Classes and Objects: Class declaration and Modifiers, Class Members, Declaration of Class Object, Object Creation, Access control for Class Members, Defining methods, Overloaded methods, Recursive methods, Constructor, Constructor overloading, static keyword, this keyword.

Inheritance: Types of Inheritance, Deriving classes using Extends keyword, Method Overloading, Super keyword, Final keyword, Polymorphism- Abstract classes and methods-Overloading-Overriding-final methods and classes

Class declaration, Declaration of Class Object, Object Creation, Defining methods, Access control for Class Members

1) Class & Object:

Class is a template or blueprint for creating objects. An object is an instance of a class.Objects have state and behavior.

To declare a class in Java, we use the Java keyword class. The keywords in Java are lowercase letters. The class keyword is followed by class name.

General syntax for declaring a class in Java is as shown below:

```
class classname
{
   datatype instancevariable1;
                 .
   datatype instancevariablen;
   returntype methodname1(parameter list)
   {
         //statements
    }
   returntype methodnamen(parameter list)
   {
          //statements
    }
}
```

Syntax for object

ClassName object-reference = new ClassName();

Eg: student s1=new student();

student is classname

s1 is object reference

new is used to allocate memory dynamically

student() is constructor

//Write a program to demonstrate Class and Objects

```
class student{
int rno=1;
String name=”jp”;
}
class sample{
public static void main(String[] args){
student s1=new student();
System.out.println(s1.rno);
System.out.println(s1.name);
}}
```

Method Declaration

Syntax:returntype method name(arg list)

```
                {
                //statements
                }
```

Eg:

- ✓ void insert(int r,String n)
```
    {
            //statements
    }
```
- ✓ int add(int a,int b)
```
    {
            //statements
            return
    }
```

//Write a program to demonstrate class methods

```
class student{
int rno;
String name;
void insert(int r,String n)
{
        rno=r;
        name=s;
}
void display()
{
        System.out.println(rno+””+name);
}
}
class sample{
public static void main(String[] args){
student s1=new student();
student s2=new student();
s1.insert(1,”jp”);
s2.insert(2,”oop”);
s1.display();
s2.display();
}}
```

Overloaded Methods:

Method overloading takes place when a class contains multiple methods with the same name but varying number of parameters or types of parameters.

```
//write a program to demonstrate method overloading
class Addition
{
  void sum(int a, int b)
  {
   System.out.println("Sum of two integers is: "+(a+b));
   }
 void sum(float a, float b)
  {
   System.out.println("Sum of two floats is: "+(a+b));
   }
  void sum(int a, int b,int c)
  {
   System.out.println("Sum of three integers is: "+(a+b+c));
   }
}
class methodoverloading{
public static void main(String[] args){
Addition a1=new Addition();
a1.sum(2,4);
a1.sum(1.0,2.4);
a1.sum(1,2,3);
}
}
```

Output:

```
Sum of two integers is:6
Sum of two floats is:3.4
Sum of three integers is:6
```

Recursive methods

Recursion in java is a process in which a method calls itself continuosly,A method that calls itself is called recursive method.

//write a program to demonstrate recursive method

```
class factorial
{
        int fact(int n)
        {
        if (n!=0)
                return (n*fact(n-1));
        else
                return 1;
        }
public static void main(String[] args){
int n=5,r;
r=fact(n);
System.out.println("factorial=" +r);
}
}
```

Output:

```
factorial=120
```

Constructor

A constructor is a special method which has the same name as class name and that is used to initialize the objects (fields of an object) of a class. A constructor has the following characteristics:

- ✓ Constructor has the same name as the class in which it is defined.
- ✓ Constructor doesn't have a return type, not even void. Implicit return type for a constructor is the class name.
- ✓ Constructor is generally used to initialize the objects of a class.

