

Unit 2- Defining your own Classes, Objects and Methods

Session 17-20

Recap

Class is a template, blue print or contract that defines what an object field and method will be

Car



Color
Mileage
Price
Model

StartEngine()
StopeEngine()
Speedup()

Class

- Class defines a new data type.

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

Example

```
class Dog{  
    String name;  
    String color;  
    String breed;  
    int age;  
    void barking()  
    {  
        -----  
    }  
    void wagging_tail()  
    {  
        -----  
    }  
}
```



Attributes
Name
Color
Breed
age

Methods
barking
Wagging tail

- The data or variables defined within a class are called instance variables.
- The code is contained within the methods.
- The methods and variables defined within a class are called members of class.
- Each instance of the class contains its own copy of these variables. Thus, the data for one object is separate and unique from the data for another.

Local variables:

- * Variables defined inside methods, constructors or blocks are called local variables.
- * The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.

Instance variables (Member variable):

- * Instance variables are variables within a class but outside any method.
- * These variables are instantiated when the class is loaded.
Instance variables can be accessed from inside any method, constructor or blocks of that particular class.

Class variables (Static Variables):

- * Class variables are variables declared with in a class, outside any method, with the static keyword.

Adding variables:

- * Data is encapsulated in a class by placing data fields inside the body of the class definition.
- * These variables are called as instance variables because they are created whenever an object of the class is instantiated.

(e.g.) class rectangle
 {
 int length;
 int width;
 }

- * No storage space has been created in the memory during declaration.
- * Instance variables are also known as member variables.

Adding Methods:

- * Methods are declared inside the body of the class but immediately after the declaration of instance variables.
- * The general form of method declaration is

```
Syntax:  type methodname(parameter_list)
        {
            method-body;
        }
```

```
(e.g.)  class rectangle
        {
            int length;
            int width;
            void getdata(int x, int y)
            {
                length=x;
                length=y;
            }
        }
```

Example for Adding Variables and Methods

- Create a class “rectangle” which contain instance variable length, and width and methods getdata() and area()

Adding Variables

```
class rectangle
{
    int length;
    int width;
}
```

Adding Methods

Adding Method getdata()

```
class rectangle
{
    int length;
    int width;
    void getdata(int x, int y)
    {
        length=x;
        length=y;
    }
}
```

Adding method area()

```
class rectangle
{
    int length, width;

    void getdata(int x,int y)
    {
        length=x;
        width=y;
    }

    int area()
    {
        int area=length*width;
        return(area);
    }
}
```

- * Instance variables and methods in classes are accessible by all the methods in the class but a method cannot access the variables in other methods.

```
class access
{
    int x;

    void method1()
    {
        int y;
        x=10;
        y=x;
    }

    void method2()
    {
        int z;
        x=5;
        z=10;
        y=1;           //illegal
    }
}
```

Object

```
(e.g.)    class student
          {
            String name;
            int usnno,m1,m2,m3;
          }
```

- * In the above example, a class declaration only creates a template, not an object
- * It does not cause any objects of type student to come into existence.

student ob=new student() // creates a student object called ob.

* An object gets created containing its own copy of instance variables defined by a class. Thus, every student object will contain its own copies of instance variables sname, usnno, m1,m2,m3.

- * To access the variables, use dot operator.

```
// ob.usnno=123;
```

Declaring Objects:

- * Obtaining objects of a class is a two-step process:
 - Firstly, declare a variable of the class type.
 - Acquire an actual, physical copy of the object and assign it to the variable.
(on using new operator)
- * The new operator dynamically allocates memory for an object and returns a reference to it. This reference is more or less the address in memory, which is then stored in the variable.

Syntax: `classname class_var = new classname();`

(e.g.) `student ob; // declare reference to object`
`ob=new student(); // allocate a student object (instantiation)`

(or)

`student ob=new student();`

- * The parenthesis after the class name specifies the constructor for the class.

Assigning object reference variables:

- * When you assign one object reference variable to another object reference variable, you are not creating a copy of the object but making a copy of the reference.

```
(e.g.) student ob1=new student();  
      student ob2=ob1;  
      .....  
      .....  
      ob1=null;
```

Methods and adding method to a class

Returning a value

Adding a method that takes parameters


```
(e.g.)  class rectangle
        {   int length, width;
            void getdata(int x,int y)
            {   length=x;    width=y; }
            int rectarea()
            {
                int area=length * width;
                return(area);
            }
        }

class rectarea
{   public static void main(String args[])
    {   int area1, area2;
        rectangle rect1=new rectangle();
        rectangle rect2=new rectangle();
        rect1.length=15;    rect1.width=10;

        area1=rect1.length*rect1.width;
        rect2.getdata(20,12);
        area2=rect2.rectarea();
        System.out.println(area1);
        System.out.println(area2);
    }
}
```

Constructors:

- * It's tedious to initialize all the variables in a class each time an instance is created.
- * Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor.
- * The constructor has the same name as the class in which it resides.
- * Once defined, the constructor is automatically called immediately after the object is created, before the new operator completes.
- * They have no return type, not even void.
- * The implicit return type of a class constructor is the class type itself.
- * If you don't type a constructor into your class code, a default constructor will be automatically generated by the compiler. The default constructor is always a no-arg constructor.

(e.g.) 1) class student
{ }

The compiler generated code

```
class student
{
    student()
    {
        super();
    }
}
```

2) class student
{
 student()
 {}
}

The compiler generated code

```
class student
{
    student()
    {
        super();
    }
}
```

- *The default constructor has the same access modifier as the class
- * The default constructor has no arguments
- * The default constructor includes a no-arg call to the super constructor.

Parameterized Constructors

- A Constructor which has **parameters** in it called as **Parameterized Constructor**, this constructor is used to assign different values for the different objects.
- We can have any number of Parameterized Constructor in our class

```
public class Car {  
    String carColor;  
    Car(String carColor) {  
        this.carColor = carColor;  
    }  
    public void disp()  
    { System.out.println("Color of the Car is : "+carColor);  
    }  
    public static void main(String args[]) { //Calling the parameterized constructor  
        Car c = new Car("Blue");  
        c.disp();  
    }  
}
```

“new” Operator

- When you are declaring a class in java, you are just creating a new data type. A class provides the blueprint for objects. You can create an object from a class. However obtaining objects of a class is a two-step process :
 - Declaration
 - Instantiation and Initialization

For example, consider the class Box

```
class Box {  
    double width;  
    double height;  
    double length;  
}
```

For creating an Object:

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

Object declaration : **Box mybox;**

- Second, acquire an actual, physical copy of the object and assign it to that variable by using the *new* operator.
- The *new* operator instantiates a class by dynamically allocating(i.e, allocation at run time) memory for a new object and returning a reference to that memory. This reference is then stored in the variable.
- Thus, in Java, **all class objects must be dynamically allocated**. The *new* operator is also followed by a call to a class constructor, which initializes the new object. A constructors defines what occurs when an object of a class is created.
- Instantiation and Initialization: **mybox = new Box();**
- Instantiation via new operator and Initialization via default constructor of class Box

Method Overloading:

- * Methods that have the same name but different parameter list and different definitions.
- * Method overloading is used when objects are required to perform similar tasks but using different input parameters.
- * When we call a method in an object, Java matches up the method name first and then the number and type of parameters to decide which one of the definitions to execute. (polymorphism).
- * Method Overloading increases the readability of the program.

