JAVA

Method OverLoading-Method Overriding-Inheritance-Abstract Class-Strategy Pattern-Template Method Pattern-Super-Final

Shoiab

Method OverLoading

"Method overloading is a feature of Java in which a class has more than one method of the same name and their parameters are different."

In other words, we can say that Method overloading is a concept of Object Oriented Analysis in which we can create multiple methods of the same name in the same class, and all methods work in different ways. When more than one method of the same name is created in a Class, this type of method is called Overloaded Methods and its helps in achieving POLYMORPHISM.

Java use a powerful feature called Dynamic Method Invocation / Virtual Method Invocation to determine which method to be called using the parameter list.

```
package hit.day8;

public class MethodOverLoadDemo {
     public static void main(String[] args)
      {
        Calculate obj = new Calculate();
        obj.multiply();
        obj.multiply(5);
        obj.multiply(2.5f);
      }
}
class Calculate
{
public void multiply()
{
     System.out.println("No Parameter Method Called");
}
public int multiply( int number )
{
     int result = number * number;
     System.out.println("Method with Integer Argument
Called:"+result);
     return result;
```

```
}
public float multiply( float number )
{
     float result = number * number;
     System.out.println("Method with float Argument
Called:"+result);
     return result;
}

}
```

In this way, we can define more than one Methods of the same name in a class, which is called Method Overloading, and

The Java compiler itself performs the appropriate Method Call for an object, based on the Data Type of the Arguments of the Methods.

Benefits of using Method Overloading

- Method overloading increases the readability of the program.
- This provides flexibility to programmers so that they can call the same method for different types of data.
- This makes the code look clean.
- This reduces the execution time because the binding is done in compilation time itself.
- Method overloading minimises the complexity of the code.
- With this, we can use the code again, which saves memory.

How to do Method Overloading?

In java, we do method overloading in two ways: –

1. By changing the number of parameters.
2. By changing data types.(above example)

- Change the number of arguments:

```
package hit.day8;

public class MethodOverLoadDemo {
```

```
    public static void main(String[] args)
      {
        Demo obj = new Demo();
        obj.multiply(2,50);
        obj.multiply(10,30,40);
      }
}
class Demo
{
  void multiply(int a, int b)
  {
    System.out.println("Result is"+(a*b)) ;
  }
  void multiply(int a, int b,int c)
  {
    System.out.println("Result is"+(a*b*c));
  }

}
```

Some points to remember about method overloading: –

- Method overloading cannot be done by changing the return type of methods.
- The most important rule of method overloading is that two overloaded methods must have different parameters.

Method overloading has nothing to do with return-type.

If there are two methods of the same signature within a class in the program, then Ambiguity Error comes, whether their return-type is different or not. This means that method overloading has no relation with return-type.

```
package hit.day8;

public class MethodOverLoadDemo {
    public static void main(String args[])
    {
```

```
        Sample s = new Sample();
        System.out.println("Value of x : " + s.disp(5));
        System.out.println("Value of y : " + s.disp(6.5));
    }
}
class Sample{
    int disp(int x){
        return x;
    }
    double disp(int y){
        return y;
    }
}
```

error: method disp(int) is already defined in class Sample

In this way, we can define more than one Methods of the same name in a class called Method Overloading. The Java Compiler itself, based on the Data Type of the Arguments of the Methods, performs the appropriate method call for an object.

Examples worked out in class. – See the video for explanation.

```
package hit.day7;

public class Service108 {
    void help(Police p) {
        System.out.println("the caller is a police man...
let connect with control room");
    }
    void help(AccidentVictim a) {
        System.out.println("the caller is a accident
victim...let connect with doctor...");
    }
    void help(Helper h) {
        System.out.println("the caller is a helper....let
us help him....");
    }
//  void help(Object o) {
```

```
//        if(o instanceof Police) {
//             System.out.println("police...");
//        }
//        else if(o instanceof AccidentVictim) {
//             System.out.println("accident victim...");
//        }
//        else if(o instanceof Helper) {
//             System.out.println("write logic for
helper...");
//        }
//   }

     public static void main(String[] args) {
          Service108 obj108=new Service108();
          Police captain=new Police();
          AccidentVictim ac=new AccidentVictim();
          Helper ramasamy=new Helper();

          obj108.help(ramasamy);
     }
}

class Police{

}
class AccidentVictim{

}
class Helper{

}
```