Syntax for creating a constructor is as shown below:

```
ClassName( [Parameters List] )
{
  //Code of the constructor
}
```

Eg:

```
sample()
{
  //Code of the constructor
}
```

//Write a program to demonstrate constructor

```
class sample{
String name;
sample()
{
        System.out.println("Zero parameter constructor");
        name="jp";
}
sample(String l)
{
        System.out.println("Parameterized constructor");
        name=l;
}
        }
class constructorDemo
{
        public static void main(String[] args){
        sample s1=new sample();
        System.out.println(s1.name);
        sample s2=new sample("oop");
        System.out.println(s2.name)
}
}
```

Types of Constructor:

Based on the number of parameters and type of parameters, constructors are of three types:
o Parameter less constructor or zero parameter constructor
o Parameterized constructor
o Copy constructor

Parameter less constructor: As the name implies a zero parameter constructor or parameter less constructor doesn't have any parameters in its signature. These are the most frequently found constructors in a Java program. Let's consider an example for zero parameter constructor:

```
class sample{
sample()
{
        System.out.println(“Zero parameter constructor”);
        name=”jp”;
}
}
```

In the above example, sample() is a zero parameter or parameter less constructor

Parameterized constructor: This type of constructor contains one or more parameters in its signature. The parameters receive their values when the constructor is called. Let’s consider an example for parameterized constructor:

```
class sample{
sample(String l)
{
        System.out.println(“Parameterized constructor”);
        name=l;
}
}
```

In the above example, the sample() constructor accepts a single parameter.

Constructor Overloading:

As constructor is a special type ofe method, constructor can also be overloaded. Several constructors can be defined in the same class given that the parameters vary in each constructor. It is similar to method overloading we can construct two or more constructors with different parameters.

//Write a program to demonstrate constructoroverloading

```
class sample{
String name;
sample()
{
        System.out.println(“Zero parameter constructor”);
        name=”jp”;
}
sample(String l)
{
        System.out.println(“Parameterized constructor”);
        name=l;
}
        }
class constructorDemo
{
        public static void main(String[] args){
        sample s1=new sample();
        System.out.println(s1.name);
        sample s2=new sample(“oop”);
        System.out.println(s2.name)
}
}
```

output:

```
Zero parameter constructor
jp
Parameterized constructor
oop
```

Static keyword-static variable and static method

In Java programs, static keyword can be used to create the following:
1. Class variables
2. Class methods
3. Static blocks

static Class Variables:

The static keyword is used to create one of the three types of variables called as class variables. A class variable is a variable declared inside a class and outside all the methods and is marked as static.
Syntax for declaring a class variable or a static variable is shown below:

```
static data-type variable-name;
```

Example for declaring a class variable is shown below:

```
static int id;
```

An instance variable is created separately for every object of the class. But a static class variable is created only once inside the memory and the same is shared among all the objects of a class.

Static Class Methods:

All the non-static methods inside a class are known as instance methods and all the static methods inside a class are known as class methods. A class method is generally used to process class variables. A class method has the following limitations:
1. A class method (static method) can access only other class methods.
2. A class method can access only class variables (static variables).
3. this and super cannot be used in class methods.

A class method can be created using the following syntax:

```
static return-type method-name(parameters-list)
{
  //statements
}
```

Consider the following code segment to demonstrate the difference between instance variables and class variables and static class method

```
class student
{
        int rno;
        String name;
        static String c="jp";
        static void valuechange()
```

```
	{
		c=”oop”;
	}
student(int r,String n)
{
	rno=r;
	name=n;
}
void display()
{
	System.out.println(rno+” “+name+” ”+c);
}
}
class staticdemo
{
public static void main(String[] args){
student s1=new student(1,”aaa”);
student s2=new student(2,”bbb”);
s1.display();
s2.display();
student.valuechange();
System.out.println(student.c);
}
}
```

Output:

```
1 aaa jp
2 bbb jp
oop
```

Static Blocks:

A static block is a block of statements prefixed with static keyword. The syntax for creating a static block is shown below:

```
static
{
  //Statements
}
```

An important property of a static block is, the statements in a static block are executed as soon as the class is loaded into memory even before the main method starts its execution. A typical use of static blocks is initializing the class variables.

