UNIT – II

CLASS FUNDAMENTALS AND INHERITANCE

Objective:

- Develop the code with the concepts of Class and Inheritance.

Syllabus:

Class Fundamentals, Declaring Objects, Methods, Constructors, This Keyword, Overloading Methods and Constructors, Access Control.

Inheritance- Basics, Types, Using Super Keyword, Method Overriding, Dynamic Method Dispatch, Abstract Classes, Using Final With Inheritance, Object Class.

Learning Outcomes:

Students will be able to

- Describe how classes, objects, methods and Constructors are created and applied in java.
- Apply different types of Inheritance and can develop simple programs using Inheritance.
- Differentiate between Method overloading and Method Overriding
- Differentiate between Abstract methods and Concrete methods.
- Demonstrate the importance of this, super and final keywords, and will be able to distinguish between them.

LEARNING MATERIAL

➢ CLASS FUNDAMENTALS:

- A class is a blueprint or prototype that defines the variables and methods common to all objects of same kind. A class can be defined as a user-defined data type and an object as a variable of that data type that can contain data and methods that manipulates the data.
- **Ex:**

Bike
boolean kickstart boolean buttonstart int gears

accelerate()
applyBrake()
changeGear()

Fig. Bike class

Manufacturers produce many bikes from the same blueprint as every bike share similar characteristics. There are many objects of same kind belonging to same classes that share certain characteristics. Bikes have attributes (speed, engine capacity, number of wheels, number of gears, brakes) behaviors (braking, accelerating, slowing down and changing gears).

- A class is a ***template*** for an object, and an object is an ***instance*** of a class. Because an object is an instance of a class, you will often see the two words *object* and *instance* used interchangeably.
- A class is declared by use of the **class** keyword.
- **Declaration of class:**

```
class classname
  {
    type instance-variable1;
    type instance-variable2;
    // ...
    type instance-variableN;
    type methodname1(parameter-list)
     {
           // body of method
     }
     type methodname2(parameter-list)
     {
           // body of method
```

```
        }
        // ...
        type methodnameN(parameter-list) {
                // body of method
        }
}
```

- The data, or variables, defined within a class are called *instance variables.*
- The code is contained within *methods.* Collectively, the methods and variables defined within a class are called *members* of the class.
- The instance variables are directly accessible by methods defined in the class.
- Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables.
- The data for one object is separate and unique from the data for another.

Example of class:

- Create a class structure that may represent the structure of a hospital or other medical organization, begin with a class called **Employee**.
- An employee has several characteristics that you can represent as **variables**, such as name, salary, and sickDays.
- Write a method **details()** which consists of three println() statements that print the values of instance variables. The Employee class can be defined as follows:

```
class Employee
{
        String name;     // Instance Variables
            int salary;       // Instance Variables
            void details()    // Instance Method
            {
                    System.out.println("Name: " + name);
                    System.out.println("Salary: " + salary);
            }
}
```

➢ <u>**CREATING OBJECTS:**</u>

- Object is an instance of a class.
- An object is created by creating an instance of a class. The type of the object is class itself.
- Creating an object for a class is a two-step process.
    - First, **declare** a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can *refer* to an object.
    - Second, acquire an actual, physical copy of the object and assign it to that variable by using the **new** operator.
- The new operator dynamically allocates memory for an object and returns a reference to it. This reference is nothing but the address in memory of the object allocated by new. This reference is then stored in the variable declared.

<u>**Syntax for creating an Object**</u>**:**

```
Classname objectname=new Classname();
```

<u>**Example:**</u>

```
Employee sam=new Employee();
```

This statement combines the two steps just described. It can be rewritten like this to show each step more clearly:

```
Employee sam;              // declare reference to object
sam=new Employee();        // allocate a Employee object
```

- The first line declares *sam* as a reference to an object of type Employee.
- After this line executes, sam contains the value null, which indicates that it does not yet point to an actual object.
- Any attempt to use sam at this point will result in a compile-time error. The next line allocates an actual object and assigns a reference to it to sam.

- After the second line executes, you can use sam as if it were an Employee object. But in reality, sam simply holds the memory address of the actual Employee object.
- The effect of these two lines of code is depicted in Figure :

Statement **Effect**

Employee sam;

sam= new Employees();

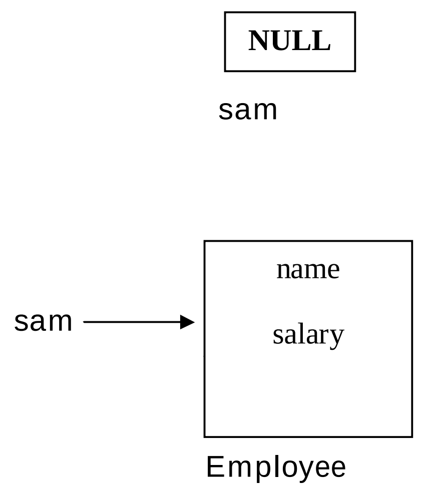


Fig 2.1: Declaring a Object type sam

Example creating an object and accessing class members via an object:

```
class Employee
{
          String name; // person's name          // Instance Variables
          double salary; // salary in dollars
          void details() // Instance Method
          {
                System.out.println("Name: " + name);
                System.out.println("Salary: " + salary);
          }
    }
  class Demo
  {
          public static void main(String[] args)
          {
            Employee ram = new Employee();
                // may be done on two lines.
                //Employee ram;     //Object declaration
                //ram = new Employee();   // Instantiation
                ram.name = "Ram"; // initialization
                ram.salary = 32000;
                // Now print out ram information using details()
                ram.details();
            }
    }
```