Different ways to overload a method:

- * There are two ways to overload a method in Java
 - By changing number of arguments
 - By changing the data type
- * Method Overloading is not possible by changing the return type of the method.

Different Scenario's

- **Method Overloading by changing the no. of arguments:**
- **Method Overloading by changing data type of argument:**
- **Method Overloading changing the return type of method:**
- **Overloading main() method:**
- **Method Overloading with TypePromotion:**
- **Method Overloading with TypePromotion of arguments if matching found:**
- **Method Overloading with TypePromotion in case ambiguity:**

Examples:

1) Method Overloading by changing the no. of arguments:

```
class Calculation
{
    void sum(int a,int b)
    {
        System.out.println(a+b);
    }
    void sum(int a,int b,int c)
    {
        System.out.println(a+b+c);
    }

    public static void main(String args[])
    {
        Calculation obj=new Calculation();
        obj.sum(10,10,10);
        obj.sum(20,20);
    }
}
```

2) Method Overloading by changing data type of argument:

```
class Calculation
{
    void sum(int a,int b)
    {
        System.out.println(a+b);
    }
    void sum(double a,double b)
    {
        System.out.println(a+b);
    }

    public static void main(String args[])
    {
        Calculation obj=new Calculation();
        obj.sum(10.5,10.5);
        obj.sum(20,20);
    }
}
```

3) Method Overloading not possible by changing the return type of method:

```
class Calculation
{
    int sum(int a,int b)
    {
        return(a+b);
    }
    double sum(int a,int b)
    {
        return(a+b);
    }

    public static void main(String args[])
    {
        Calculation obj=new Calculation();
        int result=obj.sum(20,20); //Compile Time Error
    }
}
```

4) Overloading main() method:

```
class Overloading1
{
    public static void main(int a)
    {
        System.out.println(a);
    }

    public static void main(String args[])
    {
        System.out.println("main() method invoked");
        main(10);
    }
}
```

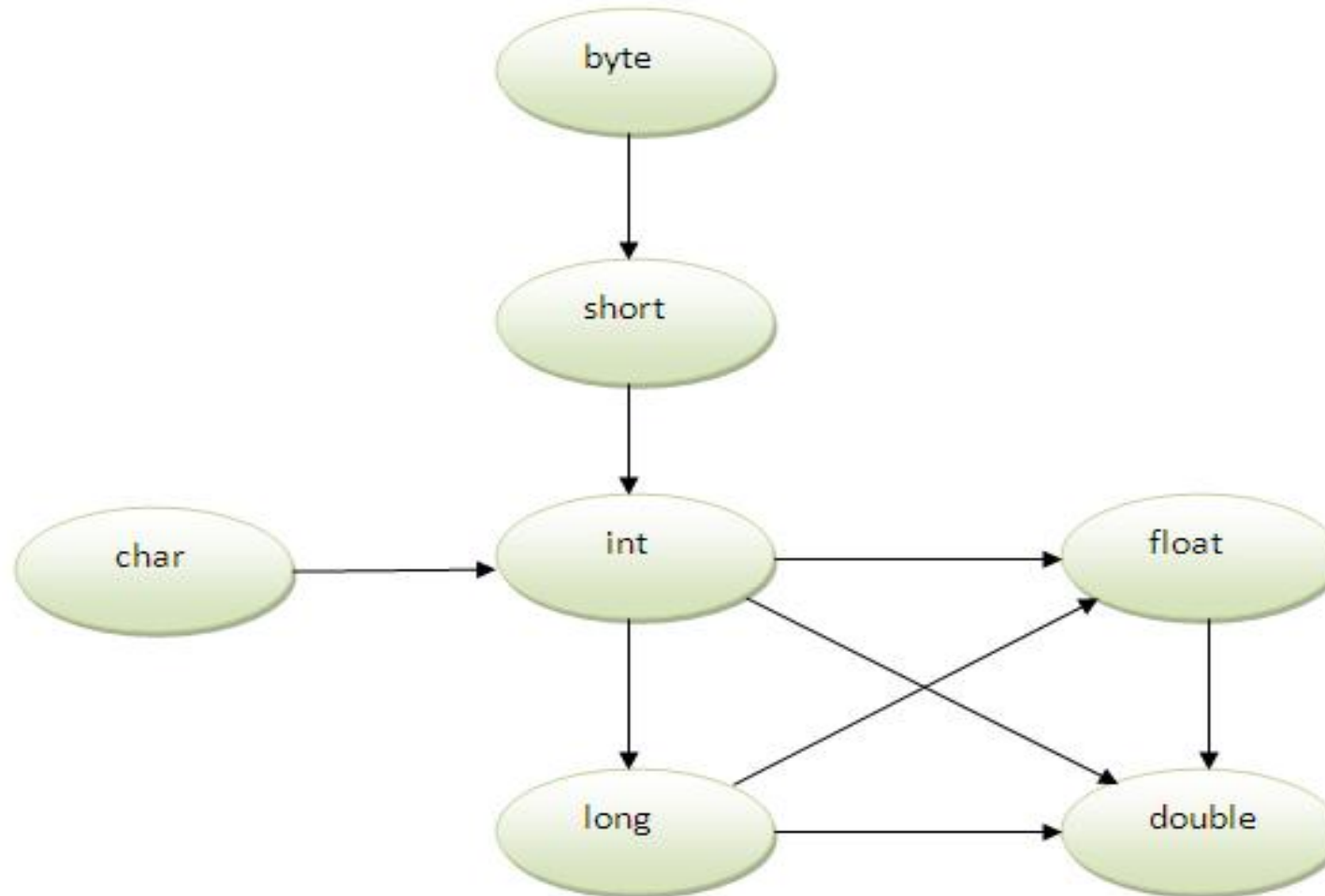
5) Method Overloading with TypePromotion:

```
class OC
{
    void sum(int a,long b)
    {
        System.out.println(a+b);
    }
    void sum(int a,int b,int c)
    {
        System.out.println(a+b+c);
    }

    public static void main(String args[])
    {
        OC obj=new OC();
        obj.sum(20,20);
        //now second int literal will be promoted to long
        obj.sum(20,20,20);
    }
}
```

Method Overloading and Type Promotion:

* One type is promoted to another implicitly if no matching datatype is found.



