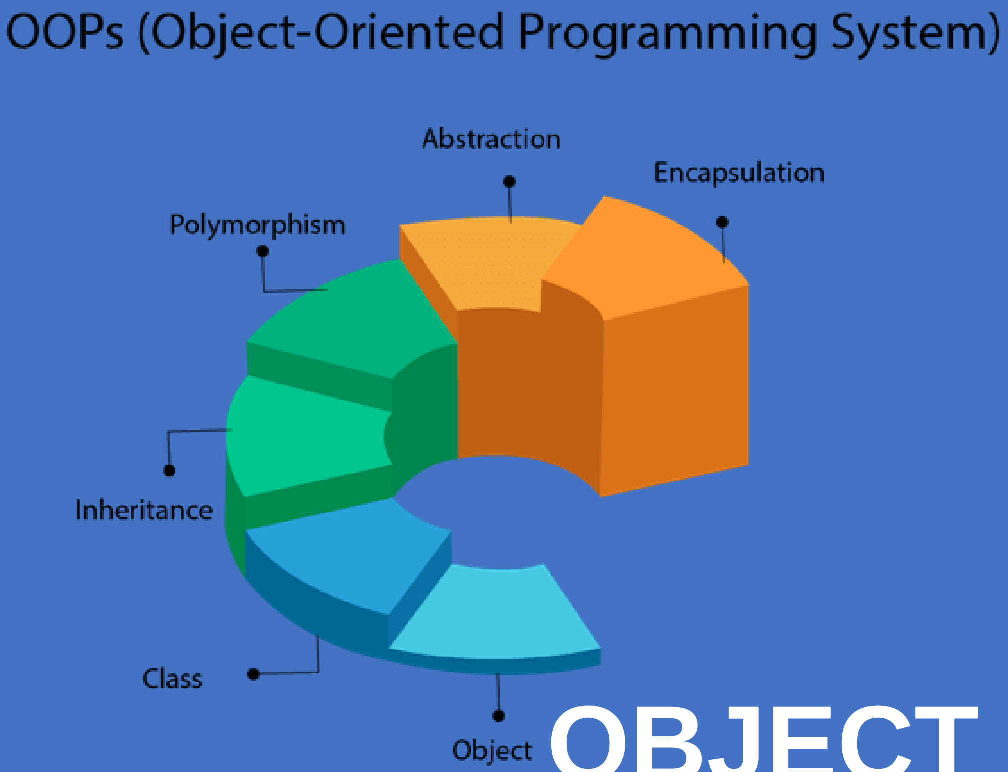


OBJECT ORIENTED PROGRAMMING

Object Oriented Programming:- Inheritance , Encapsulation , Polymorphism ,Abstraction

OOPs Concepts

Uday Sharma
mrudaysharma4600@gmail.com

OOPS in JAVA

- ## Methods in Java:-
  - Sometimes our program grows in size, and we want to separate the logic of the main method from the other methods.
  - For instance, if we calculate the average of a number pair 5 times, we can use methods to avoid repeating the logic. [DRY – Don't Repeat Yourself].

- ### *Syntax of a Method:-*

A method is a function written inside a class. Since Java is an object-oriented language, we need to write the method inside some class.

- ### *Syntax of a method :*

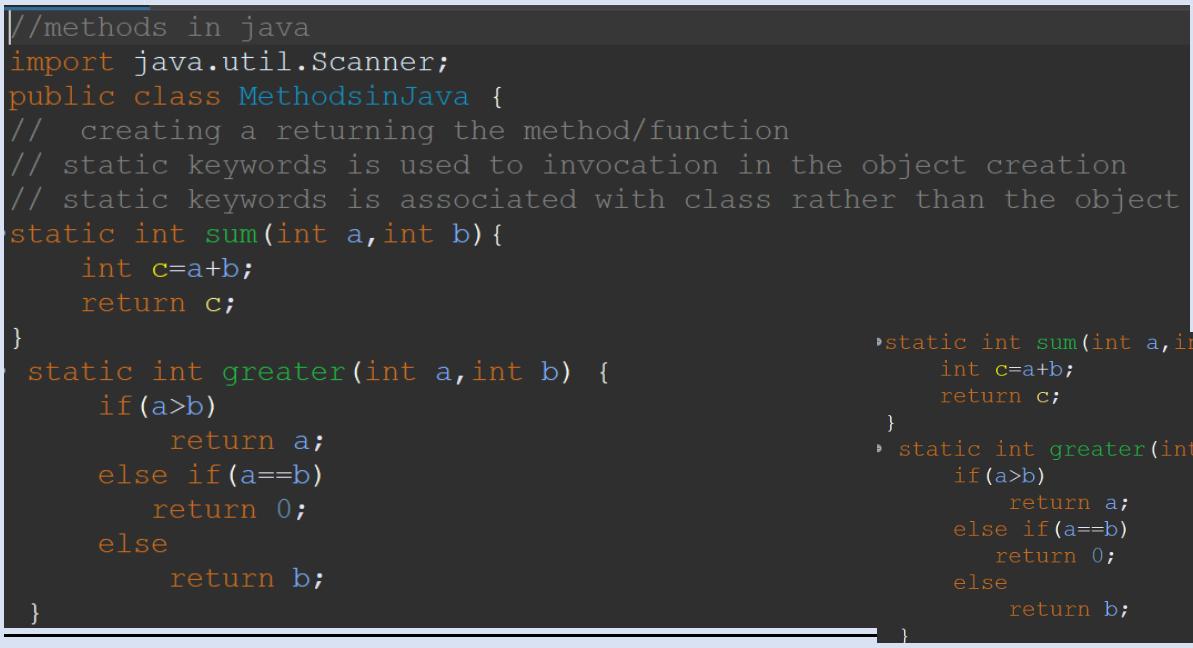
```
returnType nameOfMethod() {
//Method body
}
```

- ## *Java Static keyword : -*

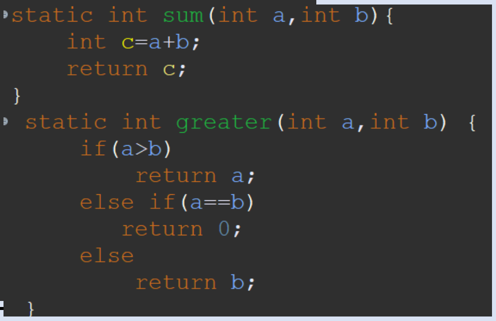
1. *Java static Methods:-*

- The static keyword is used to associate a method of a given class with the class rather than the object.
- You can call a static method without creating an instance of the class.
- In Java, the main() method is static, so that JVM can call the main() method directly without allocating any extra memory for object creation.
- All the objects share the static method in a class.

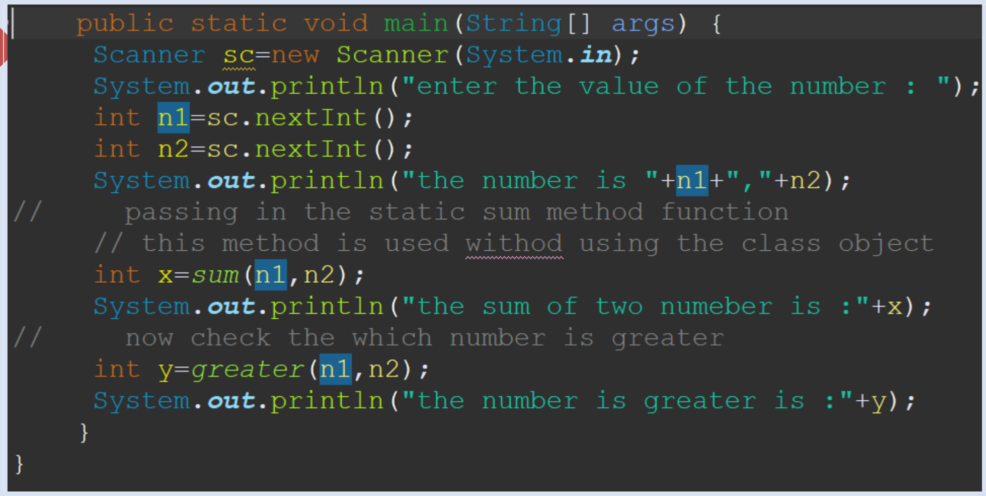
- The static can be:_
  1. Variable (also known as a class variable).
  2. Method (also known as a class method).
  3. Block.
  4. Nested class.



```
//methods in java
import java.util.Scanner;
public class MethodsinJava {
//  creating a returning the method/function
// static keywords is used to invocation in the object creation
// static keywords is associated with class rather than the object
static int sum(int a,int b){
    int c=a+b;
    return c;
}
 static int greater(int a,int b)  {
     if(a>b)
         return a;
     else if(a==b)
        return 0;
     else
         return b;
 }
```



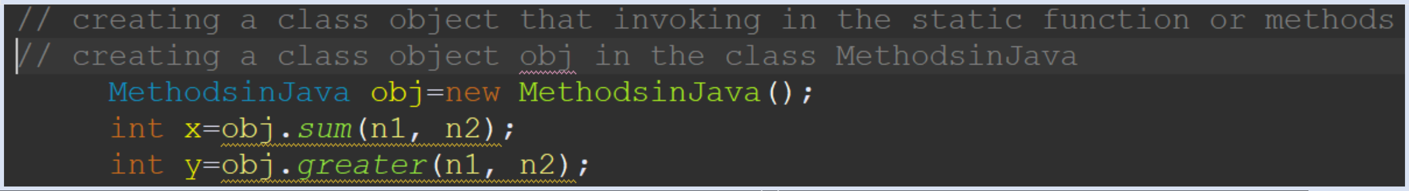
```
static int sum(int a,int b){
    int c=a+b;
    return c;
}
 static int greater(int a,int b)  {
     if(a>b)
         return a;
     else if(a==b)
        return 0;
     else
         return b;
 }
```



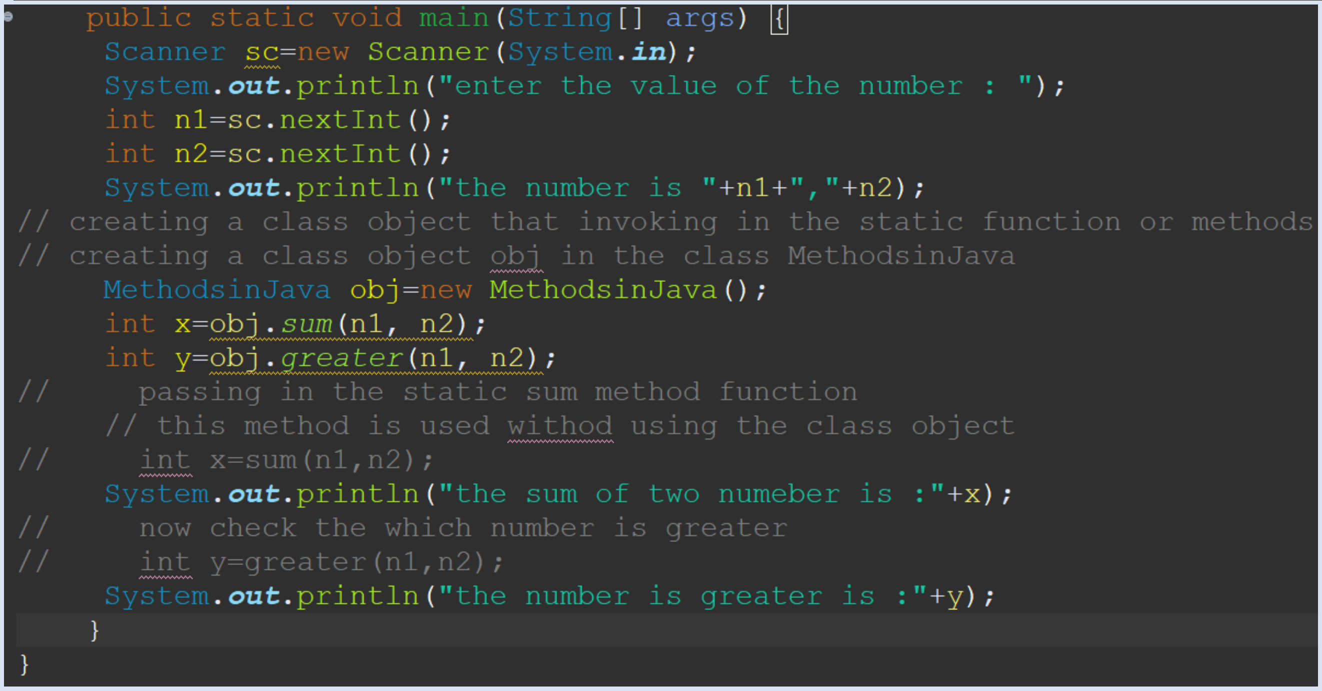
```
    public static void main(String[] args) {
     Scanner sc=new Scanner(System.in);
     System.out.println("enter the value of the number : ");
     int n1=sc.nextInt();
     int n2=sc.nextInt();
     System.out.println("the number is "+n1+","+n2);
//     passing in the static sum method function
     // this method is used withod using the class object
     int x=sum(n1,n2);
     System.out.println("the sum of two numeber is :"+x);
//     now check the which number is greater
     int y=greater(n1,n2);
     System.out.println("the number is greater is :"+y);
    }
}
```

- ***Calling a Method by creating a class object : -***

A method can be called by creating an object of the class in which the method exists followed by the method call:

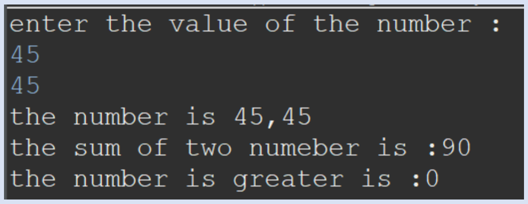


```
// creating a class object that invoking in the static function or methods
// creating a class object obj in the class MethodsinJava
     MethodsinJava obj=new MethodsinJava();
     int x=obj.sum(n1, n2);
     int y=obj.greater(n1, n2);
```



```
    public static void main(String[] args) {
     Scanner sc=new Scanner(System.in);
     System.out.println("enter the value of the number : ");
     int n1=sc.nextInt();
     int n2=sc.nextInt();
     System.out.println("the number is "+n1+","+n2);
// creating a class object that invoking in the static function or methods
// creating a class object obj in the class MethodsinJava
     MethodsinJava obj=new MethodsinJava();
     int x=obj.sum(n1, n2);
     int y=obj.greater(n1, n2);
//     passing in the static sum method function
     // this method is used withod using the class object
//     int x=sum(n1,n2);
     System.out.println("the sum of two numeber is :"+x);
//     now check the which number is greater
//     int y=greater(n1,n2);
     System.out.println("the number is greater is :"+y);
    }
}
```

Output is same in both the case:-



```
enter the value of the number :
45
45
the number is 45,45
the sum of two numeber is :90
the number is greater is :0
```

- ***Void return type : -***

When we don't want our method to return anything, we use void as the return type.

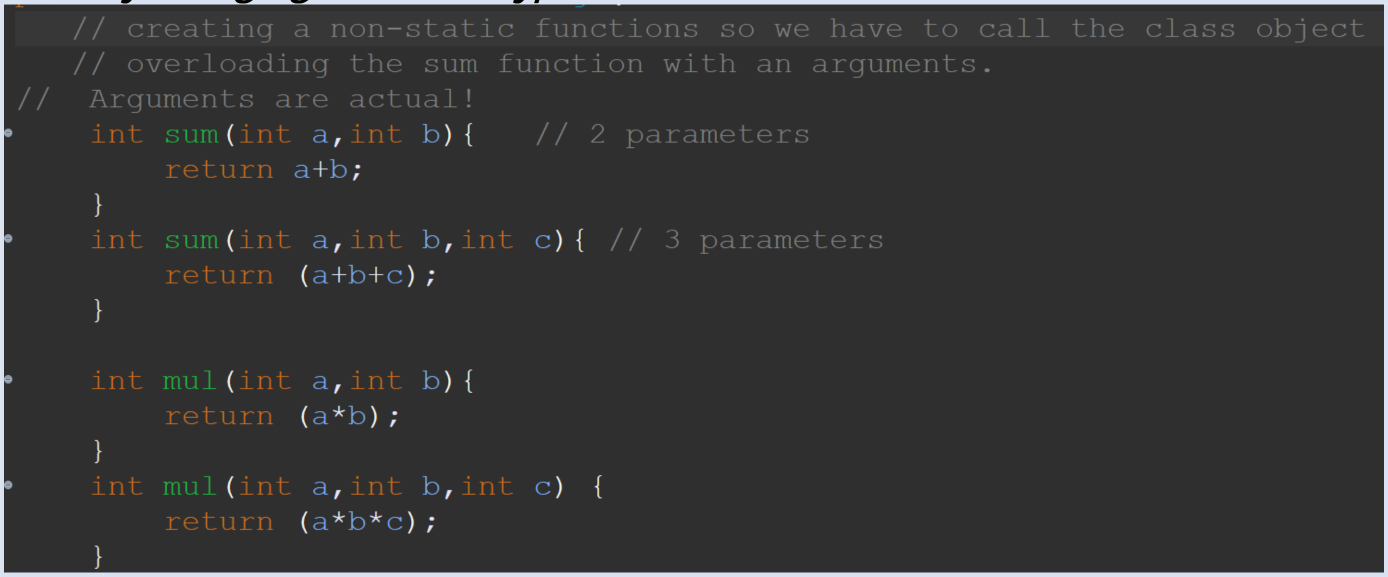
- ***Method Overloading in Java:-***
  - In Java, it is possible for a class to contain two or more methods with the same name but with different parameters. Such methods are called Overloaded methods.
  - Method overloading is used to increase the readability of the program.
- ***Ways to perform method overloading : -***

In Java, method overloading can be performed by two ways listed below :

1. By changing the **return type** of the different methods
2. By changing the **number of arguments** accepted by the method

Now, let's have an example to understand the above ways of method overloading :

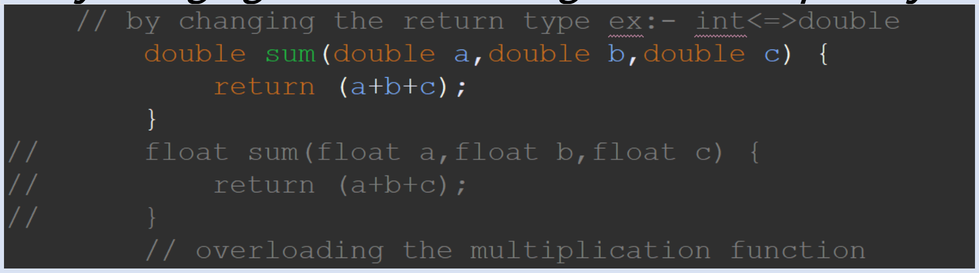
1. By changing the return type of the different methods:-



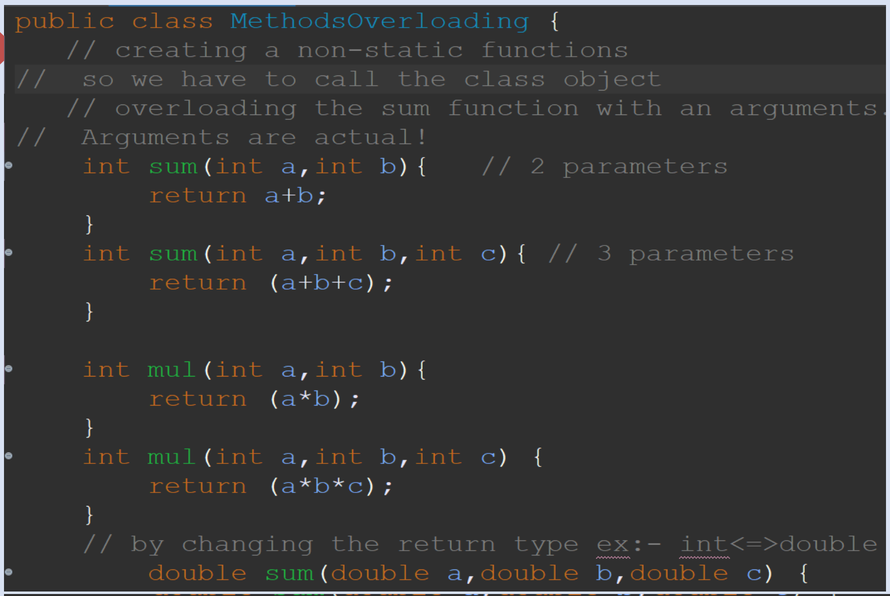
```
    // creating a non-static functions so we have to call the class object
    // overloading the sum function with an arguments.
//   Arguments are actual!
     int sum(int a,int b){    // 2 parameters
         return a+b;
     }
     int sum(int a,int b,int c){ // 3 parameters
         return (a+b+c);
     }

     int mul(int a,int b){
         return (a*b);
     }
     int mul(int a,int b,int c)  {
         return (a*b*c);
     }
```

