Dayananda Sagar College of Engineering

Department of Information Science and Engineering

Department of Information Science and Engineering

DSCE

Dept. of Information Science & Engg

Course Name: INTRODUCTION TO JAVA

Course Code:18IS6IEJVA
Semester:6

Faculty : Mrs.Radhika T V
Assistant Professor, Dept of ISE
DSCE

Module 2:
CLASSES

Class in JAVA

- A class is a user defined blueprint or prototype from which objects are created. Objects are real life entities(or) it is an instance of class.
- In general, class declarations can include following components, in order:
- **Modifiers** : A class can be public or has default access
- **Class name:**The name should begin with Capitial letter
- **Superclass(if any):**The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
- **Interfaces(if any):**A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
- **Body:**The class body surrounded by braces, { }.

Class cntd..

- An object consists of :
- **State**: It is represented by attributes of an object. It also reflects the properties of an object.
- **Behavior**: It is represented by methods of an object. It also reflects the response of an object with other objects.
- **Identity**: It gives a unique name to an object and enables one object to interact with other objects.

Example of Class

```
public class Dog
{
 String breed;
int age;
String color;
void barking()
{
 }
 void hungry()
 {
}
void sleeping()
{
 }
 }
```

Class cntd..

- A class can contain any of the following variable types.
- **Local variables** − Variables defined inside methods, constructors or blocks are called local variables.
- **Instance variables** -Instance variables are variables within a class but outside any method.
  - Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.
- **Class variables −** Class variables are variables declared within a class, outside any method, with the static keyword.

Class cntd..

Constructors

- Every class has a constructor. If we do not explicitly write a constructor for a class, the Java compiler builds a default constructor for that class.
- The main rule of constructors is that they should have the same name as the class. A class can have more than one constructor.
- Following is an example of a constructor −

```
public class Puppy
 {
 public Puppy()
{
 }
 public Puppy(String name)
{
 // This constructor has one parameter, name.
 } }
```

Examples of Types of constructors

```
//Java Program to illustrate calling a
// no-argument constructor
import java.io.*;

class Geek
{
int num;
String name;

// this would be invoked while an object
// of that class is created.
Geek()
{
System.out.println("Constructor called");
}
}
```

```
class GFG
{
public static void main (String[]
args)
{
// this would invoke default
constructor.
Geek geek1 = new Geek();

// Default constructor provides
the default
// values to the object like 0,
null
System.out.println(geek1.name)
;
System.out.println(geek1.num);
}
}
```

Output :

```
Constructor called
null
0
```

Examples of Types of constructors

```
// Java Program to illustrate calling of
// parameterized constructor.
import java.io.*;

class Geek
{
    // data members of the class.
    String name;
    int id;

    // constructor would initialize data members
    // with the values of passed arguments while
    // object of that class created.
    Geek(String name, int id)
    {
        this.name = name;
        this.id = id;
    }
}
```

```
class GFG
{
    public static void main
(String[] args)
    {
        // this would invoke the
parameterized constructor.
        Geek geek1 = new
Geek("adam", 1);

System.out.println("GeekName
:" + geek1.name +
                           " and GeekId
:" + geek1.id);
    }
}
```

Output:

```
GeekName :adam and GeekId
:1
```

Class cntd..

- Types of Constructors
  - **No argument Constructors**

    A constructor that has no parameter is known as default constructor.
  - **Parameterized Constructors**
    - A constructor that has parameters is known as parameterized constructor. If we want to initialize fields of the class with your own values, then use a parameterized constructor.

Class cntd..

Creating an Object

- A class provides the blueprints for objects. So basically, an object is created from a class.
- In Java, the new keyword is used to create new objects.
- There are three steps when creating an object from a class −
- **Declaration** − A variable declaration with a variable name with an object type.
- **Instantiation** − The 'new' keyword is used to create the object.
- **Initialization** − The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

Example of Object

```
public class Puppy
{
public Puppy(String name)
{ // This constructor has one parameter, name.
    System.out.println("Passed Name is :" + name );
}
public static void main(String []args)
{ // Following statement would create an object myPuppy Puppy
    Puppy puppy1 = new Puppy( "tommy" );
 }
}
```