6) Method Overloading with TypePromotion if matching found:

If there are matching type arguments in the method, type promotion is not performed.

```
class OC
{
    void sum(int a,int b)
    {
        System.out.println("int arg method invoked");
    }
    void sum(long a,long b)
    {
        System.out.println("long arg method invoked");
    }

    public static void main(String args[])
    {
        OC obj=new OC();
        obj.sum(20,20);
        //now int arg sum() method gets invoked
    }
}
```

7) Method Overloading with TypePromotion in case ambiguity:

If there are no matching type arguments in the method, and each method promotes similar number of arguments, there will be ambiguity.

```
class OC
{
    void sum(int a,long b)
    {
        System.out.println("a method invoked");
    }
    void sum(long a,int b)
    {
        System.out.println("b method invoked");
    }

    public static void main(String args[])
    {
        OC obj=new OC();
        obj.sum(20,20);//now ambiguity
    }
}
```


Session 21-22

Controlling access to class members

Controlling access to class members

- An access modifier restricts the access of a class, constructor, data member and method in another class.
- Java Access Modifiers are
 1. public
 2. protected
 3. private
 4. default

- We are allowed to use only “public” or “default” access modifiers with java classes.
- If a class is “public” then we can access it from anywhere, i.e from any other class located in any other packages etc.
- We can have only one “public” class in a source file and file name should be same as the public class name.
- If the class has “default access” then it can be accessed only from other classes in the same package.

Java access modifier with class members

- **public** access modifier

The members, methods and classes that are declared public can be accessed from anywhere. This modifier doesn't put any restriction on the access.

```
public class Clock{  
    public long time=0;  
}  
  
public class ClockReader{  
    Clock clock= new Clock();  
    public long readClock()  
    {  
        return clock.time;  
    }  
}
```

The variable **time** in the class Clock is marked with **public** Java access modifier. Therefore the ClockReader class can access the variable **time** in the Clock class no matter what package the ClockReader is located.

Program name- Addition.java

```
package abcpackage;  
public class Addition{  
    public int addTwoNumber(int a, int b)  
    {  
        return (a+b);  
    }  
}
```

Program name- Test.java

```
package xyz;  
import abcdpackage.*;  
class Test{  
    public static void main(String  
        args[]){  
        Addition obj= new Addition();  
        Int k= obj.addTwoNumber(100,1);  
        System.out.println(k);  
    }  
}
```

The method addTwoNumber(int a, int b) has public access modifier and the class Test is able to access the method without even extending the Addition class. **This is because public modifier has the visibility everywhere.**

- **protected** access modifier

- Protected data member and method are only accessible by **the classes of the same package** and the **subclasses present in any package**.

- You can also say that the protected access modifier is similar to default access modifier with **one exception that it has visibility in subclasses**.

Classes cannot be declared protected. This access modifier is generally used in a parent child relationship

Addition.java

```
package abcpackage;
public class Addition {
    protected int addTwoNumbers(int a, int b){
return a+b; }
}
```

Test.java

```
package xyzpackage;
import abcpackage.*;
class Test extends Addition{
    public static void main(String args[]){
        Test obj = new Test();
        System.out.println(obj.addTwoNumbers(11, 22));
    }
}
```

In this example the class Test which is present in **another package** is able to call the addTwoNumbers() method, which is declared protected. This is because the Test class **extends** class Addition and the **protected modifier allows the access of protected members in subclasses (in any packages)**

▪

Output: 33

- private access modifier

The scope of private modifier is limited to the class only.

1. Private Data members and methods are only accessible within the class
2. Class and Interface cannot be declared as private
3. If a class has private constructor then you cannot create the object of that class from outside of the class.

```
class ABC{  
    private double num = 100;  
    private int square(int a){ return a*a; }  
}  
public class Example{  
    public static void main(String args[]){  
        ABC obj = new ABC();  
        System.out.println(obj.num);  
        System.out.println(obj.square(10));  
    }  
}
```

Output
Compile Time error

This example throws compilation error because we are trying to access the private data member and method of class ABC in the class Example. **The private data member and method are only accessible within the class.**

- default access modifier

- When we do not mention any access modifier, it is called default access modifier. The scope of this modifier is limited to the package only.
- This means that if we have a class with the default access modifier in a package, **only those classes that are in this package can access this class. No other class outside** this package can access this class. Similarly, if we have a default method or data member in a class, it **would not be visible** in the class of another package.

Addition.java

```
package abcpackage;
public class Addition {
/* Since we didn't mention any access
modifier here, it would be considered as
default. */
int addTwoNumbers(int a, int b){
return a+b;
}
}
```

Test.java

```
package xyzpackage;
import abcpackage.*;
public class Test{
public static void main(String args[]){
Addition obj = new Addition();
/* It will throw error because we are trying to
access the default method in another package */
System.out.println(obj.addTwoNumbers(11, 22));
}
}
```

In this example we have two classes, Test class is trying to access the **default method** of Addition class, since class Test belongs to a different package, this program would throw compilation error, **because the scope of default modifier is limited to the same package in which it is declared.**

Call-By-Value:

- * Call-By-Value copies the value of an argument into the formal parameter of the subroutine.
- * Changes made to the parameter of the subroutine have no effect on the argument.
- * In Java, when you pass a primitive type to a method, it is passed by value.

```
class Operation
```

```
{
```

```
    int data=50;
```

```
    void change(int data)
```

```
    {    data=data+100 ;    }
```

```
    public static void main(String args[])
```

```
    {
```

```
        Operation op=new Operation();
```

```
        System.out.println("before change "+op.data);
```

```
        op.change(500);
```

```
        System.out.println("after change "+op.data);
```

```
    } }
```

Call-By-Reference:

- * A reference to an argument is passed to the parameter.
- * Changes made to the parameter of the subroutine will effect the argument specified in the call.
- * Objects are implicitly passed by use of Call-By-Reference.

```
class Operation
{
    int data=50;
    void change(Operation op1)
    {
        op1.data=op1.data+100;
    }
    public static void main(String args[])
    {
        Operation op=new Operation();
        System.out.println("before change "+op.data);
        op.change(op);//passing object
        System.out.println("after change "+op.data);
    } }
```

Returning an Object

- In Java, method can return any type of data, including objects

```
class Data {  
    int data1;  
    int data2;  
}  
class CreateData {  
    Data createData() { return new Data(); }  
}  
public class Javaapp {  
    public static void main(String[] args) {  
        CreateData cd = new CreateData();  
        Data d1 = cd.createData();  
        d1.data1 = 10;  
        d1.data2 = 20;  
        System.out.println("d1.data1 : "+d1.data1);  
        System.out.println("d1.data2 : "+d1.data2);  
    }  
}
```

Session 23-24

Recursion

The process in which the function or method calling itself directly or indirectly is called recursion and the corresponding function (method) is called recursive function (recursive method).

```
class RecursionExample{ int count=0;

    void p(){

        count++;

        if(count<=5) {

            System.out.println("hello");

            p();        }

        }

    public static void main (String []args){

RecursionExample re= new RecursionExample();

        re.p();    }}
```

Static Keyword:

- * The static keyword is used mainly for memory management.
- * The static keyword belongs to the class than instance of the class
- * The static can be:
 - Variable (Class variables)
 - Method (Class method)
 - Block

Static Variable:

- * If you declare any variable as static, it is known as static variable.
- * The static variable can be used to refer the common property of all objects (e.g.) company name of employees, college name of students, etc...
- * The static variable gets memory only once in class area at the time of class loading.