2). ***By changing the number of arguments accepted by the method:-***

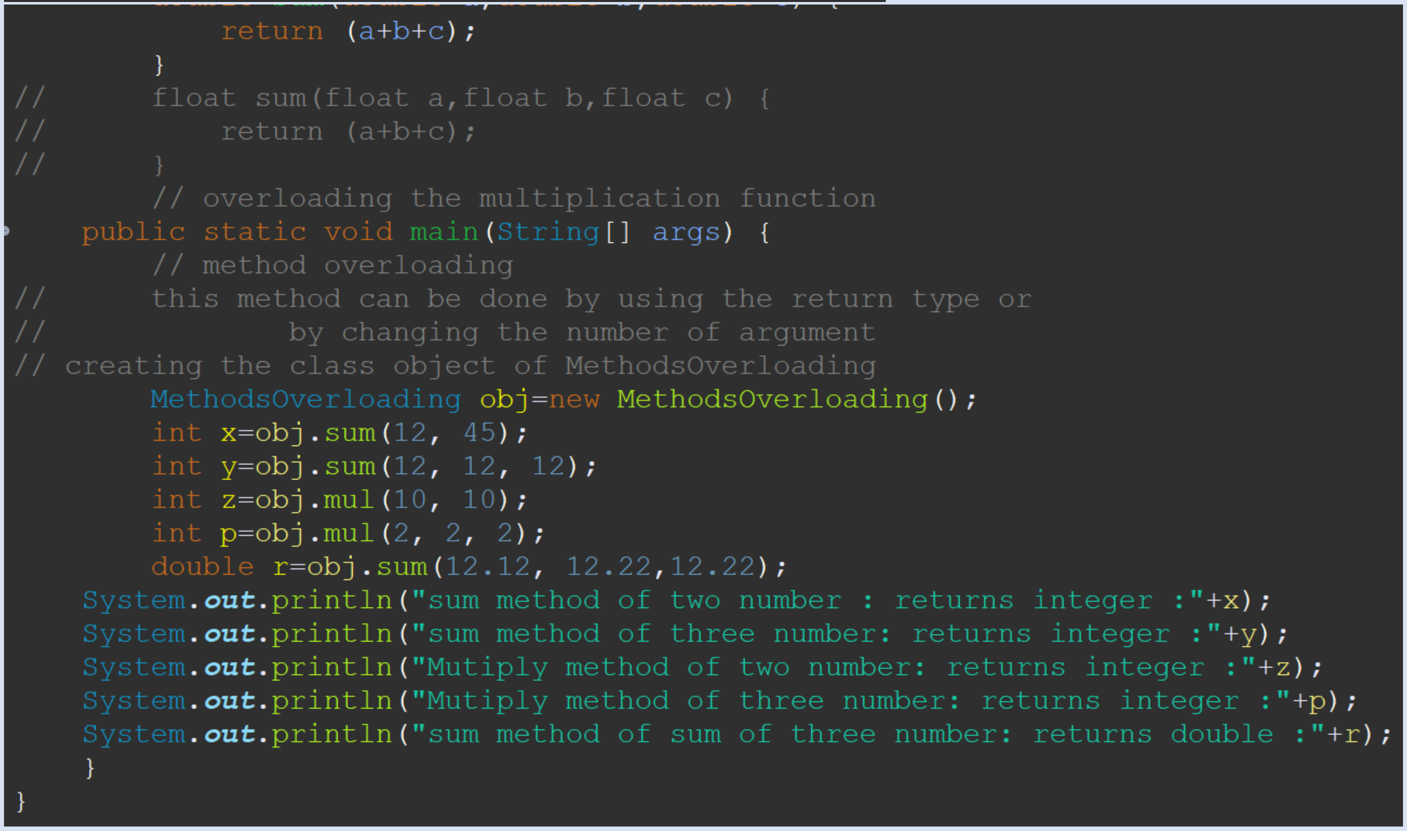


```
    // by changing the return type ex:- int<=>double
        double sum(double a,double b,double c)  {
            return (a+b+c);
        }
//      float sum(float a,float b,float c) {
//          return (a+b+c);
//      }
        // overloading the multiplication function
```



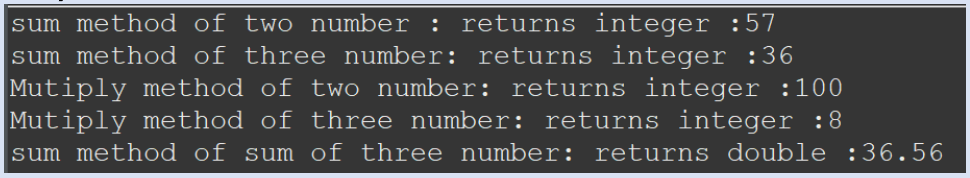
```
public class MethodsOverloading {
    // creating a non-static functions
//   so we have to call the class object
    // overloading the sum function with an arguments.
//   Arguments are actual!
     int sum(int a,int b){    // 2 parameters
         return a+b;
     }
     int sum(int a,int b,int c){ // 3 parameters
         return (a+b+c);
     }

     int mul(int a,int b){
         return (a*b);
     }
     int mul(int a,int b,int c) {
         return (a*b*c);
     }
     // by changing the return type ex:- int<=>double
         double sum(double a,double b,double c) {
```



```
            return (a+b+c);
        }
//      float sum(float a,float b,float c) {
//          return (a+b+c);
//      }
        // overloading the multiplication function
    public static void main(String[] args) {
        // method overloading
//      this method can be done by using the return type or
//              by changing the number of argument
// creating the class object of MethodsOverloading
        MethodsOverloading obj=new MethodsOverloading();
        int x=obj.sum(12, 45);
        int y=obj.sum(12, 12, 12);
        int z=obj.mul(10, 10);
        int p=obj.mul(2, 2, 2);
        double r=obj.sum(12.12, 12.22,12.22);
    System.out.println("sum method of two number : returns integer :"+x);
    System.out.println("sum method of three number: returns integer :"+y);
    System.out.println("Mutiply method of two number: returns integer :"+z);
    System.out.println("Mutiply method of three number: returns integer :"+p);
    System.out.println("sum method of sum of three number: returns double :"+r);
    }
}
```

Output:-



```
sum method of two number : returns integer :57
sum method of three number: returns integer :36
Mutiply method of two number: returns integer :100
Mutiply method of three number: returns integer :8
sum method of sum of three number: returns double :36.56
```

2). Java static variable:-

If you declare any variable as static, it is known as a static variable.

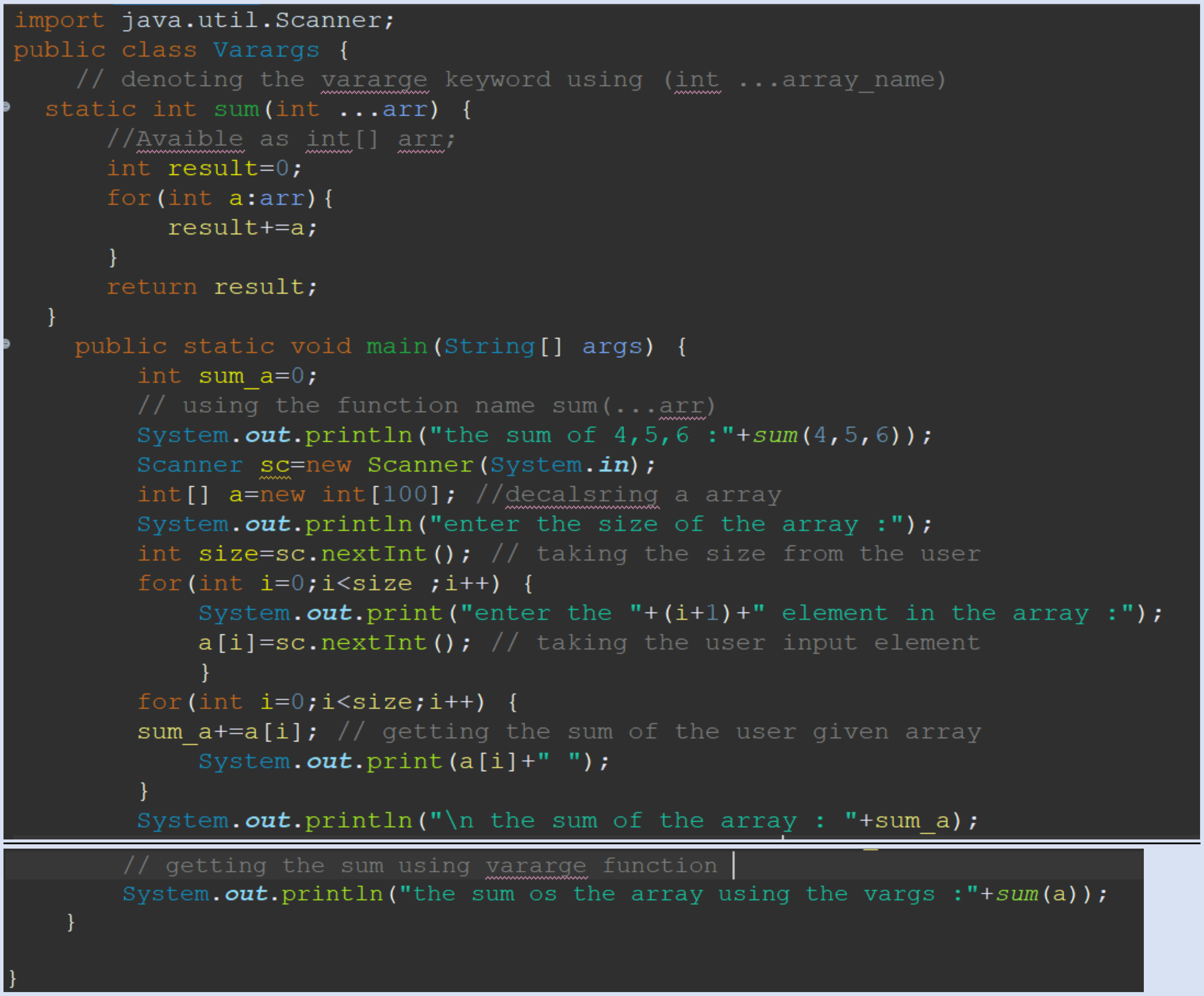
- The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.

- The static variable gets memory only once in the class area at the time of class loading.
- ***Advantages of static variable:-***

It makes your program memory efficient (i.e., it saves memory).

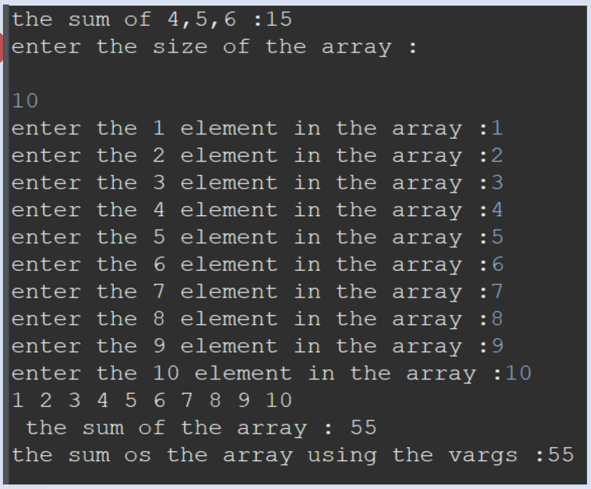
- ***Arguments (VarArgs) in Java:-***
- you want to overload an "add" method. The "add" method will accept one argument for the first time and every time the number of arguments passed will be incremented by 1 till the number of arguments is equaled to 10.
- One approach to solve this problem is to overload the "add" method 10 times. But is it the optimal approach? What if I say that the number of arguments passed will be incremented by 1 till the number of arguments is equaled to 1000. Do you think that it is good practice to overload a method 1000 times?
- To solve this problem of method overloading, Variable Arguments(Varargs) were introduced with the release of JDK 5.
- With the help of Varargs, we do not need to overload the methods.
- foo can be called with zero or more arguments like this:
    - sum(7)
    - sum(7,8,9)   // its gives the sum =7+8+9
    - sum(1,2,7,8,9)

<u>code of vargs and sum of user given array:-</u>



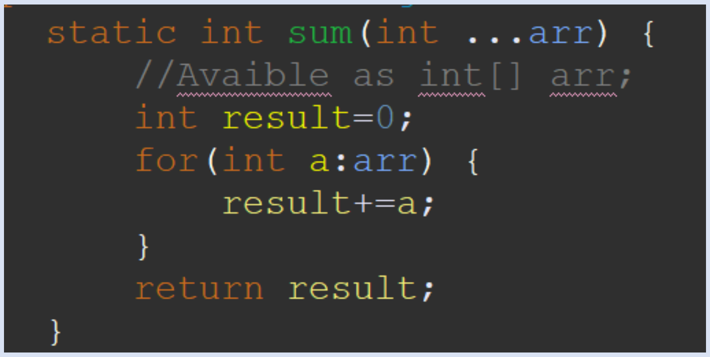
```
import java.util.Scanner;
public class Varargs {
    // denoting the vararge keyword using (int ...array_name)
  static int sum(int ...arr) {
      //Avaible as int[] arr;
      int result=0;
      for(int a:arr){
          result+=a;
      }
      return result;
  }
    public static void main(String[] args) {
        int sum_a=0;
        // using the function name sum(...arr)
        System.out.println("the sum of 4,5,6 :"+sum(4,5,6));
        Scanner sc=new Scanner(System.in);
        int[] a=new int[100]; //decalsring a array
        System.out.println("enter the size of the array :");
        int size=sc.nextInt(); // taking the size from the user
        for(int i=0;i<size ;i++) {
            System.out.print("enter the "+(i+1)+" element in the array :");
            a[i]=sc.nextInt(); // taking the user input element
            }
        for(int i=0;i<size;i++) {
        sum_a+=a[i]; // getting the sum of the user given array
            System.out.print(a[i]+" ");
        }
        System.out.println("\n the sum of the array : "+sum_a);
        // getting the sum using vararge function
        System.out.println("the sum os the array using the vargs :"+sum(a));
    }

}
```



```
the sum of 4,5,6 :15
enter the size of the array :

10
enter the 1 element in the array :1
enter the 2 element in the array :2
enter the 3 element in the array :3
enter the 4 element in the array :4
enter the 5 element in the array :5
enter the 6 element in the array :6
enter the 7 element in the array :7
enter the 8 element in the array :8
enter the 9 element in the array :9
enter the 10 element in the array :10
1 2 3 4 5 6 7 8 9 10
 the sum of the array : 55
the sum os the array using the vargs :55
```

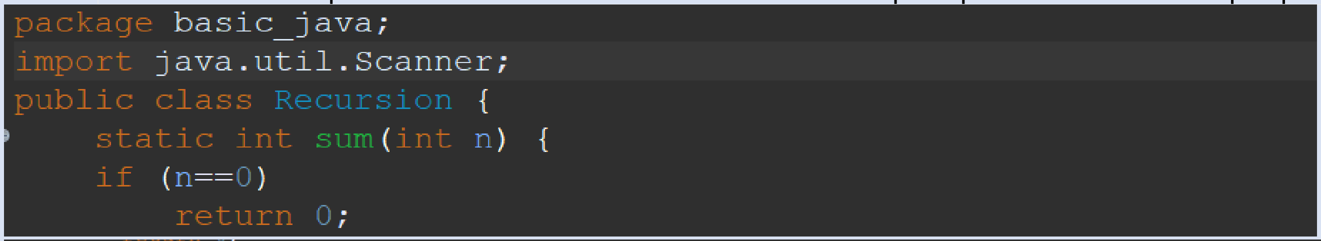


```
static int sum(int ...arr) {
    //Avaible as int[] arr;
    int result=0;
    for(int a:arr) {
        result+=a;
    }
    return result;
}
```

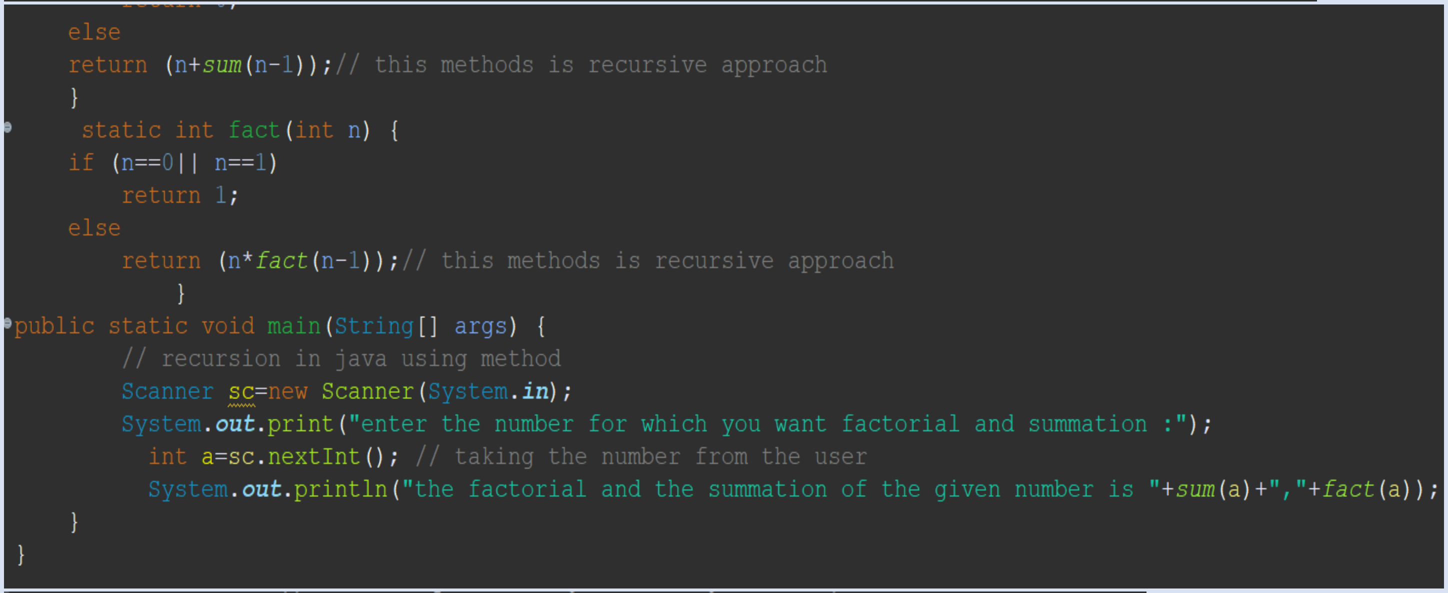
- ***Recursion in Java:-***

One does not simply understand RECURSION without understanding RECURSION.

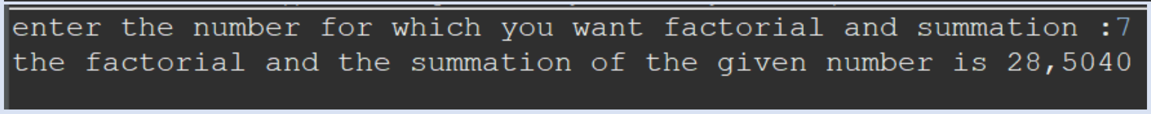
- In programming, recursion is a technique through which a function calls itself.
- With the help of recursion, we can break down complex problems into simple problems.



```
package basic_java;
import java.util.Scanner;
public class Recursion {
    static int sum(int n)  {
    if  (n==0)
        return 0;
```

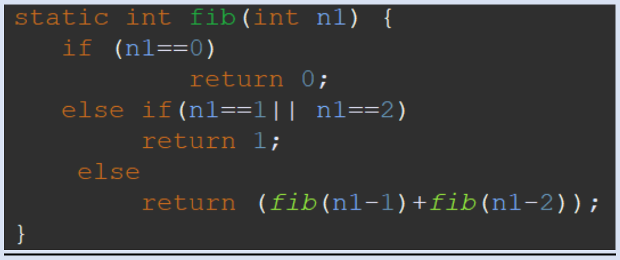


```
    else
    return (n+sum(n-1));// this methods is recursive approach
    }
     static int fact(int n) {
    if (n==0|| n==1)
        return 1;
    else
        return (n*fact(n-1));// this methods is recursive approach
            }
public static void main(String[] args) {
        // recursion in java using method
        Scanner sc=new Scanner(System.in);
        System.out.print("enter the number for which you want factorial and summation :");
          int a=sc.nextInt(); // taking the number from the user
          System.out.println("the factorial and the summation of the given number is "+sum(a)+","+fact(a));
    }
}
```

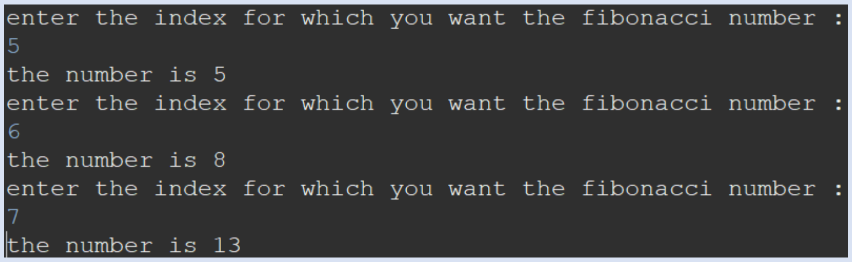


```
enter the number for which you want factorial and summation :7
the factorial and the summation of the given number is 28,5040
```

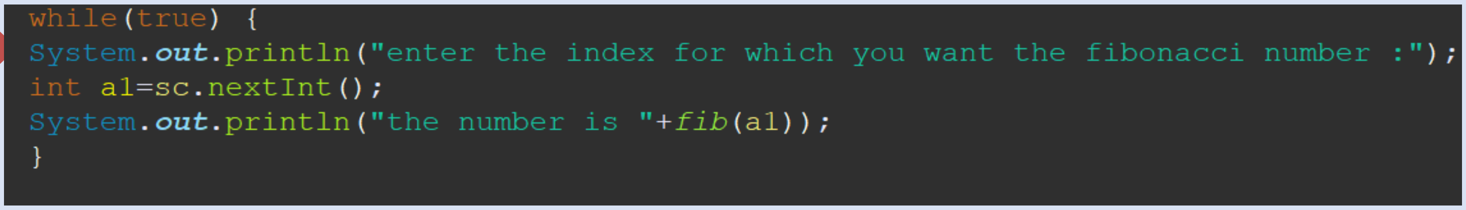
- ***Fibonacci series:-***



```
static int fib(int n1) {
   if (n1==0)
            return 0;
   else if(n1==1|| n1==2)
         return 1;
    else
         return (fib(n1-1)+fib(n1-2));
}
```



```
enter the index for which you want the fibonacci number :
5
the number is 5
enter the index for which you want the fibonacci number :
6
the number is 8
enter the index for which you want the fibonacci number :
7
the number is 13
```



```
while(true) {
System.out.println("enter the index for which you want the fibonacci number :");
int a1=sc.nextInt();
System.out.println("the number is "+fib(a1));
}
```

OBJECT ORIENTED PROGRAMMING SYSTEMS
JAVA

- **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts defined below :
- **Class** is a **user-defined data type** which defines its properties and its functions. Class is the only logical representation of the data. For example, Human being is a class. The body parts of a human being are its properties, and the actions performed by the body parts are known as functions. The class does not occupy any memory space till the time an object is instantiated.
- **Object is a run-time entity**. It is an instance of the class. An object can represent a person, place or any other item. An object can operate on both data members and member functions**.**

Four pillars of Object-Oriented-Programming Language :-

1. Abstraction :-

- Let's suppose you want to turn on the bulb in your room. What do you do to switch on the bulb. You simply press the button and the light bulb turns on. Right? Notice that here you're only concerned with your final result, i.e., turning on the light bulb. You do not care about the circuit of the bulb or how current flows through the bulb. The point here is that you press the switch, the bulb turns on! You don't know how the bulb turned on/how the circuit is made because all these details are hidden from you. This phenomenon is known as abstraction.
- More formally, data abstraction is the way through which only the essential info is shown to the user, and all the internal details remain hidden from the user.