The output produced by this program is shown here:

```
Name:Ram
Salary:32000
```

new Operator:

- The new operator dynamically allocates memory for an object. The general form is:

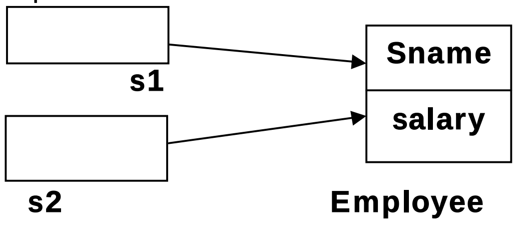
  class-var = new *classname*( );

- *class-var* is a variable of the class type being created.
- The *classname* is the name of the class that is being instantiated. The class name followed by parentheses specifies the *constructor* for the class.
- A constructor defines what occurs when an object of a class is created.
- Most real-world classes explicitly define their own constructors within their class definition. However, if no explicit constructor is specified, then Java will automatically supply a default constructor.
- In example, JVM will initialize the instance variables using a default constructor.
- Java’s primitive types are not implemented as objects and so new operator is not required for them.
- new allocates memory for an object during run time, so can create as many or as few objects as needed during the execution.

Assigning Object Reference Variables:

```
Employee s1=new Employee();
Employee s2=s1;
```

- s2 is assigned a reference to a copy of the object referred to by s1.
  s1 and s2 will both refer to the *same* object.
- With this assignment, s2 refers to the same object as s1. Thus, any changes made to the object through s2 will affect the object to which s1 is referring, since they are the same object.
  This situation is depicted here:



- A subsequent assignment to s1 will simply unhook s1 from the original object without affecting the object or affecting s2.
  For example:

```
        Employee s1=new Employee();
        Employee s2=s1;
      // ...
        s1 = null;
```

Here, s1 has been set to null, but s2 still points to the original object.

➢ <u>METHODS:</u>

- Classes usually consist of two things: instance variables and methods.

<u>General form of a method:</u>

type *name*(*parameter-list*) {

```
                // body of method
    }
```

- *type* specifies the type of data returned by the method. This can be any valid type, including class types that we create.
- If the method does not return a value, its return type must be **void**.
- The name of the method is specified by *name.* This can be any legal identifier other than those already used by other items within the current scope.
- The *parameter-list* is a sequence of type and identifier pairs separated by commas.

- Parameters are essentially variables that receive the value of the *arguments* passed to the method when it is called.
- If the method has no parameters, then the parameter list will be empty.
- Methods that have a return type other than void, return a value to the calling routine using the following form of the return statement:

  return *value*;

Here, *value* is the value returned.

➢ CONSTRUCTORS:

- Whenever an object is created for a class, the instance variables of the class needs to be given initial values.
- Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor.
- A *constructor* is a special method which initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method.
- When a constructor is not defined for a class, Java compiler provides a default constructor automatically initializes all instance variables to their default values.
- The constructor is automatically invoked as soon as the object is instantiated with the new keyword.
- Constructors have no return type, not even void. This is because the implicit return type of a class' constructor is the class type itself.
- If any constructor is defined in the class then the JVM will not provide any constructor.
- Example, a simple constructor that simply sets the dimensions of each Box to the same values.

```
/* Here, Box uses a constructor to initialize the dimensions of a box.
class Box {
    double width;
    double height;
    double depth;
```

```
        Box() {                          // This is the constructor for Box.
            System.out.println("Constructing Box");
            width = 10;
            height = 10;
            depth = 10;
        }
    double volume() {          // compute and return volume
            return width * height * depth;
        }
}
class BoxDemo6 {
        public static void main(String args[]) {
        // declare, allocate, and initialize Box objects
            Box mybox1 = new Box();
            Box mybox2 = new Box();
            double vol;
            vol = mybox1.volume();               // get volume of first box
            System.out.println("Volume is " + vol);
            vol = mybox2.volume();               // get volume of second box
            System.out.println("Volume is " + vol);
        } }
```

When this program is run, it generates the following results:

```
Constructing Box
Constructing Box
Volume is 1000.0
Volume is 1000.0
```

They initialize the details of employee to both mybox1and mybox2**.**

<u>Parameterized Constructors:</u>

- The Box( ) constructor in the preceding example initializes all boxes with the same dimensions.
- Parameters can be passed to a constructor, similar to having parameters for a method. This makes them much more useful.