Advantages of static variable:

- * The static variable makes your program memory efficient. (i.e it saves memory)

(e.g) Without static variable

```
class student
{
    int rollno;
    String name;
    String college="JAIN";
}
```

// Here all instance data members will get memory each time when object is created.

// Here, college refers to the common property of all objects.

// If we make static, this field will get memory only once.

class Student // Example on static member or variable

```
{
    int rollno;
    String name;
    static String college = "JAIN";

    Student(int r, String n)
    {
        rollno = r;
        name = n;
    }
    void display ()
    {
        System.out.println(rollno+" "+name+" "+college);    }

    public static void main(String args[])
    {
        Student s1 = new Student (111, "xxx");
        Student s2 = new Student (222, "yyy");
        s1.display();
        s2.display();
    }
}
```

Static method:

- * A static method belongs to the class rather than object of a class
- * A static method can be invoked without the need for creating an instance of a class.
- * static method can access static data member and can change the value of it.
- * The static method cannot use non static data member or call non-static method directly.

(e.g.) class A

```
{    int a=40; // non-static
    public static void main(String args[])
    {
        System.out.println(a);
    }
} // output: Compile time error
```

- * this and super keywords cannot be used in static context.

Static block:

- *It is used to initialize the static data member

- * It is executed before main method at the time of class loading.

(e.g.)

```
class A
{
    static
    {
        System.out.println("static block is invoked");
    }

    public static void main(String args[])
    {
        System.out.println("Hello");
    }
}
```

final keyword:

- * Variable declared as final prevents its contents from being modified.
- * Initialize a final variable when it is declared.

Syntax: final datatype symbolic_name = constant;

(e.g.) final int PI=3.1459;

- * There are many things which is required repeatedly and if we want to make changes then we have to make these changes in whole program where this variable is used.

Command-Line arguments:

- * Parameters that are supplied to the application program at the time of invoking its execution.
- * Parameters must be supplied at the time of its execution following the file name.
- * In the main(), the args are array of string known as string objects.
- * Any argument provided in the command line at the time of program execution, are accepted to the array args as its elements.

- * Using index the individual elements of an array can be accessed.
- * The number of elements in the array args can be retrieved with the length parameter.

(e.g.)

```
class add
{
    public static void main(String args[])
    {
        int a=Integer.parseInt(args[0]);
        int b=Integer.parseInt(args[1]);
        System.out.println(a+b);
    }
}
```


Session 25-28

Nested and Inner Classes:

- * Defining a class within another class is known as nested classes.
- * The scope of nested class is bounded by the scope of its enclosing class.
- * The nested class has access to the members, including private members of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class.

```
class outer
{
    class inner()
    {}
}
```

// On compilation, we can see two class files.

- * The class file gets generated even for the inner class. But the inner class file isn't accessible to you

```
(e.g.)  class outer
        {   private int x=7;
            class inner
            {
                public void seeinner()
                {
                    System.out.println(x);
                }
            }
        }
```

- In the above example, the inner class is accessing the private member of the Outer class. Because, the inner class is also a member of the outer class.

Instantiating an Inner class:

- * To instantiate an instance of an inner class, you must have the instance of the outer class to tie to the inner class.
- * An inner class instance can never stand alone without a direct relationship with a specific instance of the outer class.

Instantiating an inner class from within code in the outer class:

Class outer

```
{    private int x=7;
    public void makeinner()
    {
        inner in =new inner();
        in.display();
    }

    class inner
    {
        public void display()
        {
            System.out.println(x);
        }
    }
}
```

Creating an Inner class object from outside the outer class instance code

```
public static void main(String[] args)
{
    outer ob=new outer();
    outer.inner in=ob.new inner();
    in.display();
}
```

(or)

```
public static void main(String[] args)
{
    outer.inner in=new outer().new inner();
    in.display();
}
```

Method-Local Inner classes:

Class outer

```
{
    private String x="abc";

    void stuff()
    {
        class inner
        {
            public void display()
            {
                System.out.println(x);
            }
        }
    }
}
```

Method-Local Inner classes:

Class outer

```
{   private string x="abc";
    void stuff()
    {
        class inner
        {
            public void display()
            {
                System.out.println(x);
            }
        }
        inner in=new inner() //after the class
        in.display();
    } }
```

// A method-local inner class can be instantiated only within the method where the inner class is defined.

Static Nested Class:

- * A static class created inside a class is called static nested class in Java.
- * It cannot access non-static data members and methods.
- * It can be accessed by outer class name.
- * It can access static data members of outer class including private.

```
class TestOuter1
{
    static int data=30;
    static class Inner
    {
        void msg(){System.out.println("data is "+data);}
    }

    public static void main(String args[])
    {
        TestOuter1.Inner obj=new TestOuter1.Inner();
        obj.msg();
    }
}
```


[illegible]

Anonymous inner class:

- A class that have no name is known as anonymous inner class.
- An anonymous inner class can be useful when making an instance of an object with certain “extras” such as overloading methods of a class or interface, without having to actually subclass a class.
- Anonymous inner classes are useful in writing implementation classes for listener interfaces in graphics programming.

Anonymous inner class are mainly created in two ways:

1. Class (may be abstract or concrete)
2. Interface

Garbage Collection and Finalizers

- Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.
- In the Java programming language, dynamic allocation of objects is achieved using the **new** operator.
- An object once created uses some memory and the memory remains allocated till there are references for the use of the object.
- When there are no references to an object, it is assumed to be no longer needed, and the memory, occupied by the object can be reclaimed.

There are different ways to make an Object eligible for garbage collection

- Nullifying the reference variable
- Reassigning the reference variable
- Anonymous Objects
- Object created inside a method etc

```
Box mybox= new Box();  
mybox=null;
```

```
Box mybox1= new Box();  
Box mybox2= new Box();  
mybox1= mubox2;
```

```
new Box();
```

- Once we made object eligible for garbage collection, it may not destroy immediately by garbage collector. Whenever JVM runs Garbage Collector program, then only object will be destroyed. But when JVM runs Garbage Collector, we can not expect.
- We can also request JVM to run Garbage Collector. There are two ways to do it :
 - **Using *System.gc()* method** : System class contain static method *gc()* for requesting JVM to run Garbage Collector.
 - **Using *Runtime.getRuntime().gc()* method** : Runtime class allows the application to interface with the JVM in which the application is running. Hence by using its *gc()* method, we can request JVM to run Garbage Collector.