2. Polymorphism :

- One entity many forms.
- The word polymorphism comprises two words, poly which means many, and morph, which means forms.
- In OOPs, polymorphism is the property that helps to perform a single task in different ways.
- Let us consider a real-life example of polymorphism. A woman at the same time can be a mother, wife, sister, daughter, etc. Here, a woman is an entity having different forms.
- Let's take another example, a smartphone can work like a camera as well as like a calculator. So, you can see the a smartphone is an entity having different forms. Also :

3. Encapsulation :

- The act of putting various components together (in a capsule).
- In java, the variables and methods are the components that are wrapped inside a single unit named class.

- All the methods and variables of a class remain hidden from any other class.
- A automatic cold drink vending machine is an example of encapsulation.
- Cold drinks inside the machine are data that is wrapped inside a single unit cold drink vending machine.

4. Inheritance :

- The act of deriving new things from existing things.
- In Java, one class can acquire all the properties and behaviours of other some other class
- The class which inherits some other class is known as child class or sub class.
- The class which is inherited is known as parent class or super class.
- Inheritance helps us to write more efficient code because it increases the reusablity of the code.
- Example :
- Rickshaw → E-Rickshaw
- Phone → Smart Phone

Apart from these concepts, there are some other terms which are used in Object-Oriented design:

- **Coupling**
- **Cohesion**
- **Association**
- **Aggregation**
- **Composition**

- ***Coupling:-***

Coupling refers to the knowledge or information or dependency of another class. It arises when classes are aware of each other. If a class has the details information of another class, there is strong coupling. In Java, we use private, protected, and public modifiers to display the visibility level of a class, method, and field. You can use interfaces for the weaker coupling because there is no concrete implementation.

- ***Cohesion:-***

Cohesion refers to the level of a component which performs a single well-defined task. A single well-defined task is done by a highly cohesive method. The weakly cohesive method will split the task into separate parts. The java.io package is a highly cohesive package because it has I/O related classes and interface. However, the java.util package is a weakly cohesive package because it has unrelated classes and interfaces.

- ***Association:-***

Association represents the relationship between the objects. Here, one object can be associated with one object or many objects. There can be four types of association between the objects.

- One to One
- One to Many
- Many to One, and
- Many to Many

Let's understand the relationship with real-time examples. For example, One country can have one prime minister (one to one), and a prime minister can have many ministers (one to many). Also, many MP's can have one prime minister (many to one), and many ministers can have many departments (many to many).

Association can be undirectional or bidirectional.

- ***Aggregation:-***

Aggregation is a way to achieve Association. Aggregation represents the relationship where one object contains other objects as a part of its state. It represents the weak relationship between objects. It is also termed as a has-a relationship in Java. Like, inheritance represents the is-a relationship. It is another way to reuse objects.

- ***Composition:-***

The composition is also a way to achieve Association. The composition represents the relationship where one object contains other objects as a part of its state. There is a strong relationship between the containing object and the dependent object. It is the state where containing objects do not have an independent existence. If you delete the parent object, all the child objects will be deleted automatically.

• *Java Naming Convention:-*

- Java naming convention is a rule to follow as you decide what to name your identifiers such as class, package, variable, constant, method, etc.
- But, it is not forced to follow. So, it is known as convention not rule. These conventions are suggested by several Java communities such as Sun Microsystems and Netscape.
- All the classes, interfaces, packages, methods and fields of Java programming language are given according to the Java naming convention. If you fail to follow these conventions, it may generate confusion or erroneous code.
- ***Advantage of Naming Conventions in Java:-***
- By using standard Java naming conventions, you make your code easier to read for yourself and other programmers. Readability of Java program is very important. It indicates that less time is spent to figure out what the code does.

Naming Conventions of the Different Identifiers:-

The following examples shows the popular conventions used for the different identifiers.

Clas,Interface ,Constant ,Variable ,Method ,Package and many more which follows CamelCase in Java Name Convention,

Java Tutorial: Creating Our Own Java Class:-

Writing a Custom Class :

Syntax of a custom class :

```
class <class_name>{
    field;
    method;
}
```

Note: The first letter of a class should always be capital.

- Any real-world object = Properties + Behavior
- Object in OOPs = Attributes + Methods

There are many ways to create an object in java. They are:-

- **By new keyword:-**

```
Employee uday = new Employee();
```

- **By newInstance() method:-**

```
new Employee().info(125,210210210);
```

- **By clone() method:-**

```
Employee anshul=new Employee(),yadav=new Employee();
```

- **By deserialization**
- **By factory method etc.**

Example as code:-

```
//creating a first class in java
class Employee {           // craeting a class name Employee
    int ID;                 // class name is always capital
    String name;
    float salary;
    public void set_val(int n, String a) { // creating public method
        ID = n; // we can access using the object
        name = a;
    }
    public void get_val() { // another public method
        System.out.println("the ID no of the employee is " + ID + " and the name is " + name);
        System.out.println("the salary of the employee is " + sal());
    }
    public float sal() { // another method with the return argument
        return salary;
    }
    public void info(int rollno,int collage_id) {
        System.out.println("the rollno is "+rollno+" and the id is "+collage_id);
    }}
public class JavaClassIntro {
    public static void main(String[] args) {
//      Creating a class object using the new keyword
        Employee uday = new Employee();
        Employee vat = new Employee();
        uday.salary = 45000.045f;
        uday.set_val(123, "Uday Sharma"); // giving the value of the method
        uday.get_val(); // calling the method
        vat.set_val(124, "vasu tirpathi");
```

```
        vat.salary = 1230465.45f;
        vat.get_val();
        // creating a class object using a new Instance() methods
        new Employee().info(125,210210210);
//      creating a Multiple object using clone() method
         Employee anshul=new Employee(),yadav=new Employee();
         anshul.set_val(126, "ANSHUL");
         anshul.salary=55000f;
         yadav.salary=45000f;
         yadav.set_val(127, "YADAV");
         anshul.get_val();
         yadav.get_val();
}}
```

Output:-

```
the ID no of the employee is 123 and the name is Uday Sharma
the salary of the employee is 45000.047
the ID no of the employee is 124 and the name is vasu tirpathi
the salary of the employee is 1230465.5
the rollno is 125 and the id is 210210210
the ID no of the employee is 126 and the name is ANSHUL
the salary of the employee is 55000.0
the ID no of the employee is 127 and the name is YADAV
the salary of the employee is 45000.0
```

Question on java class:-

1. Create a class cellphone with methods to print “ringing…”, “vibrating…”, etc.
2. Create a class Square with a method to initialize its side, calculating area, perimeter etc.
3. Create a class Rectangle & problem 3.
4. Create a class TommyVecetti for Rockstar Games capable of hitting (print hitting…), running, firing, etc.
5. Repeat problem 4 for a circle.

Code :-

- ***Java Tutorial: Access modifiers, getters & setters in Java***

Access Modifiers

Access Modifiers specify where a property/method is accessible. There are four types of access modifiers in java :

1. private
2. default
3. protected
4. public

Access Modifier	within class	within package	outside package by subclass only	outside package
public	Y	Y	Y	Y
protected	Y	Y	Y	N
Default	Y	Y	N	N
private	Y	N	N	N

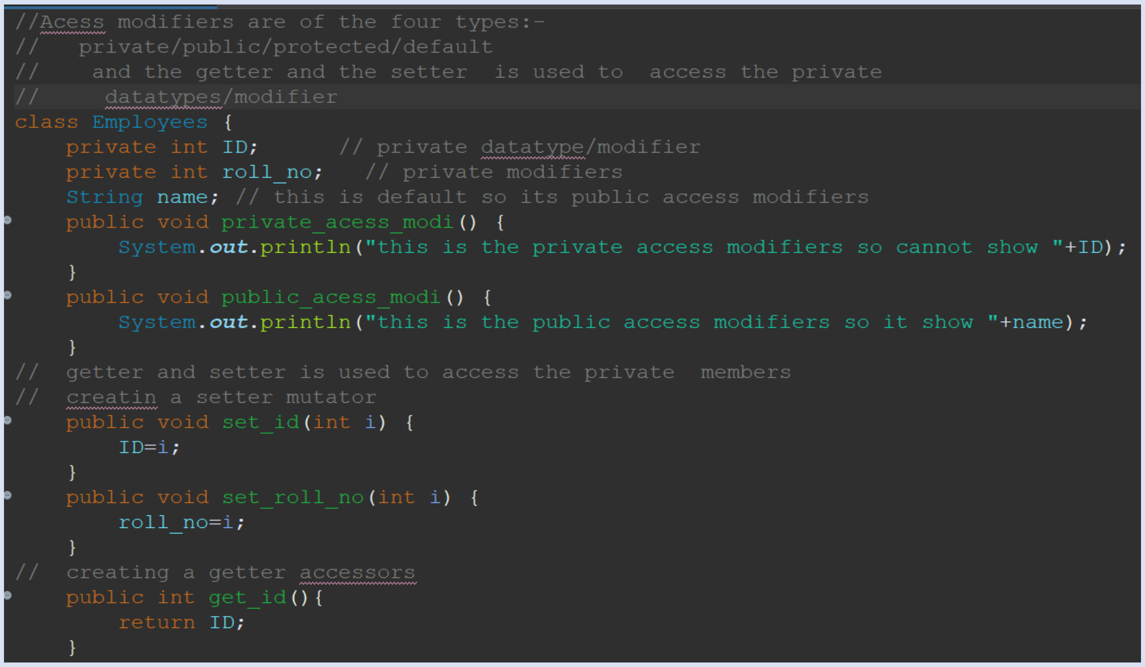
Getters and Setters :

- Getter ➳ Returns the value [accessors]
- setter ➳ Sets / updates the value [mutators]

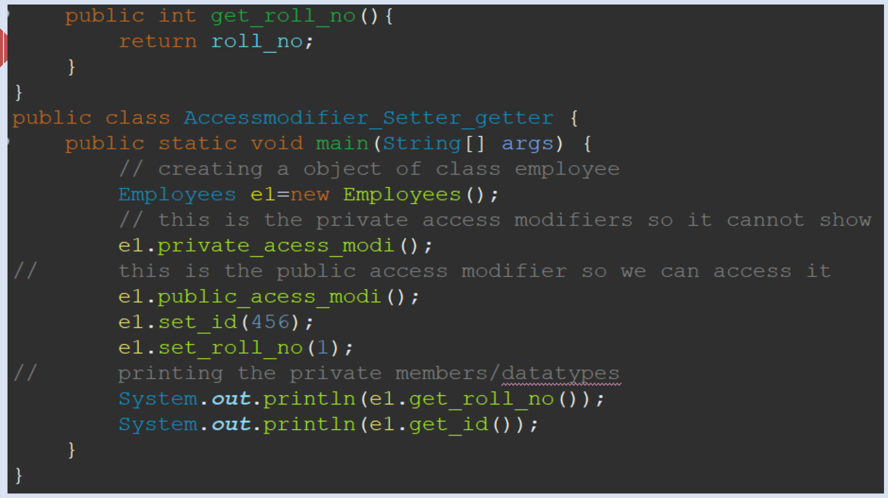
In the below code, we've created total 4 methods:

1. ***setName( ):*** The argument passed to this method is assigned to the private variable name.
2. ***getName( ):*** The method returns the value set by the setName() method.
3. ***setId( ):*** The integer argument passed to this method is assigned to the private variable id.
4. ***getId( ):*** This method returns the value set by the setId() method.

Code:-

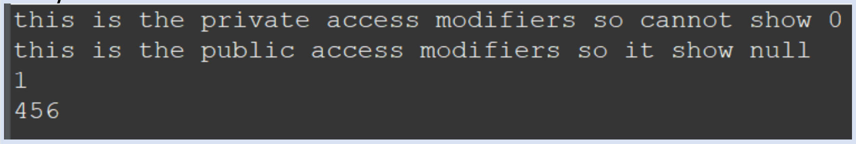


```java
//Acess modifiers are of the four types:-
//   private/public/protected/default
//    and the getter and the setter  is used to  access the private
//     datatypes/modifier
class Employees {
    private int ID;      // private datatype/modifier
    private int roll_no;   // private modifiers
    String name; // this is default so its public access modifiers
    public void private_acess_modi() {
        System.out.println("this is the private access modifiers so cannot show "+ID);
    }
    public void public_acess_modi() {
        System.out.println("this is the public access modifiers so it show "+name);
    }
//  getter and setter is used to access the private  members
//  creatin a setter mutator
    public void set_id(int i) {
        ID=i;
    }
    public void set_roll_no(int i) {
        roll_no=i;
    }
//  creating a getter accessors
    public int get_id(){
        return ID;
    }
```



```
    public int get_roll_no(){
        return roll_no;
    }
}
public class Accessmodifier_Setter_getter {
    public static void main(String[] args) {
        // creating a object of class employee
        Employees e1=new Employees();
        // this is the private access modifiers so it cannot show
        e1.private_acess_modi();
//      this is the public access modifier so we can access it
        e1.public_acess_modi();
        e1.set_id(456);
        e1.set_roll_no(1);
//      printing the private members/datatypes
        System.out.println(e1.get_roll_no());
        System.out.println(e1.get_id());
    }
}
```

Output:-



```
this is the private access modifiers so cannot show 0
this is the public access modifiers so it show null
1
456
```

Constructors in Java:-

- ***Constructors in Java : -***
  - Constructors are similar to methods,, but they are used to initialize an object.
  - Constructors do not have any return type(not even void).
  - Every time we create an object by using the new() keyword, a constructor is called.
  - If we do not create a constructor by ourself, then the default constructor(created by Java compiler) is called.
- ***Rules for creating a Constructor :-***
  1. The class name and constructor name should be the same.
  2. It must have no explicit return type.
  3. It can not be abstract, static, final, and synchronized.
- ***Types of Constructors in Java :-***

There are two types of constructors in Java :

1. ***Defaut /Non-Parameterized constructor :*** A constructor with 0 parameters is known as default constructor. A constructor which has no argument is known as non-parameterized constructor(or no-argument constructor). It is invoked at the time of creating an object. If we don't create one then it is created by default by Java.

Syntax :

```
<class_name>(){
//code to be executed on the execution of the constructor
}
```

2. ***Paramerterized constructor :*** A constructor with some specified number of parameters is known as a parameterized constructor.Constructor which has parameters is called a parameterized constructor. It is used to provide different values to distinct objects.

Syntax :

```
<class-name>(<data-type> param1, <data-type> param2,......){
//code to be executed on the invocation of the constructor
}
```

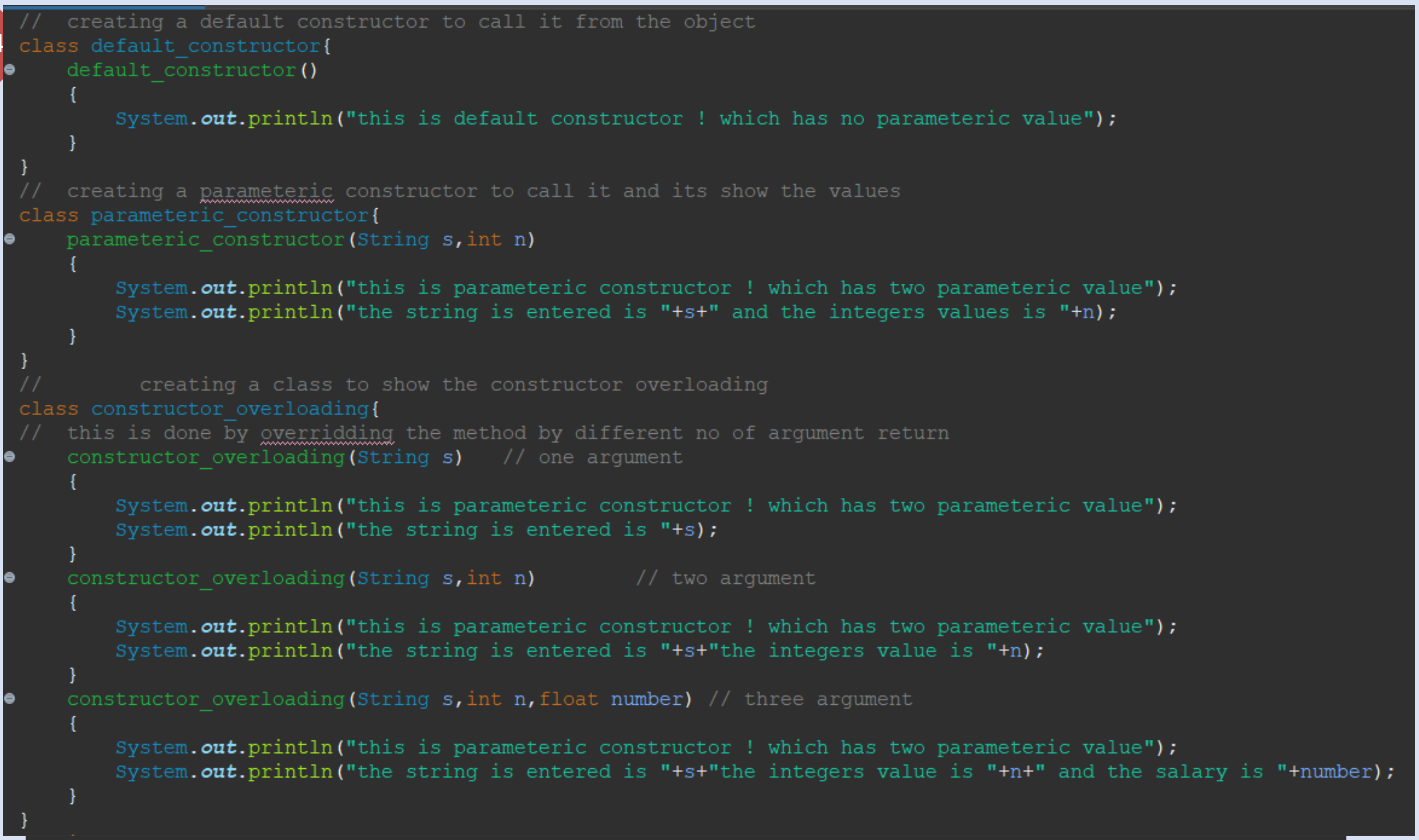
3. ***Constructor Overloading in Java :***constructor used to declare and initialize an object from another object. There is only a user defined copy constructor in Java(C++ has a default one too).
4. ***CopyConstructor:-***is used to copy the data of the other constructor.

```
public Employee (String n)
           name = n;
         }
```

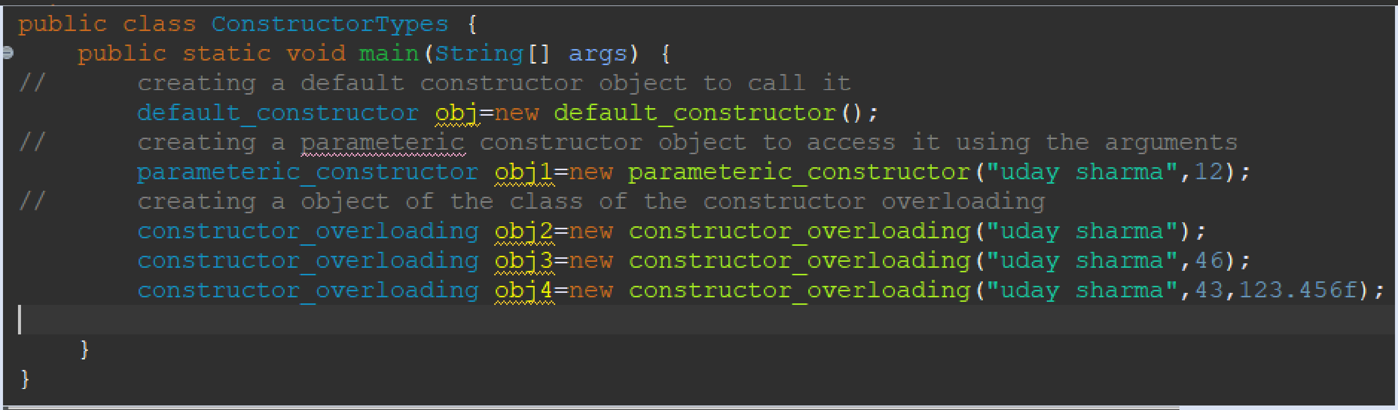
Note:-

1. Constructors can take parameters without being overloaded
2. There can be more than two overloaded constructors.
3. **Note : -** When an object is created **using** a new keyword, then space is allocated for the variable in a heap, and the starting address is stored in the stack memory.
4. **'this' keyword :** 'this' keyword in Java that refers to the current instance of the class. In OOPS it is used to:-
5. pass the current object as a parameter to another method
6. refer to the current class instance variable.

Java Constructor	**Java Method**
A constructor is used to initialize the state of an object.	A method is used to expose the behavior of an object.
A constructor must not have a return type.	A method must have a return type.
The constructor is invoked implicitly.	The method is invoked explicitly.
The Java compiler provides a default constructor if you don't have any constructor in a class.	The method is not provided by the compiler in any case.
The constructor name must be same as the class name.	The method name may or may not be same as the class name.

