Class & Inheritance

Objects and Classes in Java

In object-oriented programming, a program is design using objects and classes. An object in Java is the physical as well as a logical entity, whereas, a class in Java is a logical entity only. An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, the banking system. etc. It can be physical or logical.

An object has three characteristics:

- **State:** represents the data i.e. value of an object.
- **Behavior:** represents the behavior i.e. functionality of an object such as deposit, withdraw, etc.
- **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.

Definition of Object and class

An object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance i.e. results of a class.

Object Definitions:

- An object is *a real-world entity*.
- An object is *a runtime entity*.
- The object is *an entity which has state and behavior*.
- The object is *an instance of a class*.

Class in Java

A class is a group of objects which have common properties. Classes create objects and objects use methods to communicate between them Classes provide a convenient method for packing together a group of logically related data items and functions that work on them. A class in Java can contain:

- Fields: data items
- Methods: functions

Syntax to declare a class:

```
class classname  [extends superclassname]
{
        [variable declaration;]
        [method declaration;]

}
```

Where, classname and superclassname are any valid identifiers.
The keyword extends indicates that the properties of superclassname class are extended to the classname class. This concept is known as inheritance.

Instance variable in Java

A variable which is created inside the class but outside the method is known as an instance variable. Instance variable doesn't get memory at compile time. It gets memory at runtime when an object or instance is created. So it is known as an instance variable.

Adding Variables in class

Data is encapsulated in a class by placing data fields inside the body of the class definition. These variables are called instance variable.
For e.g.

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

Note: these variables are only declared and no storage space has been created in the memory. Instance variables are also known as member variables.

Method in Java

In Java, a method is like a function which is used to expose the behavior of an object. i.e. methods are necessary for manipulating data contained in the class.
Advantage of Method

- Code Reusability
- Code Optimization

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

```
Type methodname (parmeter-list)
{
Method-body;
}
```

Example

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

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

Creating objects

Creating an object is also known as instantiating an object. Objects in java are created using the new operator. The new operator creates an object of all specified class and returns a reference to that object.

For e.g

```
Rectangle rect1; // declare
rect1 = new Rectangle ( );  // instantiate
```

Or

```
Rectangle rect1 = new Rectangle ( );
```

Where Rectangle ( ) is default constructor of the class.

User can create any number of objects.
E.g.

```
Rectangle rect1 = new Rectangle ( );
Rectangle rect2 = new Rectangle ( );
```

Each object has its own copy of instance variable of its class. This means that any changes to the variables of one object have no effect on the variables of another.

Accessing class members

All variables must be assigned values before they are used by using . (dot) operator as shown below:

objectname .Variable name;

Or

objectname .Methodname(parameter list);

Where, objectname is the name of the object,

variablename is the name of the instance variable inside the object that we wish to access,

methodname is the method that we wish to call, and

Parameter list is a comma separated list of "actual values“ .

For e.g

Rectangle rect1 = new Rectangle ( ); // creating object

rect1.getdata(15,20) // calling method using the object

Or

rect1.length = 15;

rect1.width = 20;

```
class Rect
{
  int  l, w;
  void get(int x, int y)
   {
     l = x;
     w = y;
   System.out.println(" Given the values of length & breadth "+l+" and "+w);
    }
  int Area()
   {
     int  ar=l*w;
     return (ar);
}
}
class CalAr
{
public static void main (String args[ ])
{
  int ar1;

 Rect r1 = new Rect( );
 r1.get(12,13);
ar1=r1.Area();
System.out.println(" Area of Rectangle is "+ar1);
}
}
```

Example

```
//Java Program to illustrate how to define a class and fields
//Defining a Student class.
class Student{
 //defining fields
 int id;//field or data member or instance variable
 String name;
 //creating main method inside the Student class
 public static void main(String args[]){
  //Creating an object or instance
  Student s1=new Student();//creating an object of Student
  //Printing values of the object
  System.out.println(s1.id);//accessing member through reference variable
  System.out.println(s1.name);
 }
}
```

Constructors

A constructor in Java is a special method that is used to initialize objects. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes. Constructor have same name as class itself. Constructor do not specify a return type. This is because they return the instance of the class itself.