Assigning Object Reference Variables

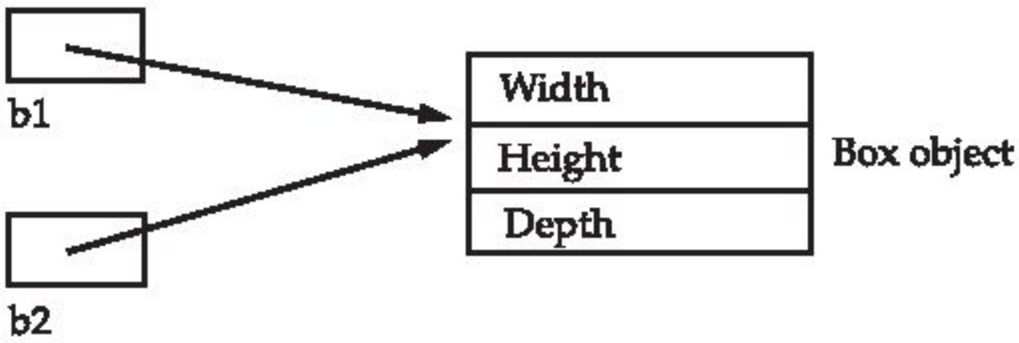
- Object reference variables act differently when an assignment takes place.

For example,

- Box b1 = **new** Box();
- Box b2 = b1;
- b1 and b2 will both refer to the same object.
- The assignment of b1 to b2 did not allocate any memory or copy any part of the original object.
- It simply makes b2 refer to the same object as does b1.
- Thus, any changes made to the object through b2 will affect the object to which b1 is referring.

Assigning Object Reference Variables

- This situation is depicted here



- Although **b1** and **b2** both refer to the same object, they are not linked in any other way.
- For example, a subsequent assignment to **b1** will simply *unhook* **b1** from the original object without affecting the object or affecting **b2**. For example:

```
Box b1 = new Box();
Box b2 = b1;
// ...
b1 = null;
```

Here, **b1** has been set to **null**, but **b2** still points to the original object

Introducing Methods

- classes usually consist of two things: instance variables and methods.
- This is the general form of a method:

```
type name(parameter-list) {      add(int a, int b)
            // body of method
                        }
```

- Methods that have a return type other than **void return a value to the calling routine using**
- the following form of the **return statement:**
- return *value;*
- Here, *value is the value returned.*

Adding a Method to the Box Class

```
class Box {
    double width;
    double height;
    double depth;
// display volume of a box
void volume() {
    System.out.print("Volume is ");
    System.out.println(width * height * depth);
            }
}
```

Cntd..

```
class BoxDemo3 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
/* assign different values to mybox2's
instance variables */
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;
// display volume of first box
mybox1.volume();
// display volume of second box
mybox2.volume();
}
}
```

Output:

```
Volume is 3000.0
Volume is 162.0
```

Returning a Value

- A better way to implement **volume( ) is to have it compute the volume of the box and return the result** to the caller.
- Ex:

```
class Box {
            double width;
            double height;
            double depth;
            // compute and return volume
            double volume() {
            return width * height * depth;
}
}
```

Returning a Value Cntd..

```
class BoxDemo4 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
/* assign different values to mybox2's
instance variables */
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

Output:

```
Volume is 3000.0
Volume is 162.0
```

Returning a Value Cntd..

- There are two important things to understand about returning values:
- The type of data returned by a method must be compatible with the return type specified by the method. For example, if the return type of some method is **boolean,** you could not return an integer.
- The variable receiving the value returned by a method (such as **vol, in this case) must** also be compatible with the return type specified for the method.

Adding a Method That Takes Parameters

- While some methods don’t need parameters, most do.
- Parameters allow a method to be generalized.
- That is, a parameterized method can operate on a variety of data and/or be used in a number of slightly different situations.
- <u>Example 1:</u>

```
int square()
{
return 10 * 10;
}
```

Adding a Method That Takes Parameters cntd..

- If we modify the method that it takes a parameter, then we can make **square( ) much more useful.**

<u>Example 2:</u>

```
int square(int i)
{
return i * i;
}
```

- That is, **square( ) is** now a general-purpose method that can compute the square of any integer value, rather than just 10.

Adding a Method That Takes Parameters cntd..

Here is an example:

