Unit-III – Classes, Objects and Methods

Introduction: Object is the physical as well as logical entity where as class is the only logical entity.

Class: Class is a blue print which is containing only list of variables and method and no memory is allocated for them. A class is a group of objects that has common properties.

A class in java contains:

1. Data Members
2. Method
3. Constructor
4. Block
5. Class and Interface

Object: Object is a instance of class, object has state and behaviors.

An Object in java has three characteristics:

State

Behavior

Identity

State: Represents data (value) of an object.

Behavior: Represents the behavior (functionality) of an object such as deposit, withdraw etc.

Identity: Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But, it is used internally by the JVM to identify each object uniquely.

Class is also can be used to achieve user defined data types.

In real world many examples of object and class like dog, cat, and cow are belong to animal's class. Each object has state and behaviors. For example a dog has state – color, name, height, age as well as behaviors – barking, eating, and sleeping.

Car, bike, truck these all are belongs to vehicle class. These Objects have also different states and behaviors. For Example car has state - color, name, model, speed, Mileage. as we;; as behaviors - distance travel

1. Defining a class

Java is a true object oriented programming language therefore each variable and method must be defined within a class. Methods and variables inside the class are known as members.

In Java variables or data members are called fields and functions are called methods.

Class creates objects and objects used methods to access data members.

The keyword **"class"** is used to define a class. A class is a user defined data type. Following is the syntax of defining a class.

Syntax:

```
class <ClassName> extends <SuperClassName>
{
  [declaration of variables;]
  [declaration of methods;]
}
```

There is no semicolon at the end of the class. The definition of class creates only the new abstract datatype.

<ClassName> and <SuperClassName> are any valid java identifier. The keyword **extends** indicates

that the properties of the super class are extended to the current class.

E.g.

```
class Test
{
  int count=1;
  void display()
  {
    System.out.println("Count="+count);
  }
}
```

Fields/Variables Declaration

Data is encapsulated in a class by placing data fields inside the body of the class definition.

```
class TestStudent
{
  int rollno;
  String name;
}
```

The variables inside the class are known as instance variables because they are created whenever an object of the class is instantiated. There is no storage space is created for class in the memory.

Methods/functions Declaration

A class only data fields has no life. The objects created by such a class can not respond to messages. We must therefore add methods that are necessary for manipulating the data contained in the class. To accessthe instance variables we have to include methods in the class.

The general form of a method declaration is:

```
access_modifier return_type methodName(list of
parameters)
{
    Statement-1;
    Statement-2;
    ::::::::
    ::::::::
    Statement-n;
}
```

Method declaration have four basic parts:

1. The name of the method.
2. The type of the value the method returns.
3. A list of parameters
4. The body of the method

If the method doesn't return a value then void is mention at the place of return type.

The parameter list always encloses within parenthesis, these parameters are separated by commas.

The body of the method describes the operations to be performed.

The class body must contain one or more methods.

Creating objects

Once a class has been defined we can create variables of that type known as objects. In Java objects are created by using **new** operator. The new operator allocates the memory at runtime and support the dynamic nature of Java. The following is the syntax of creating an object.

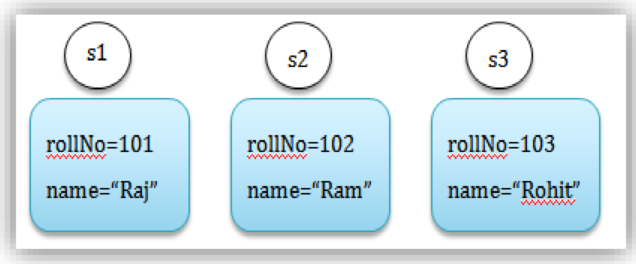
Syntax:

```
ClassName objectName=new ClassName(list of parameters);
```

E.g.

```
TestStudent s1=new TestStudent();
TestStudent s2=new TestStudent();
TestStudent s3=new TestStudent();
```

- It is important to understand that each object has its own copy of instance variables.
- A TestStudent class contains two variables i.e. rollNo and name. The s1,s2 and s3 objects maintain separate copies of this two variables as shown in following fig.



- This means that any changes to the variables of one object have no effect on the variables of another.

The following is the example of Class and object.

```
class TestStudent
{
    int rollNo=101;
    String name="Raj";
    void display()
    {
      System.out.println("Roll No: "+rollNo);
      System.out.println("Name: "+name);
    }
    public static void main(String[] args)
    {
            TestStudent s1=new TestStudent();
            s1.display();
    }
}
```

O/P

```
Roll No: 101
Name: Raj
```

Multiple classes in a single program.

A Java program may contain any number of classes but at most one class can be declared as public.

If there is a public class then the name of a program and name of public class must be the same. If there is no public class then we can use any name of the program.

If a Java program contains more than one class then program name must be the same of a class which contains main() method.

The following program shows how to create and use multiple classes in a single program.

```
class Addition
{
        int x=10;
        int y=20;
        void add()
        {
         System.out.println("Addition : "+(x+y));
        }
}
class Subtraction
{
        int x=20;
        int y=10;
        void sub()
        {
         System.out.println("Substraction : "+(x-y));
        }
}
class MultiClassTest
{
        public static void main(String[] args)
        {
                Addition a=new Addition();
                Subtraction s=new Subtraction();
                a.add();
                s.sub();
        }
}
```

O/P

```
Addition : 30
Subtraction : 10
```

2. Visibility Controls / Access Modifiers

However it may be necessary in some situations to restrict the access to certain variables and methods from outside the class. Java provides the visibility modifiers to the instance variables and methods.

Access specifiers make us to understand where to access the feature and where not to access the features. Access specifiers provides features accessing controlling mechanism among the classes and interfaces.

Access specifiers are those which are applied before data members or methods of a class. In java programming we have four access specifiers they are:

1. private
2. protected
3. public
4. default (not a keyword)

1. private

The private is an access modifier applicable for variables and methods only but not for classes.

Private Fields/variables and methods have the highest degree of protection.

Private fields and methods are accessible only within class. They can not be inherited by subclass and therefore not accessible in subclasses.

By mistake if we try to access private fields outside the class then it gives compile time error.

Simple example of private modifier.

```
class TestA
{
        private int data=40;
        private void message()
        {
                System.out.println("Hello Java");
        }
}
public class TestB
{
        public static void main(String[] args)
        {
                TestA obj=new TestA();
                System.out.println(obj.data);//CE
                obj.message();//CE
        }
 }
```

In above example we try to access private variable and method of TestA class into TestB class, it gives the compile time error.

```
        javac TestB.java
        CE: data has private access in TestA
        CE: message() has private access in TestA
```

2. **protected**

The protected is a access specifiers applicable for data members, methods and constructor but not for classes.

The protected access modifier is accessible within package and outside the package but through inheritance only.

Simple example of protected access modifier.

In this example, we have created the two packages mypack1 and mypack2. The TestA class of mypack1 package is public, so can be accessed from outside the package. But message() method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

TestA.java

```
package mypack1;
public class TestA
{
        protected int data=40;
        protected TestA()
        {
        }
        protected void message()
        {
                System.out.println("Hello Java");
        }
}
```

TestB.java

```
package mypack2;
import mypack1.*;
class TestB extends TestA
{
        TestB()
        {
        }
        public static void main(String[] args)
        {
                TestB obj=new TestB();
                obj.message();
        }
}
```

```
javac -d . TestA.java
javac -d . TestB.java
java mypack2.TestB
```

O/P

```
Hello Java
```

3. **public**

The public is a access specifier applicable for variables, methods, constructors and classes.

The public access modifier is accessible everywhere there is no restrictions. It has the widest scope among all other modifiers.

The following example shows the use of public modifier with variable, method and class.