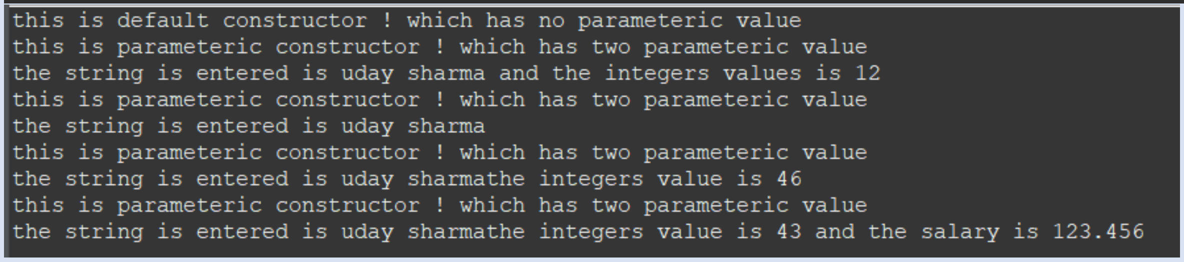


```java
//  creating a default constructor to call it from the object
class default_constructor{
    default_constructor()
    {
        System.out.println("this is default constructor ! which has no parameteric value");
    }
}
//  creating a parameteric constructor to call it and its show the values
class parameteric_constructor{
    parameteric_constructor(String s,int n)
    {
        System.out.println("this is parameteric constructor ! which has two parameteric value");
        System.out.println("the string is entered is "+s+" and the integers values is "+n);
    }
}
//        creating a class to show the constructor overloading
class constructor_overloading{
//  this is done by overridding the method by different no of argument return
    constructor_overloading(String s)   // one argument
    {
        System.out.println("this is parameteric constructor ! which has two parameteric value");
        System.out.println("the string is entered is "+s);
    }
    constructor_overloading(String s,int n)        // two argument
    {
        System.out.println("this is parameteric constructor ! which has two parameteric value");
        System.out.println("the string is entered is "+s+"the integers value is "+n);
    }
    constructor_overloading(String s,int n,float number) // three argument
    {
        System.out.println("this is parameteric constructor ! which has two parameteric value");
        System.out.println("the string is entered is "+s+"the integers value is "+n+" and the salary is "+number);
    }
}
```



```java
public class ConstructorTypes {
    public static void main(String[] args) {
//      creating a default constructor object to call it
        default_constructor obj=new default_constructor();
//      creating a parameteric constructor object to access it using the arguments
        parameteric_constructor obj1=new parameteric_constructor("uday sharma",12);
//      creating a object of the class of the constructor overloading
        constructor_overloading obj2=new constructor_overloading("uday sharma");
        constructor_overloading obj3=new constructor_overloading("uday sharma",46);
        constructor_overloading obj4=new constructor_overloading("uday sharma",43,123.456f);

    }
}
```



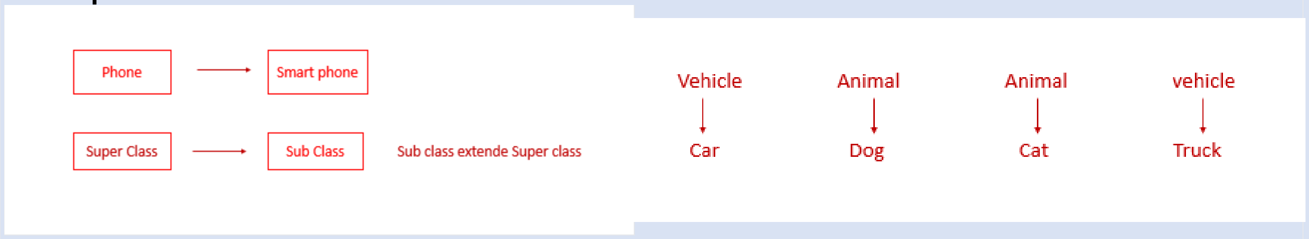
```
this is default constructor ! which has no parameteric value
this is parameteric constructor ! which has two parameteric value
the string is entered is uday sharma and the integers values is 12
this is parameteric constructor ! which has two parameteric value
the string is entered is uday sharma
this is parameteric constructor ! which has two parameteric value
the string is entered is uday sharmathe integers value is 46
this is parameteric constructor ! which has two parameteric value
the string is entered is uday sharmathe integers value is 43 and the salary is 123.456
```

• ***<u>Inheritance:-</u>***

- Inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such a way, you can reuse, extend or modify the attributes and behaviors which are defined in other classes.
- In Java, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.
- Inheritance is used to borrow properties & methods from an existing class.

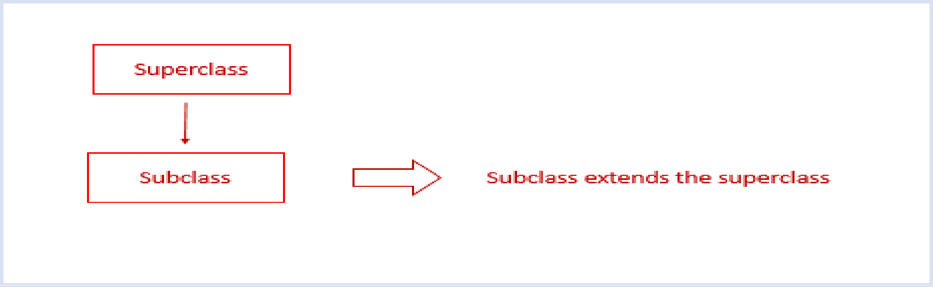
- Inheritance helps us create classes based on existing classes, which increases the code's reusability.
- Inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such a way, you can reuse, extend or modify the attributes and behaviors which are defined in other classes.
- In Java, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

Examples :



Important terminologies used in Inheritance :

1. Parent class/superclass: The class from which a class inherits methods and attributes is known as parent class.
2. Child class/sub-class: The class that inherits some other class's methods and attributes is known as child class.



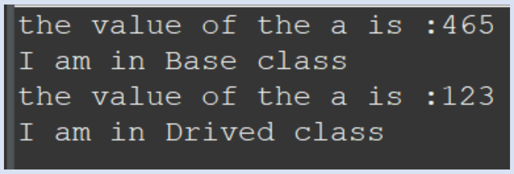
Extends keyword in inheritance :

- The **extends** keyword is used to inherit a subclass from a superclass.

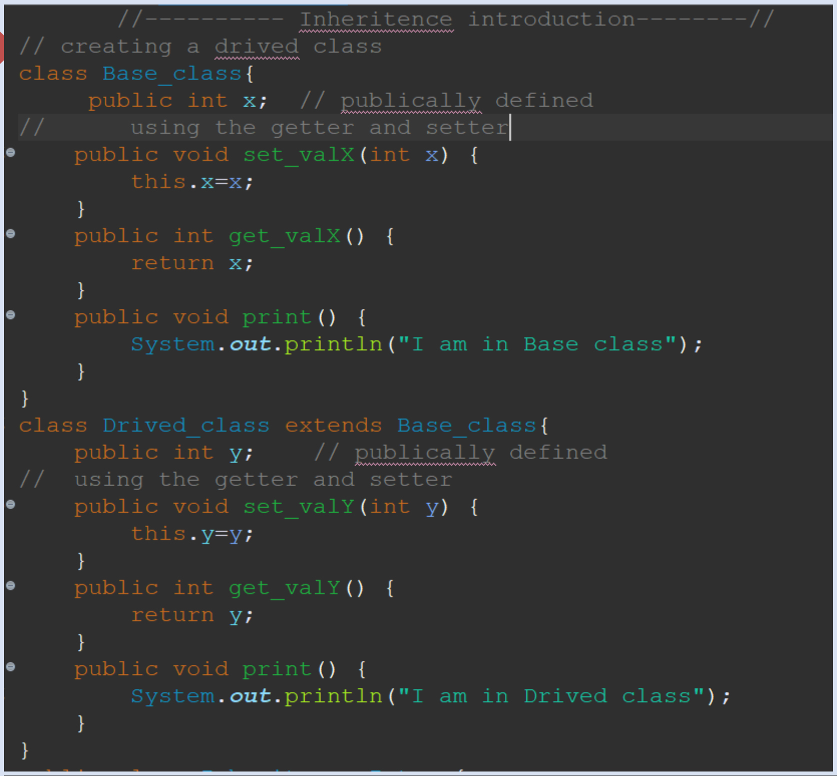
Syntax :

```
class Subclass-name extends Superclass-name
{
   //methods and fields
}
```

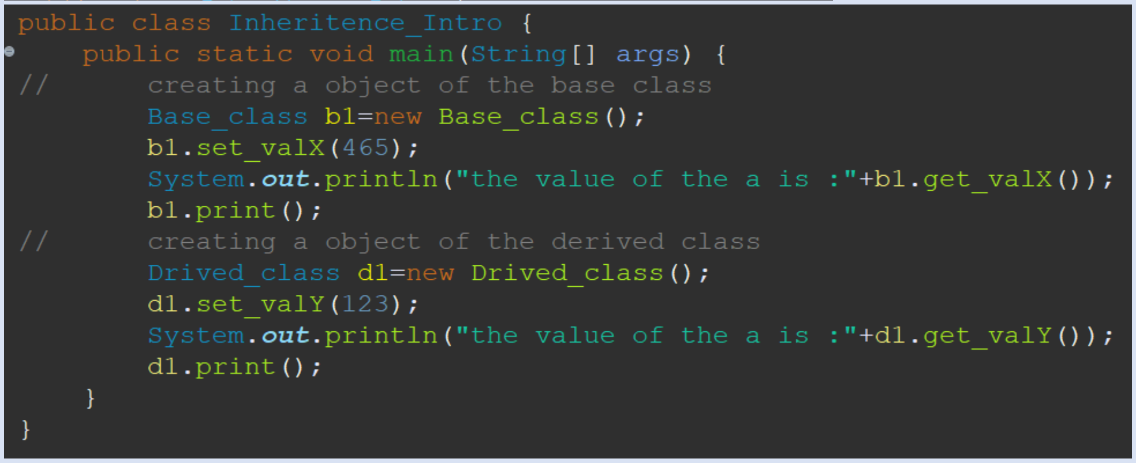
Code:-



```
the value of the a is :465
I am in Base class
the value of the a is :123
I am in Drived class
```



```
        //---------- Inheritence introduction--------//
// creating a drived class
class Base_class{
     public int x;  // publically defined
//       using the getter and setter
    public void set_valX(int x) {
        this.x=x;
    }
    public int get_valX() {
        return x;
    }
    public void print() {
        System.out.println("I am in Base class");
    }
}
class Drived_class extends Base_class{
    public int y;    // publically defined
//  using the getter and setter
    public void set_valY(int y) {
        this.y=y;
    }
    public int get_valY() {
        return y;
    }
    public void print() {
        System.out.println("I am in Drived class");
    }
}
```



```
public class Inheritence_Intro {
    public static void main(String[] args) {
//      creating a object of the base class
        Base_class b1=new Base_class();
        b1.set_valX(465);
        System.out.println("the value of the a is :"+b1.get_valX());
        b1.print();
//      creating a object of the derived class
        Drived_class d1=new Drived_class();
        d1.set_valY(123);
        System.out.println("the value of the a is :"+d1.get_valY());
        d1.print();
    }
}
```

- ***Types of Inheritance :***

1***. Single inheritance : -***When one class inherits another class, it is known as single level inheritance

```
class Shape {
  public void area() {
    System.out.println("Displays Area of Shape");
  }}
class Triangle extends Shape {
  public void area(int h, int b) {
    System.out.println((1/2)*b*h);
  }}
```

2. Hierarchical inheritance:- Hierarchical inheritance is defined as the process of deriving more than one class from a base class.

```
class Shape {
  public void area() {
     System.out.println("Displays Area of Shape");
  }}
class Triangle extends Shape {
  public void area(int h, int b) {
     System.out.println((1/2)*b*h);
  } }
class Circle extends Shape {
  public void area(int r) {
     System.out.println((3.14)*r*r);
  } }
```

3. ***Multilevel inheritance :-*** Multilevel inheritance is a process of deriving a class from another derived class.

```
class Shape {
  public void area() {
     System.out.println("Displays Area of Shape");
  }}
class Triangle extends Shape {
  public void area(int h, int b) {
     System.out.println((1/2)*b*h);
  } }
class EquilateralTriangle extends Triangle {
  int side;
}
```

4. Hybrid inheritance : -Hybrid inheritance is a combination of simple, multiple inheritance and hierarchical inheritance.

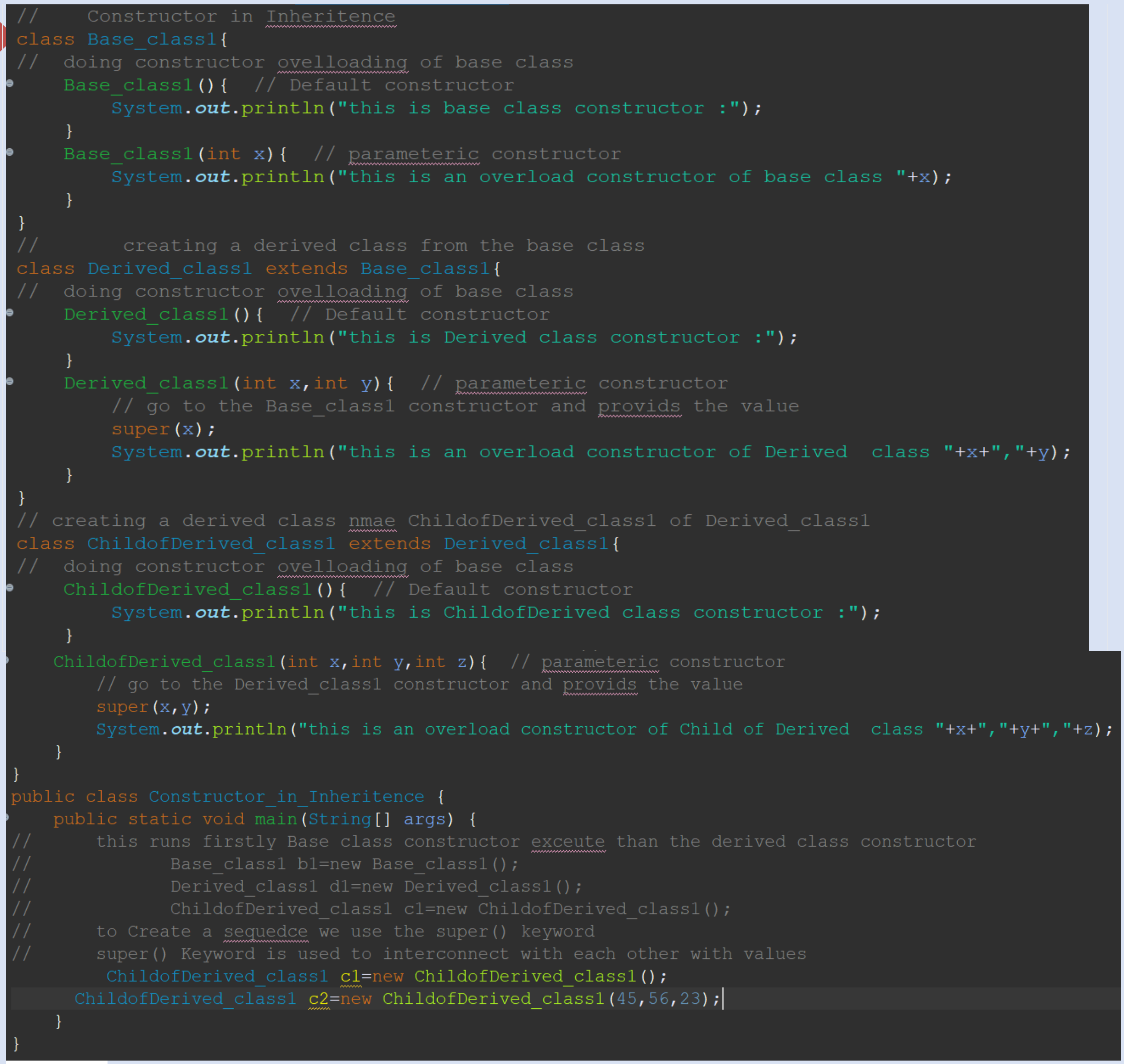
- ***Constructor order in the inheritance :-***

When a drived class is extended from the base class, the constructor of the base class is executed first followed by the constructor of the derived class. For the following Inheritance hierarchy , the constructors are executed in the order:

1. C1- Parent
2. C2 - Child
3. C3 - Grandchild

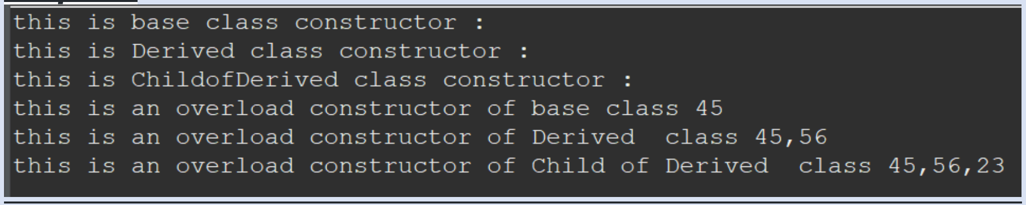
- ***Constructors during constructor overloading :-***
  - When there are multiple constructors in the parent class, the constructor without any parameters is called from the child class.
  - If we want to call the constructor with parameters from the parent class, we can use the super keyword.
  - super(a, b) calls the constructor from the parent class which takes 2 variables.

Code:-



```java
//    Constructor in Inheritence
class Base_class1{
//  doing constructor ovelloading of base class
    Base_class1(){  // Default constructor
        System.out.println("this is base class constructor :");
    }
    Base_class1(int x){  // parameteric constructor
        System.out.println("this is an overload constructor of base class "+x);
    }
}
//        creating a derived class from the base class
class Derived_class1 extends Base_class1{
//  doing constructor ovelloading of base class
    Derived_class1(){  // Default constructor
        System.out.println("this is Derived class constructor :");
    }
    Derived_class1(int x,int y){  // parameteric constructor
        // go to the Base_class1 constructor and provids the value
        super(x);
        System.out.println("this is an overload constructor of Derived  class "+x+","+y);
    }
}
// creating a derived class nmae ChildofDerived_class1 of Derived_class1
class ChildofDerived_class1 extends Derived_class1{
//  doing constructor ovelloading of base class
    ChildofDerived_class1(){  // Default constructor
        System.out.println("this is ChildofDerived class constructor :");
    }
    ChildofDerived_class1(int x,int y,int z){  // parameteric constructor
        // go to the Derived_class1 constructor and provids the value
        super(x,y);
        System.out.println("this is an overload constructor of Child of Derived  class "+x+","+y+","+z);
    }
}
public class Constructor_in_Inheritence {
    public static void main(String[] args) {
//      this runs firstly Base class constructor exceute than the derived class constructor
//             Base_class1 b1=new Base_class1();
//             Derived_class1 d1=new Derived_class1();
//             ChildofDerived_class1 c1=new ChildofDerived_class1();
//      to Create a sequedce we use the super() keyword
//      super() Keyword is used to interconnect with each other with values
        ChildofDerived_class1 c1=new ChildofDerived_class1();
      ChildofDerived_class1 c2=new ChildofDerived_class1(45,56,23);
    }
}
```

Output:-



```
this is base class constructor :
this is Derived class constructor :
this is ChildofDerived class constructor :
this is an overload constructor of base class 45
this is an overload constructor of Derived  class 45,56
this is an overload constructor of Child of Derived  class 45,56,23
```

- ***this and super keyword in Java:-***
- ***this keyword in Java :***
  - this is a way for us to reference an object of the class which is being created/referenced.
  - It is used to call the default constructor of the same class.
  - **this** keyword eliminates the confusion between the parameters and the class attributes with the same name.

- ***Super keyword :-***
  - A reference variable used to refer immediate parent class object.
  - It can be used to refer immediate parent class instance variable.
  - It can be used to invoke the parent class method.
- ***Final Keyword In Java:-***

The final keyword in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only.

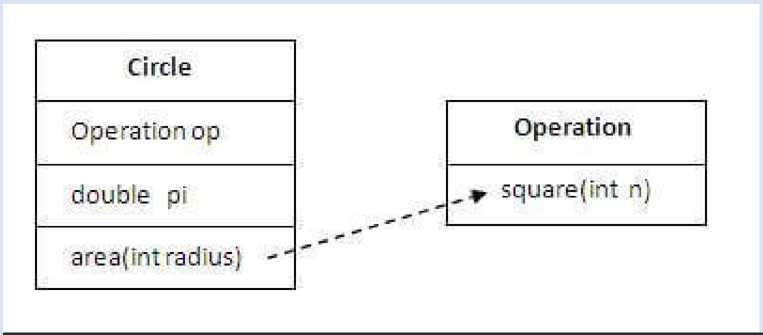
- ***Aggregation in Java:-***

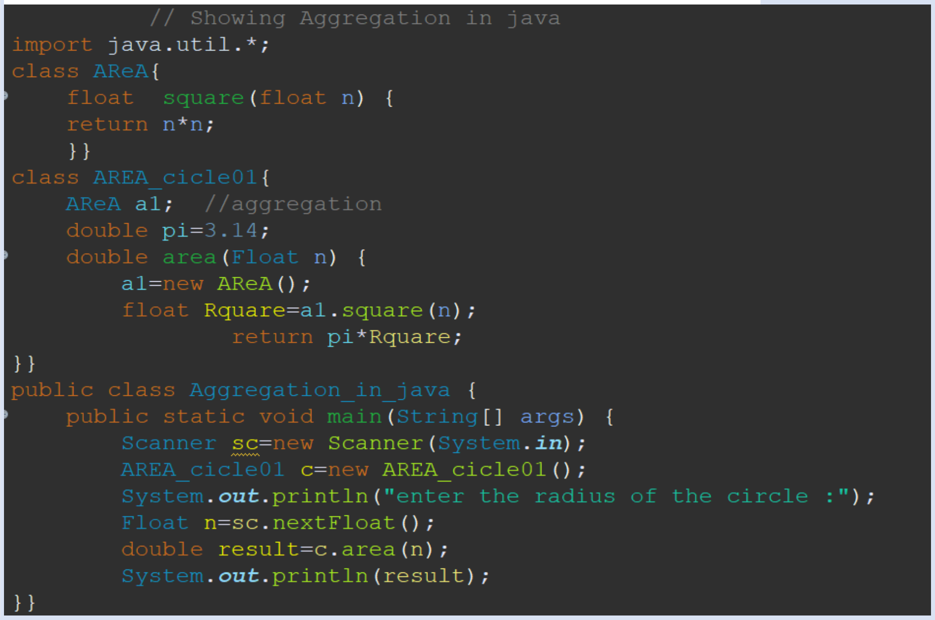
If a class have an entity reference, it is known as Aggregation. Aggregation represents **HAS-A** relationship.
Consider a situation, Employee object contains many informations such as id, name, email_Id etc. It contains one more object named address, which contains its own informations such as city, state, country, zipcode etc. as given below.

Why use Aggregation?

- For Code Reusability.

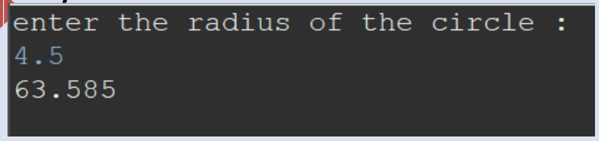
Simple Example of Aggregation





```
          // Showing Aggregation in java
import java.util.*;
class AReA{
    float  square(float n) {
    return n*n;
    }}
class AREA_cicle01{
    AReA a1;  //aggregation
    double pi=3.14;
    double area(Float n) {
        a1=new AReA();
        float Rquare=a1.square(n);
                return pi*Rquare;
}}
public class Aggregation_in_java {
    public static void main(String[] args) {
        Scanner sc=new Scanner(System.in);
        AREA_cicle01 c=new AREA_cicle01();
        System.out.println("enter the radius of the circle :");
        Float n=sc.nextFloat();
        double result=c.area(n);
        System.out.println(result);
}}
```

Output:-



```
enter the radius of the circle :
4.5
63.585
```

- ***Polymorphism:-***

Polymorphism is the ability to present the same interface for differing underlying forms (data types). With polymorphism, each of these classes will have different underlying data. Precisely, Poly means 'many' and morphism means 'forms'.