Finalize()

- Just before destroying an object, Garbage Collector calls *finalize()* method on the object to perform clean-up activities. Once *finalize()* method completes, Garbage Collector destroys that object.
- *finalize()* method is present in Object class with following prototype

protected void finalize() throws Throwable
- Based on our requirement, we can override *finalize()* method for perform our clean-up activities like closing connection from database

this Keyword

- Keyword “this” is a reference variable that refers to current object
- Following are the ways to use “this” keyword in java
 1. **Using ‘this’ keyword to refer current class instance variables**
 2. **Using this() to invoke current class constructor**
 3. **Using ‘this’ keyword to return the current class instance**
 4. **Using ‘this’ keyword as method parameter**
 5. **Using ‘this’ keyword to invoke current class method**
 6. **Using ‘this’ keyword as an argument in the constructor call**

1) this keyword can be used to refer current class instance variable:

* If there is ambiguity between the instance variable and parameter, this keyword resolves the problem of ambiguity.

```
class Student    // without using 'this' keyword
```

```
{
    int id;    String name;
    Student(int id,String name)
    {
        id = id;        name = name;
    }
    void display()
    {        System.out.println(id+" "+name);    }
    public static void main(String args[])
    {
        Student s1 = new Student(1,"K");
        Student s2 = new Student(3,"A");
        s1.display();        s2.display();
    }
}
```



```
class Student          // using 'this' keyword
{
    int id;    String name;
    Student(int id,String name)
    {
        this.id = id;          this.name = name;
    }
    void display()
    {        System.out.println(id+" "+name);    }
    public static void main(String args[])
    {
        Student s1 = new Student(1,"K");
        Student s2 = new Student(3,"A");
        s1.display();          s2.display();
    }
}
```

2) this() can be used to invoke current class constructor

- * This approach is better if you have many constructors in the class and want to reuse the constructor

```
class Student
```

```
{
```

```
    int id;    String name;
```

```
    Student()
```

```
    {
```

```
        System.out.println("Default constructor is invoked");
```

```
    }
```

```
    Student(int id, String name)
```

```
    {
```

```
        this(); // It is used to invoke current class constructor
```

```
        this.id = id;        this.name = name;
```

```
    }
```

```
    void display()
```

```
    {        System.out.println(id+" "+name);    }
```

```
}
```

```
class Student          // constructor chaining
{
    int id;          String name;          String city;
    Student(int id, String name)
    {
        this.id = id;    this.name = name;
    }
    Student(int id, String name, String city)
    {
        this(id,name);    //now no need to initialize id and name
        this.city=city;
    }

    void display(){System.out.println(id+" "+name+" "+city);}
    public static void main(String args[])
    {
        Student e1 = new Student(1,"k");
        Student e2 = new Student(2,"A","bangalore");
        e1.display();    e2.display();
    }
}
```

3) this() can be used to invoke current class method (implicitly)

```
class A
{
    void m()
    {}

    void n()
    {
        m();    // compiler implicitly adds the keyword 'this'
    }

    public static void main(String[] args)
    {
        new A().n();
    }
}
```

4) **this keyword can be passed as an argument in the method.**

Class s2

```
{
    void m(s2 obj)
    {
        System.out.println("method is invoked");
    }

    void p()
    {
        m(this);
    }

    public static void main(String[] args)
    {
        s2 s1=new s2();
        s1.p();
    }
}
```

5) this keyword can be passed as an argument in the constructor call.

```
class B
```

```
{    A4 obj;
    B(A4 obj)
    {    this.obj=obj;    }

    void display()
    {    System.out.println(obj.data);    }
}
```

```
class A4
```

```
{
    int data=10;
    A4()
    {
        B b=new B(this);
        b.display();
    }
    public static void main(String args[])
    {    A4 a=new A4();    }
}
```

6) this keyword can be used to return current class instance.

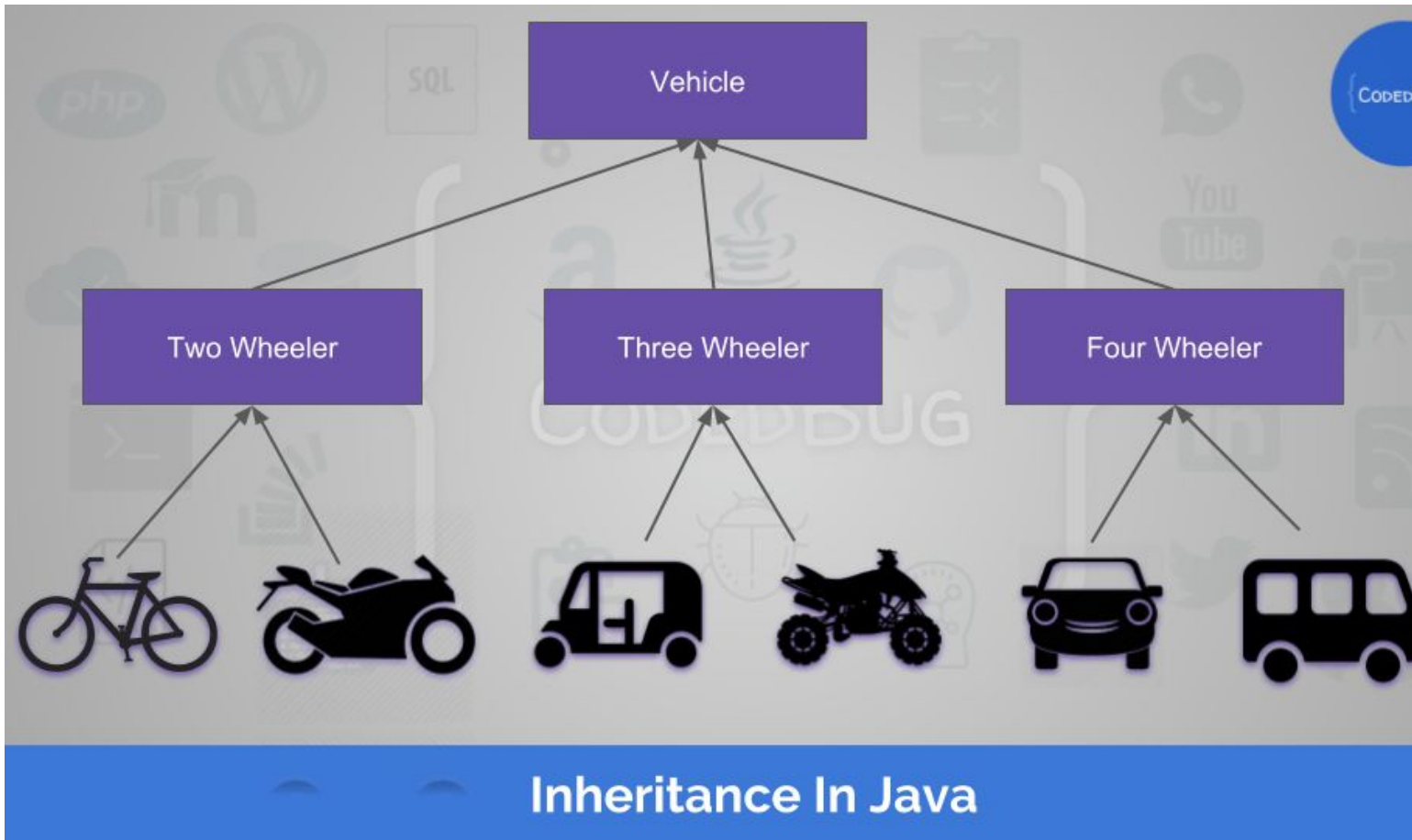
* Return type of the method must be the class type (non-primitive)

```
class A
{
    A getA()
    {        return this;        }
    void msg()
    {
        System.out.println("JU");
    }
}
class test
{
    public static void main(String[] args)
    {
        new A().getA().msg();
    }
}
```