Example of constructor

```
class Rectangle
  {
     int length;
      int width;
      Rectangle ( int x, int y) //constructor method
        {
            length = x;
             width = y;
        }
     int  reactarea()
        {
           int area = length * width;
          return(area);
        }
}
```

Application of Constructors

```
class Rectangle
  {
     int length;
      int width;
      Rectangle ( int x, int y) //constructor method
       {
           length = x;
            width = y;
       }
     int  reactarea()
       {
          return (length *  width);
       }
}
Class CalRectangle
{
    public static void main (String args [])
      {
           Rectangle r1 = new Rectangle (15,10);   //calling  constructor
            int a1= r1.reactarea( );
           System.out.println( "Area  of rectangle  is " + a1);
      }
}
```

Methods Overloading

In Java, it is possible to create methods that have the same name, but different parameter lists and different definitions. This is called method overloading. Method overloading is used when objects are required to perform similar tasks but using different input parameters.

When we call a method in an object, Java matches up the method name first and then the number and type of parameters to decide which one of the definitions to execute. This process is known as polymorphism. To create an overloaded method, all we have to do is to provide several different method definitions in the class, all with the same name, but with different parameter lists. The difference may either be in the number or type of arguments. That is, each parameter list should be unique. Note that the method's return type does not play any role in this.

Example of method overloading

```
// Java program to demonstrate working of method overloading in Java.
  class Sum
{
     // Overloaded sum(). This sum takes two int parameters
    int sum(int x, int y)
   {
      return (x + y);
   }
     // Overloaded sum(). This sum takes three int parameters
    int sum(int x, int y, int z)
   {
      return (x + y + z);
   }
     // Overloaded sum(). This sum takes two double parameters
   double sum(double x, double y)
   {
      return (x + y);
   }
     // Driver code
   public static void main(String args[])
   {
      Sum s = new Sum();
      System.out.println("The Sum is " +s.sum(10, 20));
      System.out.println(" The Sum is" +s.sum(10, 20, 30));
      System.out.println("The Sum is " +s.sum(10.5, 20.5));
   }
}
```

Constructor Overloading

In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods. Constructor overloading in Java is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

```
        //Java program to overload constructors
          class Student
         {
   int id;
   String name;
   int age;
   //creating two arg constructor
   Student(int i,String n)
    {
     id = i;
     name = n;
    }
   //creating three arg constructor
   Student(int i,String n,int a)
     {
      id = i;
      name = n;
      age=a;
     }
   void display()
   {
    System.out.println(id+" "+name+" "+age);
   }

   public static void main(String args[]){
   Student s1 = new Student(111,"Karan");
   Student s2 = new Student(222,"Aryan",25);
   Student s3 = new Student(333,"Raman",30);
   Student s4 = new Student(444,"Suman",21);
   System.out.println( "Student information is as follows\n");
   s1.display();
   s2.display();
   s3.display();
   s4.display();
  }
}
```

Difference between constructor and method

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.

Static Members

Class contains two sections

- variables declaration and
- methods declaration

These variables and methods are called instance variables and instance methods. Because every time class is instantiated, a new copy of each of them is created. They are accessed using dot operator. Let us assume that we want to define a member that is common to all the objects and accessed without using a particular object. i.e. the member belongs to the class as a whole rather than the objects created from the class. Such members can be defined as follows:

Static int count;

Static int max(int x, int y);

The members that are declared static are called static members. Since these members are associated with the class itself rather than individual objects. static variables and static methods are often referred to as class variables and class methods. Static variables are used when user want to have a variable common to all instances of a class.

```
//Java program to overload constructors
class Student
   {
   int id;
   String name;
      int age;
static String cn = " HVPM " ;
   //creating two arg constructor
   Student(int i,String n)
    {
     id = i;
     name = n;
    }
   //creating three arg constructor
   Student(int i,String n,int a)
     {
      id = i;
```

```
    name = n;
    age=a;
   }
  void display()
  {
   System.out.println(id+" "+name+" "+age+" "+ cn);
  }

  public static void main(String args[]){
  Student s1 = new Student(111,"Karan");
  Student s2 = new Student(222,"Aryan",25);
  Student s3 = new Student(333,"Raman",30);
  Student s4 = new Student(444,"Suman",21);
  System.out.println( "Student information is as follows\n");
  s1.display();
  s2.display();
  s3.display();
  s4.display();
 }
}
```

Static Method

Static method can also be called without using objects. Static methods are also available for use by other classes. Java class libraries contains a large number of class methods

For e.g. Math class of java library defines many static methods to perform math operations for e.g.

Float x= Math.sqrt(25.0);

```
Example of static method
class MathOpr
{
  static float mul (float x,  float y)
{    return x* y;
}
static float divi (float x,  float y)
{    return x/ y;
}
}
class MathApp
{
    public static void main(String args[])
{
  float a = MathOpr.mul(4.0f,5.0f);
float b = MathOpr.divi(a,2.0f);
System.out.println("a= "+a);
System.out.println("b= "+b);
}
}
```

Nesting of Methods

In java, the methods and variables which we create in a class can only be called by using the object of that class or, in case of static methods; we can directly call it by using the name of the class. The methods and variables can be called with the help of the dot operator. But there is a special case that a method can also be called by another method with the help of class name, but the condition is they should be present in the same class. This is known as nesting of methods.