```
int x, y;
x = square(5); // x equals 25
x = square(9); // x equals 81
y = 2;
x = square(y); // x equals 4
```

- It is important to keep the two terms *parameter and argument straight.*
- *A parameter is a* variable defined by a method that receives a value when the method is called.
- *An argument is a value that is passed to a method when it is invoked.*

Adding a Method That Takes Parameters cntd..

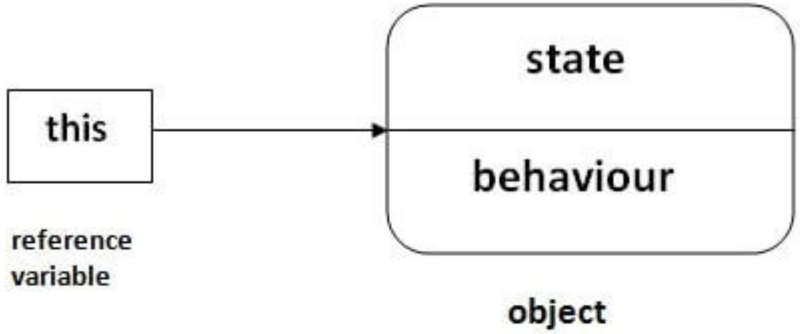
```
// This program uses a parameterized method.
class Box {
double width;
double height;
double depth;
// compute and return volume
double volume() {
return width * height * depth;
}
// sets dimensions of box
void setDim(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
}
```

Cntd..

```
class BoxDemo5 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// initialize each box
mybox1.setDim(10, 20, 15);
mybox2.setDim(3, 6, 9);
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

This keyword

- In java, this is a **reference variable** that refers to the current object.



This keyword cntd..

- Here is given the 6 usage of java this keyword.
- this can be used to refer current class instance variable.
- this can be used to invoke current class method (implicitly)
- this() can be used to invoke current class constructor.
- this can be passed as an argument in the method call.
- this can be passed as argument in the constructor call.
- this can be used to return the current class instance from the method.

Example: without this keyword

```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
rollno=rollno;
name=name;
fee=fee;
}
void display(){System.out.println(rollno+""+name+""+fee);} class TestThis1{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

Output:

```
0 null 0.0
 0 null 0.0
```

Example(using this keyword)

```
Class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
this.rollno=rollno;
this.name=name;
this.fee=fee;
}
void display(){System.out.println(rollno+""+name+""+fee);}
}
class TestThis2{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

Output:

```
111 ankit 5000
112 sumit 6000
```

this: to invoke current class met

- You may invoke the method of the current class by using the this keyword.

Example:

```
class A{
void m(){System.out.println("hello m");}
void n(){
System.out.println("hello n");
//m();//same as this.m()
this.m();
}
}
class TestThis4{
public static void main(String args[]){
A a=new A();
a.n();
}}
```

Output:

```
hello n
 hello m
```

this() : to invoke current class constructor

- The this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor.

Example:

```
class A{
A(){System.out.println("hello a");}
A(int x){
this();
System.out.println(x);
}
}
class TestThis5{
public static void main(String args[]){
A a=new A(10);
}}
```

Output:

```
hello a
 10
```

Garbage Collection in Java

- But in Java, the programmer need not to care for all those objects which are no longer in use. Garbage collector destroys these objects.

- Garbage collector is best example of Daemon thread as it is always running in background.

- Main objective of Garbage Collector is to free heap memory by destroying **unreachable objects**.

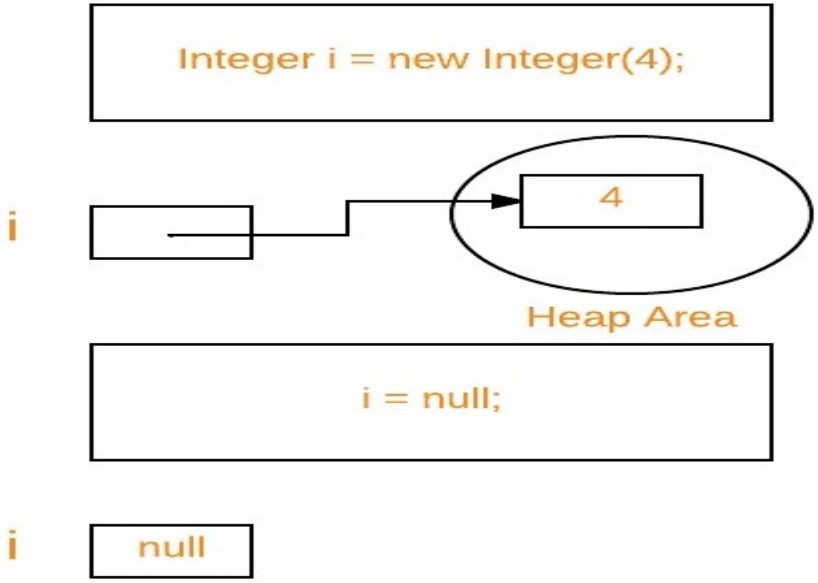
Garbage Collection in Java

Important terms :

- **Unreachable objects**
- **<u>Ex:</u>**Integer i = new Integer(4);

// the new Integer object is reachable via the reference in 'i' i = null;

// the Integer object is no longer reachable.



