


# 304185

# Unit 2: Class and Objects

## TE E&TE

# Syllabus

Pune Institute of Computer Technology, Pune

- Class Fundamentals, Creating Objects, Accessing Class members, Assigning Object reference variables, Methods, Constructors, using objects as parameters, Argument passing, returning objects, Method Overloading, static members, Nesting of Methods , this keyword, Garbage collection, finalize methods, final variables and methods, final class.

# Java Program

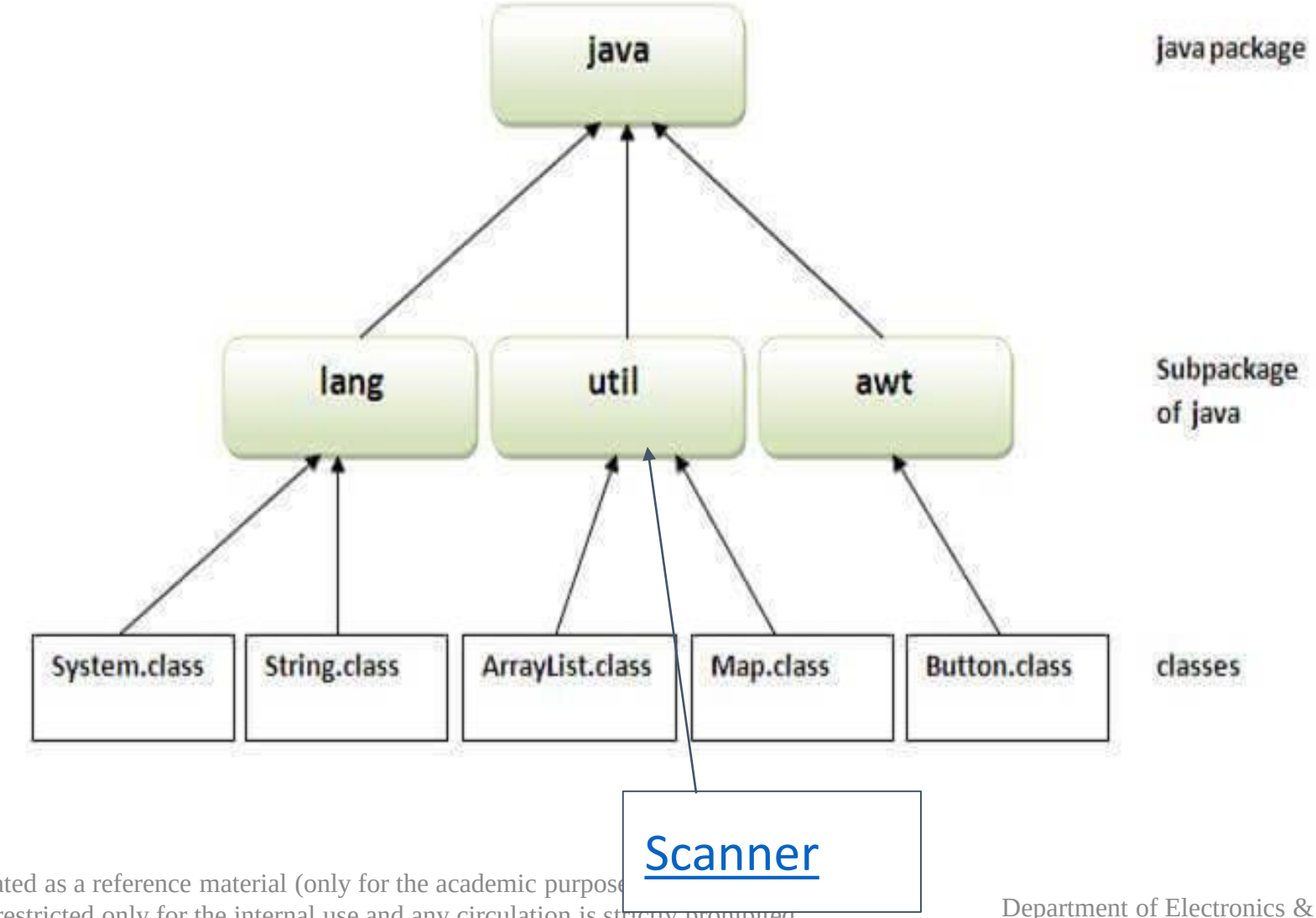


A typical Java file looks like:

**Packages**

```
import java.lang.*;
import java.util.*;

public class SomethingOrOther {
  // object definitions go here
  . . .
}
```



# Class



- The class is at the core of Java.
- It is the logical construct upon which the entire Java language is built because it defines the shape and nature of an object.
- Any concept you wish to implement in a Java program must be encapsulated within a class.
- While defining a class, it is declared using exact form and nature, by specifying the data that it contains and the code that operates on that data.
- While very simple classes may contain only code or only data, most real-world classes contain both.