Constructor OverLoading – Listen to my video for understanding

```
package hit.day7;

//Constructor overloading..
```

```
public class ConsDemo {
	public ConsDemo() {
		System.out.println("cons called...");
	}

	public ConsDemo(int i) {
		System.out.println("overloaded cons called...");
	}

	public ConsDemo(String s) {
		System.out.println("string constructor called...");
	}

	public ConsDemo(int s,String str) {
		System.out.println("multi parameter constructor
called...");
	}
	public static void main(String[] args) {
		ConsDemo obj1=new ConsDemo(100,"hello");
		ConsDemo obj2=new ConsDemo();

		obj2.met();
		obj2.met(1);
		//the name of the concept is - Virtual Method
Invocation (VMI)
		//only one constructor can be called when one
object is created...
	}

	void met() {
		System.out.println("method without parameter
called...");
	}
	void met(int i) {
		System.out.println("method with parameter
called...");
	}
```

```
}
```

Method Overriding

What is Overriding in Java?
Overriding is when a child class has its method implementation for the method already present in the parent class.

Technically, overriding is a function that requires a subclass or child class to provide a variety of method implementations, that are already provided by one of its superclasses or parent classes, in any object-oriented programming language.

When a method in a subclass has the same name and signature as in its super-class, the subclass is originated from the super-class.

One of the ways that Java manages Run Time Polymorphism is by method overriding.

The object that is used to trigger a method specifies the variant of the process that is executed. If it implements a method with an object from a parent class, the parent class's version will be used. But if the method is triggered with an object from a subclass, the child class's version will be used.

What Are the Rules for Method Overriding in Java?

Laws of Method Overriding in JAVA:

1. The method name should be common and the same as it is in the parent class.
2. The method signature (parameter list, return type) in the method must be the same as in the parent class.
3. There must be an inheritance connection between classes.
4. All the abstract methods in the parent class should be overridden in the child class.
5. If it declared the methods as static or final, then those methods cannot be overridden.

Handle Access-Modifiers in Overriding:

An overriding method's activities increase will give more access than the overridden method. A protected method in the parent class may be made public but not private in the child class. If you force it to be done, it will cause a compile-time error.

The goal of Approach Overriding in Java is transparent in this situation. The child class has to have its

implementation of this process.

For Final methods:

If you declare any method as final, it cannot override it. It is impossible to override final methods.

For Constructor methods:

Obviously, the constructor will have its class name, so it also cannot be overridden.

For Abstract methods:

If the abstract method is in the interface or any other class, the child class should override.

For Static methods:

Similar to the final methods, the static methods also cannot be overridden. The static method in the parent class will be hidden from the child class.

Activating an Overridden Method:

The super keyword will be used to invoke the parent class function in an overriding method.

Error Handling in Overriding in JAVA:

There are three ways how overriding deals with exception-handling.

- When the parent class doesn't declare an exception, the child class can declare only unchecked exceptions.
- When the parent class has declared an exception, the child class can declare the same exception, not any other exceptions.
- When the parent class has declared an exception, the child class can declare without exceptions.

Note:

1. By overriding, the child class extends the capabilities of the parent class.
2. Method overriding implements both polymorphism and inheritance for code scalability.