Example of Nesting Methods

```
class Nesting
{
 int a,b;
Nesting(int p, int q) // constructor method
{
 a=p;
 b=q;
}
int greatest()
{
 if(a>=b)
   return(a);
 else
   return(b);
}
void display()
{
 int great=greatest(); //calling a method
 System.out.println("The Greatest Value="+great);
}
}
class Temp
{
 public static void main(String args[])
{
 Nesting t1=new Nesting(25,60);
 t1.display();
}
}
```

Inheritance in Java

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system). The idea behind inheritance in Java is that user can create

new classes that are built upon existing classes. When you inherit from an existing class, user can reuse methods and fields of the parent class. Moreover, user can add new methods and fields in the current class also.

Terms used in Inheritance

- Class: A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- Sub Class/Child Class: Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- Super Class/Parent Class: Super class is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- Reusability: As the name specifies, reusability is a mechanism which facilitates user to reuse the fields and methods of the existing class when user create a new class. user can use the same fields and methods already defined in the previous class.

The syntax of Java Inheritance

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

The extends keyword indicates that user are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

For e.g.

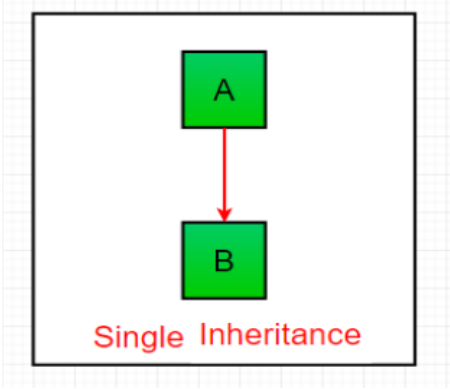
```
class Employee
{
 float salary=40000;
}
class Programmer extends Employee
{
 int bonus=10000;
 public static void main(String args[])
{
   Programmer p=new Programmer();
   System.out.println("Programmer salary is:"+p.salary);
   System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

Types of inheritance in java

Inheritance may take different forms:

- Single inheritance (only one super class)
- Multilevel inheritance (Derived from a derived
- Hierarchical inheritance (one super class, many subclasses)
- Multiple inheritance (several super classes)
- Hybrid inheritance

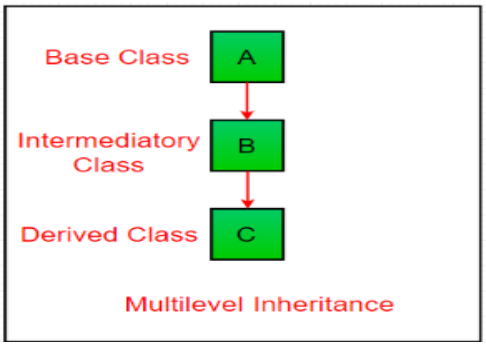
Single Inheritance: In single inheritance, subclasses inherit the features of one superclass. In the image below, class A serves as a base class for the derived class B.



Single Inheritance Example

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

Multilevel Inheritance: In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class. In the below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C.

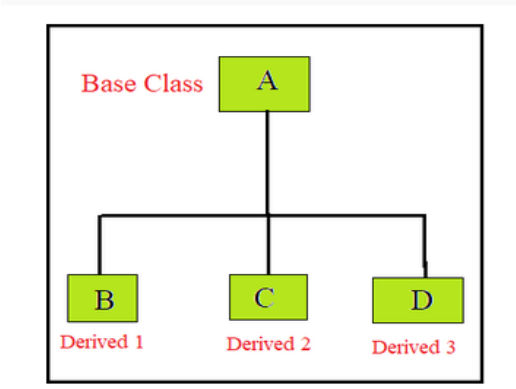


Multilevel Inheritance Example

```
class Animal
{
void eat()
{
System.out.println("eating...");}
}
class Dog extends Animal
{
void bark()
{
System.out.println("barking...");
}
}
class BabyDog extends Dog
{
void weep()
{System.out.println("weeping...");
}
}
class TestInheritance2
{
public static void main(String args[])
{
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}
}
```

Hierarchical Inheritance:

In Hierarchical Inheritance, one class serves as a super class (base class) for more than one subclass. In the below image, class A serves as a base class for the derived class B, C and D.

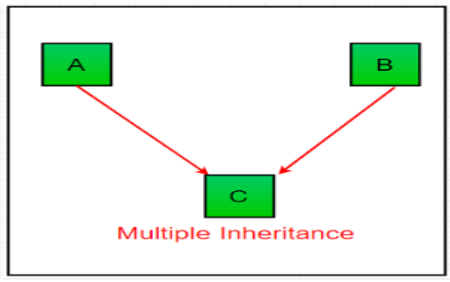


Hierarchical Inheritance Example