Garbage Collection in Java cntd.,

- **Eligibility for garbage collection :**
- An object is said to be eligible for GC(garbage collection) iff it is unreachable.
- There are generally four different ways to make an object eligible for garbage collection.
  - Nullifying the reference variable
  - Re-assigning the reference variable
  - Object created inside method
  - Island of Isolation

Finalize method()

- The finalize() method is invoked each time before the object is garbage collected.
- This method can be used to perform cleanup processing.
- This method is defined in Object class as:

```
protected void finalize()
{     }
```

Or

```
protected void finalize throws Throwable{}
```

Chapter :A Closer Look at Methods and Classes:

Method Overloading

- If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.
- If we have to perform only one operation, having same name of the methods increases the readability of the program.
- Advantage of method overloading: Method overloading *increases the readability of the program*.
- There are two ways to overload the method in java
  - By changing number of arguments
  - By changing the data type

Method Overloading: changing n
arguments

- The below example, two methods are created,first add() method performs addition of two numbers and second add method performs addition of three numbers.

Ex:
```
class Adder{
static int add(int a,int b){return a+b;}
static int add(int a,int b,int c){return a+b+c;}
}
class TestOverloading1{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(11,11,11));
}}
```

Output:

```
22
33
```

Method Overloading: changing c type of arguments

- In this example, we have created two methods that differs in data type.

Example:

```
class Adder{
static int add(int a, int b){return a+b;}
static double add(double a, double b){return a+b;}
}
class TestOverloading2{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(12.3,12.6));
}}
```

Output:

```
22
24.9
```

Constructor Overloading in Java

- 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.

Example of Constructor Overloading

```
//Java program to overload constructors
class Student5{
    int id;
    String name;
    int age;
    //creating two arg constructor
    Student5(int i,String n){
    id = i;
    name = n;
    }
   //creating three arg constructor

    Student5(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[])
 {
   Student5 s1 = new Student5(111,"Karan");
  Student5 s2 = new Student5(222,"Aryan",25);
    s1.display();
    s2.display();
   }
 }
```

Output:

```
111 Karan 0
 222 Aryan 25
```

Using Objects as Parameters

- It is common to pass objects to methods.
- Example:

```
// Objects may be passed to methods.
class Test {
int a, b;
Test(int i, int j) {
a = i;
b = j;
}
// return true if o is equal to the invoking object
boolean equals(Test o) {
if(o.a == a && o.b == b) return true;
else return false;
}
}
```

```
class PassOb {
public static void main(String args[]) {
Test ob1 = new Test(100, 22);
Test ob2 = new Test(100, 22);
Test ob3 = new Test(-1, -1);
System.out.println("ob1 == ob2: " + ob1.equals(ob2));
System.out.println("ob1 == ob3: " + ob1.equals(ob3));
}
}
```

Output:

```
ob1 == ob2: true
ob1 == ob3: false
```

Using Objects as Parameters cntd..

- One of the most common uses of object parameter s involves constructors.

Example:

```
// Here, Box allows one object to initialize another.
class Box {
double width;
double height;
double depth;
// Notice this constructor. It takes an object of type Box.
Box(Box ob) { // pass object to constructor
width = ob.width;
height = ob.height;
depth = ob.depth;
```

- }

Cntd..

```
// constructor used when all dimensions specified
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// constructor used when no dimensions specified
Box() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}
// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
}
```

Cntd..,

```
class OverloadCons2 {
public static void main(String args[]) {
// create boxes using the various constructors
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box();
Box mycube = new Box(7);
Box myclone = new Box(mybox1); // create copy of mybox1
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
// get volume of cube
vol = mycube.volume();
System.out.println("Volume of cube is " + vol);
// get volume of clone
vol = myclone.volume();
System.out.println("Volume of clone is " + vol);
}
}
```

Returning Objects

- Amethod can return any type of data, including class types that you create.
- For example, in the following program, the **incrByTen( )** method returns an object in which the value of **a** is ten greater than it is in the invoking object.