//Write a program to demonstrate static block

```
class staticblock{
static String c;
static int a;
static
{
        c=”jp”;
        a=1;
}
public static void main(String[] args){
System.out.println(c);
System.out.println(“Value is ”+a);
}
}
```

Output:

```
jp
Value is 1
```

This keyword

this keyword in Java is used to refer current object on which a method is invoked. Using such property, we can refer the fields and other methods inside the class of that object.
The ‘this’ keyword has two main uses which are listed below:
1. It is used to eliminate ambiguity between fields and method parameters having the same name.
2. It is used for chaining constructors.

program with name ambiguity

```
class sample
{
int a;
sample(int a)
{
        a=a;//ambiguity
}
}
class constructor
{
public static void main(String[] args){
sample s1=new sample(8);
System.out.println(“The value is ”+s1.a);
}
}
Output:error
```

```
class sample
{
int a;
void display(int a)
{
      a=a;//ambiguity
}
}
class constructor
{
public static void main(String[] args){
sample s1=new sample();
s1.display(8);
System.out.println(“The value is ”+s1.a);
}
}
Output:error
```

Note:To remove name ambiguity ‘this’ keyword is used.

```
class sample
{
int a;
sample(int a)
{
        this.a=a;
}
}
class constructor
{
public static void main(String[] args){
sample s1=new sample(8);
System.out.println("The value is "+s1.a);
}
}
```

```
class sample
{
int a;
void display(int a)
{
      this.a=a;
}
}
class constructor
{
public static void main(String[] args){
sample s1=new sample();
s1.display(8);
System.out.println("The value is "+s1.a);
}
}
```

Access control/Access Modifiers

Access control is a mechanism, an attribute of encapsulation which restricts the access of certain members of a class to specific parts of a program. Access to members of a class can be controlled using the access modifiers. There are four access modifiers in Java. They are:

- o public
- o protected
- o default
- o private

If the member (variable or method) is not marked as either public or protected or private, the access modifier for that member will be default.

We can apply access modifiers to classes also. Among the four access modifiers, private is the most restrictive access modifier and public is the least restrictive access modifier.

Syntax for declaring a access modifier is shown below:

access-modifier data-type variable-name;

Example for declaring a private integer variable is shown below:

private int side;

In a similar way we can apply access modifiers to methods or classes although private classes are less common.

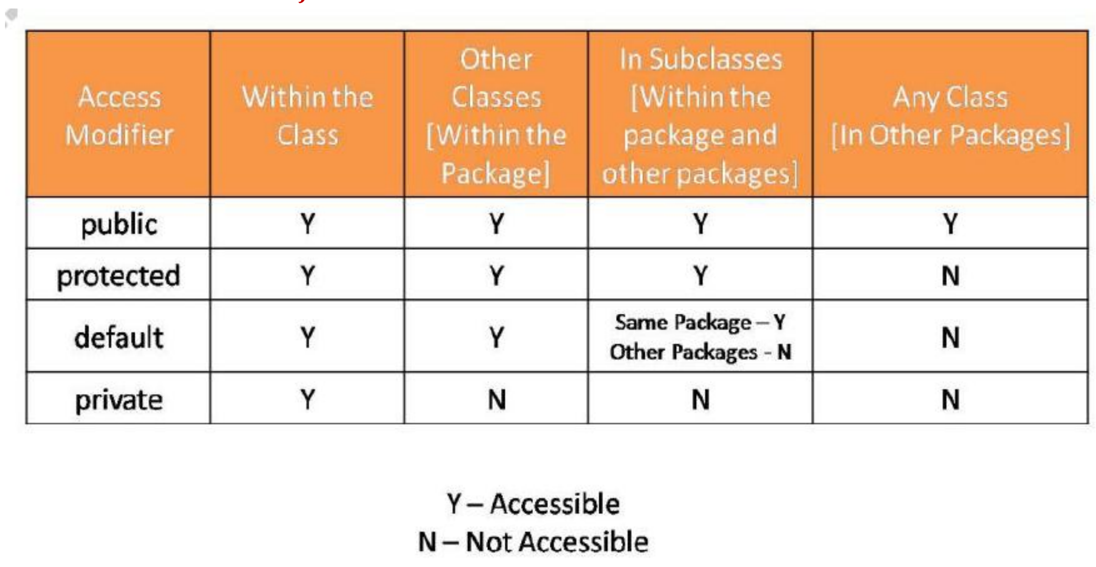
//Demonstration of accessmodifier

```
class data
{
  private String name;
  public String getname()
   {
        return this.name;
   }
  public void setname(String name)
  {
          this.name=name;
  }
class sample
{
          public static void main(String[] args){
          data d=new data();
          d.setname("JP");
          System.out.println(d.getname());
}
}
```