```
class Animal
{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal
{
void bark(){System.out.println("barking...");
}
}
class Cat extends Animal
{
void meow(){System.out.println("meowing...");
}
}
class TestInheritance3
{
public static void main(String args[])
{
Cat c=new Cat();
c.meow();
c.eat();
//c.bark();//C.T.Error
}
}
```

Multiple Inheritance:

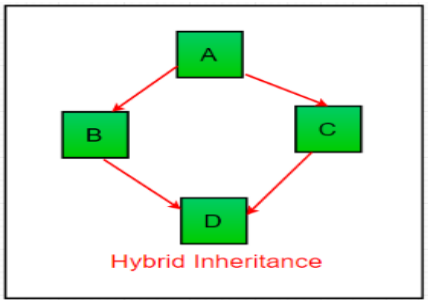
Multiple Inheritance In Multiple inheritances, one class can have more than one superclass and inherit features from all parent classes.



Note: Java does **not** support multiple inheritances with classes. In java, it can achieve multiple inheritances only through Interfaces.

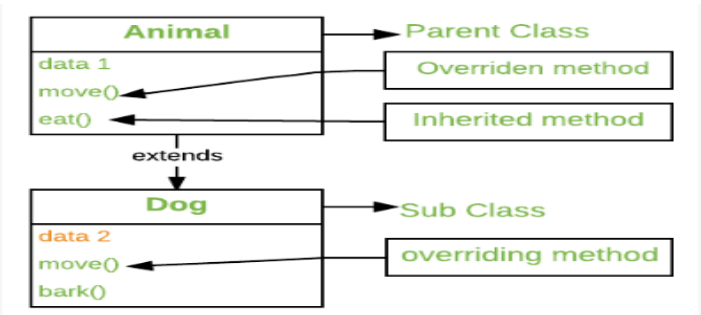
Hybrid Inheritance:

It is a mix of two or more of the above types of inheritance. Since java doesn't support multiple inheritances with classes, hybrid inheritance is also not possible with classes. In java, we can achieve hybrid inheritance only through Interfaces.



Overriding in Java:

In any object-oriented programming language, Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, same parameters and same return type as a method in its super-class, then the method in the subclass is said to *override* the method in the super-class.



Method overriding is one of the way by which java achieve Run Time Polymorphism. The version of a method that is executed will be determined by the object that is used to

invoke it. If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed. In other words, *it is the type of the object being referred to* (not the type of the reference variable) that determines which version of an overridden method will be executed.

```
//Java Program to illustrate the use of Java Method Overriding
//Creating a parent class.
class Vehicle
{
  //defining a method
  void run()
{
System.out.println("Vehicle is running");
}
}
//Creating a child class
class Bike extends Vehicle
{
  //defining the same method as in the parent class
  void run()
{
System.out.println("Bike is running safely");
}
}
class Tested
{
  public static void main(String args[])
{
Vehicle v1 = new Vehicle ();
  v1.run(); //calling parent method
  Bike obj = new Bike();//creating object
  obj.run();//calling subclass method
   }
}
```

Abstraction in Java:

Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, weight and cost, and methods that operate on these attributes. They encapsulate all the essential properties of the objects that are to be created.

In java, abstraction is achieved by interfaces and abstract classes.

ABSTRACT METHODS AND CLASSES:

We have seen that by making a method final we ensure that the method is not redefined in a subclass. That is, the method can never be subclassed.

Java allows us to do something that is exctly opposite to this. That is, we can indicate that a method must always be redefined in a subclass, thus making overriding compulsory.

This is done using the modifier keyword abstract in the method definition.

- Example:

```
abstract class  Shape
{……
……..
Abstract void draw();
……..
…….
}
```

- When a class contains one or more abstract methods, it should also be declared abstract methods as shown in the example above. While using abstract classes, we must satisfy the following conditions:
- We cannot use abstract classes to instantiate objects directly.

For example,

Shape s= newShape();

is illegal because shape is an abstract class.

- The abstract methods of an abstract class must be defined in its subclass.

• We cannot declare abstract constructors or abstract static methods.

Abstract classes and Abstract methods:

- An abstract class is a class that is declared with abstract keyword.
- An abstract method is a method that is declared without implementation.
- An abstract class may or may not have all abstract methods. Some of them can be concrete methods
- A method defined abstract must always be redefined in the subclass, thus making overriding compulsory OR either make subclass itself abstract.
- Any class that contains one or more abstract methods must also be declared with abstract keyword.
- There can be no object of an abstract class. That is, an abstract class can not be directly instantiated with the *new operator*.
- An abstract class can have parameterized constructors and default constructor is always present in an abstract class.

Example of abstract