In Java, method overriding occurs when a subclass (child class) has the same method as the parent class. In other words, method overriding occurs when a subclass provides a particular implementation of a method declared by one of its parent classes. The ability for a subclass to override a method allows a class to inherit from a superclass with "near enough" actions and then change it as required. The name, number,

and type of parameters, and return type of the overriding method are identical to those of the method it overrides.

```
package hit.day10;

public class InherDemo {
	public static void main(String[] args) {
		Employee p=new Engineer();
		Engineer eng=new Engineer();

		p.met(2);
		eng.met(2);
	}
}
class Human{
	public void blabla() {
		System.out.println("bla bla method called....");
	}
}
class Employee extends Human{
	public int met(int i) {
		System.out.println("met method of employee
called...");
		return i;
	}
}
//rules of method overriding
//1. the parameters, method name and return type shoud be
same to same
//2. the accept2 specifier cannot be reduced - visibility
cannot be reduced but can be increased.
//inheritance or generalization
class Engineer extends Employee{//child is a kind of
parent,... child can replace parent
	public int met(int i) {//method overriding..
		System.out.println("engineer met method
called....");
```

```
            super.met(4);//super is a key word which refers the
parent class...
            super.blabla();
            return i;
        }
}
class Attender extends Employee{

}
```

```
package hit.day12;
/*
 * bad - code which is open for modification and closed for
extension - bad brush
 * good - code which is closed for modification and open for
extension -good brush
 * 1. Whenever you find if-else-if conditions then apply
this solution
 * 1. Convert the condition into classes
 * 2. Group them under a hierarchy
 * 3. create a association between the using class
(PaintBrush) and the hierarchial class(Paint)
 *  ooa principles used
 *  1. Inheritance 2. Association
 *  what we achieved -
 *  1. open close principle
 *  2. Object Reusability
 */
public class GoodBrush {
    public static void main(String[] args) {
        GoodPaintBrush brush=new GoodPaintBrush();
        RedPaint rp=new RedPaint();
        BluePaint bp=new BluePaint();
        GreenPaint gp=new GreenPaint();

        brush.rp=gp;
        brush.doPaint();
```

```
	}
}

class GoodPaintBrush{// in this paintbrush the code is not modified whenever a new color is chosen
	Paint rp;
	public void doPaint() {
		System.out.println(rp);
	}
}

class BadPaintBrush{// in this paintbrush the code gets modified whenever a new color is chosen
	public void doPaint(int choice) {

		if(choice==1) {
			System.out.println("red colour...");
		}
		else if(choice==2) {
			System.out.println("blue colour...");
		}
		else if(choice==3) {
			System.out.println("green colour...");
		}
	}

}
/*
 * 100% the parent class should be declared abstract
 * abstract class is a special class used to represent the parent class, its a classfier class
 * it provided a overview of the heirarchy which u create by extending this class..
 * You cannot create an object of this class
 * There is no compulsion to have any code inside the abstract class
 */
```

```
abstract class Paint{

}
class RedPaint extends Paint{

}
class BluePaint extends Paint{

}
class GreenPaint extends Paint{

}
```

```
package hit.day12;
/*
 * Why Inheritance
 * 1. Object Reusability
 * 2. create a part-whole hierarchy
 * 3. composition
 * 4. eliminate if-else-if condition statements -
 * 5. Polymorphic queries - 90%
 * 6. Code Reusability - 1%
 */
public class MultipleInherDemo {
     public static void main(String[] args) {
          Shoiab obj=new Shoiab();

     }
}
class LivingObject{

}
class Human extends LivingObject{

}
class Dracula{

}
```

```
class Dog{

}

class Shoiab extends Human{

}
```

```
package hit.day13;

import java.util.Scanner;

/*
 * 1. Convert the condition to classes
 * 2. Group them under a hierarchy - Inheritance
 * 3. Create a Association between the using class and the
hierarchy class - association
 *
 * Design Patterns - Strategy Pattern
 * A strategy to implements openclose principle
 * A strategy to remove if-else-condition
 */
public class GoodFanDemo {
     public static void main(String[] args) {
          Scanner scan=new Scanner(System.in);
          //BadFan shaitan=new BadFan();
          GoodFan khaitan=new GoodFan();
          while(true) {
               System.out.println("Press a key and then Enter
for pulling chain..");
               scan.next();
               khaitan.pull();
          }
     }
}
/*
 * Paint brush followed uni directional relationship
 * Fan will follow bi-directional relationship
```

```
 */
class GoodFan{