output:

```
JP
```

Access Modifiers in java



Access Modifier	Within the Class	Other Classes [Within the Package]	In Subclasses [Within the package and other packages]	Any Class [In Other Packages]
public	Y	Y	Y	Y
protected	Y	Y	Y	N
default	Y	Y	Same Package – Y Other Packages - N	N
private	Y	N	N	N

Y – Accessible
N – Not Accessible

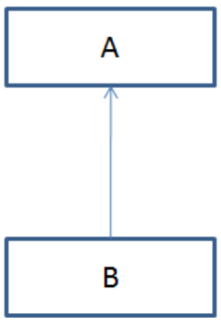
Inheritance: Types of Inheritance, Deriving classes using Extends keyword,

Types of inheritance:

1) Simple inheritance
2) Multi level inheritance
3)Multiple inheritance
4)Hierarchial inheritance
5)Hybrid inheritance

Simple inheritance:

This is the most frequently used and most simple of the five types of inheritance. In this type of inheritance there is a single derived class which inherits from a single base class. This can be represented pictorially as shown below:



In the above figure, A is base class and B is derived class. This can be converted to Java code as given below:

```
class A
{
//Members of class A
}
class B extends A
{
//Members of class B
}
```

//write a program to demonstrate single inheritance

$$\begin{tikzcd} \textbf{parent} \\ \textbf{child} \arrow[u] \end{tikzcd}$$

```
class parent
{
        void parentmethod()
        {
                System.out.println("parent method");
        }
}
class child extends parent
{
        void childmethod()
        {
                System.out.println("child method");
        }
}
class singleinheritance{
```

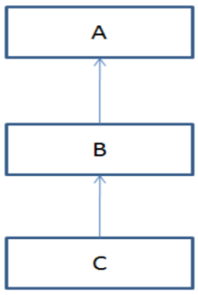
```
public static void main(String[] args){
child c =new child();
c.parentmethod();
c.childmethod();
}
}
```

Output:

```
parent method
child method
```

Multi-level Inheritance

In this type of inheritance there are several classes in the hierarchy forming multiple levels. Each level contains a base class and a derived class. Multi-level inheritance can be represented as shown below:



```
class A
{
  //Members of class A
}
class B extends A
{
 //Members of class B
}
class C extends B
{
 //Members of class C
}
```

//Write a program to demonstrate multiple inheritance

$$\begin{tikzcd}
\text{Grandfather} \\
\text{Father} \arrow[u] \\
\text{Son} \arrow[u]
\end{tikzcd}$$

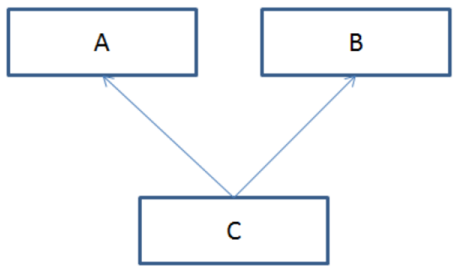
```
class GrandFather
{
  GrandFather()
  {
   System.out.println("I am the grandfather!");
  }
}
class Father extends GrandFather
{
 String name="JP";
  Father()
      {
       System.out.println("I am the father! I inherit from Grandfather");
       }
}
public class Son extends Father
 {
  Son()
      {
    System.out.println("I am the son and I inherit from my father.");
       }
}
class multiplelevelinheritance
{
   public static void main(String[] args) {
   Son s1 = new Son();
   System.out.println(s1.name)
}
}
```