TestX.java

```
package packTestX;
public class  TestX
```

```
{
 public int count=10;
 public void showMessage()
  {
   System.out.println("In showMessage() of TestX");
}
}
```

TestY.java

```
package packTestY;
import packTestX.*;
class TestY
{
public static void main(String[] args)
{
TestX t=new TestX();
System.out.println("TestX.count : "+t.count);
t.showMessage();
}
}
```

javac -d . TestX.java

javac -d . TestY.java

java packTestY.TestY

O/P

```
TestX.count : 10
In showMessage() of TestX
```

In the above example we have declared public variable, public method and public class and accessing these from outside the package.

3. Use of 'this' Keyword

"this" is a reference variable that refers to the current object. It is a keyword in java language represents current class object.

this keyword can be used to refer current class instance variable.

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

Why use this keyword ?

The main purpose of using this keyword is to differentiate the formal parameter and data members of class, whenever the formal parameter and data members of the class are similar then JVM get ambiguity (no clarity between formal parameter and member of the class)

To differentiate between formal parameter and data member of the class, the data member of the class must be preceded by "this".

If local variables(formal arguments) and instance variables are different, there is no need to use this keyword.

Syntax

this.data_member_of_current_class.

E.g.

this.emp_id;

this.emp_name;

Note: If any variable is preceded by "this" JVM treated that variable as class variable.

Following example shows the use of "this" keyword

```
class Student
{
 int emp_id;
 String emp_name;
 Student(int emp_id, String emp_name)
  {
    this.emp_id=emp_id;
    this.emp_name=emp_name;
  }
  void display()
```

```
   {
    System.out.println("Emp ID : "+emp_id);
    System.out.println("Emp Name : "+emp_name);
    }
}
class TestThis
{
 public static void main(String[] args)
  {
   Student s1=new Student(101,"Raj");
    Student s2=new Student(102,"Ram");
    s1.display();
    s2.display();
 }
}
```

O/P

```
Employee ID : 101
Employee Name : Raj
Employee ID : 102
Employee Name : Ram
```

The second use of **"this"** keyword is to invoke the current class method.

You may invoke the method of the current class by using the this keyword. If you don't use the this keyword, compiler automatically adds this keyword while invoking the method.

Syntax

```
this.methodOfCurrentClass();
```

E.g.

```
this.display();
this.calculate();
```

__this() can be used to invoked current class constructor.__

The this() constructor call can be used to invoke the current class constructor (constructor chaining).

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

The following program shows the use of this() method

```
class Student13
{
   int id;
   String name;
   Student13()
    {
   System.out.println("default constructor is invoked");
    }
   Student13(int id,String name)
   {
   this ();
   this.id = id;
   this.name = name;
   }
   void display()
    {
     System.out.println(id+" "+name);
    }
   public static void main(String args[])
   {
     Student13 e1 = new Student13(111,"karan");
     Student13 e2 = new Student13(222,"Aryan");
     e1.display();
     e2.display();
  }
}
```

O/P

```
default constructor is invoked
default constructor is invoked
111 karan
222 Aryan
```

4. Method Parameters

The term call by value means that the method gets just the value that caller provides. In contrast, call by reference means that the method gets the location of the variable that the caller provides. Thus a method can modify the value stored in a variable that is passed by reference but not in one that passed by the value.

The Java language always uses call by value. That means, the method gets a copy of all parameter values. In particular, the method can not modify the contents of any parameter variables that are passed to it. For example consider the following call.

```
Double percent=10;
raj.raiseSalary(percent);
```

In above example no matter how the method is implemented, we know that after the method call, the value of percent is still 10.

Let us look a little more closely at this situation. Suppose a method tried to triple the value of a method parameter.

```
public static void tripleValue(double x)
{
	x=3*x;
}
```

Let's call this method:

```
double percent=10;
tripleValue(percent);
```

However, this does not work. After the method call, the value of percent is still 10. Here is what happens:

1. x is initialized with a copy of the value of percent i.e. 10.
2. x is tripled- it is now 30. But percent is still 10.
3. The method ends, and the parameter variable x is no longer in use.

There are two kinds of parameters:

1. Primitive method parameters
2. Object reference

In above example we have seen that it is impossible for a method to change a primitive type parameter. The situation is different for object reference parameters. You can easily implement a method that triples the salary of an employee.

```
public static void tripleSalary(Employee x)
{
	x.raiseSalary(200);
}
```

When you call

```
raj=new Employee(...);
tripleSalary(harry);
```

Then the following happens

1. x initialized with a copy of the value of harry, that is, an object reference.
2. The raiseSalary() method applied to that object reference. The Employee object to which both x and harry refer gets its salary raised by 200.
3. The method ends , and the parameter variable x is no longer in use. Of course, the object reference variable harry continuous to refer to the object whose salary was tripled.

5. Overloading

Whenever same method name is exiting multiple times in the same class with different number of parameter or different order of parameters or different types of parameters is known as method overloading.

Why method Overloading

Suppose we have to perform addition of given number but there can be any number of arguments, if we write method such as a(int,

int)for two arguments, b(int, int, int) for three arguments then it is very difficult for you and other programmer to understand purpose or behaviors of method they can not identify purpose of method. So we use method overloading to easily figure out the program. For example above two methods we can write sum(int, int) and sum(int, int, int) using method overloading concept.

Syntax:

```
class  Class_Name
{
Returntype method()
{
..........
..........
}
Returntype  method(datatype1 variable1)
{
..........
..........
}
Returntype  method(datatype1 variable1,
datatype2 variable2)
{
..........
..........
}
Returntype  method(datatype2 variable2)
{
..........
..........
}
Returntype  method(datatype2 variable2,
datatype1 variable1)
{
..........
..........
}
}
```

Different ways to overload the method

There are two ways to overload the method in java

1. By changing number of arguments or parameters
2. By changing the data type

1. By changing number of arguments

In this example, we have created two overloaded methods, first sum method performs addition of two numbers and second sum method performs addition of three numbers.

```
class Addition
{
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[])
{
Addition obj=new Addition();
obj.sum(10, 20);
obj.sum(10, 20, 30);
}
}
```

Output

```
30
60
```

2. By changing the data type

In this example, we have created two overloaded methods that differs in data type. The first sum method receives two integer arguments and second sum method receives two float arguments.

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

```
void sum(float a, float b)
{
System.out.println(a+b);
}
public static void main(String args[])
{
Addition obj=new Addition();
obj.sum(10, 20);
obj.sum(10.05, 15.20);
}
}
```

Output

```
30
25.25
```

Why Method Overloaing is not possible by changing the return type of method?

In java, method overloading is not possible by changing the return type of the method because there may occur ambiguity. Let's see how ambiguity may occur:

because there was problem:

```
class Addition
{
 int sum(int a, int b)
 {
 System.out.println(a+b);
 }
 double sum(int a, int b)
 {
 System.out.println(a+b);
 }
 public static void main(String args[])
 {
 Addition obj=new Addition();
 int result=obj.sum(20,20); //Compile Time Error
 }
}
```

Explanation of Code

```
int result=obj.sum(20,20);
```

Here how can java determine which sum() method should be called

Note: The scope of overloading is within the class.

Any object reference of class can call any of overloaded method.

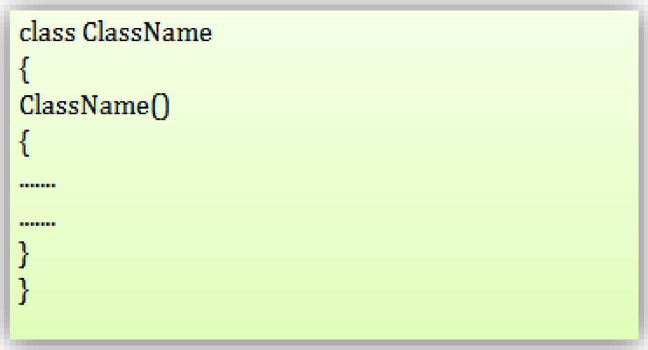
As for as real time application are concern overloaded method should exists with same name but with different implementation part.

6. Constructor and Overloading Constructors

A **constructor** is a special member method which will be called implicitly (automatically) by the JVM whenever an object is created.