- **Types of Polymorphism:-**

1. Compile Time Polymorphism (Static).
2. Runtime Polymorphism (Dynamic) .

- ***Compile Time Polymorphism*** : The polymorphism which is implemented at the compile time is known as compile-time polymorphism. Example - Method Overloading .
- ***Method Overloading*** : Method overloading is a technique which allows you to have more than one function with the same function name but with different functionality. Method overloading can be possible on the following basis:

1. The type of the parameters/argument passed to the function.
2. The number of parameters passed to the function.
3. We can check the overloding of method in the beginning of the notes of OOPS in JAVA programming .

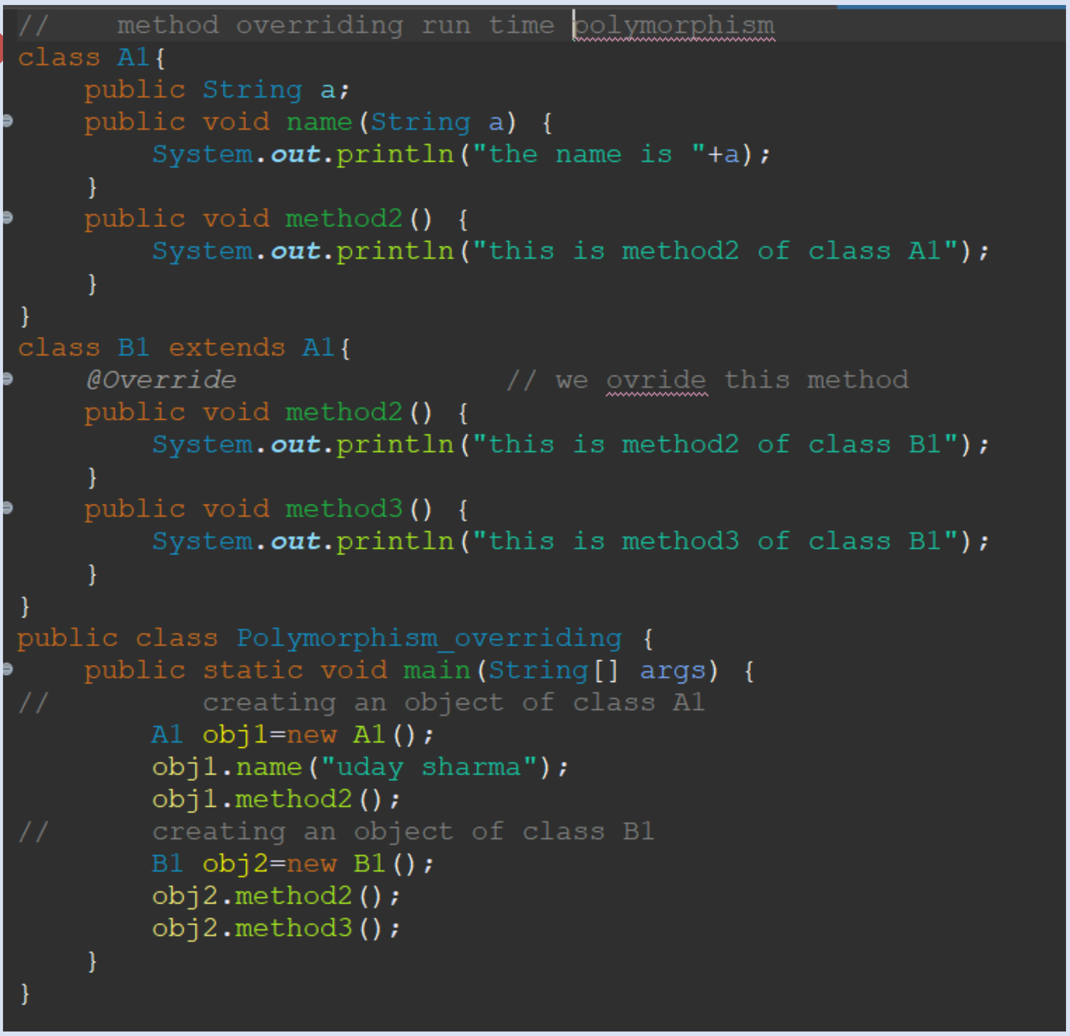
- ***Runtime Polymorphism :*** Runtime polymorphism is also known as dynamic polymorphism. Function overriding is an example of runtime polymorphism. Function overriding means when the child class contains the method which is already present in the parent class. Hence, the child class overrides the method of the parent class. In case of function overriding, parent and child classes both contain the same function with a different definition. The call to the function is determined at runtime is known as runtime polymorphism.
- ***Method Overriding in Java:-***

1. If the child class implements the same method present in the parent class again, it is know as method overriding.
2. Method overriding helps us to classify a behavior that is specific to the child class.
3. The subclass can override the method of the parent class only when the method is not declared as final.

Example :
In the below code, we've created two classes: class A & class B.
Class B is inheriting class A.

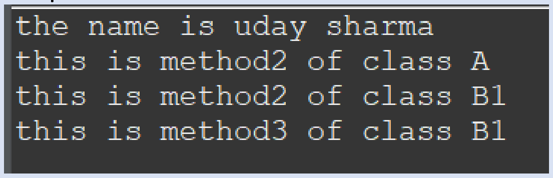
4. In the main() method, we've created one object for both classes. We're running the meth1() method on class A and B objects separately, but the output is the same because the meth1() is defined in the parent class, i.e., class A.
5. Overriding method is denoted by @override which denotes this method is override ,if we override the method then we use override to detect the error . @overriding is not necessary but it is recommended to use in the code .

Code:-



```
//    method overriding run time polymorphism
class A1{
    public String a;
    public void name(String a) {
        System.out.println("the name is "+a);
    }
    public void method2() {
        System.out.println("this is method2 of class A1");
    }
}
class B1 extends A1{
    @Override                    // we ovride this method
    public void method2() {
        System.out.println("this is method2 of class B1");
    }
    public void method3() {
        System.out.println("this is method3 of class B1");
    }
}
public class Polymorphism_overriding {
    public static void main(String[] args) {
//          creating an object of class A1
        A1 obj1=new A1();
        obj1.name("uday sharma");
        obj1.method2();
//      creating an object of class B1
        B1 obj2=new B1();
        obj2.method2();
        obj2.method3();
    }
}
```

Output:-



```
the name is uday sharma
this is method2 of class A
this is method2 of class B1
this is method3 of class B1
```

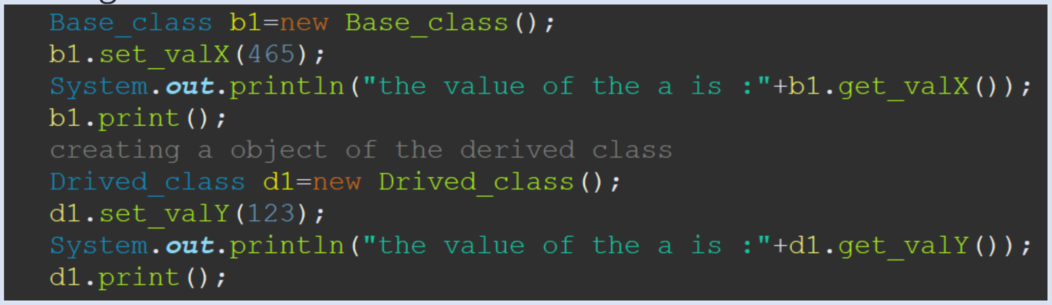
- ***Static Binding and Dynamic Binding***

static binding and dynamic binding in java
Connecting a method call to the method body is known as binding.
There are two types of binding:-

- ***Static Binding*** (also known as Early Binding).
- ***Dynamic Binding*** (also known as Late Binding).

1. ***static binding:-***

When type of the object is determined at compiled time(by the compiler), it is known as static binding.



```
Base_class b1=new Base_class();
b1.set_valX(465);
System.out.println("the value of the a is :"+b1.get_valX());
b1.print();
creating a object of the derived class
Drived_class d1=new Drived_class();
d1.set_valY(123);
System.out.println("the value of the a is :"+d1.get_valY());
d1.print();
```

If there is any private, final or static method in a class, there is static binding.

2. Dynamic Binding Method Dispatch in Java:-

1. Dynamic method dispatch is also known as run time polymorphism.
2. It is the process through which a call to an overridden method is resolved at runtime.
3. This technique is used to resolve a call to an overridden method at runtime rather than compile time.
4. To properly understand Dynamic method dispatch in Java, it is important to understand the concept of upcasting because dynamic method dispatch is based on upcasting.

Upcasting :

- It is a technique in which a superclass reference variable refers to the object of the subclass.

Example :

```
class Animal{}

class Dog extends Animal{}

Animal a=new Dog();//upcasting
```

Example to demonstrate the use of Dynamic method dispatch :

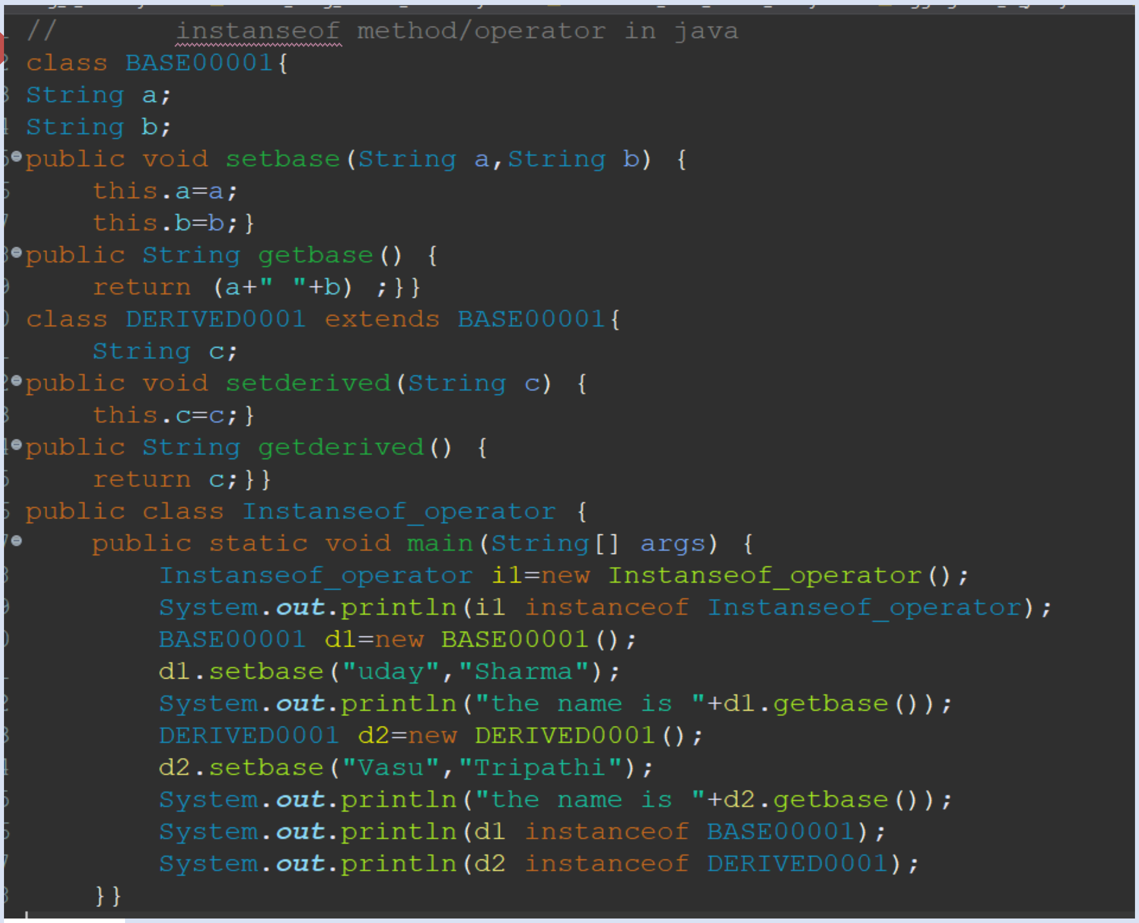
- In the below code, we've created two classes: **Base & Derived**.
- The **Base** is the parent class and the **Derived** is the child class.
- The method **on()** of the parent class is overridden inside the child class.
- Inside the main() method, we've created an object **obj** of the **Derived ()** class by taking the reference of the **Base()** class.
- When **obj.method()** will be executed, it will call the **method()** method of the **Derived ()** class because the reference variable obj is pointing towards the object of class **Derived ()**.

```
public static void main(String[] args) {
        //Base obj = new Base(); // Allowed
        //Derived obj = new Derived(); // Allowed
        // obj.method();
        Base obj = new Derived(); // Yes it is allowed
        //Derived obj2 = new Base(); // Not allowed
        obj.method1();
        obj.method2();
        // obj.method3(); // Derived class method Not Allowed
    }
}
```

- ***Java instanceof operaotor or method:-***

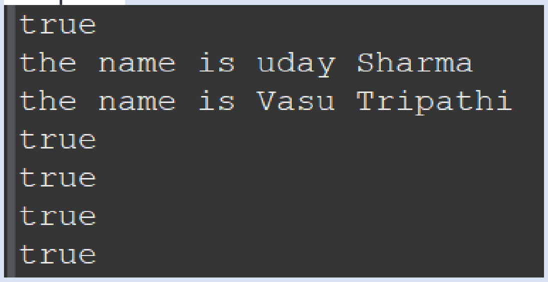
- The java **instanceof** operator is used to test whether the object is an instance of the specified type (class or subclass or interface).
- The **instanceof** in java is also known as type comparison operator because it compares the instance with type. It returns either true or false. If we apply the **instanceof** operator with any variable that has null value, it returns false.

Code:-



```java
//       instanseof method/operator in java
class BASE00001{
String a;
String b;
public void setbase(String a,String b) {
    this.a=a;
    this.b=b;}
public String getbase() {
    return (a+" "+b) ;}}
class DERIVED0001 extends BASE00001{
    String c;
public void setderived(String c) {
    this.c=c;}
public String getderived() {
    return c;}}
public class Instanseof_operator {
    public static void main(String[] args) {
        Instanseof_operator i1=new Instanseof_operator();
        System.out.println(i1 instanceof Instanseof_operator);
        BASE00001 d1=new BASE00001();
        d1.setbase("uday","Sharma");
        System.out.println("the name is "+d1.getbase());
        DERIVED0001 d2=new DERIVED0001();
        d2.setbase("Vasu","Tripathi");
        System.out.println("the name is "+d2.getbase());
        System.out.println(d1 instanceof BASE00001);
        System.out.println(d2 instanceof DERIVED0001);
    }}
```

Output:-



```
true
the name is uday Sharma
the name is Vasu Tripathi
true
true
true
true
```

- ***Downcasting without the use of java instanceof:-***

Downcasting can also be performed without the use of instanceof operator as displayed in the following code:-

Abstraction:-

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

Data binding : Data binding is a process of binding the application UI and business logic. Any change made in the business logic will reflect directly to the application UI.

Abstraction is achieved in 2 ways :-

- **Abstract class.**
- **Interfaces (Pure Abstraction).**
- **Abstract vs Interface.**

What does Abstract mean?

Abstract in English means existing in through or as an idea without concrete existence.

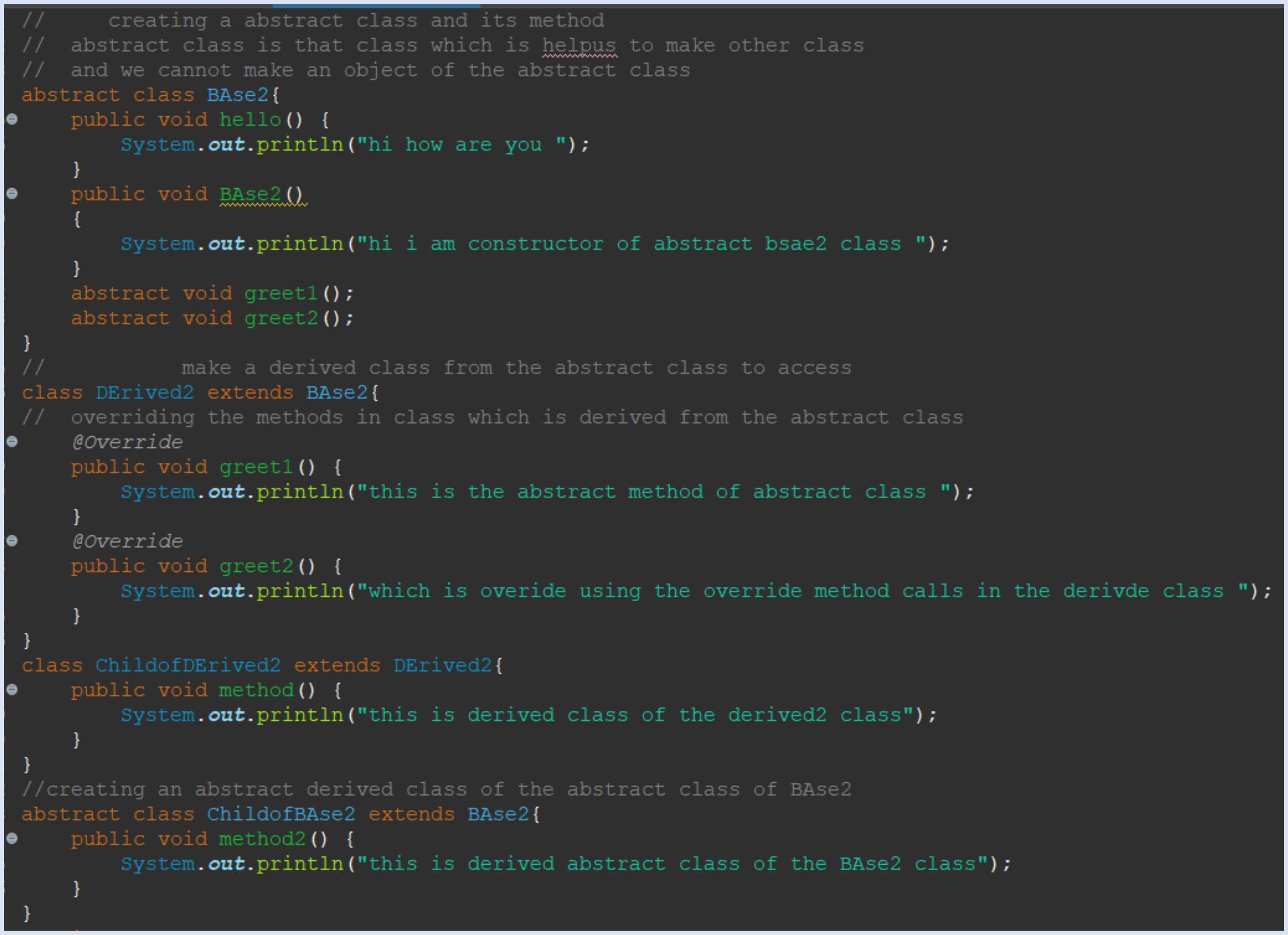
- ***Abstract Class:-***

1. An abstract class must be declared with an abstract keyword.
2. It can have abstract and non-abstract methods.
3. It cannot be instantiated.
4. It can have constructors and static methods also.
5. Abstract class are used when we want to achieve security & abstraction(hide certain details & show only necessary details to the user)
6. It can have final methods which will force the subclass not to change the body of the method.

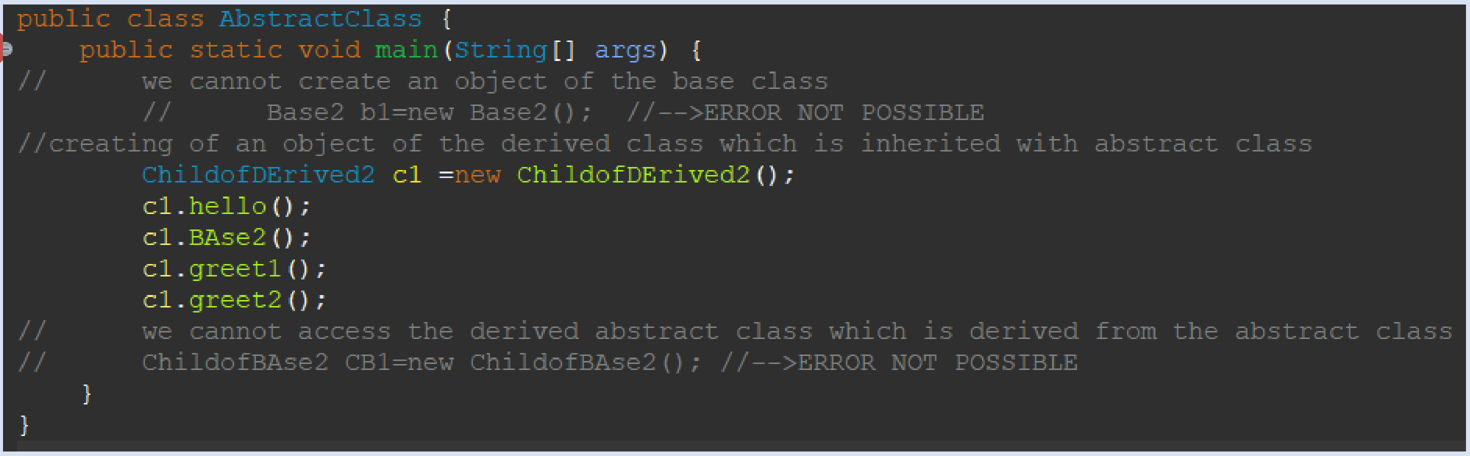
- ***Abstract method :***
  - A method that is declared without implementation is known as the abstract method.
  - An abstract method can only be used inside an abstract class.
  - The body of the abstract method is provided by the class that inherits the abstract class in which the abstract method is present.

```
public abstract class phone Model {
        abstract void switch off ();
         || more code
         }
```