Output:

```
I am the grandfather!
I am the father! I inherit from Grandfather
I am the son and I inherit from my father.
JP
```

Multiple Inheritance

In this type of inheritance a single derived class can inherit from two or more base classes. Multiple inheritance can be represented as shown below:



In the above figure A and B are base classes and C is the derived class. Above figure can be converted to Java code as given below:

```
class A
{
  //Members of class A
}
class B
{
 //Members of class B
}
class C extends A,B
{
 //Members of class C
}
```

It is important to remember that in Java, we can inherit from more than one class. So in the above code class C extends A, B is syntactically incorrect.

What types of inheritance does Java support?

Java supports all types of inheritance mentioned above except multiple inheritance. Java doesn't allow multiple inheritance in case of classes but it allows multiple inheritance in case of interfaces. As interfaces are not yet introduced, I will cover this in future articles.

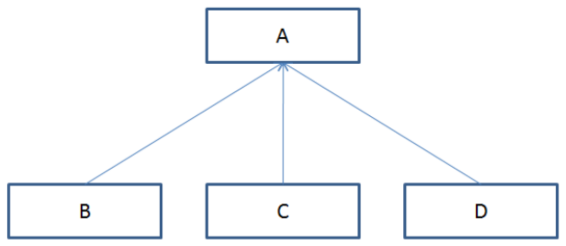
Why multiple inheritance (among classes) has been removed in Java? Well, there are certain problems in multiple inheritance. For example, consider that a derived class C inherits from classes A and B (multiple inheritance). Suppose there are is a variable by the name X in both classes A and B. Now, if I access X in class C (which is valid due to inheritance) where does X value come from. Does it come from A or B?

Above problem is also valid for methods and not only for variables. Such situations leads to ambiguity. To eliminate such ambiguous situations, Java designers voted for removing multiple inheritance in the case of classes.

Hierarchical Inheritance

In this type of inheritance, two or more derived classes inherit from a common base class.

Hierarchical inheritance can be represented as shown below:



In the above figure A is the base class and B, C and D are derived classes. Above figure can be converted to Java code as given below:

```
class A
{
//Members of class A
}
class B extends A
{
//Members of class B
}
class C extends A
{
//Members of class C
}
class D extends A
{
//Members of class D
}
//write a program to demonstrate hierarchial inheritance
```

$$\begin{tikzcd}
 & \text{father} & \\
\text{Son} \arrow[ur] & & \text{Daughter} \arrow[ul]
\end{tikzcd}$$

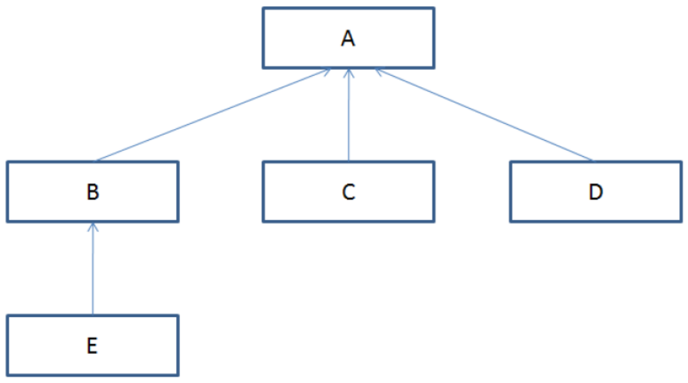
```
class Father {
 String name;
 String addr;
 Father() {
   name = "jp";
   addr = "bvrm";
 }
}
class Son extends Father {
 Son() {
   System.out.println("I am the Son");
   System.out.println("name is " + name + " and I am from " +addr);
 }
}
class Daughter extends Father {
 Daughter() {
   System.out.println("I am the Daughter");
   System.out.println("name is " + name + " and I am from " +addr);
 }
}
class hierarchialinheritance {
 public static void main(String[] args) {
   Son s = new Son();
   Daughter d = new Daughter();
 }
}
```