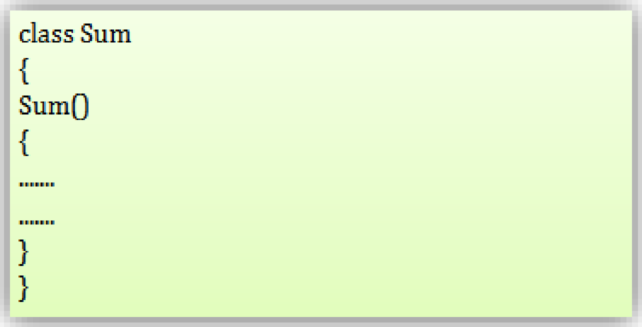
The purpose of constructor is to initialize an object called object initialization. Constructors are mainly create for initializing the object. Initialization is a process of assigning user defined values at the time of allocation of memory space.

The following is the syntax of defining constructor.



```
class ClassName
{
ClassName()
{
.......
.......
}
}
```

E.g.



```
class Sum
{
Sum()
{
.......
.......
}
}
```

Advantages of constructors:

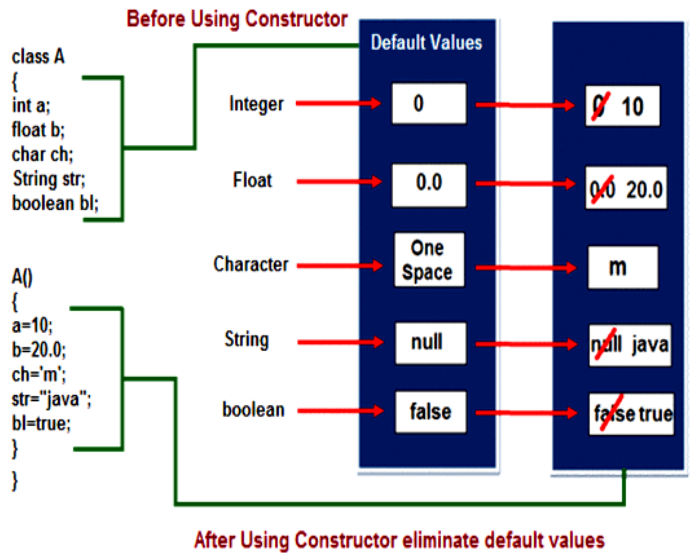
- A constructor eliminates placing the default values.
- A constructor eliminates calling the normal or ordinary method implicitly.

Rules or properties of a constructor

- Constructor will be called automatically when the object is created.
- Constructor name must be similar to name of the class.
- Constructor should not return any value even void also. Because basic aim is to place the value in the object. (if we write the return type for the constructor then that constructor will be treated as ordinary method).
- Constructor definitions should not be static. Because constructors will be called each and every time, whenever an object is creating.
- Constructor should not be private provided an object of one class is created in another class (Constructor can be private provided an object of one class created in the same class).
- Constructors will not be inherited from one class to another class (Because every class constructor is create for initializing its own data members).
- The access specifier of the constructor may or may not be private.

1. If the access specifier of the constructor is private then an object of corresponding class can be created in the context of the same class but not in the context of some other classes.
2. If the access specifier of the constructor is not private then an object of corresponding class

can be created both in the same class context and in other class context.



	Method	Constructor
1	Method can be any user defined name	Constructor must be class name
2	Method should have return type	It should not have any return type (even void)
3	Method should be called explicitly either with object reference or class reference	It will be called automatically whenever object is created
1	Method is not provided by compiler in any case.	The java compiler provides a default constructor if we do not have any constructor.

7. Types of constructors

Based on creating objects in Java constructor are classified in two types. They are

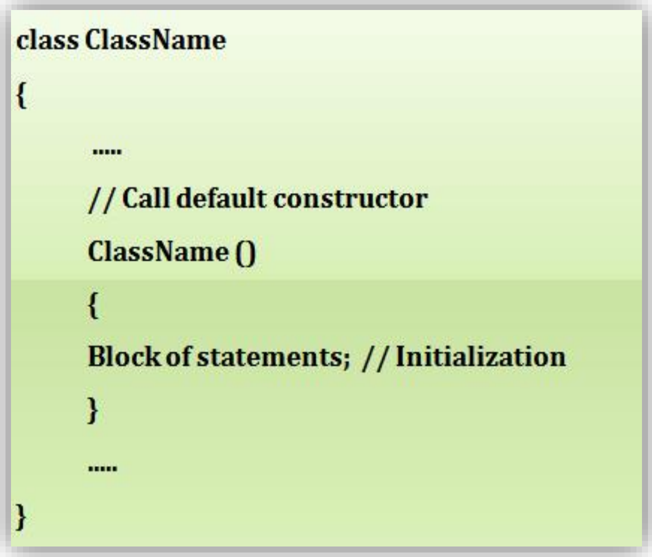
I. Default or no argument Constructor
II. Parameterized constructor.

Default Constructor

A constructor is said to be default constructor if and only if it never take any parameters.

If any class does not contain at least one user defined constructor than the system will create a default constructor at the time of compilation it is known as system defined default constructor.

Syntax:



```
class ClassName
{
        .....
        // Call default constructor
        ClassName ()
        {
        Block of statements; // Initialization
        }
        .....
}
```

Note: System defined default constructor is created by java compiler and does not have any statement in the body part. This constructor will be executed every time whenever an object is created if that class does not contain any user defined constructor.

Example of default constructor.

In below example, we are creating the no argument constructor in the Test class. It will be invoked at the time of object creation.

Example

```
//TestDemo.java
class Test
{
int a, b;
Test ()
{
```

```
System.out.println("I am from default Constructor...");
a=10;
b=20;
System.out.println("Value of a: "+a);
System.out.println("Value of b: "+b);
}
};
class TestDemo
{
public static void main(String [] args)
{
Test t1=new Test ();
}
};
```

Output

```
Output:
I am from default Constructor...
Value of a: 10
Value of b: 20
```

Purpose of default constructor?

Default constructor provides the default values to the object like 0, 0.0, null etc. depending on their type (for integer 0, for string null).

```
class Student
{
int roll;
float marks;
String name;
void show()
{
System.out.println("Roll: "+roll);
System.out.println("Marks: "+marks);
System.out.println("Name: "+name);
}
}
class TestDemo
{
public static void main(String [] args)
{
Student s1=new Student();
s1.show();
}
}
```

Output

```
Roll: 0
Marks: 0.0
Name: null
```

Explanation: In the above class, we are not creating any constructor so compiler provides a default constructor. Here 0, 0.0 and null values are provided by default constructor.

Parameterized constructor

Def1: If any constructor contain list of variable in its signature is known as paremetrized constructor.

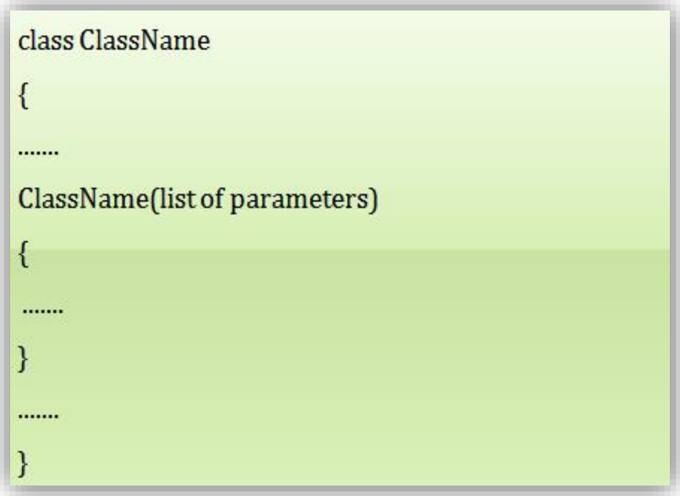
Def2: A constructor that have parameters is known as parameterized constructor.

A parameterized constructor is one which takes some parameters.

Why use parameterized constructor?

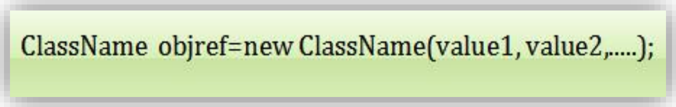
Parameterized constructor is used to provide different values to the distinct objects.