Code :-

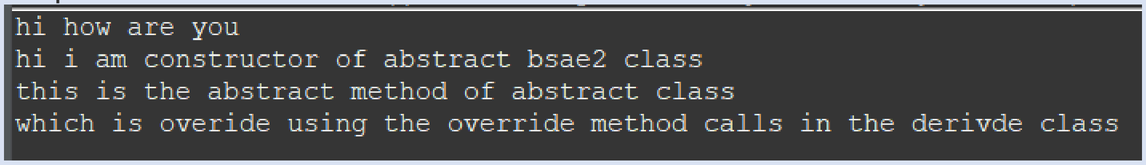


```
//     creating a abstract class and its method
//  abstract class is that class which is helpus to make other class
//  and we cannot make an object of the abstract class
abstract class BAse2{
    public void hello() {
        System.out.println("hi how are you ");
    }
    public void BAse2()
    {
        System.out.println("hi i am constructor of abstract bsae2 class ");
    }
    abstract void greet1();
    abstract void greet2();
}
//            make a derived class from the abstract class to access
class DErived2 extends BAse2{
//  overriding the methods in class which is derived from the abstract class
    @Override
    public void greet1() {
        System.out.println("this is the abstract method of abstract class ");
    }
    @Override
    public void greet2() {
        System.out.println("which is overide using the override method calls in the derivde class ");
    }
}
class ChildofDErived2 extends DErived2{
    public void method() {
        System.out.println("this is derived class of the derived2 class");
    }
}
//creating an abstract derived class of the abstract class of BAse2
abstract class ChildofBAse2 extends BAse2{
    public void method2() {
        System.out.println("this is derived abstract class of the BAse2 class");
    }
}
```



```
public class AbstractClass {
    public static void main(String[] args) {
//      we cannot create an object of the base class
        //      Base2 b1=new Base2();  //-->ERROR NOT POSSIBLE
//creating of an object of the derived class which is inherited with abstract class
        ChildofDErived2 c1 =new ChildofDErived2();
        c1.hello();
        c1.BAse2();
        c1.greet1();
        c1.greet2();
//      we cannot access the derived abstract class which is derived from the abstract class
//      ChildofBAse2 CB1=new ChildofBAse2(); //-->ERROR NOT POSSIBLE
    }
}
```

Output:-

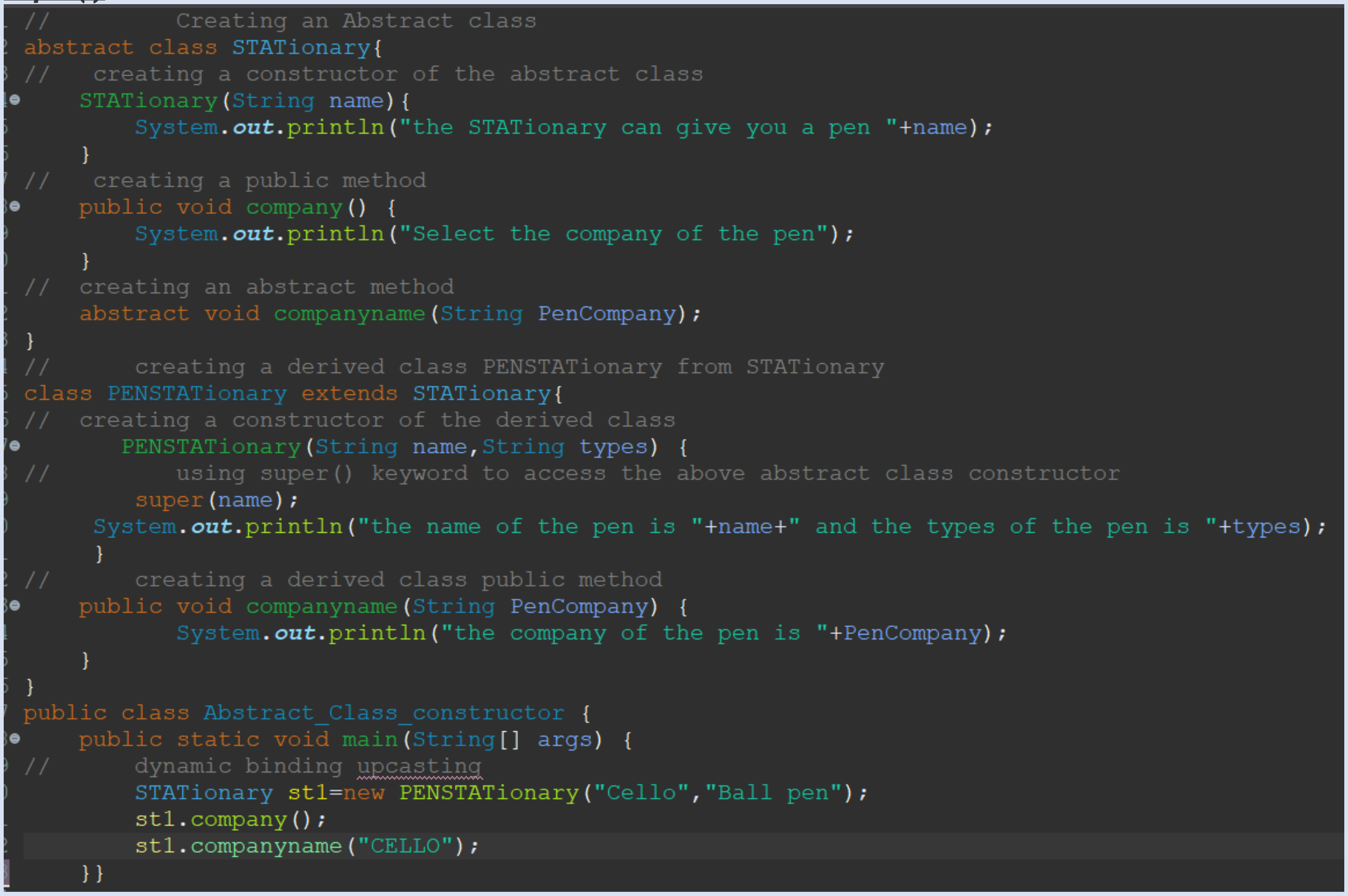


```
hi how are you
hi i am constructor of abstract bsae2 class
this is the abstract method of abstract class
which is overide using the override method calls in the derivde class
```

- ***Abstract class having constructor, data member and methods:-***

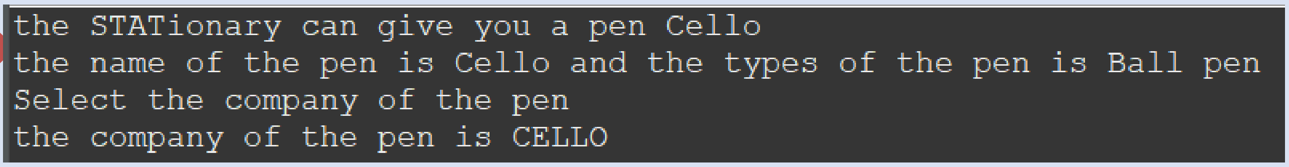
An abstract class can have a data member, abstract method, method body (non-abstract method), constructor, and even main() method.

Code:- Using parametric constructor in both class parent abstract class and the derived class. Using ***super( )*** method to access both constructor.



```
//          Creating an Abstract class
abstract class STATionary{
//   creating a constructor of the abstract class
    STATionary(String name){
        System.out.println("the STATionary can give you a pen "+name);
    }
//   creating a public method
    public void company() {
        System.out.println("Select the company of the pen");
    }
//  creating an abstract method
    abstract void companyname(String PenCompany);
}
//      creating a derived class PENSTATionary from STATionary
class PENSTATionary extends STATionary{
//  creating a constructor of the derived class
       PENSTATionary(String name,String types) {
//          using super() keyword to access the above abstract class constructor
        super(name);
      System.out.println("the name of the pen is "+name+" and the types of the pen is "+types);
       }
//      creating a derived class public method
    public void companyname(String PenCompany) {
          System.out.println("the company of the pen is "+PenCompany);
    }
}
public class Abstract_Class_constructor {
    public static void main(String[] args) {
//      dynamic binding upcasting
        STATionary st1=new PENSTATionary("Cello","Ball pen");
        st1.company();
        st1.companyname("CELLO");
    }}
```

Output:-



```
the STATionary can give you a pen Cello
the name of the pen is Cello and the types of the pen is Ball pen
Select the company of the pen
the company of the pen is CELLO
```

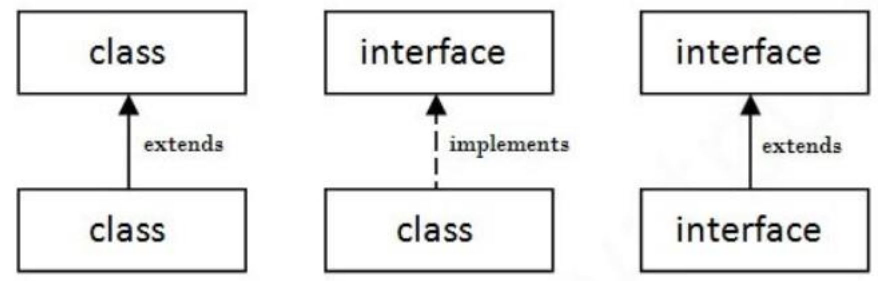
- ***Interfaces***

- All the fields in interfaces are public, static and final by default.
- All methods are public & abstract by default.
- A class that implements an interface must implement all the methods declared in the interface(which is abstract by nature).
- Just like a class in java is a collection of the related methods, an interface in java is a collection of abstract methods.
- The interface is one more way to achieve abstraction in Java.
- An interface may also contain constants, default methods, and static methods.
- All the methods inside an interface must have empty bodies except default methods and static methods.
- We use the **interface** keyword to declare an interface.
- There is no need to write **abstract** keyword before declaring methods in an interface because an interface is implicitly abstract.
- An interface **cannot** contain a **constructor** (as it cannot be used to create objects)
- In order to implement an interface, java requires a class to use the **implement** keyword.
- Interfaces support the functionality of **multiple inheritance.**
- Java Interface also represents the **IS-A** relationship.

The relationship between classes and interfaces:-

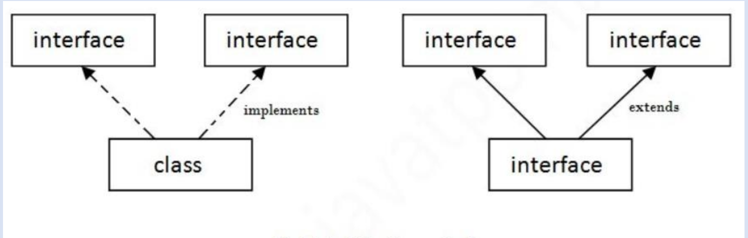
- ***The relationship between class and interface:-***

As shown in the figure given below, a class extends another class, an interface extends another interface, but a class implements an interface.



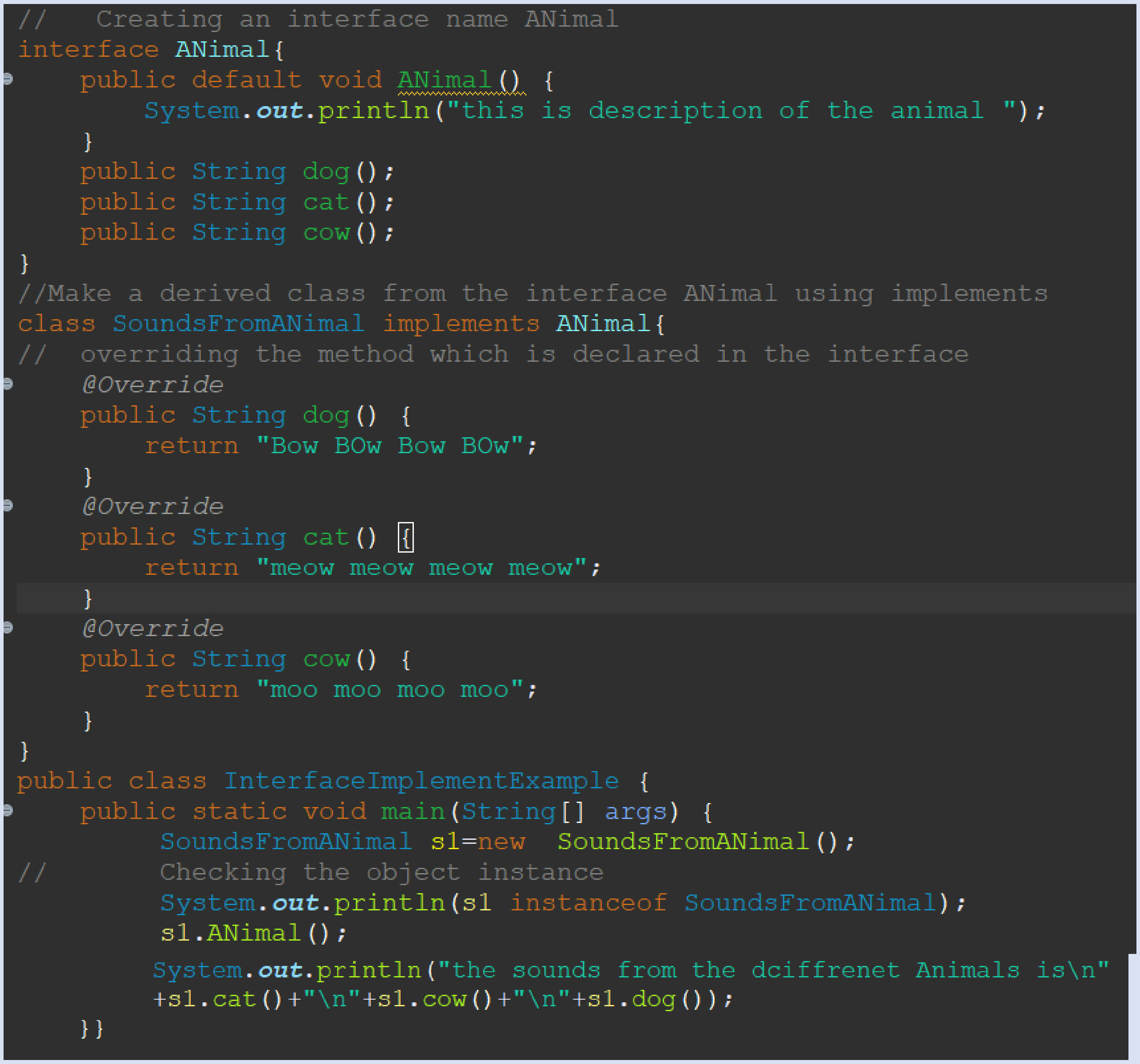
Simple Examples of interface implements by class.

Diagram:-



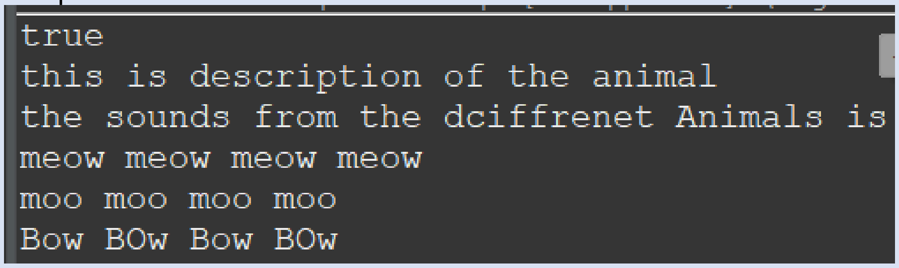
Multiple Inheritance in Java

Output-



```
//   Creating an interface name ANimal
interface ANimal{
    public default void ANimal() {
        System.out.println("this is description of the animal ");
    }
    public String dog();
    public String cat();
    public String cow();
}
//Make a derived class from the interface ANimal using implements
class SoundsFromANimal implements ANimal{
//  overriding the method which is declared in the interface
    @Override
    public String dog() {
        return "Bow BOw Bow BOw";
    }
    @Override
    public String cat() {
        return "meow meow meow meow";
    }
    @Override
    public String cow() {
        return "moo moo moo moo";
    }
}
public class InterfaceImplementExample {
    public static void main(String[] args) {
         SoundsFromANimal s1=new  SoundsFromANimal();
//       Checking the object instance
         System.out.println(s1 instanceof SoundsFromANimal);
         s1.ANimal();
        System.out.println("the sounds from the dciffrenet Animals is\n"
        +s1.cat()+"\n"+s1.cow()+"\n"+s1.dog());
    }}
```

Outputs:-

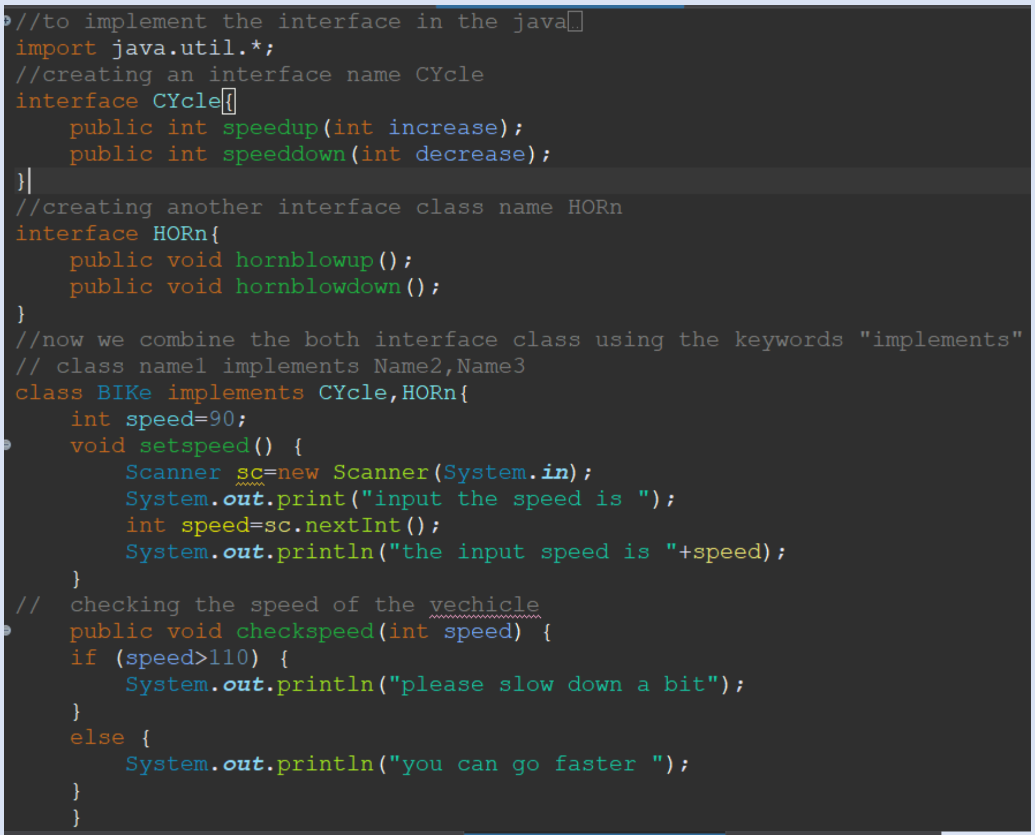


```
true
this is description of the animal
the sounds from the dciffrenet Animals is
meow meow meow meow
moo moo moo moo
Bow BOw Bow BOw
```

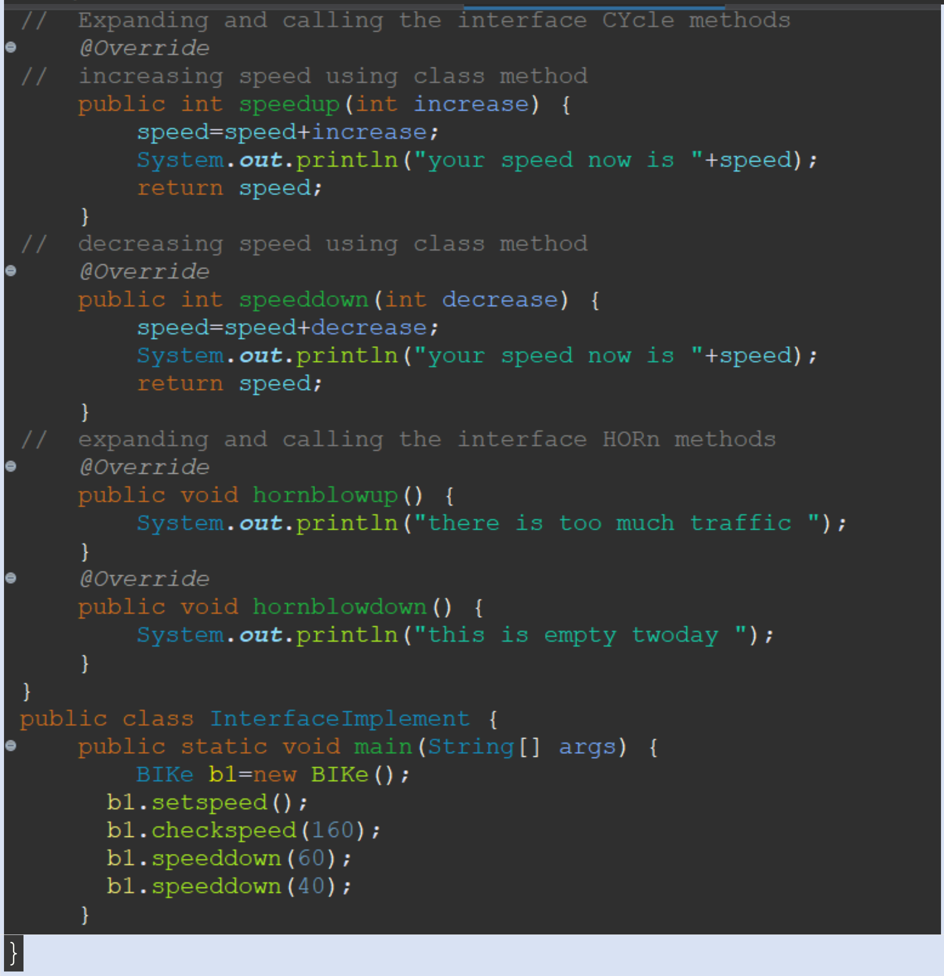
- ***Multiple inheritance in Java by interface:-***

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.