Output:

```
I am the Son
name is JP and I am from bvrm
I am the Daughter
name is JP and I am from bvrm
```

Hybrid Inheritance

As the name itself implies, hybrid inheritance is a combination of any two or more of the above mentioned four types of inheritance. Hybrid inheritance can be represented as shown below:



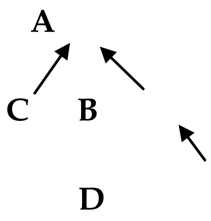
In the above figure you can see hierarchical, single and multi-level inheritance. Above figure can be converted to Java code as given below:

Note: It is not mandatory to write the classes in a particular order in Java. You can write in any order that you want.

```
class A
{
//Members of class A
}
class B extends A
{
//Members of class B
}
class C extends A
{
//Members of class C
}
class D extends A
{
```

```
//Members of class D
}
class E extends B
{
   //Members of class E
}
```

//Write a program to demonstrate hybrid inheritance



```
class A
{
  A() {
    System.out.println("I am super class");
  }
}
class B extends A
{
  B() {
    System.out.println("I am the class B ");
  }
}
class C extends A
{
  C() {
    System.out.println("I am the class C ");
  }
}
class D extends B
{
  D() {
    System.out.println("I am the D class ");
  }
}
class Hybrid {
  public static void main(String[] args) {
    D d = new D();
    B b = new B();
    C c = new C();
  }
}
```

Output:

```
I am the super class
I am the class B
I am the class D
I am the super class
I am the class B

I am the super class
I am the class C
```

✓ **Advantages of Inheritance**

Following are the advantages of using inheritance in programs:

- Reusability: The code and methods declared in the base class can be re used in the derived class.
- Extensibility: Derived classes can be extended to provide new functionality of their own.
- Data Hiding: Base class can hide some of its data and code by making them private.
- Overriding: Derived classes can have methods with same signature as in base class. The methods in the derived class provides suitable functionality which might be different from the methods available in the base class.

✓ **Disadvantages of Inheritance**

Following are the disadvantages of using inheritance:

- More time taken for the control to reach the base class from derived classes when there are several levels of inheritance.
- Tight coupling between the base class and derived class.
- Increase in maintenance time as changes done to base class may require changes to be performed in the derived class.

---

super keyword

Super keyword (in sub class) is to access the hidden members of its immediate super class.
Following are the uses of super keyword:

- To refer the immediate super class constructor
- To refer the immediate super class members.
  To refer the immediate parents instance variables.

To understand this let's consider the following example:

//Demonstartion of super keyword

```
class A
{
int x;
 public void display()
 {
        System.out.println("This is display in A");
 }
}
class B extends A
{
        int x;
        public void display()
        {
                System.out.println("Value of x in A is: " + super.x);
                super.display();
                System.out.println("This is display in B");
        }
}
class Main
{
public static void main(String[] args)
{
B obj = new B();
obj.display();
}
}
```

output:

```
Value of x in A is: 0
This is display in A
This is display in B
```

//Write a program to refer class constructor

```
class A
{
        A()
        {
        System.out.println("I am constructor in class A");
        }
}
class B extends A
{
        B()
        {
        super();
        System.out.println("I am constructor in class B");
        }
}
class superconstructor
{
public static void main(String[] args)
{
        B b1=new B();
}
}
```

Output:

```
I am constructor in class A
I am constructor in class B
```

----

final Keyword

Following are the uses of final keyword in Java:

- To declare constant values
- To make a method non-overridable
- To make a class non-inheritable

The second and third uses of final are applicable only in the context of inheritance.
Let's look at each of the uses of final keyword in detail.