Syntax:



```
class ClassName
{
.......
ClassName(list of parameters)
{
 .......
}
.......
}
```

Syntax to call parametrized constructor



```
ClassName  objref=new ClassName(value1, value2,.....);
```

Example of Parametrized Constructor

```
class Test
{
int a, b;
Test(int n1, int n2)
{
System.out.println("I am from Parameterized Constructor...");
a=n1;
b=n2;
System.out.println("Value of a = "+a);
System.out.println("Value of b = "+b);
}
};
class TestDemo1
{
public static void main(String k [])
{
Test t1=new Test(10, 20);
}
};
```

Important points Related to Parameterized Constructor

- Whenever we create an object using parameterized constructor, it must be define parameterized constructor otherwise we will get compile time error. Whenever we define the objects with respect to both parameterized constructor and default constructor, It must be define both the constructors.
- In any class maximum one default constructor but 'n' number of parameterized constructors.

Constructor Overloading

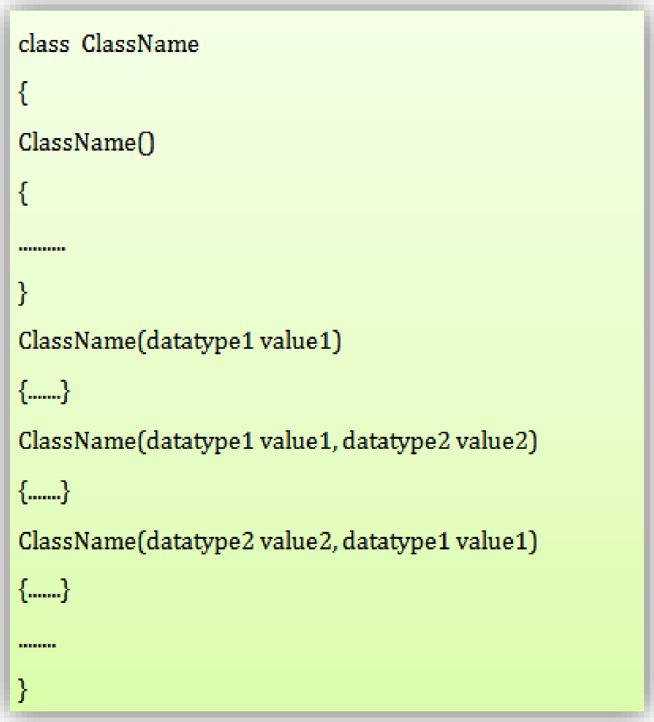
Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter lists.

The compiler differentiates these constructors by taking the number of parameters, and their type.

In other words whenever same constructor is existing multiple times in the same class with different number of parameters or order of parameters or type of parameters is known as Constructor overloading.

In general constructor overloading can be used to initialized same or different objects with different values.

Syntax



```
class  ClassName
{
ClassName()
{
..........
}
ClassName(datatype1 value1)
{.......}
ClassName(datatype1 value1, datatype2 value2)
{.......}
ClassName(datatype2 value2, datatype1 value1)
{.......}
........
}
```

8. Static Fields/variables and methods

Static Fields/variables

The **static** keyword is used in java mainly for memory management. Static keyword are used with variables, methods. If you declare any variable as static, it is known static variable.

If the value of a variable is not varied from object to object we should declare those variables at the class level by using static modifier.

In the case of instance variables for every object a separate copy will be created but in the case of static variables at the class level a single copy will be created and shared that copy by all objects of that class.

Static variables can be created at the time of class loading and destroyed at the time of class unloading hence the scope of static variables is exactly same as the scope of .class file.

We can access static variables either by using object reference or by using class name but usage of class name is recommended.

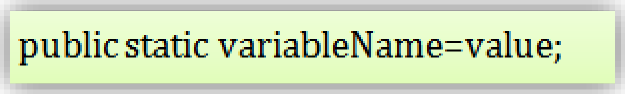
Within the same class it is not required to use class name also we can access static variables directly.

When and why we use static variable

Suppose we want to store record of all employee of any company, in this case employee id is unique for every employee but company name is common for all.

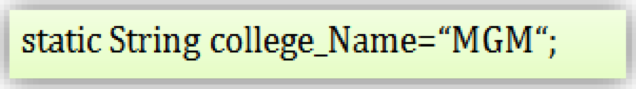
When we create a static variable as a company name then only once memory is allocated otherwise it allocates a memory space each time for every employee. The following is the syntax of declaring static variables.

Syntax:



```
public static variableName=value;
```

E.g



```
static String college_Name="MGM";
```

The following example shows the use of static variables:

```
class Student1
{
        int seatNo;
        String name;
        static String collegeName="MGM";
        Student1(int seatNo, String name)
{
 this.seatNo=seatNo;
 this.name=name;
 }
 void display()
 {
 System.out.println(seatNo+" "+name+"
                    "+collegeName);
 }
public static void main(String[] args)
  {
   Student1 s1=new Student1(101,"Karan");
   Student1 s2=new Student1(102,"Arjun");
   s1.display();
   s2.display();
   }
}
```

O/P

```
101 Karan MGM
102 Arjun MGM
```

	Non-static variable	Static variable
1	These variable should not be preceded by any static keyword Example: class A { int a; }	These variables are preceded by static keyword. Example: class A { static int b; }
2	Memory is allocated for these variable whenever an object is created	Memory is allocated for these variable at the time of loading of the class.
3	Memory is allocated multiple time whenever a new object is created.	Memory is allocated for these variable only once in the program.
4	Non-static variable also known as instance variable while because memory is allocated whenever instance is created.	Memory is allocated at the time of loading of class so that these are also known as class variable.
5	Non-static variable are specific to an object	Static variable are common for every object that means there memory location can be sharable by every object reference or same class.
6	Non-static variable can access with object reference. Syntax obj_ref.variable_name	Static variable can access with class reference. Syntax: class_name.variable_name

Static Methods

If you apply static keyword with any method, it is known as static method.

In case of non static method memory is allocated multiple times whenever method is calling. But in case of static method memory is allocated only once at the time of class loading.

A static method can be invoked without the need for creating an instance of a class.

A static method can access static data member and can change the value of it.

Syntax for declare static method:

```
public static  returnType methodName()
{
.......
.......
}
```

E.g.

```
public static  void  calculate()
{
.......
.......
}
```

The following example demonstrate the static method.

```
class Student2
 {
   int rollno;
   String name;
   static String college = "ITM";
   static void change()
   {
   college = "MGM";
   }
  Student2(int r, String n)
  {
    rollno = r;
    name = n;
  }
  void display ()
  {
   System.out.println(rollno+" "+name+" "+college);
  }
  public static void main(String args[])
  {
  Student2.change();

  Student2 s1 = new Student2 (111,"Karan");
  Student2 s2 = new Student2 (222,"Arjun");
  Student2 s3 = new Student2 (333,"Devilal");
  s1.display();
  s2.display();
  s3.display();
  }
}
```

O/P

```
111 Karan MGM
222 Arjun MGM
333 Devilal MGM
```

9. finalize() Method (Garbage Collection)

We know that Java is automatic garbage collected system. It automatically frees up the memory resources used by the objects. But objects may hold other non object resources such as Files.

The garbage collector can not free these resources. In order to free these resources we must use finalize() method. This is similar to destructors in C++.

The finalize() method

Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-java resources such as a file handles or character font, then you might want to make sure these resources are freed before an object is destroyed. To handle such situations java provides a mechanism called **finalization.** By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.

To add a finalizer to a class you simply define the finalise method. The Java calls that method whenever it is about to recycle an object of that class. Inside the finalize() method, you will specify those actions that must be performed before an object is destroyed.

There are two ways for requesting JVM to run garbage collector.

1. By System class.

2. By Runtime class

   - System class contain a static method to call a garbage collector.

   E.g.