Code:-

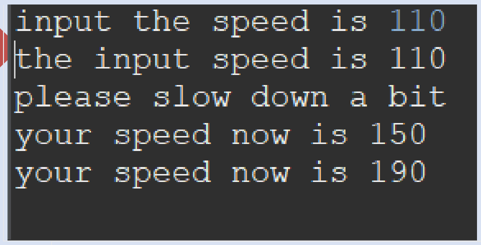


```
//to implement the interface in the java
import java.util.*;
//creating an interface name CYcle
interface CYcle{
    public int speedup(int increase);
    public int speeddown(int decrease);
}
//creating another interface class name HORn
interface HORn{
    public void hornblowup();
    public void hornblowdown();
}
//now we combine the both interface class using the keywords "implements"
// class name1 implements Name2,Name3
class BIKe implements CYcle,HORn{
    int speed=90;
    void setspeed() {
        Scanner sc=new Scanner(System.in);
        System.out.print("input the speed is ");
        int speed=sc.nextInt();
        System.out.println("the input speed is "+speed);
    }
//  checking the speed of the vechicle
    public void checkspeed(int speed) {
    if (speed>110) {
        System.out.println("please slow down a bit");
    }
    else {
        System.out.println("you can go faster ");
    }
    }
```



```
//  Expanding and calling the interface CYcle methods
    @Override
//  increasing speed using class method
    public int speedup(int increase) {
        speed=speed+increase;
        System.out.println("your speed now is "+speed);
        return speed;
    }
//  decreasing speed using class method
    @Override
    public int speeddown(int decrease) {
        speed=speed+decrease;
        System.out.println("your speed now is "+speed);
        return speed;
    }
//  expanding and calling the interface HORn methods
    @Override
    public void hornblowup() {
        System.out.println("there is too much traffic ");
    }
    @Override
    public void hornblowdown() {
        System.out.println("this is empty twoday ");
    }
}
public class InterfaceImplement {
    public static void main(String[] args) {
        BIKe b1=new BIKe();
      b1.setspeed();
      b1.checkspeed(160);
      b1.speeddown(60);
      b1.speeddown(40);
    }
}
```

Output:-



```
input the speed is 110
the input speed is 110
please slow down a bit
your speed now is 150
your speed now is 190
```

- ***<u>Is multiple inheritance allowed in Java?</u>***
  - Multiple inheritance faces problems when there exists a method with the same signature in both the superclasses.
  - Due to such a problem, java does not support multiple inheritance directly, but the similar concept can be achieved using interfaces.
  - A class can implement multiple interfaces and extend a class at the same time.

<u>Some Important points :</u>

1. Interfaces in java are a bit like the class but with a significantly different.
2. An Interface can only have method signatures field and a default method.
3. The class implementing an interface needs to declare the methods (not field)
4. You can create a reference of an interface but not the object
5. Interface methods are public by default

- ***<u>Default methods In Java:-</u>***
  - An interface can have static and default methods.
  - Default methods enable us to add new functionality to existing interfaces.
  - This feature was introduced in java 8 to ensure backward compatibility while updating an interface.
  - A class implementing the interface need not implement the default methods.
  - Interfaces can also include private methods for default methods to use.
  - You can easily override a default method like any other method of an interface.

<u>Another Example:-</u>

```
interface Animal{
   // Default method
   default void say(){
      System.out.println("Hello, this is default method");
   }
   // Abstract method
   void bark();
}
public class CWH implements Animal{

   @Override
   public void bark() {
      System.out.println("Dog barks!");
   }
   public static void main(String[] args) {
      CWH obj1 = new CWH();
      obj1.bark();
      obj1.say();
   }}
```

<u>Another Example:-</u>

package com.company;

```
interface MyCamera{
   void takeSnap();
   void recordVideo();
   private void greet(){
      System.out.println("Good Morning");
   }
   default void record4KVideo(){
      greet();
      System.out.println("Recording in 4k...");
   }}
interface MyWifi{
   String[] getNetworks();
   void connectToNetwork(String network);
}
class MyCellPhone{
   void callNumber(int phoneNumber){
      System.out.println("Calling "+ phoneNumber);
   }
   void pickCall(){
      System.out.println("Connecting... ");
   }}
class MySmartPhone extends MyCellPhone implements MyWifi, MyCamera{
   public void takeSnap(){
      System.out.println("Taking snap");
   }
   public void recordVideo(){
      System.out.println("Taking snap");
   }
//   public void record4KVideo(){
//      System.out.println("Taking snap and recoding in 4k");
//   }
   public String[] getNetworks(){
      System.out.println("Getting List of Networks");
      String[] networkList = {"Harry", "Prashanth", "Anjali5G"};
      return networkList;}
   public void connectToNetwork(String network){
      System.out.println("Connecting to " + network);
   }}
public class cwh_57_default_methods {
   public static void main(String[] args) {
      MySmartPhone ms = new MySmartPhone();
      ms.record4KVideo();
      // ms.greet(); --> Throws an error!
      String[] ar = ms.getNetworks();
      for (String item: ar) {
         System.out.println(item);
      } }}
```

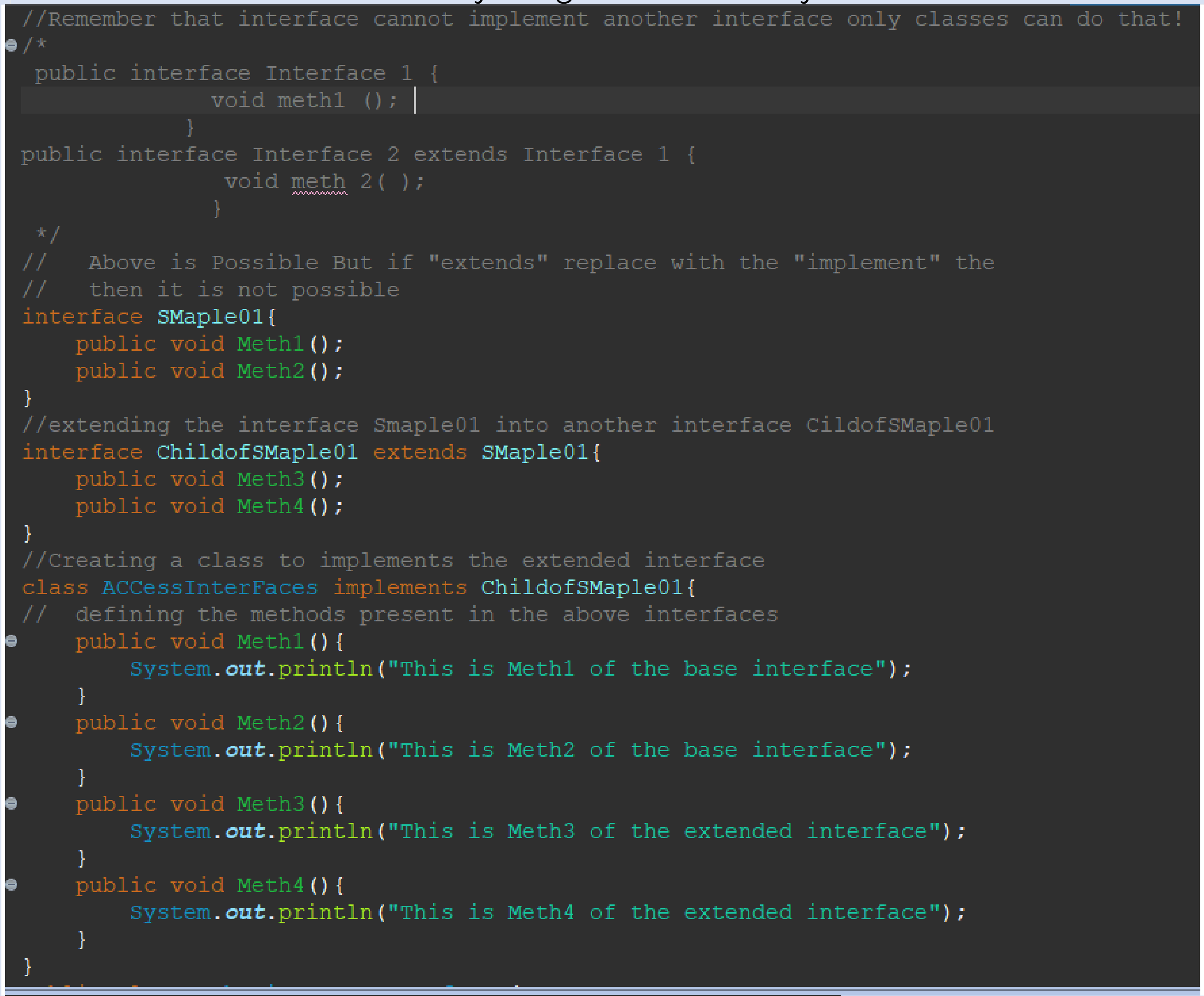
• ***Inheritance in Interfaces***

Interfaces can extend other interfaces as shown below :

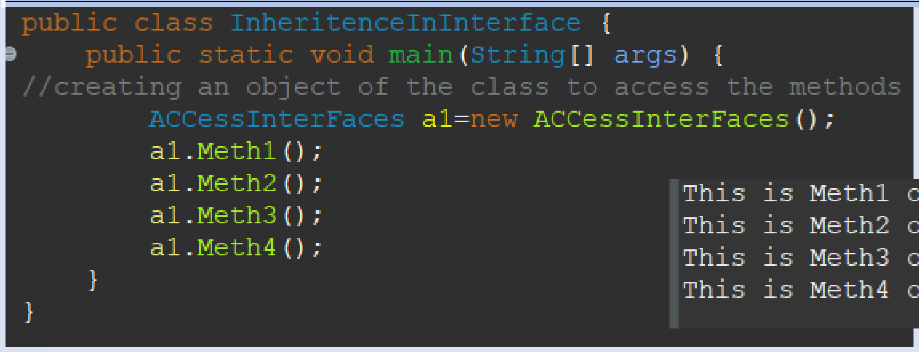
```
public interface Interface 1 {
        void meth1 (); }
```

```
public interface Interface 2 extends Interface 1 {
          void meth 2( ); }
```

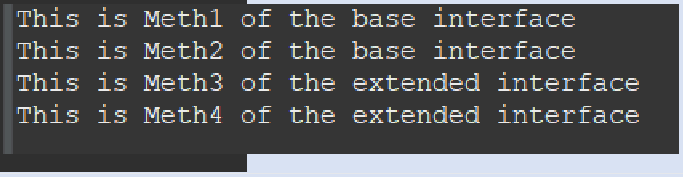
Note: Remember that interface cannot implement another interface only classes can do that! But the interface can be Extends by using the “extends” Keyword .



```
//Remember that interface cannot implement another interface only classes can do that!
/*
 public interface Interface 1 {
              void meth1 ();
            }
public interface Interface 2 extends Interface 1 {
               void meth 2( );
             }
 */
//   Above is Possible But if "extends" replace with the "implement" the
//   then it is not possible
interface SMaple01{
    public void Meth1();
    public void Meth2();
}
//extending the interface Smaple01 into another interface CildofSMaple01
interface ChildofSMaple01 extends SMaple01{
    public void Meth3();
    public void Meth4();
}
//Creating a class to implements the extended interface
class ACCessInterFaces implements ChildofSMaple01{
//  defining the methods present in the above interfaces
    public void Meth1(){
        System.out.println("This is Meth1 of the base interface");
    }
    public void Meth2(){
        System.out.println("This is Meth2 of the base interface");
    }
    public void Meth3(){
        System.out.println("This is Meth3 of the extended interface");
    }
    public void Meth4(){
        System.out.println("This is Meth4 of the extended interface");
    }
}
```



```
public class InheritenceInInterface {
    public static void main(String[] args)  {
//creating an object of the class to access the methods
        ACCessInterFaces a1=new ACCessInterFaces();
        a1.Meth1();
        a1.Meth2();
        a1.Meth3();
        a1.Meth4();
    }
}
```

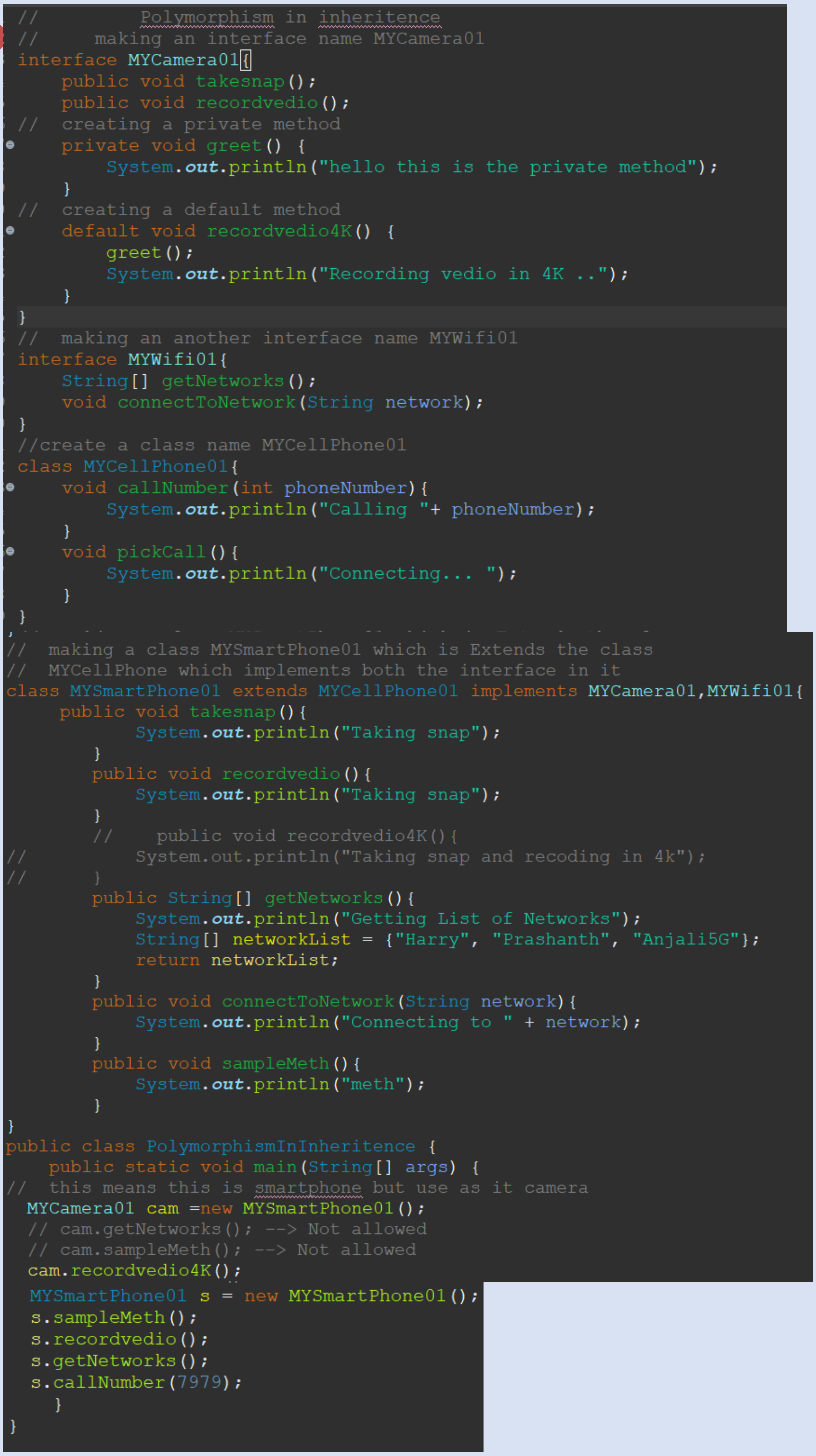


```
This is Meth1 of the base interface
This is Meth2 of the base interface
This is Meth3 of the extended interface
This is Meth4 of the extended interface
```

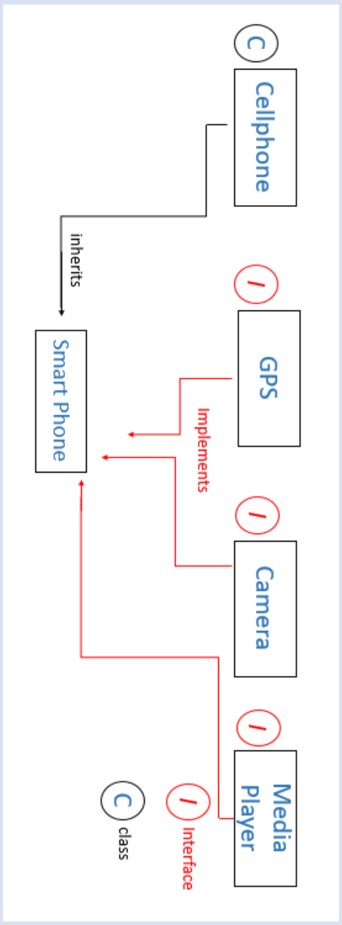
- ***Polymorphism in Interfaces***

- **GPS g = new Smartphone ( ); can only use GPS method**
  Smartphones = new Smartphone ( ); can only use smartphone methods
- Implementing an Interface force method implementation

Code:-



```
//          Polymorphism in inheritence
//      making an interface name MYCamera01
interface MYCamera01{
    public void takesnap();
    public void recordvedio();
//  creating a private method
    private void greet() {
        System.out.println("hello this is the private method");
    }
//  creating a default method
    default void recordvedio4K() {
        greet();
        System.out.println("Recording vedio in 4K ..");
    }
}
//  making an another interface name MYWifi01
interface MYWifi01{
    String[] getNetworks();
    void connectToNetwork(String network);
}
//create a class name MYCellPhone01
class MYCellPhone01{
    void callNumber(int phoneNumber){
        System.out.println("Calling "+ phoneNumber);
    }
    void pickCall(){
        System.out.println("Connecting... ");
    }
}
//  making a class MYSmartPhone01 which is Extends the class
//  MYCellPhone which implements both the interface in it
class MYSmartPhone01 extends MYCellPhone01 implements MYCamera01,MYWifi01{
    public void takesnap(){
        System.out.println("Taking snap");
    }
    public void recordvedio(){
        System.out.println("Taking snap");
    }
    //    public void recordvedio4K(){
//      System.out.println("Taking snap and recoding in 4k");
//  }
    public String[] getNetworks(){
        System.out.println("Getting List of Networks");
        String[] networkList = {"Harry", "Prashanth", "Anjali5G"};
        return networkList;
    }
    public void connectToNetwork(String network){
        System.out.println("Connecting to " + network);
    }
    public void sampleMeth(){
        System.out.println("meth");
    }
}
public class PolymorphismInInheritence {
    public static void main(String[] args) {
//  this means this is smartphone but use as it camera
  MYCamera01 cam =new MYSmartPhone01();
  // cam.getNetworks(); --> Not allowed
  // cam.sampleMeth(); --> Not allowed
  cam.recordvedio4K();
  MYSmartPhone01 s = new MYSmartPhone01();
  s.sampleMeth();
  s.recordvedio();
  s.getNetworks();
  s.callNumber(7979);
    }
}
```



● *Java Tutorial: Abstract Classes Vs Interfaces*

Abstract class	Interface
1) Abstract class can have **abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An interface can extend another Java interface only.
7) An **abstract class** can be extended using keyword "extends".	An interface can be implemented using keyword "implements".
8) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

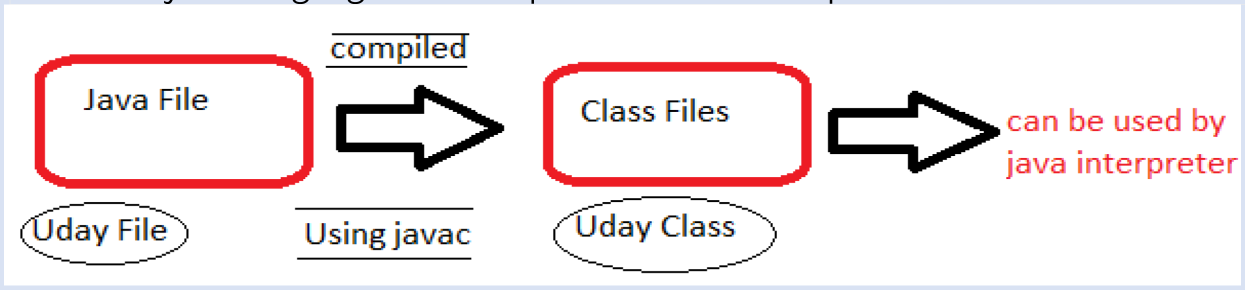
9)Example:
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

- ***Interpreted vs Compiled Languages:-***
- ***Interpreter Vs Compliler : -***

The interpreter translates one statement at a time into machine code. On the other hand, the compiler scans the entire program and translates the whole of it into machine code

- ***Interpreter :-***
  1. one statement at a time
  2. An interpreter is needed every time
  3. Partial execution if an error occurs in the program.
  4. Easy for programmers.
- ***Compiler :-***
  1. Entire program at a time
  2. Once compiled, it is not needed
  3. No execution if an error occurs
  4. Usually not as easy as interpreted once
- ***Is Java interpreted or compiled?***

Java is a hybrid language both compiled as well as interpreted.



- A JVM can be used to interpret this bycode.

- This bytecode can be taken to any platform (win/ mac / Linux) for education.
- Hence java is platform-independent (write once run everywhere).

- **Executing a java program**

java Uday java - compiles.
java Uday class - Interpreted .

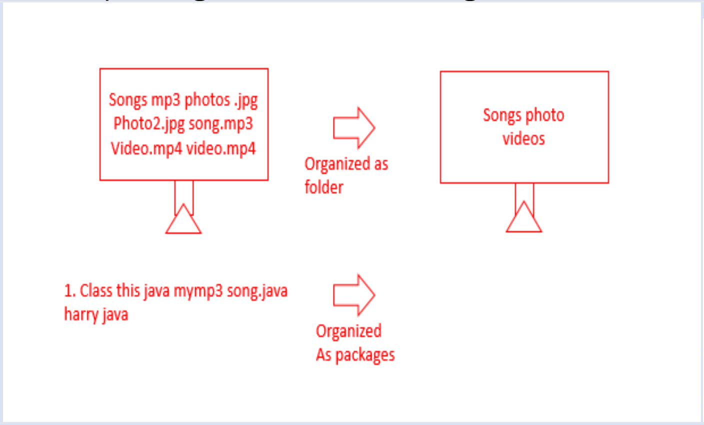
• ***Encapsulation :-***

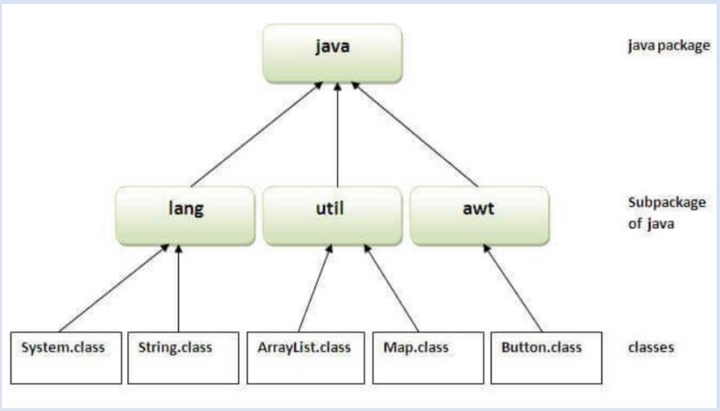
Encapsulation is the process of combining data and functions into a single unit called class. In Encapsulation, the data is not accessed directly; it is accessed through the functions present inside the class. In simpler words, attributes of the class are kept private and public getter and setter methods are provided to manipulate these attributes. Thus, encapsulation makes the concept of data hiding possible.(Data hiding: a language feature to restrict access to members of an object, reducing the negative effect due to dependencies. e.g. "protected", "private" feature in Java).