Example:

```
// Here, Box uses a parameterized constructor to initialize the dimensions
   of a box.
class Box {
        double width;
```

```
            double height;
            double depth;
            Box(double w, double h, double d)//This is the constructor for Box
            {
                width = w;
                height = h;
                depth = d;
           }
        double volume() {                              // compute and return volume
                return width * height * depth;
          }
}
class BoxDemo7 {
            public static void main(String args[]) {
                 // declare, allocate, and initialize Box objects
                Box mybox1 = new Box(10, 20, 15);
                Box mybox2 = new Box(3, 6, 9);
                double vol;
                vol = mybox1.volume();  //// get volume of first box
                System.out.println("Volume is " + vol);
                vol = mybox2.volume();  // get volume of second box
                System.out.println("Volume is " + vol);
            }
}
```

The output from this program is shown here:

```
Volume is 3000.0
Volume is 162.0
```

- Each object is initialized with values specified in the parameters to its constructor.
  For example, in the following line,
  `Box mybox1 = new Box(10, 20, 15);`
- The values 10, 20, and 15 are passed to the **Box( )** constructor when **new** creates the object.Thus, mybox1's copy of width, height, and depth will contain the values 10, 20, and 15,respectively.

➢ **'this' KEYWORD:**

- Java defines the '**this'** keyword. **this** can be used inside any instance method to refer to the *current* object.
- **this** is always a reference to the object on which the method was invoked.
- Example:

```
Box(double w, double h, double d) {  // A redundant use of this.
        this.width = w;
        this.height = h;
        this.depth = d;
 }
```

Instance Variable Hiding

- It is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes.
- But the names of local variables, including formal parameters to methods, may overlap with the names of the class' instance variables.
- So, when a local variable has the same name as an instance variable, the local variable *hides* the instance variable.
- This is why width, height, and depth were not used as the names of the parameters to the Box( ) constructor inside the Box class.
- **Example:**

Here is another version of Box( ), which uses width, height, and depth for parameter names and then uses 'this' to access the instance variables by the same name:

```
// Use this to resolve name-space collisions.
     Box(double width, double height, double depth) {
       this.width = width;
       this.height = height;
       this.depth = depth;
     }
```

- 'this' can be used for constructor chaining, means a constructor can be called from another constructor.

```
/* First Constructor */
Box()
{
    //constructor chained
   this(14,12,10);
}
/*   Second Constructor  */
Box(double w, double h, double depth) {
            width = w;
            height = h;
            depth = d;
      }
```

OVERLOADING METHODS AND CONSTRUCTORS:

- Method overloading is one way of achieving polymorphism in java.
- Each method in a class is uniquely identified by its name and parameter list, means two or more methods with same name, but with a different parameter list. This feature called as method overloading.
- Overloaded methods must differ in the type and/or number of their parameters.
- While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method.
- When Java encounters a call to an overloaded method, it simply executes the version of the method whose number and type of parameters match the arguments used in the call.

```
// Demonstrate method overloading.
class OverloadDemo
{
        void test()
```

```
    {
            System.out.println("No parameters");
        }
    void test(int a)
    {                         // Overload test for one integer parameter.
            System.out.println("a: " + a);
}
void test(int a, int b)
{                    // Overload test for two integer parameters.
            System.out.println("a and b: " + a + " " + b);
}
double test(double a)
{                            // overload test for a double parameter

            System.out.println("double a: " + a);
            return a*a;
}

 }

    class Overload

        {

    public static void main(String args[])
    {
            OverloadDemo ob = new OverloadDemo();
            double result;
            // call all versions of test()
            ob.test();
            ob.test(10);
            ob.test(10, 20);
            result = ob.test(123.25);
            System.out.println("Result of ob.test(123.25): " + result);
    }
```

```
}
```

output:

```
No parameters
a: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625
```

- **test( )** is overloaded four times. The first version takes no parameters,the second takes one integer parameter, the third takes two integer parameters, and the fourth takes one double parameter.
- The fact that the fourth version of test( ) also returns a value is of no consequence relative to overloading, since return types do not play a role in overload resolution.
- overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters.
- However, this match need not always be exact. In some cases, Java's automatic type conversions can play a role in overload resolution.

Example:

```
            // Automatic type conversions apply to overloading.
class OverloadDemo
{
    void test()
    {
                System.out.println("No parameters");
            }
            void test(int a, int b)
            {     // Overload test for two integer parameters.
                System.out.println("a and b: " + a + " " + b);
            }
            void test(double a)
            {    // overload test for a double parameter
                System.out.println("Inside test(double) a: " + a);
            }
}
class Overload
{
```

```
            public static void main(String args[])
            {
                OverloadDemo ob = new OverloadDemo();
                int i = 88;
                ob.test();
                ob.test(10, 20);
                ob.test(i); // this will invoke test(double)
                ob.test(123.2); // this will invoke test(double)
            }
}
```

output:

```
            No parameters
            a and b: 10 20
            Inside test(double) a: 88
            Inside test(double) a: 123.2
```

- *OverloadDemo* does not define **test(int)**. So java will automatically convert int to **double**.

OVERLOADING CONSTRUCTORS:

- Similar to methods constructor can also be overloaded.
- Constructors for a class having the same name as that of class, but with different signatures I.e., different number of arguments or different types of arguments.

Example:

```
class Box
{
      double width;
      double height;
      double depth;
      Box(double w, double h, double d)
      {// This is the constructor for Box.
            width = w;
            height = h;
            depth = d;
        }
        double volume()
        {// compute and return volume
            return width * height * depth;
        } }
```

- The Box( ) constructor requires three parameters. This means that all declarations of Box objects must pass three arguments to the Box( ) constructor.
- For example,the following statement is currently invalid.

  Box ob = new Box();

  because Box( ) requires three arguments.