   System.gc();

   - We can also call the garbage collector using Runtime class by creating object of Runtime class.

   E.g.

   Runtime r1=Runtime.getRuntime();

   Once we get runtime object then we can call the following method on that object.

   1. **freeMemory();**
   2. **totalMemory();**
   3. **gc();**

```
class FinalizeDemo
{
public static void main(String[] args)
{
FinalizeDemo fd=new FinalizeDemo();
fd=null;
System.gc();
System.out.println(fd);
System.out.println("End of main() method");
}
public void finalize()
{
System.out.println("finalize() method called");
}
}
```

O/P

```
javac FinalizeDemo.java
java FinalizeDemo
null
finalize() method called
End of main() method
```

10. Inheritance

Inheritance Basics:- The process of accessing the data members and methods from one class to another class is known as **inheritance**. It is one of the fundamental features of object-oriented programming.

The idea behind inheritance in java is that you can create new classes that are built upon existing classes.

- In the inheritance the class which is give data members and methods is known as base or super or parent class.
- The class which is taking the data members and methods is known as sub or derived or child class.
- The data members and methods of a class are known as features.
- The concept of inheritance is also known as re-usability or extendable classes or sub classing or derivation.

Imp: When you inherit from an existing class, you can reuse **non private methods and non private fields** of parent class, and you can add new methods and fields also.

Why use Inheritance?

- For Method Overriding (used for Runtime Polymorphism).
- It's main uses are to enable polymorphism and to be able to reuse code for different classes by putting it in a common super class
- For code Re-usability

Advantage of inheritance

If we develop any application using concept of Inheritance than that application have following advantages,

- Application development time is less.
- Application takes less memory.
- Application execution time is less.
- Application performance is enhance (improved).
- Redundancy (repetition) of the code is reduced or minimized so that we get consistence results and less storage cost.

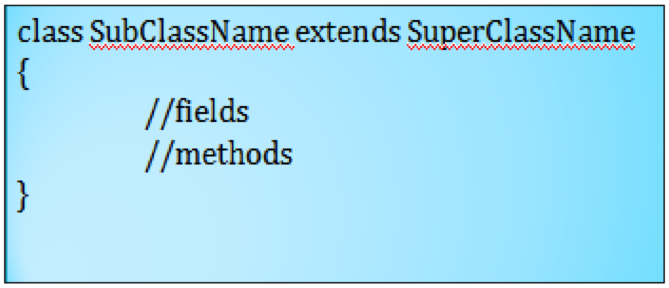
The "extends" keyword:

- Suppose we are having the one already existing class, we call it as "Super class". Suppose we need to create another class having exactly same attributes to that of the parent class and some extra attributes then we use **extends** keyword to extend the class.
- We can extend the class by using **extends** keyword in a class declaration after the class name and before the parent class.

The keyword **extends** indicates that you are making a new class that derives from an existing class.

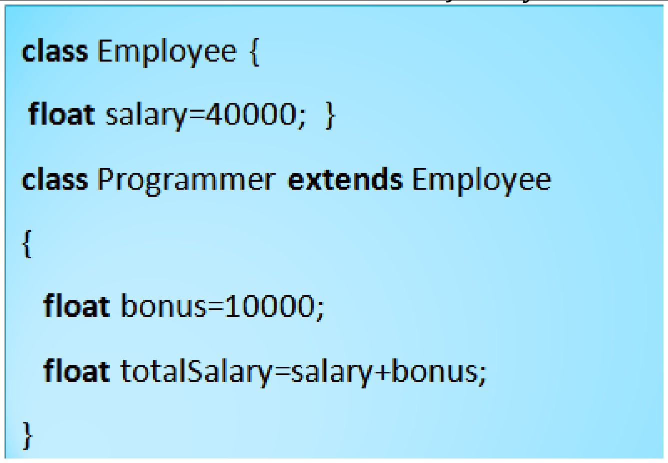
The following is the syntax of creating a new class from existing one.

Syntax:



```
class SubClassName extends SuperClassName
{
        //fields
        //methods
}
```

E.g.



```
class Employee {
 float salary=40000; }
class Programmer extends Employee
{
  float bonus=10000;
  float totalSalary=salary+bonus;
}
```

The following is the example of inheritance.

```
class Employee
{
float salary=20000;
void empSalary()
{
 System.out.println("Salary of Employee: "+salary);
}}
class Programmer extends Employee
{
float bonus=10000;
float totalSalary=salary+bonus;
void progSalary()
{
 System.out.println("Total Salary of Programmer: "+totalSalary);
}
public static void main(String[] args)
{
Programmer p=new Programmer();
p.empSalary();
p.progSalary();
}}
```

O/P

```
javac Programmer.java
```

java Programmer

Salary of Employee: 20000.0

Total Salary of Programmer: 30000.0

11. Types of Inheritance

Based on number of ways inheriting the feature of base class into derived class we have five Inheritance types they are:

1. Single inheritance
2. Multiple inheritance
3. Hierarchical inheritance
4. Multilevel inheritance
5. Hybrid inheritance

1. Single Inheritance

When a sub-class is derived simply from it's super-class then this mechanism is known as Single inheritance.

In case of single inheritance there is only a sub-class and its super-class. It is also called as one-level inheritance.

The following is the syntax of single inheritance:

Syntax:

```
class SuperClassName
{
//fields and methods
}
class SubClassName extends SuperClassName
{
//Fields and methods
}
```

E.g.

```
class A
{
   int a=10;
}
class B extends A
{
 int b=a*a;
}
```

The following is the example of single inheritance

```
class Parent
{
        int x=10;
        int y=20;
        void getP()
        {
                System.out.println("In Parent class");
                System.out.println("x="+x);
                System.out.println("y="+y);
        }
}
class Child extends Parent
{
        void getC()
        {
                System.out.println("In Child class");
                System.out.println("x="+x);
                System.out.println("y="+y);
        }
        public static void main(String[] args)
        {
                Child p=new Child();
                p.getC();
                p.getP();
        }
}
```

O/P

```
javac Child.java
java Child
In Child class
x=10
y=20
In Parent class
x=10
y=20
```

- In the above example we have declared two classes i.e. Parent and Child.
- In Parent class we have declared two variables and assigned a values to that variables i.e. x and y.
- According to inheritance Child class can access the non private data members of Parent class.
- If we declare x and y as a private then child class can not access x and y variables.
- If we try to access x and y variables in Child class then it gives **CE** saying:
    - ✗ x has private access in Parent
    - ✗ y has private access in Parent

2. Multiple inheritance

The mechanism of inheriting the features of more than one base class into a single class is known as multiple inheritance.

Java doesn't support multiple inheritance through classes but it can be achieved by using the interfaces.

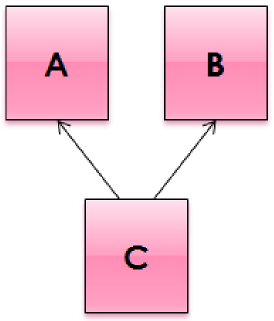


Fig. Multiple inheritance

```
class ClassName1
{
}
class ClassName2
{
}
class Child extends ClassName1, ClassName2
{
}
```

E.g.

```
class A
{
}
class B
{
}
class C extends A,B
{
}
```

If we try to compile the above program we will get **CE** saying:

```
C.java:7: '{' expected
class C extends A,B
```

3. Hierarchical inheritance

The mechanism of inheriting the features of one super-class into more thanone sub-class is known as hierarchical inheritance.