     State state=new SwitchOffState();//association
     public void pull() {
          state.pull(this);
     }
}
abstract class State{
     public abstract void pull(GoodFan fan);// {}
}
class SwitchOffState extends State{// condition class
grouped under a category
     public void pull(GoodFan fan) {
          System.out.println("Switch on state....");
          fan.state=new SwitchOnState();
     }
}
class SwitchOnState extends State{
     @Override
     public void pull(GoodFan fan) {
          System.out.println("medium speed
state.............");
          fan.state=new MediumSpeedState();
     }
}
class MediumSpeedState extends State{
     @Override
     public void pull(GoodFan fan) {
          System.out.println("high speed state..........");
          fan.state=new HighSpeedState();
     }
}
class HighSpeedState extends State{
     @Override
     public void pull(GoodFan fan) {
          System.out.println("swtitc off state....");
          fan.state=new SwitchOffState();
```

```
	}
}

class BadFan{
	int state;
	public void pull() {
		if(state==0) {
			System.out.println("switch on state...");
			state=1;
		}
		else if(state==1) {
			System.out.println("Medium speed state...");
			state=2;
		}
		else if(state==2) {
			System.out.println("High speed state...");
			state=3;
		}
		else if(state==3) {
			System.out.println("switch off state...");
			state=0;
		}

	}
}
```

```
package hit.day13;

public class AbstractDemo {
	public static void main(String[] args) {
		//condtion 1 - You cannot create an object of
abstract class
		//new Parent();//this will throw me an error
		new Child();
	}
}
/*
 * final variables values cannot be changed
```

```
 * final classes cannot be inherited
 * final methods cannot be overriden
 */
/*
 * when an abstract class extends another abstract class,
the child class need not override
 * the parent abstract class method...
 */
abstract class GrandParent{
     public abstract void met();
}
abstract class Parent extends GrandParent{
     //condition 2 - Can I have a constructor inside abstract
class
     //ans: yes you can have....
     final int i=10;//its a constant
     public Parent() {
          System.out.println("abstract class Parent cons
called...");
     }
     public abstract void yoursSayHello();
     final public void mySayHello() {//final methods cannot
be overriden
          System.out.println("saying hello as per the
abstract class norms...");
     }
}
//when you create an object of child class, the parent
abstract class constructor gets called

class Child extends Parent{
     public void met(){}
     public Child() {
          System.out.println("child class of abstract parent
object created...");
     }
     @Override
```

```
	public void yoursSayHello() {
		System.out.println("say hello as per the child
class logic....or norms...");
	}
	@Override
	public void mySayHello() {
		System.out.println("write my own logic..");
	}
}
```

```
package hit.day13;

public class TemplateMethodDemo {
	public static void main(String[] args) {
		Sangeetha sangeethaHotel=new
ShoiabSangeethaRestaurant();
		sangeethaHotel.process();
	}
}

abstract class Sangeetha{
	public final void uniform() {
		System.out.println("uniform defined by
sangeetha...");
	}
	public final void serve() {
		System.out.println("sangeetha defines how to
serve...");
	}
	public abstract void makeIdly();

	public final void createIdlyPlate() {
		System.out.println("sangeetha define how to create
a idly plate...");
	}
	public final void charge() {
		System.out.println("sangeetha define how much to
```

```
charge....");
	}
	public final void process() {//Template method...
		uniform();
		makeIdly();
		createIdlyPlate();
		serve();
		charge();
	}
}
class ShoiabSangeethaRestaurant extends Sangeetha{

	@Override
	public void makeIdly() {
		System.out.println("my grandma makes the idly in my
hotel.....");
	}

}
```

```
package hit.day10;

public class ConsInherDemo {
	public static void main(String[] args) {
		Shoe shoe=new LeatherShoe(100);
	}
}


class Shoe{
	int i;
	public Shoe() {
		System.out.println("cons of shoe called....");
	}
	public Shoe(int i) {
		System.out.println("parametric constructor of shoe
called....");
	}
```

```
     public void met() {
          System.out.println("parent shoe class method
called...");
     }
}

class LeatherShoe extends Shoe{
     //there is no concept of over riding in constructor
     //always the parent class default constructor is
called...
     public LeatherShoe(int i) {
          super(343);//super class constructor - should be
the first line in the constructor
          super.i=100;//super can refer to super class
variable
          super.met();//super can refer to super class method
          System.out.println("cons of leather shoe
called....");
     }
}
```

