
        System.out.println("Radius=" + r);
        System.out.println("Height=" + h);
    }
}
class InterfaceDemo2
{
    public static void main(String args[])
    {
        show ref1=null;
        calculations ref2=null;
        cylinder obj1 = new cylinder(10,20);
        sphere obj2 = new sphere(10);
        cone obj3 = new cone(10,20);
        for(int i=1;i<=3;i++)
        {
            switch(i)
            {
                case 1:  ref1=obj1;
                         ref2=obj1;
                         break;
                case 2:  ref1=obj2;
                         ref2=obj2;
                         break;
                case 3:  ref1=obj3;
                         ref2=obj3;
                         break;
            }
            ref1.display();
            System.out.println("Surface area=" + ref2.surfaceArea());
            System.out.println("Volume=" + ref2.volume() + '\n');
        }
    }
}
```



Output –

```
Cylinder
Radius=10.0
Height=20.0
Surface area=1884.954
Volume=6283.18

Sphere
Radius=10.0
Surface area=1256.636
Volume=4188.786666666666

Cone
Radius=10.0
Height=20.0
Surface area=1016.6398797433565
Volume=2094.393333333333
```

Explanation –

- The above program use two interfaces named as show and calculations. The interface show declares one method called as display( ). The intereface calculations declares two methods named as area( ) and volume( ).
- The program uses three independent classes named as cylinder, sphere and cone. All three classes implements both interfaces show and calculations i.e. it implements display( ), surface_area( ) and volume( ).
- Since the display( ) of interface show and surface_area( ) and volume( ) of interface calculations are all available to the three classes cylinder, sphere and cone the concept similar to the multiple inheritance is obtained.

7.6 Extending the Interfaces –

The way we can extend the clsses, it is also possible to extend the interfaces. Consider the following example.

```
interface counting
{
      void increment();
}
interface moreCounting extends counting
{
      void decrement();
}
```

```
interface show
{
      void display();
}
class Counter implements moreCounting,show
{
      int count;
      Counter(int c)
      {
            count=c;
      }
      public void increment()
      {
            count++;
      }
      public void decrement()
      {
            count--;
      }
      public void display()
      {
            System.out.println(count);
      }
}
class Time implements counting,show
{
      int hour,min;
      Time(int h,int m)
      {
            hour=h;
            min=m;
      }
      public void increment()
      {
            min++;
            if(min>=60)
            {
                  min=0;
                  hour++;
            }
            if(hour>=24) hour=min=0;
      }
      public void display()
      {
            System.out.println(hour + ":" + min);
      }
}
```

```
class Demo                                                    multi-
{
      public static void main(String args[])
      {
            Counter c1=new Counter(10);
            Time t1=new Time(23,58);

            c1.display();
            c1.increment();
            c1.display();
            c1.decrement();
            c1.display();

            t1.display();
            t1.increment();
            t1.display();
            t1.increment();
            t1.display();

            counting ref1=null;
            moreCounting ref2=null;
            show ref3=null;

            Counter c2=new Counter(25);
            time t2=new time(12,59);
            ref2=c2;

            ref3=c2;
            ref3.display();
            ref2.increment();
            ref3.display();
            ref2.decrement();
            ref3.display();

            ref1=t2;
            ref3=t2;
            ref3.display();
            ref1.increment();
            ref3.display();
            ref1.increment();
            ref3.display();
      }
}
```

Explanation –

- One interface can inherit another by use of the keyword extends. The syntax is similar to the inheritance with classes. When a class implements the interface which is inherited from the other interface then the class must implement all the methods defined within the interface inheritance chain.
- In the above program, interfaces counting and moreCounting forms inheritance. The program defines one more independent interface called as shsow.
- The class Counter implements the interfaces named as moreCounting and show. The class Time implements the interfaces named as counting and show.

7.7 Difference between Abstract Class and Interface –

	Abstract Class		Interface
1.	The abstract class must have at least one abstract method. It can have two or more abstract method also. However, all the methods within the abstract class need not be abstract methods.	1.	All the methods declared within the interface are compulsorily abstrct i.e. without any implementation.
2.	The abstract methods of abstract class must be implemented by every sub class of the abstract class.	2.	All the methods declared within the interface must be implemented by the classes implementing that interface.
3.	The abstract methods need not be implemented as public method.	3.	The interface methods must be implemented as public method.
4.	The abstract classes are useful to implement run time polymorphism with the classes related to each other by hierarchical inheritance.	4.	The interfaces are useful to implement the run time polymorphism with the classes which are unrelated to each other in terms of hierarchy.
5.	One sub class can be inherited only from one abstract class.	5.	The class can implement one or more interfaces.
6.	The keyword **abstract** and **class** are used in the header of abstract class.	6.	The keyword **interface** is used in the header of interface.