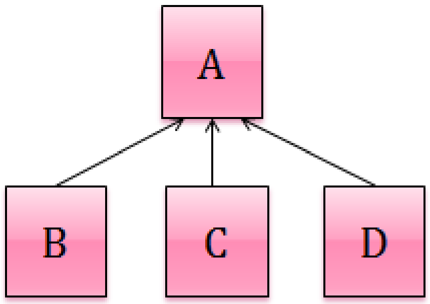


Fig. Hierarchical Inheritance

Syntax:

```
class BaseClassName
{
//fields and methods
}
class ChildClass1 extends BaseClassName
{
//fields and methods
}
 class ChildClass2 extends BaseClassName
{
//fields and methods
}
 class ChildClass3 extends BaseClassName
{
//fields and methods
}
```

E.g.

```
class A
{
 int a=10;
}
class B extends A
{
}
class C extends A
{
}
class D extends A
{
}
```

1. W.a.p to demonstrate the hierarchical inheritance.

```
class A
{
        int a=10;
}
class B extends A
{
        int b=20;
        void showB()
        {
         System.out.println("In showB()");
         System.out.println("a="+a);
         System.out.println("b="+b);
        }
}
class C extends A
{
        int c=30;
        void showC()
        {
         System.out.println("In showC()");
         System.out.println("a="+a);
         System.out.println("b="+c);
        }
}
class D extends A
{
        int d=40;
        void showD()
        {
         System.out.println("In showD()");
         System.out.println("a="+a);
         System.out.println("b="+d);
        }
}
```

```
class Test
{
	public static void main(String[] args)
	{
		B b1=new B();
		C b2=new C();
		D b3=new D();
		b1.showB();
		b2.showC();
		b3.showD();
	}
}
```

O/P

```
javac Test.java
java Test
In showB()
a=10
b=20
In showC()
a=10
b=30
In showD()
a=10
b=40
```

```
class Faculty
{
	float salary=50000.00f;
}
class Science extends Faculty
{
float bonus=20000.00f;
void displaySalary()
{
System.out.println("Faculty : Science");
System.out.println("Salary: "+salary);
System.out.println("Bonus: "+bonus);
}
}
class Commerce extends Faculty
{
float bonus=15000.00f;
void displaySalary()
{
System.out.println("Faculty : Commerce");
System.out.println("Salary: "+salary);
System.out.println("Bonus: "+bonus);
}
}
class Art extends Faculty
{
float bonus=10000.00f;
void displaySalary()
{
System.out.println("Faculty : Art");
System.out.println("Salary: "+salary);
System.out.println("Bonus: "+bonus);
}
}
class SalaryDetails
{
public static void main(String[] args)
{
	Science s=new Science();
	Commerce c=new Commerce();
	Art a=new Art();
	s.displaySalary();
	c.displaySalary();
	a.displaySalary();
	}}
```

```
javac SalaryDetails.java
java SalaryDetails
```

O/P

```
Faculty : Science
Salary: 50000.0
Bonus: 20000.0
Faculty : Commerce
Salary: 50000.0
Bonus: 15000.0
Faculty : Art
Salary: 50000.0
Bonus: 10000.0
```

4. Multilevel inheritance

In Multilevel inheritances there exists single base class, single derived class and multiple intermediate base classes.

Single base class + single derived class + multiple intermediate base classes.

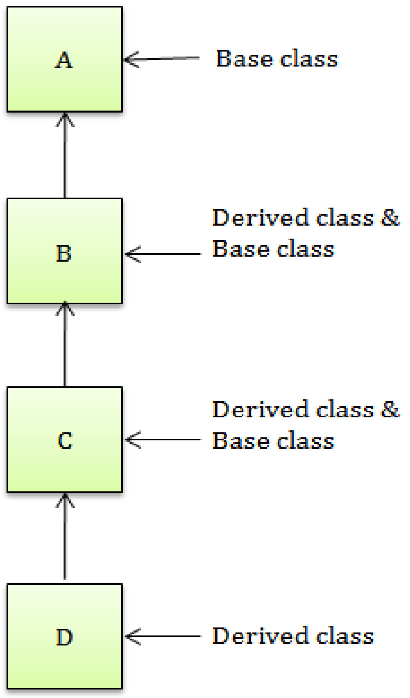


Fig. Multilevel Inheritance

When a sub-class is derived from a derived class then this mechanism is called as multilevel inheritance. Multilevel inheritance can go up to any number of levels.

E.g.

```
class A
{
}
class B extends A
{
}
class C extends B
{
}
class D extends C
{
}
```

1. The following program demonstrates the simple example of multilevel inheritance:

```
class A
{
        int a=10;
}
class B extends A
{
        int b=20;
        void showB()
        {
        System.out.println("In showB()");
        System.out.println("a="+a);
        System.out.println("b="+b);
        }
}
class C extends B
{
        int c=30;
        void showC()
        {
```

```
        System.out.println("In showC()");
        System.out.println("a="+a);
        System.out.println("c="+c);
        }
}
class D extends C
{
        int d=40;
        void showD()
        {
        System.out.println("In showD()");
        System.out.println("a="+a);
        System.out.println("d="+d);
        }
        public static void main(String[] args)
        {
        D d1=new D();
        d1.showB();
        d1.showC();
        d1.showD();
        }
}
```

```
javac D.java
java D
```

O/P

```
In showB()
a=10
b=20
In showC()
a=10
c=30
In showD()
a=10
d=40
```

2. The following program demonstrates the multilevel inheritance:

```
class Student
{
int rollNo;
String name;
void setStudent(int rollNo, String name)
{
this.rollNo=rollNo;
this.name=name;
}
public void showStudent()
{
System.out.println("Roll No : "+rollNo);
System.out.println("Name : "+name);
}
}
class Marks extends Student
{
int s1,s2,s3,s4;
public void setMarks(int s1, int s2, int s3, int s4)
{
this.s1=s1;
this.s2=s2;
this.s3=s3;
this.s4=s4;
}
public void showMarks()
{
System.out.println("S1 : "+s1);
System.out.println("S2 : "+s2);
System.out.println("S3 : "+s3);
System.out.println("S4 : "+s4);
}
}
class Result extends Marks
{
int total;
float percentage;
public void calculate()
{
total=s1+s2+s3+s4;
percentage=total/4;
}
public void showResult()
{
System.out.println("Total Marks : "+total);
System.out.println("Percentage : "+percentage);
}
public static void main(String[] args)
{
        Result r=new Result();
        r.setStudent(101,"Raj");
        r.showStudent();
        r.setMarks(50,40,65,36);
        r.showMarks();
        r.calculate();
        r.showResult();
}
}
```

```
javac Result.java
java Result
```

O/P

```
Roll No : 101
Name : Raj
S1 : 50
S2 : 40
S3 : 65
S4 : 36
Total Marks : 191
Percentage : 47.0
```

Important Points for Inheritance:

1. In java programming one derived class can extends only one base class because java programming does not support multiple inheritance through the concept of classes, but it can be supported through the concept of Interface.
2. Whenever we develop any inheritance application first create an object of bottom most derived class but not for top most base class.
3. When we create an object of bottom most derived class, first we get the memory space for the data members of top most base class, and then we get the memory space for data member of other bottom most derived class.
4. Bottom most derived class contains logical appearance for the data members of all top most base classes.
5. If we do not want to give the features of base class to the derived class then the definition of the base class must be preceded by final hence final base classes are not reusable or not inheritable.
6. If we are do not want to give some of the features of base class to derived class than such features of base class must be as private hence private features of base class are not inheritable or accessible in derived class.
7. Data members and methods of a base class can be inherited into the derived class but constructors of base class can not be inherited because every constructor of a class is made for initializing its own data members but not made for initializing the data members of other classes.

- An object of base class can contain details about features of same class but an object of base class never contains the details about special features of its derived class (this concept is known as scope of base class object).
- For each and every class in java there exists an implicit predefined super class called java.lang.Object. because it providers garbage collection facilities to its sub classes for collecting un-used memory space and improved the performance of java application.

12. Super Keyword

Super keyword in java is a reference variable that is used to refer parent class object. Super is an implicit keyword creates by JVM and supply each and every java program for performing important role in three places.

1. Accessing Super class variables
2. Accessing Super classmethods
3. Accessing Super classconstructor

Need of super keyword:

Whenever the derived class is inherits the base class features, there is a possibility that base class features are similar to derived class features and JVM gets an ambiguity. In order to differentiate between base class features and derived class features must be preceded by super keyword.

Syntax

super.superMemberName

1. Accessing Super class variables

Whenever the derived class inherits base class data members there is a possibility that base class data member are similar to derived class data member and JVM gets an ambiguity.