PassByValue and PassByReference

```
package hit.day12;

public class PbvPbr {
     public static void main(String[] args) {
          Laddu laddu=new Laddu();
          laddu.setSize(10);
          System.out.println("Initial size of
laddu..:"+laddu.getSize());

          PBV pbvObj=new PBV();
          pbvObj.modifySizeOfLaddu(laddu.size);

          System.out.println("Size of Laddu after
PBV...:"+laddu.getSize());
```

```
            PBR pbrObj=new PBR();
            pbrObj.modifySizeOfLaddu(laddu);

            System.out.println("Sizeof Laddu after
PBR...:"+laddu.getSize());
      }
}
//all primitives types, a copy is made when it is passed -
viz. int,short,byte,long,char etc
//all complex types viz.laddu is always pass by reference,
does not make a copy.
class PBV{
      public void modifySizeOfLaddu(int size) {
            size=size-5;
      }
}
class PBR{
      public void modifySizeOfLaddu(Laddu laddu) {
            laddu.setSize(5);
      }
}
class Laddu{
      int size;
      public void setSize(int size) {
            this.size=size;
      }
      public int getSize() {
            return this.size;
      }
}
```

```
package hit.day12;

import com.sun.jdi.LocalVariable;

public class PBRArrayDemo {
      static int classVariable;//class variable
      int instanceVariable;//instance variable
```

```
     static int arr[]=new int[5];
     int array[]=new int[5];
     public static void main(String[] args) {
          System.out.println(classVariable);
          PBRArrayDemo pba=new PBRArrayDemo();
          System.out.println(pba.instanceVariable);

          int localVariable=0;//local variable -- local
variables are not initialized by default...
          System.out.println(localVariable);

          for(int i:arr) {
               System.out.print(i+"\t");
          }
          System.out.println();
          for(int i:pba.array) {
               System.out.print(i+"\t");
          }
          System.out.println();
          int localArray[]=new int[5];//arrays are
initialized by default even if they are locally declared
          for(int i:localArray) {
               System.out.print("\tlocal..:"+i);
          }

     }
}
```

```
package hit.day12;
//arrays are object type and the belive in Pass by
reference.
public class PBRArrayDemo2 {
     public static void main(String[] args) {
          int arr[]=new int[5];
          for(int i:arr) {
               System.out.print(i+"\t");
          }
```

```
            System.out.println();
            AcceptArray obj=new AcceptArray();
            obj.modifyArray(arr);
            System.out.println("After Modification...");
            for(int i:arr) {
                 System.out.print(i+"\t");
            }
            System.out.println();

      }
}

class AcceptArray{
      public void modifyArray(int a[]) {
            a[2]=300;
      }
}
```

String Class

Java – String Class and its methods explained with examples

String is a sequence of characters, for e.g. “Hello” is a string of 5 characters. In java, string is an immutable object which means it is constant and can cannot be changed once it has been created. In this tutorial we will learn about String class and String methods in detail along with many other Java String tutorials.