// Here, Box defines three constructors to initialize the dimensions of a box various ways.

```
class Box
{
            double width;
            double height;
            double depth;
        Box(double w, double h, double d)
        {   //constructor used when all dimensions specified
                width = w;
                height = h;
                depth = d;
            }
            Box()
            {       // constructor used when no dimensions specified
                width = -1;         // use -1 to indicate
                height = -1;        // an uninitialized
                depth = -1;         // box
            }
            Box(double len)
            {       // constructor used when cube is created
```

```
                        width = height = depth = len;
                }
                double volume()
                {// compute and return volume
                        return width * height * depth;
                }
}
class OverloadCons
{
    public static void main(String args[])
    {// create boxes using the various constructors
                Box mybox1 = new Box(10, 20, 15);
                Box mybox2 = new Box();
                Box mycube = new Box(7);
                double vol;
                vol = mybox1.volume();      // get volume of first box
            System.out.println("Volume of mybox1 is " + vol);
                vol = mybox2.volume();       // get volume of second box
                System.out.println("Volume of mybox2 is " + vol);
                vol = mycube.volume();        // get volume of cube
                System.out.println("Volume of mycube is " + vol);
                }
        }
```

output:

```
Volume of mybox1 is 3000.0
Volume of mybox2 is -1.0.
Volume of mycube is 343.0
```

- The appropriate overloaded constructor is called based upon the parameters specified when **new** is executed.

➢ ACCESS CONTROL:

- Encapsulation links data with the code that manipulates it. Encapsulation provides another important attribute: ***access control***.
- Through encapsulation, we can control what parts of a program, can access the members of a class.
- How a member can be accessed is determined by the ***access specifier*** that modifies its declaration. *Java supplies a rich set of access specifiers*. Some aspects of access control are related mostly to inheritance or packages.
- Java's access specifiers are **public**, **private**, and **protected**. Java also defines a default access level.
- **protected** applies only when inheritance is involved.
- A member of a class is modified by the **public** specifier, then that member can be accessed by any other code.
- When a member of a class is specified as **private**, then that member can only be accessed by other members of its class.
- No access specifier is used, then by **default** the member of a class is **public** within its own package, but cannot be accessed outside of its package.

```
// This program demonstrates the difference betweenpublic and private.
class Test
{
        int a;                  // default access
        public int b;           // public access
        private int c;   // private access
        // methods to access c
        void setc(int i)
        {   // set c's value
                c = i;
        }
        int getc()
        {   // get c's value
                return c;
}}
class AccessTest
{
        public static void main(String args[])
        {
                Test ob = new Test();
                // These are OK, a and b may be accessed directly
                ob.a = 10;
                ob.b = 20;
                // This is not OK and will cause an error
                // ob.c = 100; // Error!
                // You must access c through its methods
                ob.setc(100); // OK
                System.out.println("a, b, and c: " + ob.a + " " +
                ob.b + " " + ob.getc());
}}
```

- Inside the **Test** class, **a** uses default access, which for this example is the same as specifying **public**. **b** is explicitly specified as **public**.
- Member **c** is given private access. This means that it cannot be accessed by code outside of its class.
- So, inside the **AccessTest** class, **c** cannot be used directly. It must be accessed through its public methods: **setc( )** and **getc( )**.

INHERITANCE

BASICS:

- Inheritance is the process by which one class acquires the properties (instance variables, methods) of another class.
- A deeply inherited subclass (descendent) inherits all of the properties from each of its ancestors in the class hierarchy.
- Inheritance should create an *is-a relationship*, meaning the child *is a* more specific version of the parent.

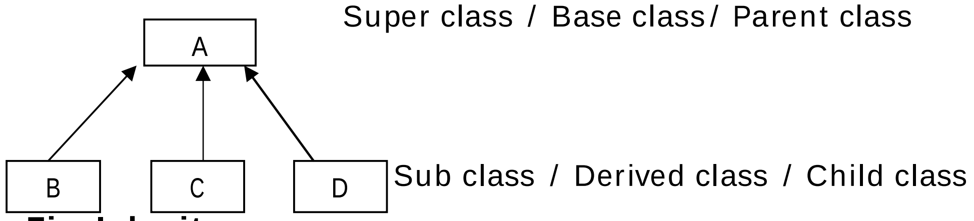

Super class / Base class/ Parent class
A
B
C
D
Sub class / Derived class / Child class

Fig: Inheritance

Main purpose of Inheritance:

1. Reusability.
2. Abstraction.

- **“**extends” keyword is used to inherit the properties from one class to another class**.**

Member Access

- Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as **private** and accessing restrictions are as follows,
  - public: variable/method can be accessed anywhere
  - private: variable/method can be accessed only within this class (but NOT within subclasses)
  - protected: variable/method can be accessed:
    – Within this class.
    – Within any class/subclass in the same *package.*
    – Within any subclass of this class in other package.