Session 29-33

Inheritance, Polymorphism and Abstraction

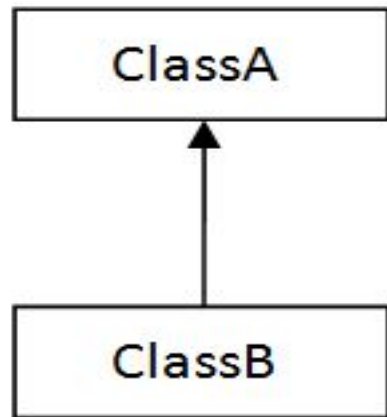
Recap - Inheritance



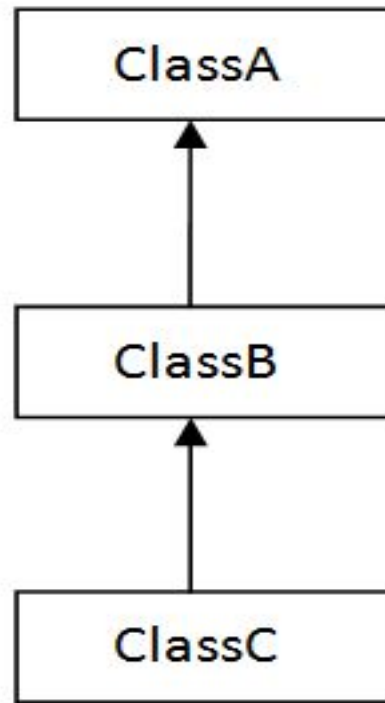
Inheritance In Java is the process by which one class acquires the properties of another class.

- **Super Class:** The class whose features are inherited is known as a superclass(or a base class or a parent class).
- **Sub Class:** The class that inherits the other class is known as a subclass(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the super-class fields and methods.
- **Reusability:** Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.
- The keyword used for inheritance is **extends**.