# Class

- A class consists of
  - a collection of fields, or variables, very much like the named fields of a struct
  - all the operations (called methods) that can be performed on those fields
- Class defines a new data type. Once defined, this new type can be used to create objects of that type.
- Thus, a class is a template for an object, and an object is an instance of a class.
- A class describes objects and operations defined on those objects.

# Example



New data type named Person

*Variable*

*Method*

```
class Person {
    String name;
    int age;

    void birthday ( ) {
        age++;
        System.out.println (name + " is now " + age);
    }
}
```

# Class

- A class is declared by use of the ***class*** keyword.
- The data, or variables, defined within a class are called ***instance variables***.
- The code is contained within ***methods***.
- Collectively, the methods and variables defined within a class are called ***members of the class***.

```
class classname {
type instance-variable1;
type instance-variable2;
// ...
type instance-variableN;
type methodname1(parameter-list) {
// body of method
}
type methodname2(parameter-list) {
// body of method
}
// ...
type methodnameN(parameter-list) {
// body of method
}
}
```

```
class Box {
   double width;
   double height;
   double depth;
}
```

# Class

- Variables defined within a class are called ***instance variables*** because each instance of the class (object) contains its own copy of these variables.
- Thus, the data for one object is separate and unique from the data for another.

```
class Box {
double width;
double height;
double depth;
}
class BoxDemo {
public static void main(String args[]) {
Box mybox = new Box();
double vol;
// assign values to mybox's instance variables
mybox.width = 10;
mybox.height = 20;
mybox.depth = 15;
// compute volume of box
vol = mybox.width * mybox.height * mybox.depth;
System.out.println("Volume is " + vol);
}
}
```

# Declaring object in Java



- When you create a class, you are creating a new data type. You can use this type to declare objects of that type.
- Obtaining objects of a class is a two-step process:
  - First, you must declare a variable of the class type.
    - This variable does not define an object.
    - Instead, it is simply a variable that can refer to an object.
  - Second, you must acquire an actual, physical copy of the object and assign it to that variable.
    - This is achieved using the new operator.

# Declaring object in Java



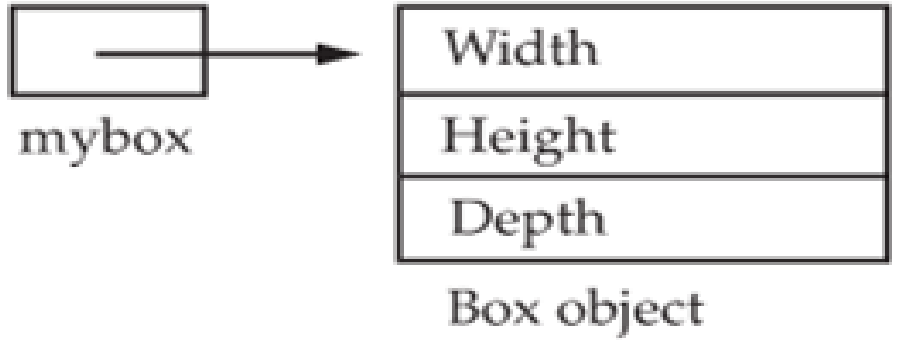
- The ***new*** operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it.
- This reference is, more or less, the address in memory of the object allocated by new.
- This reference is then stored in the variable.

```
Box mybox = new Box();


Box mybox;                  //declare reference to object
mybox = new Box();          //allocate a Box object
```

# Declaring object in Java



| Statement | Effect |
| --- | --- |
| Box mybox; |  mybox |
| mybox = new Box(); |  mybox; Width; Height; Depth; Box object |

# Declaring object

- The class name followed by parentheses specifies the constructor for the class.
- A constructor defines what occurs when an object of a class is created.
- Constructors are an important part of all classes and have many significant attributes.
- However, if no explicit constructor is specified, then Java will automatically supply a default constructor.