Creating a String

There are two ways to create a String in Java

1. String literal
2. Using new keyword

String literal

In java, Strings can be created like this: Assigning a String literal to a String instance:

```
String str1 = "Welcome";
String str2 = "Welcome";
```

The problem with this approach: As I stated in the beginning that String is an object in Java. However we have not created any string object using new keyword above. The compiler does that task for us it creates a string object having the string literal (that we have provided , in this case it is “Welcome”) and assigns it to the provided string instances

But if the object already exist in the memory it does not create a new Object rather it assigns the same old object to the new instance, that means even though we have two string instances above(str1 and str2) compiler only created on string object (having the value “Welcome”) and assigned the same to both the instances. For example there are 10 string instances that have same value, it means that in memory there is only one object having the value and all the 10 string instances would be pointing to the same object.

What if we want to have two different object with the same string? For that we would need to create strings using **new keyword**.

Using New Keyword

As we saw above that when we tried to assign the same string object to two different literals, compiler only created one object and made both of the literals to point the same object. To overcome that approach we can create strings like this:

```
String str1 = new String("Welcome");
String str2 = new String("Welcome");
```

In this case compiler would create two different object in memory having the same string.

A Simple Java String Example

```
class Example{
        public static void main(String args[]){
          //creating a string by java string literal
          String str = "HaarisInfotech";
          char arrch[]={'h','e','l','l','o'};
          //converting char array arrch[] to string str2
          String str2 = new String(arrch);

          //creating another java string str3 by using new
keyword
          String str3 = new String("Java String Example");

          //Displaying all the three strings
          System.out.println(str);
```

```
            System.out.println(str2);
            System.out.println(str3);
        }
}
```

String Methods

```
package hit.day13;

public class StringDemo {
     public static void main(String[] args) {
            String s="hello";//recommended approach by default
            String st=new String("hello");
            System.out.println(s.hashCode());
            System.out.println(st.hashCode());
            System.out.println(s==st);
            System.out.println(s.equals(st));//if both the
hascodes are same then the result will be true

            //string objects are non mutable

            String temp="hello new temp world..."+s;
            //+ - is a concatenation operator
            System.out.println(temp);
            System.out.println(s);

            String mystore=s;

            s=s+"new value....";
            System.out.println(s);

            System.out.println(mystore);
            //string is non mutable... once u declare a
string.. the string value cannot be changed...
            //when u change then a new copy of string is
created...
```

```
        //So be very carefull when you operations like
concat/split/reverse etc on a string in indefinite loop

        char c=s.charAt(2);
        System.out.println(c);
        System.out.println(s.substring(2, 4));
        System.out.println(mystore==st);
        System.out.println(mystore.equals(st));
    }
}
```

```
class Main {
      public static void main(String[] args) {

        // create a string
        String greet = "Hello! World";
        System.out.println("String: " + greet);

        // get the length of greet
        int length = greet.length();
        System.out.println("Length: " + length);
      }
}
```

Join two Strings

We can join two strings in Java using the concat() method. For example,

```
class Main {
      public static void main(String[] args) {

        // create first string
        String first = "Java ";
        System.out.println("First String: " + first);

        // create second
        String second = "Programming";
        System.out.println("Second String: " + second);
```

```
        // join two strings
        String joinedString = first.concat(second);
        System.out.println("Joined String: " +
joinedString);
    }
}
```

Compare two Strings

In Java, we can make comparisons between two strings using the equals() method. For example,

```
class Main {
    public static void main(String[] args) {

        // create 3 strings
        String first = "java programming";
        String second = "java programming";
        String third = "python programming";

        // compare first and second strings
        boolean result1 = first.equals(second);
        System.out.println("Strings first and second are
equal: " + result1);

        // compare first and third strings
        boolean result2 = first.equals(third);
        System.out.println("Strings first and third are
equal: " + result2);
    }
}
```

```
class StringMethodsDemo

{

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

```
 {

   String s1="HAARISINFOTECH";

   String s2=" TRAINING DEPARTMENT";

   System.out.println("the concatenation of two strings is
"+s1.concat(s2));

   System.out.println("the character at 7th place of string
2 is "+s2.charAt(7));

   System.out.println("the comparison of two strings is
"+s1.compareTo(s2));