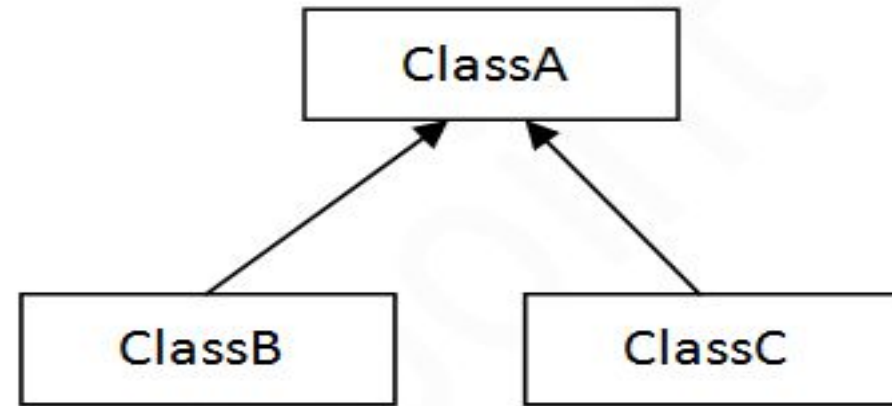
Types of Inheritance



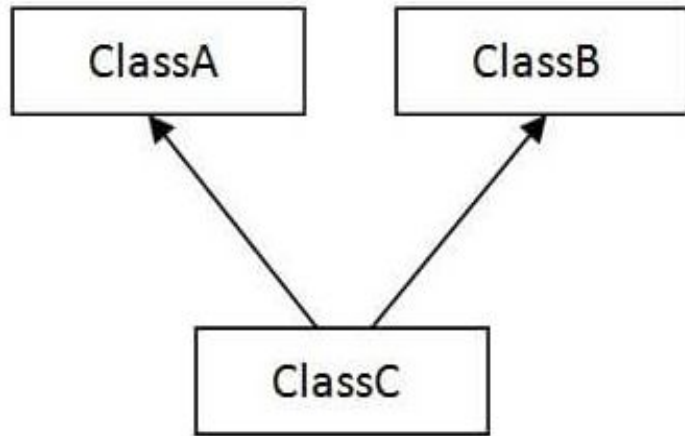
1) Single



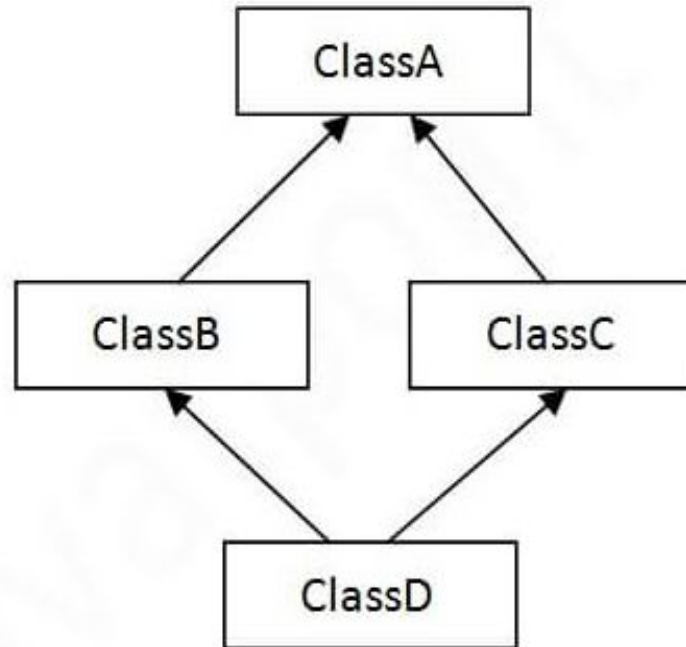
2) Multilevel



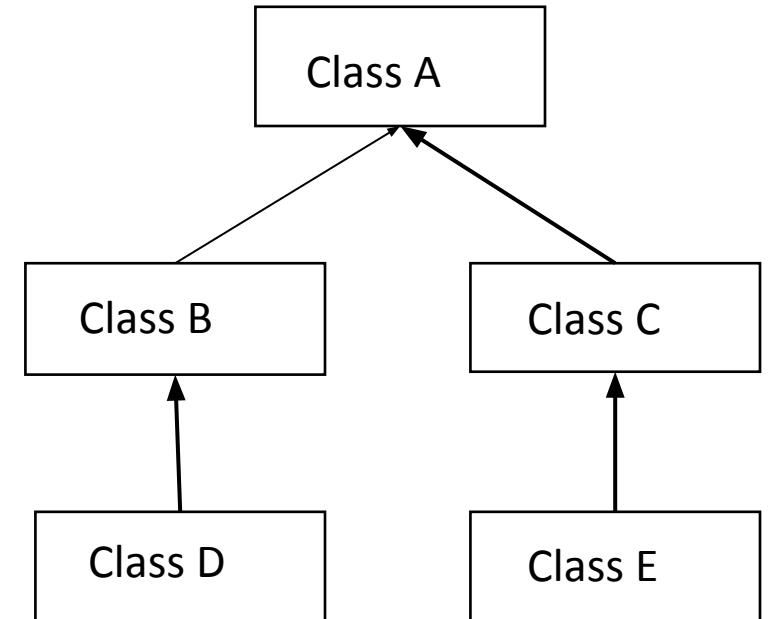
3) Hierarchical



4) Multiple



a



b

5) Hybrid

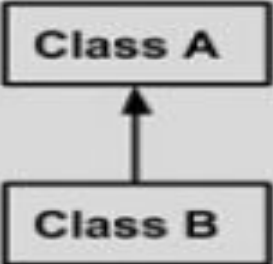
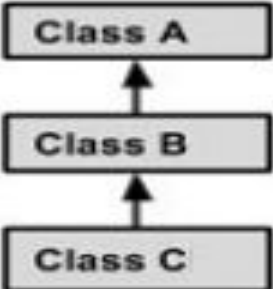
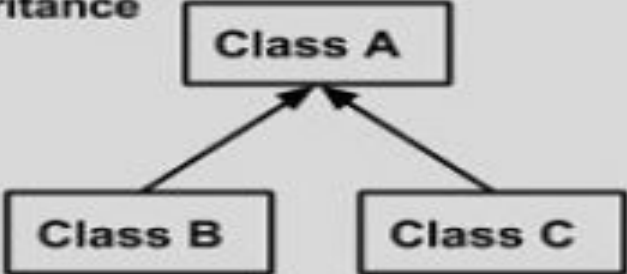
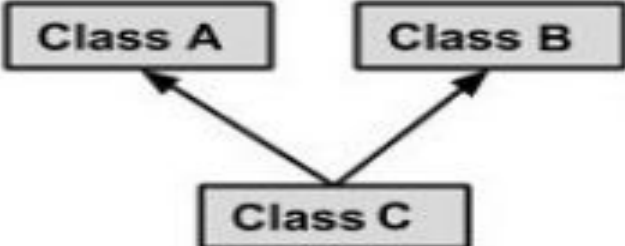
- To reduce the complexity and simplify the language, multiple inheritance is not supported in java.
- Consider a scenario where A, B and C are three classes. The C class inherits A and B classes. If A and B classes have same method and you call it from child class object, there will be ambiguity to call method of A or B class.
- Since compile time errors are better than runtime errors, java renders compile time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error now.

Defining a subclass:

```
class subclassname extends superclassname
{
    variables declaration;
    methods declaration;
}
```

// Keyword “extends” signifies that the properties of the superclassname
// are extended to the subclassname

- * The subclass will contain its own variables and methods as well of the superclass. This kind of situation occurs when we want to add some more properties to an existing class without actually modifying it.

<p>Single Inheritance</p>  <pre> graph BT B[Class B] --> A[Class A] </pre>	<pre> public class A { } public class B extends A { } </pre>
<p>Multi Level Inheritance</p>  <pre> graph BT C[Class C] --> B[Class B] B --> A[Class A] </pre>	<pre> public class A {} public class B extends A {.....} public class C extends B {.....} </pre>
<p>Hierarchical Inheritance</p>  <pre> graph BT B[Class B] --> A[Class A] C[Class C] --> A </pre>	<pre> public class A {} public class B extends A {.....} public class C extends A {.....} </pre>
<p>Multiple Inheritance</p>  <pre> graph BT C[Class C] --> A[Class A] C --> B[Class B] </pre>	<pre> public class A {} public class B {.....} public class C extends A,B { } // Java does not support mutiple Inheritance </pre>

Example:

```
class Calculation {
    int z;
    public void addition(int x, int y){
        z = x + y;
        System.out.println("The sum of the given numbers:"+z);
    }
    public void Subtraction(int x, int y) {
        z = x - y;
        System.out.println("The difference between the given numbers:"+z);
    }
}

class My_Calculation extends Calculation {
    public void multiplication(int x, int y) {
        z = x * y;
        System.out.println("The product of the given numbers:"+z);
    }
}

public class ExampleCalculation {
    public static void main(String args[]) {
        int a = 20, b = 10;
        My_Calculation demo = new My_Calculation();
        demo.addition(a, b);
        demo.Subtraction(a, b);
        demo.multiplication(a, b);
    }
}
```

What You can do in Subclass

- The inherited fields can be **used directly**, just like any other fields.
- You can declare a field in the subclass with the same name as the one in the superclass, thus *hiding* it (not recommended).
- You can declare new fields in the subclass that are not in the superclass.
- The inherited methods can be used directly as they are.
- You can write a new *instance* method in the subclass that has the same signature as the one in the superclass, thus *overriding* it.
- You can write a new *static* method in the subclass that has the same signature as the one in the superclass, thus *hiding* it.
- You can declare new methods in the subclass that are not in the superclass.
- You can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword `super`.

Member Access and Inheritance

- A subclass inherits all of the public and protected members of its parents.
- A subclass does not inherit the private members of the parent class.
- A nested class has access to all the private members of its enclosing class ie both attributes and methods. Therefore, public or protected nested class inherited by a subclass has **indirect access** to the all the private members of the superclass (parent class).

Constructors and Inheritance

- If a class is inheriting the properties of another class, the subclass automatically acquires the default constructor of the superclass.
- The superclass constructor can be called **explicitly** using the [super keyword](#).

super();

- If you want to call a parameterized constructor of the superclass, you need to use the super keyword as shown below.

super(values)

Default execution of constructors

```
class parent{
    parent(){
        System.out.println("Parent Class");
    }
}

class child extends parent{
    child(){
        System.out.println("Child Class");
    }
}

public class Constructor {

    public static void main(String[] args) {
        child ch= new child();
    } }
```

Explicit use of super keyword

```
class parent{
    int age;
    parent(int k){
        age=k;
        System.out.println("Parent Class");}
    void getAge(){
        System.out.println("Parent class Age="+age);}
}

class child extends parent{
    child(int m){
        super(m);
        System.out.println("Child Class"); }
}

public class Constructor {
    public static void main(String[] args) {
        child ch= new child(40);
        ch.getAge();
    } }
```

Using supper to Access Superclass Members

1. super is used to refer immediate parent class instance variable.

```
class Animal{
    String color="white";
}
class Dog extends Animal{
    String color="black";
    void printColor(){
        System.out.println(color);//prints color of Dog class
        System.out.println(super.color);//prints color of Animal class
    }
}
class TestSuper1{
    public static void main(String args[]){
        Dog d=new Dog();
        d.printColor();
    }
}
```

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

2. super can be used to invoke parent class method

```
class Animal{
    void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
    void eat(){System.out.println("eating bread...");}
    void bark(){System.out.println("barking...");}
    void work(){
        super.eat();
        bark();
    }
}
class TestSuper2{
    public static void main(String args[]){
        Dog d=new Dog();
        d.work();
    }
}
```

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

Creating a Multilevel Hierarchy

- In simple inheritance a subclass or derived class derives the properties from its parent class, but in multilevel inheritance a subclass is derived from a derived class.
- One class inherits only single class. Therefore, in multilevel inheritance, every time ladder increases by one.
- The lower most class will have the properties of all the super classes.