In order to differentiate between the data member of base class and derived class, in the context of derived class the base class data members must be preceded by super keyword.

Syntax

super.superclass_datamember_name

If we are not writing super keyword before the base class data member name than it will be referred as current class data member name and base class data member are hidden in the context of derived class.

The following example demonstrate the use of super keyword for accessing super class variables.

```
class Employee
{
float salary=10000;
}
class HR extends Employee
{
float salary=20000;
void display()
{
System.out.println("Base class Salary: "+super.salary);
System.out.println("Child class Salary: "+salary);
}
}
class Supervarible
{
public static void main(String[] args)
{
HR obj=new HR();
obj.display();
}
}
```

```
javac Supervarible.java
```

```
java Supervarible
```

O/P

```
Base class Salary: 10000.0
Child class Salary: 20000.0
```

2. **Accessing Super class methods**

The **super keyword** can also be used to invoke or call parent class method. It should be use in case of method overriding.

In other word **super keyword** use when base class method name and derived class method name have same name.

The following example demonstrate the use of super keyword for accessing super class method.

```
class Parent
{
int bikeSpeed=50;
public void showSpeed()
{
System.out.println("Parent's speed : "+bikeSpeed);
}
}
class Child extends Parent
{
int speed=100;
public void showSpeed()
{
System.out.println("Child's speed : "+speed);
super.showSpeed();
}
public static void main(String[] args)
{
        Child c1=new Child();
        c1.showSpeed();
}
}
```

```
javac Child.java
java Child
```

O/P

```
Child's speed : 100
Parent's speed : 50
```

In case there is no method in subclass as parent, there is no need to use super.

3. **Accessing Super class constructor**

The super keyword can also be used to invoke or call the parent class constructor. Constructor are calling from bottom to top and executing from top to bottom.

To establish the connection between base class constructor and derived class constructors JVM provides two implicit methods they are:

1. super()
2. super(parameterList)

1. **super()**

It is used for calling super class default constructor from the derived class constructors.

The following example demonstrate how to call super class constructor from sub-class constructor:

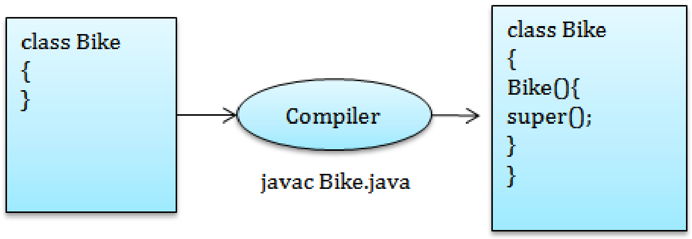
```
class Vehicle
{
 Vehicle()
 {
 System.out.println("Vehicle is created");
 }
}
 class Bike extends Vehicle
 {
 Bike()
 {
  super();//will invoke parent class constructor
  System.out.println("Bike is created");
 }
 public static void main(String args[])
 {
  Bike b=new Bike();
 }
 }
```

```
javac Bike.java
java Bike
```

O/P

```
Vehicle is created
Bike is created
```

- The first line inside constructor should be either super() or this().
- If we are not writing then compiler will always generate super keyword.



We can use super() only at the first line of constructor, if we are using anywhere else we will get **CE:**

call to super must be first statement in constructor.

13. Method Overriding

Polymorphism is an important feature of OOP. The process of representing one Form in multiple forms is known as Polymorphism. Here one form represent original form or original method always resides in base class and multiple forms represents overridden method which resides in derived classes.

Polymorphism is derived from 2 greek words: **poly** and **morphs**. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

Real life example of polymorphism

Suppose if you are in class room that time you behave like a student, when you are in market at that time you behave like a customer, when you at your home at that time you behave like a son or daughter, Here one person present in different-different behaviors.

How to achieve Polymorphism in Java ?

In java programming the Polymorphism principal is implemented with method overriding concept of java.

Polymorphism principal is divided into two sub principal they are:

- Static or Compile time polymorphism
- Dynamic or Runtime polymorphism

Static polymorphism is implemented by using method overloading and dynamic polymorphism is implemented by using method overriding.

What is method overriding?

A subclass inherits methods from a superclass. However in certain situations, the subclass need to modify the implementation (code) of a method defined in the superclass without changing the parameter list. This is achieved by overriding or redefining the method in the subclass.

If sub class has the same method as declared in the parent class, it is known as **"method overrding".**

In other words, if the sub-class provides specific implementation of the method that has been provided by one of its parent class, it is known as "**method overriding**".

If we doesn't satisfy with the parent class implementation then we should override that method in the sub-class in our own way.

Advantages of method overriding?

1. Method overriding is used to provide specific implementation of a method that is already provided by its super class.
2. Method overriding is used for Runtime Polymorphism.

Rules for method overriding

1. Method must have same name as in the parent class.
2. Method must have same parameter as in the parent class.
3. The accessibility must not be more restrictive than original method.
4. Must have the **IS-A** relationship(inheritance is must).

The following is the syntax of method overriding:

Syntax:

```
class BaseClass
{
 return_type methodName()
 {
  ===========
  ===========
}
}
class ChildClass extends BaseClass
{
  return_type methodName()
  {
   ============
   ============
  }
}
```

E.g.

```
class Parent
{
 void speed()
 {
   System.out.println("60km/hr");
 }
}
class Child extends Parent
{
void speed()
 {
   System.out.println("100km/hr");
 }
}
```

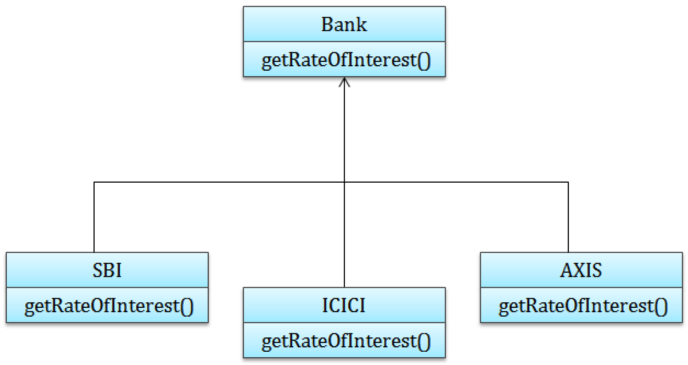
To override a subclass's method, we simply redefine the method in the subclass with the same name, same return type and same parameter list as its superclass counterpart.

When the method with the same name exists in both the superclass as well as in the subclass, the object of the subclass will invoke the method of the subclass, not the method inherited from the base class.

- Private methods are not visible in child classes hence overriding concept is applicable for private methods.
- Based on our requirement we can take exactly same private method in child class it is valid but it is not overriding.
- Final methods can not be overridden in child classes. But non final method can be overridden as final.
- While overriding we can't decrease access modifiers if we want we can increase.

Real-time example of Java Method Overriding

Consider a scenario, Bank is a class that provides functionality to get rate of interest. But, rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7% and 9% rate of interest.



The following program shows the real-time example of method overriding:

```
class Bank
{
    int getRateOfInterest(){return 0;}
}
class SBI extends Bank
{
    int getRateOfInterest(){return 8;}
}
class ICICI extends Bank
{
    int getRateOfInterest()
    {
    return 7;
    }
}
class AXIS extends Bank
{
    int getRateOfInterest()
    {
    return 9;
    }
    }
    class Test2
    {
    public static void main(String args[])
    {
        SBI s=new SBI();
        ICICI i=new ICICI();
        AXIS a=new AXIS();
        System.out.println("SBI Rate of Interest: "+s.ge
        tRateOfInterest());
        System.out.println("ICICI Rate of Interest: "+i.g
        etRateOfInterest());
        System.out.println("AXIS Rate of Interest: "+a.g
        etRateOfInterest());
    }
    }
    javac Test2.java
    java Test2
        SBI Rate of Interest: 8
        ICICI Rate of Interest: 7
```

AXIS Rate of Interest: 9

14. final method

The "**final**" is a keyword or access modifier applicable for variables, methods and classes.