  Note: if you do not include an access specifier (default), the Variable or method has *package access.*

➢ TYPES OF INHERITANCE:

There are 5 types of Inheritance

- Single Inheritance.
- Multilevel Inheritance.
- Hierarchical Inheritance.
- Multiple Inheritance
  - JAVA does not support, need to use Interface. 'extends' can be used with only one class.
- Hybrid Inheritance

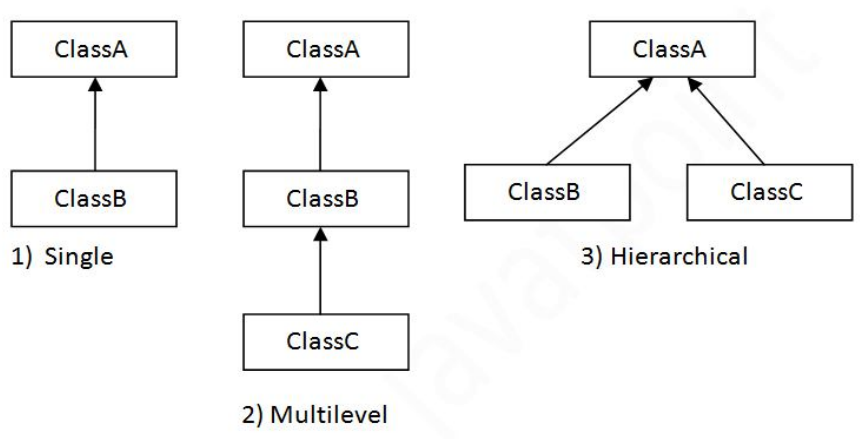


Fig: Types of Inheritance

- <u>**Program for Single Inheritance:**</u>

```
/* simple inheritance */
import java.io.*;
class A
 {
    public int i;
    public A()
     {
        System.out.println("\n \t default constructor A() is called");
        i=10;
     }
    public void Adisplay()
     {
        System.out.println("\n \t in A class i= "+i);
     }
 }
class B extends A
 {
    public int j;
    public B()
     {
        System.out.println("\n \t default constructor B() is called");
        j=20;
     }
    public void Bdisplay()
     {
        j=i+1;
        System.out.println("\n \t in B class j= "+j);
     }
 }
class Simple
 {
     public static void main(String ar[])throws IOException
      {
         System.out.println("\n \t start of main()");

         B b=new B();
         b.Adisplay();
         b.Bdisplay();
         System.out.println("\n \t end of main()");
      }
 }
```

OUTPUT:

```
    start of main()
    default constructor A() is called
    default constructor B() is called
    in A class i= 10
```

```
   in B class j= 11
   end of main()
```

- **Program for Multilevel Inheritance:**

```
/* multilevel inheritance */
import java.io.*;
class A
 {
   public int i;
   public A()
    {
      System.out.println("\n \t default constructor A() is called");
      i=10;
    }
   public void Adisplay()
    {
      System.out.println("\n \t in A class i= "+i);
    }
 }
class B extends A
 {
   public int j;
   public B()
    {
      System.out.println("\n \t default constructor B() is called");
      j=20;
    }
   public void Bdisplay()
    {
      j=i+1;
      System.out.println("\n \t in B class j= "+j);
    }
 }
class C extends B
 {
    public int k;
    public C()
     {
        System.out.println("\n \t default constructor C() is called");
        k=30;
     }
    public void Cdisplay()
     {
        k=i+j;
        System.out.println("\n \t in C class k= "+k);
     }
 }
 class MulLevel
  {
     public static void main(String ar[])throws IOException
      {
         System.out.println("\n \t start of main()");

         C c=new C();
         c.Adisplay();
         c.Bdisplay();
         c.Cdisplay();
         System.out.println("\n \t end of main()");
      }   }
```

OUTPUT:

```
   start of main()
   default constructor A() is called
   default constructor B() is called
   default constructor C() is called
```

```
in A class  i= 10
in B class  j= 11
 in C class k= 21
end of main()
```

A Super class Variable Can Reference a Subclass Object

- A reference variable of a super class can be assigned a reference to any subclass derived from that super class.
- You will find this aspect of inheritance quite useful in a variety of situations . For example, consider the following:

```
// This program uses inheritance to extend Box.
class Box
 {
   double width;
   double height;
   double depth;

// construct clone of an object
   Box(Box ob)
   {
             // pass object to constructor
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
   }
// constructor used when all dimensions specified
  Box(double w, double h, double d)
  {
    width = w;
    height = h;
    depth = d;
  }
// constructor used when no dimensions specified
```

```
  Box()
  {
     width = -1; // use -1 to indicate
     height = -1; // an uninitialized
     depth = -1; // box
  }
   // constructor used when cube is created
 Box(double len)
  {
      width = height = depth = len;
  }
    // compute and return volume
 double volume()
   {
       return width * height * depth;
   }
}
// Here, Box is extended to include weight.
class BoxWeight extends Box
  {
     double weight; // weight of box
     // constructor for BoxWeight
     BoxWeight(double w, double h, double d, double m)
     {
        width = w;
        height = h;
        depth = d;
        weight = m;
     }
}
class DemoBoxWeight
 {
   public static void main(String args[])
    {
        BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
        BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
        double vol;
        vol = mybox1.volume();
         System.out.println("Volume of mybox1 is " + vol);
         System.out.println("Weight of mybox1 is " + mybox1.weight);
         System.out.println();
         vol = mybox2.volume();
         System.out.println("Volume of mybox2 is " + vol);
         System.out.println("Weight of mybox2 is " + mybox2.weight);
      }
 }
```

Output:

```
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3
Volume of mybox2 is 24.0
Weight of mybox2 is 0.076
```