Returning Objects cntd..

```
// Returning an object.
class Test {
int a;
Test(int i) {
a = i;
}
Test incrByTen() {
Test temp = new Test(a+10);
return temp;
}
}
```

Returning Objects cntd..

```
class RetOb {
public static void main(String args[]) {
Test ob1 = new Test(2);
Test ob2;
ob2 = ob1.incrByTen();
System.out.println("ob1.a: " + ob1.a);
System.out.println("ob2.a: " + ob2.a);
ob2 = ob2.incrByTen();
System.out.println("ob2.a after second increase: "
+ ob2.a);
}
}
```

Returning Objects cntd..

Output:

```
ob1.a: 2
ob2.a: 12
ob2.a after second increase: 22
```

Introducing Access Control

- Encapsulation links data with the code that manipulates it.
- Java’s access specifiers are **public, private, and protected**.
- **Acess specifiers**

  1. **public:**
     - A class, method, constructor, interface etc declared public can be accessed from any other class.
     - Therefore fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe.

Introducing Access Contro.

2. Private:

- Methods, Variables and Constructors that are declared private can only be accessed within the declared class itself.
- Class and interfaces cannot be private.
- Variables that are declared private can be accessed outside the class if public getter methods are present in the class.

Introducing Access Control

3. Protect:

- Variables, methods and constructors which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class.
- The protected access modifier cannot be applied to class and interfaces.

4. Default: :

- Default access modifier means we do not explicitly declare an access modifier for a class, field, method, etc.
- A variable or method declared without any access control modifier is available to any other class in the same package

Example for access control

```
class Test
{
int a; // default access
public int b; // public access
private int c; // private access
// methods to access c
void setc(int i)  // set c's value
{
c = i;
}
int getc()  // get c's value
{
return c;
}
}
```

Example for access control cntd…;

```
class AccessTest
{
public static void main(String args[])
{
Test ob = new Test();
// These are OK, a and b may be accessed directly
ob.a = 10;
ob.b = 20;
// This is not OK and will cause an error
// ob.c = 100; // Error!
// You must access c through its methods
ob.setc(100); // OK
System.out.println("a, b, and c: " + ob.a + " " +
ob.b + " " + ob.getc());
}
}
```

Understanding static

- There will be times when you will want to define a class member that will be used independently of any object of that class.
- However, it is possible to create a member that can be used by itself, without reference to a specific instance.
- To create such a member, precede its declaration with the keyword **static.**
- The most common example of a static member is

main( ).

Understanding static cntd.,

- Instance variables declared as **static** are, essentially, global variables.
- When objects of its class are declared, no copy of a **static** variable is made.
- Instead, all instances of the class share the same **static** variable.
- Methods declared as **static** have several restrictions:
  - They can only call other **static** methods.
  - They must only access **static** data.
  - They cannot refer to **this** or **super** in any way.

Example 1 of static

```
// Demonstrate static variables, methods, and blocks.
class UseStatic {
static int a = 3;
static int b;
static void meth(int x) {
System.out.println("x = " + x);
System.out.println("a = " + a);
System.out.println("b = " + b);
}
static {
System.out.println("Static block initialized.");
b = a * 4;
}
public static void main(String args[]) {
meth(42);
}
}
```

Output:

```
Static block initialized.
x = 42
a = 3
b = 12
```

Example 2 of static

//Inside **main( )**, the **static** method **callme( )** and the **static** variable **b** are accessed through their class name **StaticDemo**.

```
class StaticDemo {
static int a = 42;
static int b = 99;
static void callme() {
System.out.println("a = " + a);
}
}
class StaticByName {
public static void main(String args[]) {
StaticDemo.callme();
System.out.println("b = " + StaticDemo.b);
}
}
```

Output:

```
a = 42
b = 99
```

Introducing final

- A variable can be declared as **final**. Doing so prevents its contents from being modified.
- This means that you must initialize a **final** variable when it is declared.
- For example:

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
final int FILE_SAVEAS = 4;
final int FILE_QUIT = 5;
```

Introducing final

- In Java class can be declared as final using final keyword.
- If the final keyword is used in the class declaration the class becomes unable to be sub-classed.
- No any class can extend the final class i.e. features of a final class can't be inherited
- Syntax:

```
public final class FinalClassName
 {
   ……
 }
```

Introducing final cntd..,

- **Example for using final keyword:**

```
public class FinalVariableDemo
 {
 final int number=10;
public void showFinalValue()
     {
      System.out.println("Final vbariable value : "+number);
   }
public static void main(String[] args) {
      FinalVariableDemo obVariableDemo=new FinalVariableDemo();
      obVariableDemo.showFinalValue();
      }
}
```

Output:

```
Final variable value : 10
```