# Class and Object

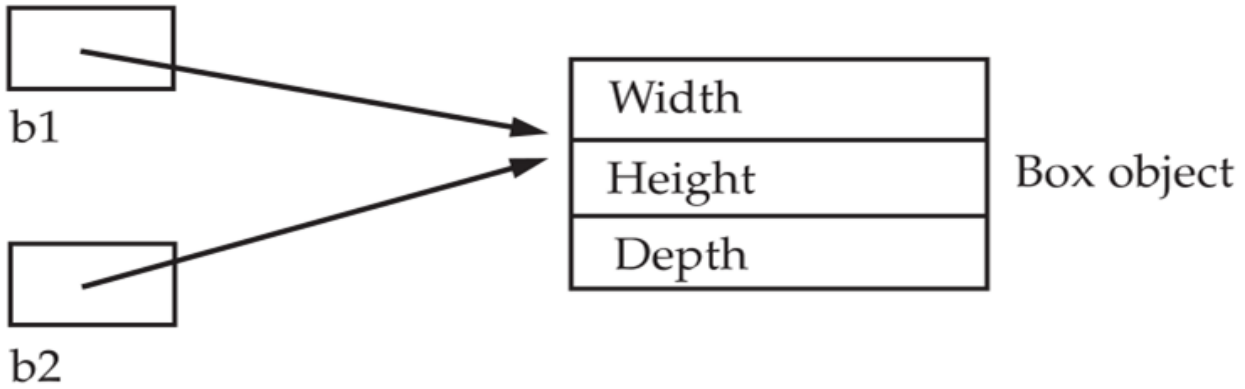
- A class creates a new data type that can be used to create objects.
- That is, a class creates a logical framework that defines the relationship between its members.
- When you declare an object of a class, you are creating an instance of that class.
- Thus, ***a class is a logical construct. An object has physical reality.***

# Object Reference Variable



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, since they are the same object.

# Object Reference Variable



Although b1 and b2 both refer to the same object, they are not linked in any other way.

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

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

**null** is a primitive value that represents the intentional absence of any object value.

REMEMBER :When you assign one object reference variable to another object reference variable, you are not creating a copy of the object, you are only making a copy of the reference.

# Initialization of Object

1. By using Reference variable
2. By using Method
3. By using Constructor

# Using Reference Variable



```
class Box
{
double width =10;
double height=20;
double depth=15;
}
```

```
class BoxDemo
{
public static void main(String
args[]) {
Box mybox = new Box();
double vol;
// compute volume of box
vol = mybox.width * mybox.height
* mybox.depth;
System.out.println("Volume is "
+ vol);
}
}
```

# Using Reference Variable

Pune Institute of Computer Technology, Pune

```
class Box
{
double width;
double height;
double depth;
}
```

```
class BoxDemo
{
public static void main(String
args[]) {
Box mybox = new Box();
double vol;
// assign values to mybox's instance
variables
mybox.width = 10;
mybox.height = 20;
mybox.depth = 15;
// compute volume of box
vol = mybox.width * mybox.height *
mybox.depth;
System.out.println("Volume is " +
vol);
}}
```

# Using Method



```
class Box
{
double width;
double height;
double depth;
// sets dimensions of box
void setDim(double w,
double h, double d)
{
width = w;
height = h;
depth = d;
}}
```

```
class BoxDemo{
public static void main(String
args[]) {
Box mybox = new Box();

double vol;
mybox.setDim(10, 20, 15);
vol = mybox.width * mybox.height
* mybox.depth;
System.out.println("Volume is "
+ vol);
}
}
```

# Methods



Syntax:

```
type name(parameter-list) {
// body of method
}
```

***type*** specifies the type of data returned by the method.

This can be any valid type, including class types that you create.

If the method does not return a value, its return type must be void.

# Methods



- The name of the method is specified by ***name***.
  - This can be any legal identifier other than those already used by other items within the current scope.
- The ***parameter-list*** is a sequence of type and identifier pairs separated by commas.
  - Parameters are essentially variables that receive the value of the arguments passed to the method when it is called.
  - If the method has no parameters, then the parameter list will be empty.
- 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.

# Method Components



**Access Modifier:** Defines the **access type** of the method i.e. from where it can be accessed in your application. In Java, there are 4 types of access specifiers:

- **public:** Accessible in all classes in your application.
- **protected:** Accessible within the package in which it is defined and, in its subclass, **(es) (including subclasses declared outside the package)**.
- **private:** Accessible only within the class in which it is defined.
- **default (declared/defined without using any modifier):** Accessible within the same class and package within which its class is defined.

# Access Modifiers in JAVA



| | default | private | protected | public |
|---|---|---|---|---|
| Same Class | Yes | Yes | Yes | Yes |
| Same package subclass | Yes | No | Yes | Yes |
| Same package non-subclass | Yes | No | Yes | Yes |
| Different package subclass | No | No | Yes | Yes |
| Different package non-subclass | No | No | No | Yes |

# Method Components



- **The return type**: The data type of the value returned by the method or void if it does not return a value.
- **Method Name**: The rules for field names apply to method names as well, but the convention is a little different.
- **Parameter list**: Comma-separated list of the input parameters that are defined, preceded by their data type, within the enclosed parentheses. If there are no parameters, you must use empty parentheses ().
- **Exception list**: The exceptions you expect the method to throw. You can specify these exception(s).
- **Method body**: It is the block of code, enclosed between braces, that you need to execute to perform the required operation.