- **BoxWeight** inherits all of the characteristics of **Box** and adds to them the **weight** component. It is not necessary for **BoxWeight** to re-create all of the features found in **Box**. It can simply extend **Box** to meet its own purposes.
- A major advantage of inheritance is that once you have created a superclass that defines the attributes common to a set of objects, it can be used to create any number of more specific subclasses.
- For example, the following class inherits **Box** and adds a color attribute:

```
// Here, Box is extended to include color.
class ColorBox extends Box
{
     int color; // color of box
```

```
 ColorBox(double w, double h, double d, int c)
 {
    width = w;
    height = h;
    depth = d;
    color = c;
}
  }
```

- Remember, once you have created a superclass that defines the general aspects of an object, that superclass can be inherited to form specialized classes. Each subclass simply adds its own, unique attributes. This is the essence of inheritance.

```
    class RefDemo
{
   public static void main(String args[])
    {
      BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
       Box plainbox = new Box();
       double vol;
       vol = weightbox.volume();
       System.out.println("Volume of weightbox is " + vol);
       System.out.println("Weight of weightbox is " +
       weightbox.weight);
       System.out.println();
       plainbox = weightbox;  // assign BoxWeight reference to Box
reference
       vol = plainbox.volume(); // OK, volume() defined in Box
       System.out.println("Volume of plainbox is " + vol);
/* The following statement is invalid because plainbox does not define a
    weight member. */
    // System.out.println("Weight of plainbox is " + plainbox.weight);
}
}
```

- What members can be accessed is determined based on the type of the reference variable, not on the type of the object that it refers to.
- That is, when a reference to a subclass object is assigned to a superclass reference variable, you will have access only to those parts of the object defined by the superclass.

➢ **SUPER KEYWORD:**

- 'super' is used when a subclass wants to refer to its **immediate** super class members.
- 'super' has two general forms.
  - ✓ To make a call to the super class constructor from sub class constructor.
  - ✓ The second is used to access a member of the superclass that has been hidden by a member of a subclass.

program for super keyword:

```
/* super keyword */
import java.io.*;
class A
 {
   public int i;
   public A()
    {
      System.out.println("\n \t default constructor A() is called");
      i=10;
    }
   public void display()
    {
      System.out.println("\n \t in A class i= "+i);
    }
 }
class B extends A
 {
   public int i;
```

```
    public B()
     {                                // invoking the super class constructor .
        super();                      // always must be first statement and it is default
        System.out.println("\n \t default constructor B() is called");
        this.i=20;
        super.i=30;                   //  points the super class instance variable.
     }
    public void display()
     {
        super.display();        // calling super class method.
        this.i=this.i+super.i;
        System.out.println("\n \t in B class subofi+supofi =  "+this.i);
     }
}
class Super
 {
    public static void main(String ar[])throws IOException
     {
       System.out.println("\n \t start of main()");
       B b=new B();
       b.display();
       System.out.println("\n \t end of main()");
     }
 }
```

OUTPUT:

```
        start of main()
        default constructor A() is called
        default constructor B() is called
        in A class i= 30
        in B class subofi+supofi = 50
        end of main
```

➢ **<u>METHOD OVERRIDING</u>:**

- In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass.

- When an overridden method is called within a subclass, it will always refer to the version of that method defined by the subclass. The version of ***the method defined by the superclass will be hidden***.

Program for Method Overriding:

```
/* overriding(Run-time polymorphism) */
import java.io.*;
class A
 {
   public int i;
   public A()
    {
       System.out.println("\n \t default constructor A() is called");
       i=10;
    }
   public void display()
    {
       System.out.println("\n \t in A class i= "+i);
    }
 }
class B extends A
 {
   public int j;
   public B()
    {
       System.out.println("\n \t default constructor B() is called");
       j=20;
    }
   public void display()
    {
       j=i+1;
       System.out.println("\n \t in B class j= "+j);
    }
 }
 class OverRiding
  {
    public static void main(String ar[])throws IOException
    {
       System.out.println("\n \t start of main()");
       B b=new B();
       b.display();   // display()  in B class overrides the display() in A class
       System.out.println("\n \t end of main()");
    }
  }
```

OUT PUT

```
   start of main()
   default constructor A() is called
   default constructor B() is called
    in B class j= 21
   end of main
```

➢ <u>DYNAMIC METHOD DISPATCH:</u>

- Runtime polymorphism or Dynamic Method Dispatch is a process in which a call to an overridden method is resolved at runtime rather than at compile-time.
- Using this feature, java implements Runtime Polymorphism in Java.
- "A super class reference variable can refer to a sub class object", so resolves call to a overridden method during runtime.
- When an overridden method is called through a super class reference, java determines which version of overridden method to execute based upon the type of being refereed to at the time the call occurs.
- It is the type of object being referred to at the time of call occurs, not the type of reference variable that determines which version of an overridden method will be executed.