• ***Packages in Java:-***

- A java package is a group of similar types of classes, interfaces and sub-packages.
- Package in java can be categorized in two form, built-in package and user-defined package.
- There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.
- Here, we will have the detailed learning of creating and using user-defined packages.
- ***Advantage of Java Package:-***

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
2) Java package provides access protection.
3) Java package removes naming collision..





- ***Using a java package :***

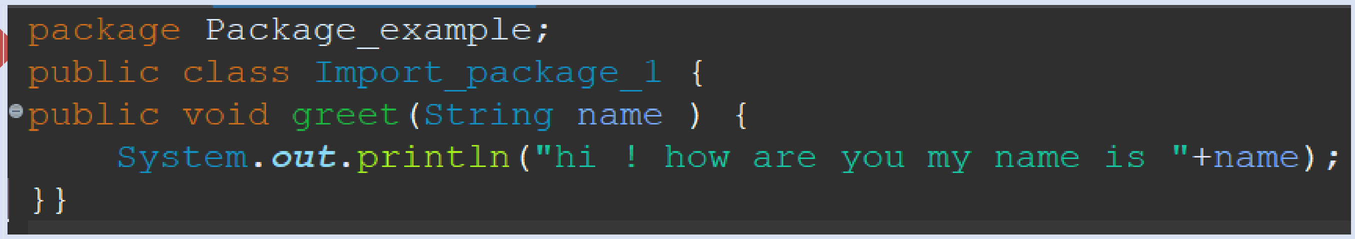
Import keyword is used to import packages in the java program. Example :

- Import java. Lang. * - import
- import java.lang. string - import string from java long
- s= new java long string ("uday") - use without importing .

code:-

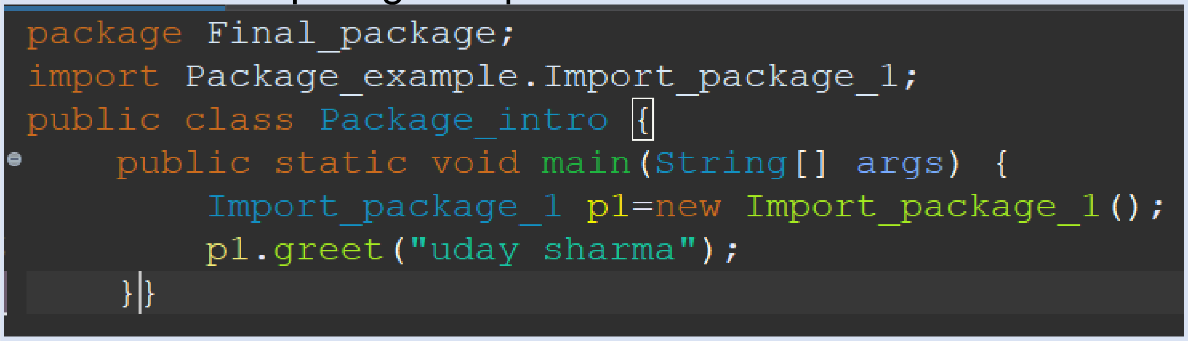
1. ***import package.classname;***

importing package should be create without the ***public static void main(String[ ] args***) and the class in which we import the package have ***the public static void main(String[ ] args) .***
class which is used to import:-



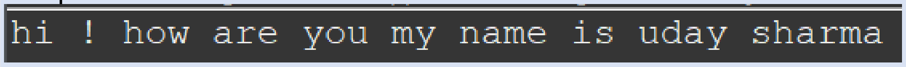
```
package Package_example;
public class Import_package_1 {
public void greet(String name ) {
     System.out.println("hi ! how are you my name is "+name);
}}
```

Class in which the package is impoerted:-



```
package Final_package;
import Package_example.Import_package_1;
public class Package_intro {
     public static void main(String[] args) {
         Import_package_1 p1=new Import_package_1();
         p1.greet("uday sharma");
     }}
```

Output:-



```
hi ! how are you my name is uday sharma
```

How to access package from another package?

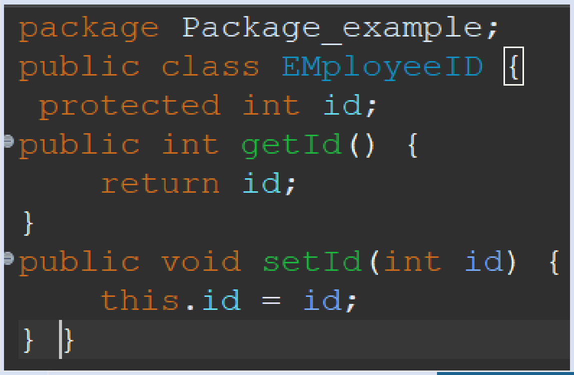
There are three ways to access the package from outside the package.

- import package.*;
- import package.classname;
- fully qualified name.

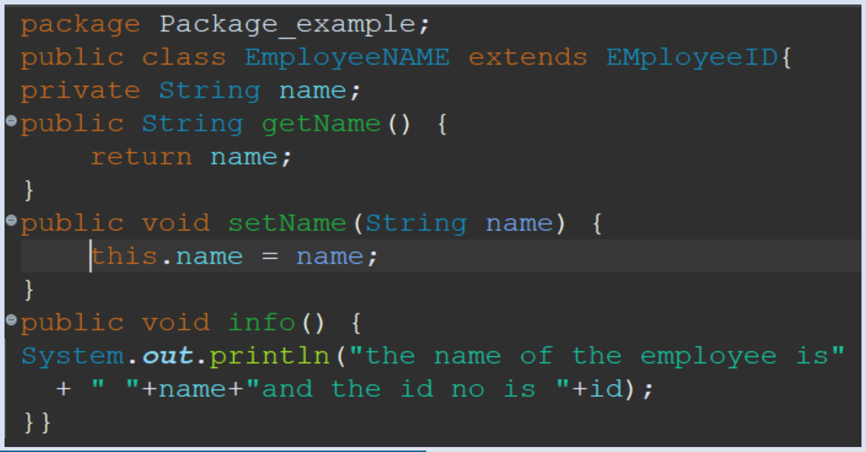
1) ***Using packagename.* :-***

Code:-

Make a Class **EMployeeID** which is extends in the another class **EmployeeNAME:-**



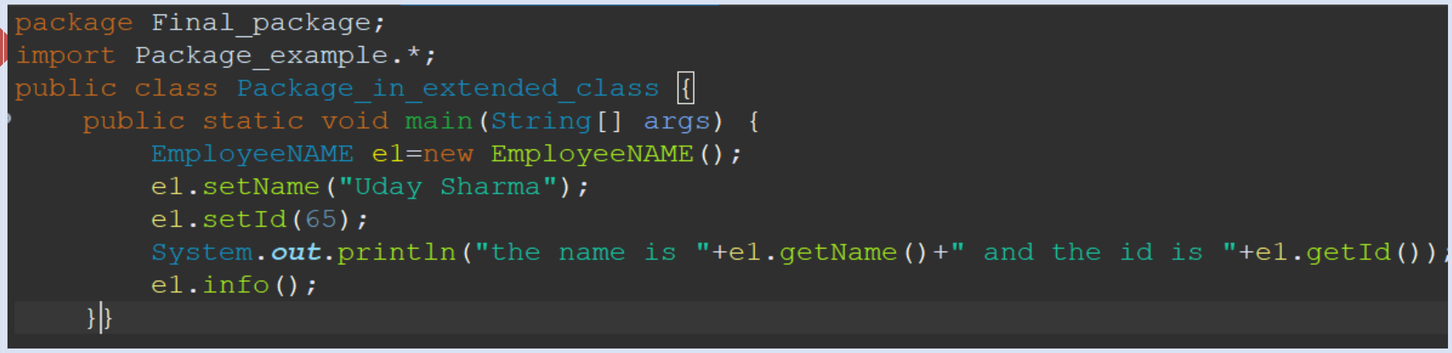
```
package Package_example;
public class EMployeeID {
 protected int id;
public int getId() {
     return id;
}
public void setId(int id) {
     this.id = id;
} }
```



```
package Package_example;
public class EmployeeNAME extends EMployeeID{
private String name;
public String getName() {
     return name;
}
public void setName(String name) {
     this.name = name;
}
public void info() {
System.out.println("the name of the employee is"
  + " "+name+"and the id no is "+id);
}}
```

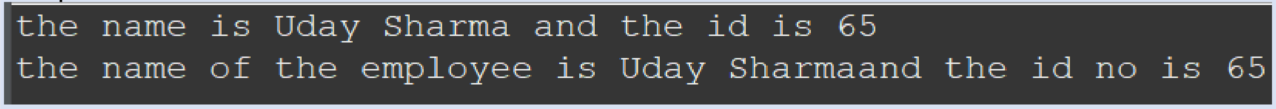
Now the main class name Package_in_extended_class in which import both the classes using **import package.*;**

In this the ID is protected because protected access modifiers can be pass through the same package but if the ID is private it cannot be accessed.



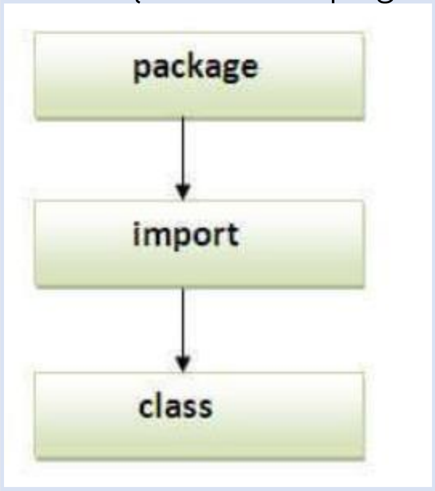
```
package Final_package;
import Package_example.*;
public class Package_in_extended_class {
    public static void main(String[] args) {
        EmployeeNAME e1=new EmployeeNAME();
        e1.setName("Uday Sharma");
        e1.setId(65);
        System.out.println("the name is "+e1.getName()+" and the id is "+e1.getId());
        e1.info();
    }}
```

Output:-



```
the name is Uday Sharma and the id is 65
the name of the employee is Uday Sharmaand the id no is 65
```

Note: Sequence of the program must be package then import then class.



Subpackage in java.

Package inside the package is called the subpackage. It should be created to categorize the package further.

- ***Access Modifiers in Java:-***

Access modifiers determine whether other classes can use a particular field or invoke a particular method can be public, private, protected, or default (no modifier). See the table given below :

Access Modifier	within class	within package	outside package by subclass only	outside package
public	Y	Y	Y	Y
protected	Y	Y	Y	N
Default	Y	Y	N	N
private	Y	N	N	N

- ***Encapsulation in Java:-***

Encapsulation in Java is a *process of wrapping code and data together into a single unit*, for example, a capsule which is mixed of several medicines.

We can create a fully encapsulated class in Java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.

The **Java Bean** class is the example of a fully encapsulated class.

- ***Advantage of Encapsulation in Java:-***
- By providing only a setter or getter method, you can make the class **read-only or write-only**. In other words, you can skip the getter or setter methods.
- It provides you the **control over the data**. Suppose you want to set the value of id which should be greater than 100 only, you can write the logic inside the setter method. You can write the logic not to store the negative numbers in the setter methods.
- It is a way to achieve **data hiding** in Java because other class will not be able to access the data through the private data members.
- The encapsulate class is **easy to test**. So, it is better for unit testing.
- The standard IDE's are providing the facility to generate the getters and setters. So, it is **easy and fast to create an encapsulated class** in Java.

Note:- In the encapsulation the private datatypes is can be access by the getter() and the setter() Method:-
Awesome example of using getter( ) and the setter( ) methods:-
We using the two interface `Interface_student_2` and `Interface_student_1`in the java and implements in the class-`public class Combining_interfaces_Student_1_2 implements Interface_student_1,Interface_student_2` in the same package .
Then the package is imported in the different class `public class package_in_interfaces_student` of the different package which is in same project folder name as the Encapsulation in java.

➢ From the package package_in_example:-

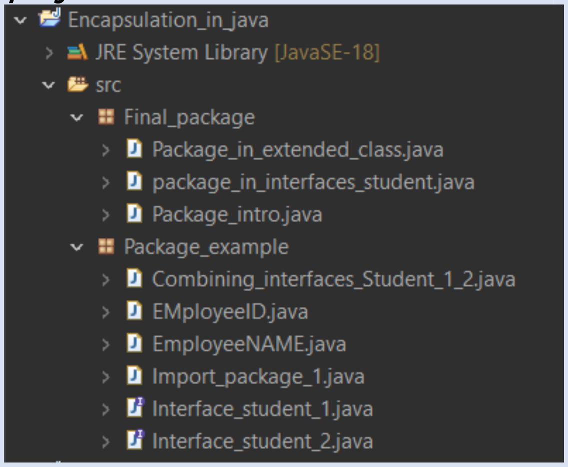
❖ **Interface 1:-**

```
package Package_example;
public interface Interface_student_1 {
      default void info() {
          System.out.println("this is the 1st interface ");
     }
      public void setName(String name);
      public String getName();
      public void setID(int id);
      public int getID();
}
```

❖ **Interface-2:-**

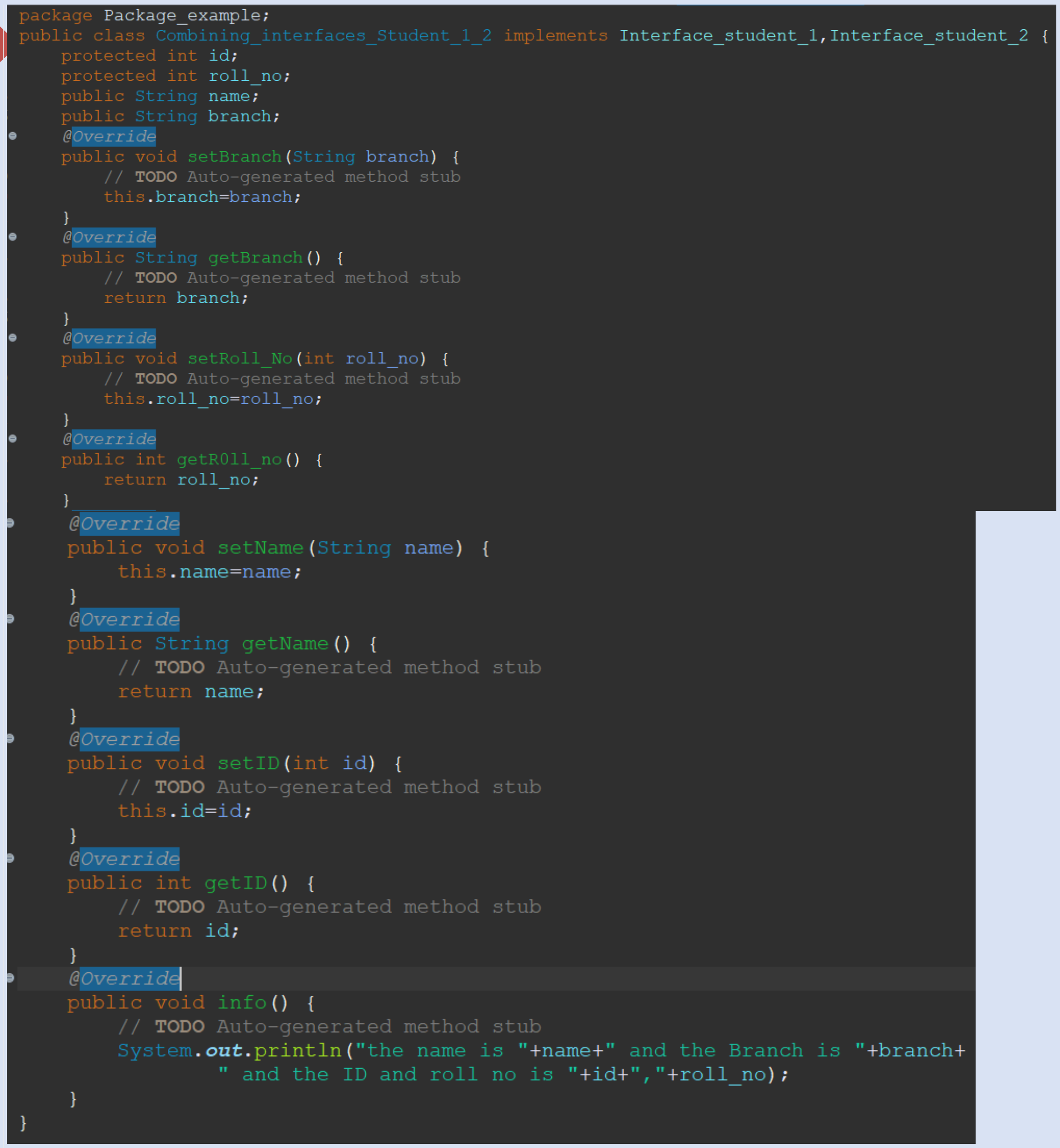
```
package Package_example;
public interface Interface_student_2 {
default void info() {
     System.out.println("this is the interface 2");
}
public void setBranch(String branch);
public String getBranch();
public void setRoll_No(int roll_no);
public int getR0ll_no();
}
```

project is lookalike this:-



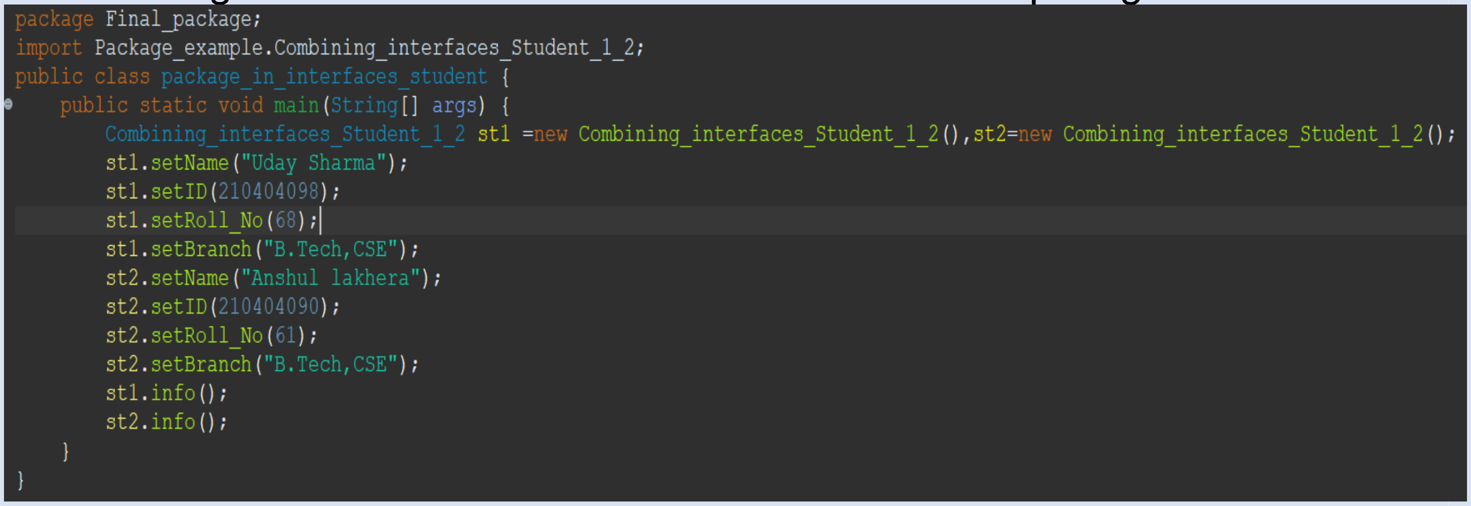
```
Encapsulation_in_java
  > JRE System Library [JavaSE-18]
  v src
    v Final_package
      > Package_in_extended_class.java
      > package_in_interfaces_student.java
      > Package_intro.java
    v Package_example
      > Combining_interfaces_Student_1_2.java
      > EMployeeID.java
      > EmployeeNAME.java
      > Import_package_1.java
      > Interface_student_1.java
      > Interface_student_2.java
```

❖ **Combining the both interface :-**



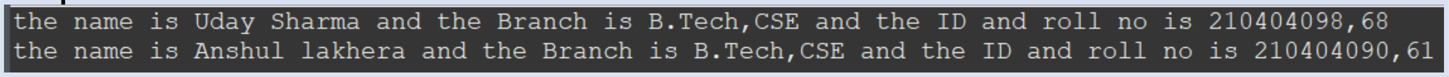
```java
package Package_example;
public class Combining_interfaces_Student_1_2 implements Interface_student_1,Interface_student_2 {
    protected int id;
    protected int roll_no;
    public String name;
    public String branch;
    @Override
    public void setBranch(String branch) {
        // TODO Auto-generated method stub
        this.branch=branch;
    }
    @Override
    public String getBranch() {
        // TODO Auto-generated method stub
        return branch;
    }
    @Override
    public void setRoll_No(int roll_no) {
        // TODO Auto-generated method stub
        this.roll_no=roll_no;
    }
    @Override
    public int getR0ll_no() {
        return roll_no;
    }
    @Override
    public void setName(String name) {
        this.name=name;
    }
    @Override
    public String getName() {
        // TODO Auto-generated method stub
        return name;
    }
    @Override
    public void setID(int id) {
        // TODO Auto-generated method stub
        this.id=id;
    }
    @Override
    public int getID() {
        // TODO Auto-generated method stub
        return id;
    }
    @Override
    public void info() {
        // TODO Auto-generated method stub
        System.out.println("the name is "+name+" and the Branch is "+branch+
                " and the ID and roll no is "+id+","+roll_no);
    }
}
```

❖ Now calling the class in the different class which is in different package:-



```java
package Final_package;
import Package_example.Combining_interfaces_Student_1_2;
public class package_in_interfaces_student {
    public static void main(String[] args) {
        Combining_interfaces_Student_1_2 st1 =new Combining_interfaces_Student_1_2(),st2=new Combining_interfaces_Student_1_2();
        st1.setName("Uday Sharma");
        st1.setID(210404098);
        st1.setRoll_No(68);
        st1.setBranch("B.Tech,CSE");
        st2.setName("Anshul lakhera");
        st2.setID(210404090);
        st2.setRoll_No(61);
        st2.setBranch("B.Tech,CSE");
        st1.info();
        st2.info();
    }
}
```

Output:-



```
the name is Uday Sharma and the Branch is B.Tech,CSE and the ID and roll no is 210404098,68
the name is Anshul lakhera and the Branch is B.Tech,CSE and the ID and roll no is 210404090,61
```

- Go check out my ***LinkedIn profile*** for more notes and other resources content

@Uday Sharma

mrudaysharma4600@gmail.com

https://www.linkedin.com/in/uday-sharma-602b33267