```
abstract class Base {
   abstract void fun();
}
class Derived extends Base {
   void fun()
   {
```

```
        System.out.println("Derived fun() called");
    }
}
class Main
{
    public static void main(String args[])
    {

        // Uncommenting the following line will cause
        // compiler error as the line tries to create an
        // instance of abstract class. Base b = new Base();

        // We can have references of Base type.
      Derived  b = new Derived();
        b.fun();
    }
}
```

Final Keyword in Java

final variables and methods

All methods and variables can be overridden by default in subclasses. If user wish to prevent the subclasses from overriding the members of the superclass by using the keyword final as a modifier.

Example:

```
final int SIZE = 100;
 final void showstatus ()
{……..
  ……}
```

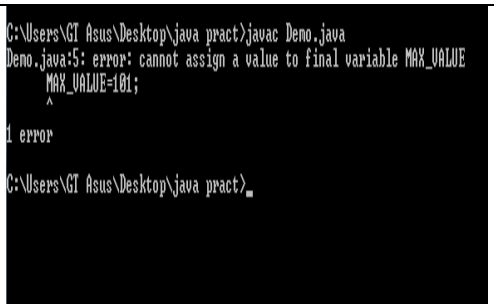
Making a method final ensures that the functionality defined in this method will never be altered in any way. Similarly, the value of a final variable can never be changed. Final variables, behave like class variables and they do not take any space on individual objects of the class.

Final variable

final variables are nothing but constants. User cannot change the value of a final variable once it is initialized.

```
class Demo
{
   final int MAX_VALUE=99;
   void myMethod(){
      MAX_VALUE=101;
   }
   public static void main(String args[]){
```

```
    Demo obj=new  Demo();
    obj.myMethod();
  }
}
```



final method

A final method cannot be overridden. Which means even though a sub class can call the final method of parent class without any issues but it cannot override it.

e.g.

```
class XYZ
{
  final void demo()
{
    System.out.println("XYZ Class Method");
  }
}

class ABC extends XYZ
{
  void demo()
{
    System.out.println("ABC Class Method");
  }

  public static void main(String args[]){
    ABC obj= new ABC();
    obj.demo();
  }
}
```

Final classes

Sometimes user may like to prevent a class being further subclassed for security reasons. A class that cannot be subclassed is called a final class. This is achieved in Java using the keyword final as follows:

final class Aclass {.....}

final class Bclass extends Someclass {.....}

Any attempt to inherit these classes will cause an error and the compiler will not allow it. Declaring a class final prevents any unwanted extensions to the class. It also allows the compiler to perform some optimisations when a method of a final class is invoked.

Example

```
class XYZ
{
  final void demo()
{
    System.out.println("XYZ Class Method");
  }
}

class ABC extends XYZ
{
  void demo()
{
    System.out.println("ABC Class Method");
  }

  public static void main(String args[]){
    ABC obj= new ABC();
    obj.demo();
  }
}
```

StringBuffer delete() Method in Java

The java.lang.StringBuffer.delete() is an inbuilt method in Java which is used to remove or delete the characters in a substring of this sequence. The substring starts at a specified index start_point and extends to the character at the index end_point.

Syntax :

```
public StringBuffer delete(int start_point, int end_point)
```

Parameters : The method accepts two parameters of integer type:
start_point – This refers to the beginning index and is included in the count.
end_point – This refer to the ending index and is excluded from the count.

Return Value : The method returns the string after deleting the substring formed by the range mentioned in the parameters.

Exceptions : StringIndexOutOfBoundsException occurs if the *start_point* is negative, greater than length(), or greater than the *end_point*.

Examples :

Input: String = "Apple”

start_point = 2 end_point = 4

Output: Ape

Input: String = "GeeksforGeeks"

start_point = 2 end_point = 7

Output: GerGeeks

```
// Java praogram to illustrate the
// java.lang.StringBuffer.delete()
import java.lang.*;
  public class geeks
{
     public static void main(String[] args)
   {
         StringBuffer sbf = new StringBuffer("Geeksforgeeks");
       System.out.println("string buffer = " + sbf);
       sbf.delete(6, 8);
       System.out.println("After deletion string buffer is = " + sbf);
   }
}
```