<u>Program for Dynamic method dispatch</u>:

```
 // Dynamic Method Dispatch
 class A
          {
               void callme()
```

```
                {
                        System.out.println("Inside A's callme method");
                }
        }
class B extends A
    {
            // override callme()
           void callme()
           {
                 System.out.println("Inside B's callme method");
           }
    }
 class C extends A
    {
          // override callme()
         void callme()
         {
               System.out.println("Inside C's callme method");
         }
    }
 class Dispatch
   {
         public static void main(String args[]) {
                A a = new A();              // object of type A
                B b = new B();             // object of type B
                C c = new C();             // object of type C
                     A r;                  // obtain a reference of type A
                     r  = a;               // r refers to an A object
                     r.callme();           // calls A's version of callme
                     r = b;                // r refers to a B object
                     r.callme();           // calls B's version of callme
                     r = c;                // r refers to a C object
```

```
                    r.callme();            // calls C's version of callme
    }
}
```

Output:

```
Inside A's callme method
Inside B's callme method
Inside C's callme method
```

- This program creates one superclass called **A** and two subclasses of it, called **B** and **C**. Subclasses **B** and **C** override **callme( )** declared in **A**.
- Inside the **main( )** method, objects of type **A**, **B**, and **C** are declared. Also, a reference of type **A**, called **r**, is declared.
- The program then assigns a reference to each type of object to **r** and uses that reference to invoke **callme( )**.
- As the output shows, the version of **callme( )** executed is determined by the type of object being referred to at the time of the call, determined by the type of the reference variable, **r**, from three calls to **A**'s **callme( )** method.

- **ABSTRACT CLASS:**
  - An abstract class contain one or more abstract methods.
  - An *abstract method* is method without a body, i.e., only declared but not defined.
  - The keyword "abstract" is used to indicate a method/class as abstract ones.
  - Abstract classes cannot be instantiated.
  - Abstract methods needs to be defined in subclasses of the abstract class.

Program for abstract class:

```
  /* abstract keyword */
import java.io.*;
abstract class A
 {
   public int i;
```

```
    public A()
     {
        System.out.println("\n \t default constructor A() is called");
        i=10;
     }
    public abstract void add(); // no definition since abstract
    public void Adisplay()
     {
        System.out.println("\n \t in A class i= "+i);
     }
 }
class B extends A
 {
    public int j;
    public B()
     {
        System.out.println("\n \t default constructor B() is called");
        j=20;
     }
    public void Bdisplay()
     {
        System.out.println("\n \t in B class j= "+j);
     }
    public void add()
     {
         System.out.println("\n \t in B class add() is called");
         j=j+i;
     }
 }
 class Abstract
  {
     public static void main(String ar[])throws IOException
      {
         System.out.println("\n \t start of main()");
         B b=new B();
         b.add();
         b.Bdisplay();
         System.out.println("\n \t end of main()");
      }
  }
```

OUTPUT:

```
   start of main()

   default constructor A() is called

   default constructor B() is called

   in B class add() is called

   in B class j=30

   end of main()
```

- Note: abstract classes must be inherited.
- Note: abstract classes must be override

➢ **USING FINAL WITH INHERITANCE:**

- The final keyword is used in three ways
  - To variables, those become constants.
  - To methods, those should not be override.
  - To the class, then that class should not be inherited.

Program which illustrates the usage of final keyword

```
   /* final keyword to variable, method */
import java.io.*;
class A
 {
    public int i;
    final int SPEED_LIMIT=60;
    public A()
     {
        System.out.println("\n \t default constructor A() is called");
        i=10;
     }
    public final void Aadd() // not overrided since final
     {
        System.out.println("\n \t in A class final add() method is called ");
        i=i+10;
     }
    public void Adisplay()
     {
        System.out.println("\n \t in A class i= "+i);
     }
 }
class B extends A
 {
    public int j;
    public B()
     {
        System.out.println("\n \t default constructor B() is called");
        j=20;
     }
    public void Badd()
     {
         System.out.println("\n \t in B class Badd() is called");
         j=j+i;
     }
    public void Bdisplay()
     {
        System.out.println("\n \t in B class j= "+j);
     }
 }
 class FinalMethod
  {
     public static void main(String ar[])throws IOException
      {
         System.out.println("\n \t start of main()");
         B b=new B();
         b.Aadd();
         b.Adisplay();
         b.Badd();
         b.Bdisplay();
         System.out.println("\n \t end of main()");
      }
  }
```

OUTPUT

```
   start of main()

   default constructor A() is called

   default constructor B() is called

   in A class final add() method is called

   in A class i=20

   in B class Badd() is called

   in B class j=40

   end of main()
```

1. /* final keyword to class */

```
import java.io.*;
final class A       // not inherited
 {
   public int i;
   public A()
    {
      System.out.println("\n \t default constructor A() is called");
      i=10;
    }
    public final void Aadd() // not overrided since final
     {
        System.out.println("\n \t in A class final add() method is called ");
        i=i+10;
     }
    public void Adisplay()
     {
        System.out.println("\n \t in A class i= "+i);
     }
 }
 class FinalCV
  {
     public static void main(String ar[])throws IOException
      {
         System.out.println("\n \t start of main()");
         final int f=100;  // like const variable
         System.out.println("\n \t in main() the value of final f= "+f);
         A ob=new A();
         ob.Aadd();
         ob.Adisplay();
         System.out.println("\n \t end of main()");
      }
  }
```

OUT PUT:

```
   start of main()
   in main() the value of final f=100
   default constructor A() is called
   in A class final add() method is called
   in A class i=20
   end of main()
```

➢ THE OBJECT CLASS:

- There is one special class, **Object**, defined by Java.
- All other classes are subclasses of **Object**, **Object** is a superclass of all other classes.
- This means that a reference variable of type **Object** can refer to an object of any other class