If we declare a method as a final then we can't override that method in child class i.e. overriding is not possible with final methods.

We can simply add the keyword final at the start of the method declaration in a super class.

The following is the syntax of declaring final method.

Syntax:

```
final return_type  methodName(parameterList)
{
  =========
  =========
}
```

E.g.

```
final void  m1()
{
  =========
  =========
}
```

The following example demonstrate the use of final method:

```
class Account
{
final void getDetails()
{
System.out.println("Base class method");
}
}
class Clerk extends Account
{
void getDetails()
{
System.out.println("Child class method");
}
public static void main(String[] args)
{
        Clerk c1=new Clerk();
        c1.getDetails();
}
}
```

If we try to compile the above program then it gives **CE** saying:

getDetails() in Clerk cannot override getDetails() in Account; overridden method is final

15. final class

If we declare a class as a final then we can't extend that class i.e. inheritance concept is not applicable with final class.

If we declare a class as a final then all the variables and methods in that class are by default final.

When we want to restrict inheritance then make a class as a final.

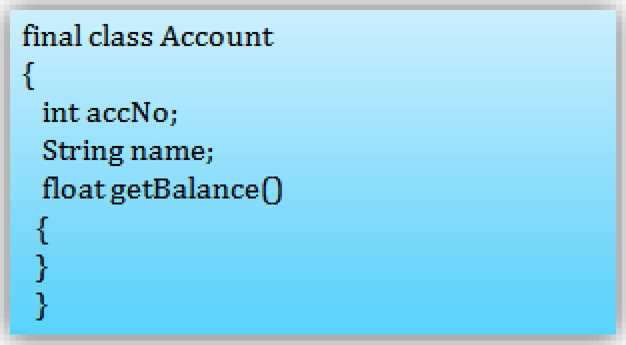
We can simply add final keyword at the class declaration.

The following is the syntax of declaring final class:

Syntax:

```
final class ClassName
{
  // fields
  // methods
}
```

E.g.



```
final class Account
{
  int accNo;
  String name;
  float getBalance()
  {
  }
  }
```

The following example demonstrate the use of final with class.

```
final class Account
{
  int accNo=1001;
  float balance=50000;
  float getBalance()
 {
 System.out.println("Balance : "+balance);
 }
 }
class Customer extends Account
{
  public static void main(String a[])
  {
          Customer c1=new Customer();
          c1.getBalance();
   }
}
```

In the above example we have created two classes i.e. Account and Customer. We have declared Account class as a final it means we can't extend this class but here we are trying to extends Account class therefore it gives **CE** saying:

javac Customer.java

cannot inherit from final Account

16. Use of Abstract class:-

What is abstraction?

Hiding the internal implementation and highlight the set of services is the concept of "**abstraction**".

Hiding of data is known as **data abstraction**. In object oriented programming language this is implemented automatically while writing the code in the form of class and object.

Abstraction shows only important things to the user and hides the internal details. For example when we ride a bike, we only know about how to ride bike but can not know about how it work? and also we do not know internal functionality of bike.

Abstraction solves the problem in the design side while Encapsulation is the Implementation.

By using interfaces and abstract classes we can achieve abstraction.

Advantages of abstraction:

1. **Security:** As we are not highlighting our internal implementation.
2. Enhancement will become very easy because without affecting the external things we can perform any changes in our internal design.

How to achieve Abstraction?

There are two ways to achieve abstraction in java:

1. **Abstract class (0 to 100%)**
2. **Interface (Achieve 100% abstraction)**

1. **What is Abstract class (0 to 100%)?**

We know that every java program must start with a concept of class that is without classes concept there is no java program perfect.

In java programming we have two types of classes they are

1. Concrete class
2. Abstract class

Concrete class

A concrete class is one which is containing fully defined methods or implemented method.

Example

```
class Helloworld
 {
        void display()
        {
        System.out.println("Good Morning........");
        }
 }
```

Here Helloworld class is containing a defined method and object can be created directly.

```
        Helloworld obj=new Helloworld();
        obj.display();
```

Every concrete class have specific feature and these classes are used for specific requirement but not for common requirement.

If we use concrete classes for fulfill common requirements than such application will get the following limitations.

1. Application will take more amount of memory space (main memory).
2. Application execution time is more.
3. Application performance is decreased.

To overcome above limitation you can use abstract class.

Abstract class

A class that is declared with abstract keyword is known as **abstract class**

An abstract class is one which is containing some concrete methods and some abstract methods.

A class is made abstract by putting the keyword abstract in front of class keyword in the first line of class definition.

The following is the syntax of declaring abstract class:

Syntax:

```
abstract class ClassName
{
 ========;
 ========;
}
```

E.g.

```
abstract class Vehicle
{
 ========;
 ========;
}
```

Abstract Method

A method that is declared as abstract and does not have implementation is known as **abstract method.**

To create an abstract method, simply specify the modifier abstract followed by the method declaration and end the method with semicolon.

The following is the syntax of abstract method.

Syntax:

```
abstract return_type methodName( );
```

E.g

```
abstract int getMileage();
```

The following is the example of abstract class:

```
abstract class Vehicle
{
 abstract void speed();
}
class Bike extends Vehicle
{
void speed()
{
System.out.println("Speed limit is 50 km/hr..");
}
}
class Car extends Vehicle
{
void speed()
{
System.out.println("Speed limit is 80 km/hr..");
}
}
class AbsTest
{
public static void main(String args[])
{
 Bike b = new Bike();
 Car c=new Car();
 b.speed();
 c.speed();
 }
}
javac AbsTest.java
java AbsTest
```

O/P

```
Speed limit is 50 km/hr..
Speed limit is 80 km/hr..
```

If you extend the abstract class then you must have to provide implementation for each and every abstract method otherwise we will get **CE:**

E.g.

In above example if you doesn't override speed() method in Bike class then compiler gives an error **CE:**

Bike is not abstract and does not override abstract method speed () in Vehicle

---

If there is any abstract method in a class that class must be declared as abstract class.

If you are extending any abstract class that has abstract methods, you must either provide the implementation for the abstract methods or make this class abstract otherwise we will get **CE** saying:

Vehicle is not abstract and does not override abstract method speed() in Vehicle

---

An abstract modifier is applicable only for methods and classes not for variables. If we declare a variable as a abstract then we will get **CE:**

modifier abstract not allowed here

An abstract class can have data members, normal methods, constructors and even main method also.

E.g

```
abstract class Vehicle
{
 abstract void speed();
 int wheels;
 int gears;
 void show()
```

```
        {
                System.out.println("In show() method of Vehicle");
   }
}
class Bike extends Vehicle
{
void speed()
{
System.out.println("In Bike class");
System.out.println("Speed limit is 50 km/hr..");
System.out.println("Wheels : 2");
System.out.println("Gears : 4");
}
}
class Car extends Vehicle
{
void speed()
{
System.out.println("In Car class");
System.out.println("Speed limit is 80 km/hr..");
System.out.println("Wheels : 4");
System.out.println("Gears : 4");
}
}
class AbsTest1
{
public static void main(String args[])
{
 Bike b = new Bike();
 Car c=new Car();
 b.speed();
 c.speed();
 }
}
```

```
javac AbsTest.java
java AbsTest
```

O/P

```
In Bike class
Speed limit is 50 km/hr..
Wheels : 2
Gears : 4
In Car class
Speed limit is 80 km/hr..
Wheels : 4
Gears : 4
```

<u>Advantage of abstract class</u>

1. Less memory space for the application
2. Less execution time
3. More performance

<u>Important points of abstract class:</u>

1. Abstract class of java always contains common features.
2. Every abstract class participate in inheritance.
3. Abstract classes definitions should not be made as final because abstract classes always participate in inheritance classes.
4. An object of abstract class can not be created directly but it can be created indirectly.
5. All the abstract classes of java makes use of polymorphism along with method overriding for business logic development and makes use of dynamic binding for execution logic.