   System.out.println("Boolean comparison ignoring case
considerations "+

    s1.equalsIgnoreCase(s2));

   System.out.println("hash code of string 2 is
"+s2.hashCode());

   System.out.println("index of E in string 2 is
"+s2.indexOf('e',3));

   System.out.println("index of E in string 2 from last is
"+s2.lastIndexOf('e',16));

   System.out.println("length of string 2 is "+s2.length());

   System.out.println("after replacing e with k in string 1,
string 1 = "+s1.replace('H','P'));

   System.out.println("substring of string 2 =
"+s2.substring(4));
```

```
   System.out.println("substring of string 2 from 4 to
9"+s2.substring(4,9));

   System.out.println("lower case of string1 is
="+s1.toLowerCase());

   System.out.println("UPPER case of string 2 =
"+s2.toUpperCase());

   System.out.println("after trimming, string 2
="+s2.trim());

  }

}
```

```
import java.util.StringJoiner;
class StringDemo {
     public static void main(String[] args) {
          String s="hello world";
          String ss=new String("hello world");

          StringJoiner joinstr=new StringJoiner(",");
          joinstr.setEmptyValue("its a empty string..");
          System.out.println(joinstr);
          joinstr.add("hello");
          joinstr.add("hai");
          System.out.println(joinstr);

          joinstr=new StringJoiner(",","[","]");
          joinstr.add("hello");
          joinstr.add("hai");
          System.out.println(joinstr);

          StringBuffer sbf=new StringBuffer();//synchronized
          sbf.append("hello");
          StringBuilder sb=new StringBuilder();//non
```

```
synchronized..
          sb.append("hai");
     }
}
```

```
class StringSplitTest {
   public static void main(String[] args) {
      String str = "Monday Tuesday Wednesday Thursday Friday
Saturday Sunday";
      String[] tokens = str.split("\\s");  // white space
'\s' as delimiter
      StringBuilder sb = new StringBuilder();
      for (int i = 0; i < tokens.length; ++i) {
         sb.insert(0, tokens[i]);
         if (i < tokens.length - 1) {
            sb.insert(0, " ");
         }
      }
      String strReverse = sb.toString();
      System.out.println(strReverse);
   }
}
```

```
/*
 * % is a special character denoting that a formatting
instruction follows.
 * [flag]-special formatting instruction. – that is pad on
the right , + pad on the left
 * [width] denotes the minimum number of output characters
for that Object.
 * [.precession] denotes the precision of floating point
 * type along with % - For integers that is d,
 * for strings that is s, for floating point numbers that is
f
 */
public class StringFormatDemo {
public static void main(String[] args) {
     System.out.printf("Integer : %d\n",15);
```

```
     System.out.printf("Floating point number with 3 decimal
digits: %.3f\n",1.21312939123);
     System.out.printf("Floating point number with 8 decimal
digits: %.8f\n",1.21312939123);
     System.out.printf("String: %s, integer: %06d, float:
%.6f", "Hello World",89,9.231435);
     String s=String.format("String: %s, integer: %d, float:
%.6f", "Hello World",89,9.231435);
     System.out.println("\n"+s);
     System.out.printf("%-12s%-12s%s\n","Column 1","Column
2","Column3");
     System.out.printf("%-12.5s%s", "Hello World","World");
}
}
/*
 * Integer formatting
%d       : will print the integer as it is.
%6d    : will pint the integer as it is. If the number of
digits is less than 6, the output will be padded on the
left.
%-6d  : will pint the integer as it is. If the number of
digits is less than 6, the output will be padded on the
right.
%06d : will pint the integer as it is. If the number of
digits is less than 6, the output will be padded on the left
with zeroes.
String formatting
%s       : will print the string as it is.
%15s    : will pint the string as it is. If the string has
less than 15 characters, the output will be padded on the
left.
%-15s  : will pint the string as it is. If the string has
less than 6 characters, the output will be padded on the
right.
%.8d : will print maximum 8 characters of the string.
 */
```