METHOD	**PURPOSE**
protected Object cone()	Creates a new object that is same as the object being cloned
boolean equals(Object ob)	Determines whether one object is equal to another
protected void finalize()	Called before an unused object is recycled
final class getClass()	Obtains the class of an object at runtime
int hashCode()	Returns the hashcode associated with the invoking object
void notify()	Resumes execution of a thread waiting on the invoking object
void notifyAll()	Resumes execution of all threads waiting on the invoking object
String toString()	Returns a string that describes the object
void wait()	
void wait(long milliseconds)	Waits on another thread of execution
void wait(long milliseconds, int nanoseconds)	

UNIT-II

Assignment-Cum-Tutorial Questions

SECTION-A

Objective Questions

1) Which of the following is the correct syntax for creating Object [ ]

(a)Classname objName=new Classname;

(b)Classname objName=new Classname();

(c)Classname objName=Classname();

()objName classname=new objName();

2) ___________is a keyword that refers to the current object that invoked the method.

3) _______ is the process of reclaiming the runtime unused memory automatically.

4) _____is the process of defining 2 or more methods within same class that have same name but different parameter declarations. [ ]

(a) Method overriding (b) Method overloading

(c) Method hiding (d) None of the above

5) Which of these is correct way of inheriting class A by class B? [ ]

(a) class B class A { } (b) class B inherits class A { }

(c) class B extends A { } (d) class B extends class A { }

6) Run-time polymorphism is achieved by using_____ [ ]

(a) Method Overloading (b) Constructor Overloading

(c) Method Overriding (d) this keyword

7) _______ is the Super class for all the classes in Java

8) What is the output of this program? [ ]

```
class box{
```

```
int width;

int height;

int length;

int vol;

box(){

width = 5;

height = 5;

length = 6;  }

void volume() {

vol = width*height*length; } }

class constructor_output {

public static void main(String args[]) {

box obj = new box();

obj.volume();

System.out.println(obj.vol); } }
```

(a) 100 (b) 150 (c) 200 (d) 250

9) Consider the following code [ ]

```
class A {

private int i;

public int j;   }

class B extends A {
```

```
int k;

void show() {

k=i+j;

System.out.println("sum of " +i+ "and" +j+"="+k);  }

public static void main(String arg[])  {

B b1=new B();   } }
```

(a)B gets only the member j through inheritance from A

(b)B gets both i, j through inheritance from A

(c)A is the sub class and B is the super class

(d)None of the above

10) what is the output of this program? [ ]

```
     class overload {
int x;

int y;

void add(int a) {

x =  a + 1;   }

void add(int a, int b) {

x =  a + 2; } }

   class Overload_methods {

public static void main(String args[]) {

overload obj = new overload();

int a = 0;

obj.add(6,7);
System.out.println(obj.x);    }  }
```

(a) 5 (b)8 (c)7 (d) 6

11) The following code prints ________ [ ]

```
class A {

int i;

int j;

A() {

i = 1;

j = 2;  } }

class Output {

public static void main(String args[]) {

A obj1 = new A();

System.out.print(obj1.toString()) } }
```

a. true
b. String associated with object
c) false
d) Compilation Error

12) Predict the output of following Java Program. [ ]

```
   class Grandparent   {

public void Print( ) {

System.out.println("Grandparent's Print()");   }     }

class Parent extends Grandparent {

public void Print( ) {

System.out.println("Parent's Print()");

System.exit(0); }    }
```

class Child extends Parent {

public void Print() {

super.Print();

System.out.println("Child's Print()"); } }

public class Main {

public static void main(String[] args) {

Child c = new Child();

c.Print(); } }

(a)Grandparent's Print()

(b)Parent's Print()

(c)Child's Print()

(d)Runtime Error

13) What is the output of the following Java program? [ ]

class Test {

int i; }

class MainDemo {

public static void main(String args[]) {

Test t = new Test();

System.out.println(t.i); } }

(a)0 (b)garbage value c) compiler error d) runtime error

14) What is the output of the following Java program? [ ]

```
class Point {

int m_x, m_y;

public Point(int x, int y) {

m_x = x;     m_y = y;  }

public static void main(String args[]) {

Point p = new Point();  } }
```

(a)1

(b) garbagevalue

(c) compilererror

(d) runtime error

SECTION-B

SUBJECTIVE QUESTIONS

1. Define class. Write the steps for creating class and object? Explain it with an example?
2. Define constructor? Can we overload a constructor? If so, explain with an example?
3. Explain the usage of following keywords with examples?

a) this b) super c) final

4. List Different types of Inheritance? Explain with example programs?
5. To read an integer n and then print the nth table as below:

1 x n = n

2 x n = 2n

. . . . .

10 x n = 10n

6. To read the details of a student like name, age, phone number in a method called getData() and then write another method called putData() to display the details.
7. To find factorial of a given number using recursion?
8. (a) Implement Method overloading with the following example?

   (b) To overload a method area() which computes the area of a geometrical figure based on number of parameters. If number of parameters is 1 and is of type float it should calculate the area of circle, if it is of type int it should calculate area of square. If the number of parameters is 2 and they are of type float calculate area of triangle, if they are of int calculate area of rectangle.
9. Implement dynamic method dispatch with an example.
10. Define Abstract class. Differentiate abstract method and concrete method?