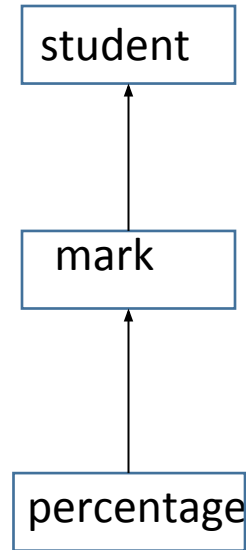

```

class student
{
    int rollno;
    String name;

    student(int r, String n)
    {
        rollno = r;
        name = n;
    }
    void dispdatas()
    {
        System.out.println("Rollno = " + rollno);
        System.out.println("Name = " + name);
    }
}

class marks extends student
{
    int total;
    marks(int r, String n, int t)
    {
        super(r,n); //call super class (student) constructor
        total = t;
    }
    void dispdatam()
    {
        dispdatas(); // call dispdatap of student class
        System.out.println("Total = " + total);
    }
}

```



Output

Rollno = 102689
Name = RATHEESH
Total = 350
Percentage = 70

```
class percentage extends marks
{
    int per;

    percentage(int r, String n, int t, int p)
    {
        super(r,n,t); //call super class(marks) constructor
        per = p;
    }
    void dispdatap()
    {
        dispdatam(); // call dispdatap of marks class
        System.out.println("Percentage = " + per);
    }
}
class Multi_Inhe
{
    public static void main(String args[])
    {
        percentage stu = new percentage(102689, "RATHEESH", 350, 70); //call constructor percentage
        stu.dispdatap(); // call dispdatap of percentage class
    }
}
```

Superclass References and Subclass Objects

- A sub class has an 'is a' relationship with its superclass. This means that a sub class is a special kind of its super class.
- When we talk in terms of objects, a sub class object can also be treated as a super class object. And hence, we can assign the reference of a sub class object to a super class variable type.
- However, the reverse is not true. A reference of a super class object may not be assigned to a sub class variable.

```
class Base
{
    public void display()
    {
        System.out.println("Base class display method is called");
    }
}

class Derv1 extends Base
{
    public void display()
    {
        System.out.println("Derv1 class display method is called");
    }
}

class derv2 extends Base
{
    public void display()
    {
        System.out.println("Derv2 class display method is called");
    }
}

class polymorphism
{
    public static void main(String[] args)
    {
        Base ptr; //Bae class reference variable
        Derv1 d1 = new Derv1();
        Derv1 d2 = new Derv2();
        ptr = d1; // ptr contain reference of Derv1 object
        ptr.display();
        ptr = d2; // ptr contain reference of derv2 object
        ptr.display();
    }
}
```

Session 34-36

Method overriding:

- * 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.
- * Method overriding occurs only when the names and the type signatures of the two methods are identical. If they are not, then the two methods are simply overloaded.

super keyword:

- * using “super” keyword, we can access the superclass version of an overridden method.
- * “super” keyword can be used within a subclass constructor to invoke the superclass constructor. A subclass constructor is used to construct the instance variables of both the subclass and the superclass.
- * It can also be used in situations where member names of a subclass hide members by the same name in the superclass.

final keyword:

“final” variable:

- * Variable declared as final prevents its contents from being modified.
- * Initialize a final variable when it is declared.

Syntax: **final datatype symbolic_name = constant;**
(e.g.) final int PI=3.1459;

- * There are many things which is required repeatedly and if we want to make changes then we have to make these changes in whole program where this variable is used.
- * final variables behave like class variables and they do not take any space on individual objects.

“final” method: (prevent overriding)

- * All methods and variables can be overridden by default in subclasses.
- * Preventing the subclasses from overriding the members of the superclass.

“final” classes: (prevent inheritance)

- * Preventing a class being further subclassed for security reasons (or)
A class that cannot be subclassed is called a final class.

(e.g.) final class c1 {.....}
 final class c2 extends c3 {.....}

// Any attempt to inherit the above classes will cause an error and the
// compiler will not allow it

Dynamic method dispatch: This is the mechanism by which a call to an overridden method is resolved at run time, rather than compile-time.

```
class A
{
    void callme()
    {
        System.out.println("Inside A");
    }
}
class B extends A
{
    void callme()
    {
        System.out.println("Inside B");
    }
}
class C extends A
{
    void callme()
    {
        System.out.println("Inside C");
    }
}
```

Session 37-39

Abstract classes:

- * Abstract classes can't be instantiated but can be subclassed.
- * Abstraction refers to the ability to make a class abstract in OOP. It does not provide 100% abstraction(hiding complexity) because it can also have concrete method.
- * If the class is abstract and cannot be instantiated, the class does not have much use unless it is a subclass.
- * A parent class contains the common functionality of a collection of child classes, but the parent class itself is too abstract to be used on its own.
- * An abstract classes can have both abstract methods and concrete methods.
- * Abstract classes can have constructors, member variables and normal methods as similar to a class
- * When you extend abstract class with abstract method, you must define the abstract method in the child class, or make the child class abstract.

```
abstract class Animal{
.....
}
abstract class Animal-with-Backbones extends Animal{
.....
}
class Mamal extends Animal-with-Backbones{
Mamal(String name) {}
.....
}
class Abc{
Public static void main(String arg[]){
Animal a = new Animal(); //Not Valid

Mamal j= new Mamal(); // Valid
}
```

When to use abstract methods and abstract classes?

- * Abstract methods are usually declared where two or more subclasses are expected to do a similar thing in different ways through different implementations. These subclasses extend the same abstract class and provide different implementations for the abstract methods.
- * Abstract classes are used to define generic types of behaviors at the top of an Object-Oriented programming class hierarchy, and use its subclasses to provide implementation details of the abstract class.

Declaring a method as abstract has two results:

- * The class must also be declared abstract. If a class contains an abstract method, the class must be abstract as well.
- * Any child class must either override the abstract method or declare itself abstract.

Object Class

- **Object** class is present in **java.lang** package. Every class in Java is directly or indirectly derived from the **Object** class.
- If a Class does not extend any other class then it is direct child class of **Object** and if extends other class then it is an indirectly derived. Therefore the Object class methods are available to all Java classes. Hence Object class acts as a root of inheritance hierarchy in any Java Program.

List of Java Object Class methods

- `clone()`- Create and return a copy of this object.
- `equals()`- Indicates whether some other object is “equal” to this object.
- `finalize()`- Called by Garbage collector on an object when garbage collection determines that there are no more reference to the object.
- `getClass()`- returns the runtime class of an object.
- `hashCode()`- return the hash code value for the object.
- `notify()`- Wakes up a single thread that is waiting on this object’s monitor.
- `notifyAll()`- Wake up all the threads that are waiting on this object’s monitor.
- `toString()`-Returns a string representation of the object.
- `wait()`- Causes current thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object.