Declaring constant values

The first use of final is to declare constant values in Java programs.
Syntax for declaring a constant is shown below:

```
final data-type variable-name = value;
```

An example for declaring constants is given below:

```
final float PI = 3.1415;
```

In Java, you have to assign the value for the constant in the declaration time itself.

final for methods

The final keyword can be used to make a method in base class non-overridable. Syntax for making a method non-overridable is given below:

```
final return-type method-name(parameters-list)
 {
   //Body of the method
  }
```

So, we can say that an abstract method cannot be declared as final.

final for classes

The final keyword can be used to make a class non-inheritable or non-extendable. Syntax for making a class non-inheritable is given below:

```
final class Class-Name
{
 //Members of class
}
```

//Write a program to demonstrate final in inheritance

$$\begin{tikzcd}
& \text{Shape} & \\
\text{Circle} \arrow[ur] & & \text{Rectangle} \arrow[ul]
\end{tikzcd}$$

```
final class Shape
{
  void area()
  {
   System.out.println("Area of shape");
  }
}
class Circle extends Shape
{
  void area()
  {
    System.out.println("Area of circle");
  }
}
class Rectangle extends Shape
{
  void area()
  {
    System.out.println("Area of rectangle");
   }
}
public class finalkeyword
{
public static void main(String[] args)
{
Shape s;
s = new Circle();
s.area();
s = new Rectangle();
s.area();
}
}
```

Above code gives errors because the classes Circle and Rectangle are trying to inherit the Shape class which was declared final (non-inheritable).

Method overriding:

It is a process of overriding or redefining a method that was already defined in the parent/super class.

Method overriding is possible only when the following things are satisfied:

- A class inherits from another class (inheritance).
- Both super class and sub class should contain a method with same signature.

Note: Method overriding does not depend up on the return type of the method and the access specifiers like (public, private, protected etc..).

```
class A
{
        public void display()
        {
                System.out.println("This is display method of class A");
        }
}
class B extends A
{
  public void display()
  {
    System.out.println("This is display method of class B");
  }
}
public class Methodoverriding
{
public static void main(String[] args)
{
  B obj = new B();
  obj.display();
}
}
```

In the above example the method display in both classes A and B have same method signature (method name + number of parameters). So, the display method in class B hides the display method in class A.

Output of the above program is:

```
This is display method of class B
```

Abstraction

Abstraction is a feature of OOPs. The feature allows us to hide the implementation detail from the user and shows only the functionality of the programming to the user. Because the user is not interested to know the implementation. It is also safe from the security point of view.

Let's understand the abstraction with the help of a real-world example. The best example of abstraction is a car. When we derive a car, we do not know how is the car moving or how internal components are working? But we know how to derive a car. It means it is not necessary to know how the car is working, but it is important how to derive a car. The same is an abstraction.

The same principle (as we have explained in the above example) also applied in Java programming and any OOPs.

Abstract classes and methods

abstract keyword

Following are the uses of abstract keyword in Java:

- Used to create abstract methods

- Used to create abstract classes

Creating abstract methods

Sometimes while creating hierarchies, a method inside a super class might not be suitable to have any kind of implementation. Such methods can be declared as abstract using the abstract keyword.

Syntax for creating an abstract method is as follows:

abstract return-type method-name(parameters-list);

Following are the rules associated with abstract methods:

- Abstract methods doesn't contain any body. It contains only method prototype.
- Abstract methods can only be declared inside abstract classes.

Creating abstract classes

A class declared using the abstract keyword is known as an abstract class.

The syntax forcreating an abstract class is as shown below:

```
abstract class ClassName
{
        //Members of the class
}
```

Following are the important points that should be remembered about abstract classes:

- A class which contains at least one abstract method should be declared as abstract class.
- An abstract class can contain both concrete (non-abstract) and abstract methods.
- Abstract classes cannot be instantiated i.e, objects cannot be created for abstract classes.
- All abstract methods in a base class must be overridden in the derived class.

//example which demonstrates the use of abstract classes:

```
abstract class Shape
{
   abstract void area();
}
class Circle extends Shape
{
  void area()
  {
  System.out.println("Area of circle");
   }
}
class Rectangle extends Shape
{
 void area()
  {
   System.out.println("Area of rectangle");
  }
}
public class abstractdemo
{
public static void main(String[] args)
{
Circle c= new Circle();
c.area();
Rectangle r = new Rectangle();
r.area();
}
}
```

Output of above Java code is:

```
Area of circle
 Area of rectangle
```