# Adding Methods to Class



```
// This program includes a method inside 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);
}
}
```

# Adding Methods to Class



```
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();
}
}
```

```
Volume is 3000.0

Volume is 162.0
```

# Adding Methods



- Inside the volume( ) method: the instance variables width, height, and depth are referred to directly, without preceding them with an object name or the dot operator.
- When a method uses an instance variable that is defined by its class, it does so directly, without explicit reference to an object and without use of the dot operator.
- A method is always invoked relative to some object of its class.
- Once this invocation has occurred, the object is known.
- Thus, within a method, there is no need to specify the object a second time.
- This means that width, height, and depth inside volume( ) implicitly refer to the copies of those variables found in the object that invokes volume( ) .

# Returning a Value

```
class Box {
  double width;
  double height;
  double depth;

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

```
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);
}
}
```

```
class Rect {

   int len;

   int bth;

   int area()

    {

       int y;

       y = len * bth;

       return y;

     }

  }
```

```
public class example
{
 public static void main(String[] args)
 {
  Rect r = new Rect();
  r.len = 5;
  r.bth = 2;
  int x;

  x = r.area();
  System.out.println("Area of rectangle
  is: " + x);
 }
}
```

# Method with Parameter



- 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.

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

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

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

```
class Rect {

   int len;

   int bth;

   void setDim(int x, int y){
      len = x;

       bth = y;

   }

   int area() {

       int y;

       y = len * bth;

       return y;

     }

  }
```

```
public class example {

public static void main(String[] args)
   {

       Rect r = new Rect();

       r.setDim(5,2);

       int x;

       x = r.area();

       System.out.println("Areaof
       rectangle is: " + x);

}

}
```

```
// 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;
}
}
```

```
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);
}
}
```

# Example of Initialize Object using 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;
  }
}
```

```
class BoxDemo{
public static void main(String args[])
{
      Box mybox = new Box();

      double vol;
      mybox.setDim(10, 20, 15);
      vol = mybox.width * mybox.height
      * mybox.depth;

      System.out.println("Volume is " +
      vol);
}
}
```

# Constructors



- Initialization of objects can be tiresome.
- Use of functions like setDim( ) might help but it would be simpler and more concise, if all the setup done at the time the object is first created.
- Because the requirement for initialization is so common, Java allows objects to initialize themselves when they are created.
- This automatic initialization is performed using a **constructor.**
- A constructor initializes an object immediately upon creation.

# Constructors



- It has the ***same name as the class*** in which it resides and is syntactically similar to a method.
- ***Once defined***, the constructor is ***automatically called*** when the object is created, before the new operator completes.
- Constructors have ***no return type***, not even void.
- This is because the implicit ***return type of a class' constructor is the class type itself***.
- It is the constructor's job to ***initialize the internal state of an object*** so that the code creating an instance will have a fully initialized, usable object immediately.

```
class Box {
  double width;
  double height;
  double depth;

  // This is the constructor for Box.
  Box() {
    System.out.println("Constructing Box");
    width = 10;
    height = 10;
    depth = 10;
  }

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

```
class BoxDemo6 {
  public static void main(String args[]) {
    // declare, allocate, and initialize Box objects
    Box mybox1 = new Box();
    Box mybox2 = new Box();

    double vol;

    // 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);
  }
}
```

When this program is run, it generates the following results:

```
Constructing Box
Constructing Box
Volume is 1000.0
Volume is 1000.0
```

# Constructors



- Java creates a default constructor for the class, when a constructor is not defined explicitly for a class.

- The default constructor automatically initializes all instance variables to their default values, which are zero, null, and false, for numeric types, reference types, and boolean, respectively.

# Parameterized Constructors



- It may be necessary to initialize the various data elements of different objects with different values when they are created.
- This is achieved by passing arguments to the constructor function when the objects are created.
- The constructors that can take arguments are called parameterized constructors.

```
class Box {
  double width;
  double height;
  double depth;

  // This is the constructor for Box.
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

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

```
class BoxDemo7 {
  public static void main(String args[]) {
    // declare, allocate, and initialize Box objects
    Box mybox1 = new Box(10, 20, 15);
    Box mybox2 = new Box(3, 6, 9);

    double vol;

    // 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);
  }
}
```

The output from this program is shown here:

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

# Parameterized Constructors

```
class Example{
    //Default constructor
    Example(){
        System.out.println("Default constructor");
    }
    /* Parameterized constructor with
     * two integer arguments
     */
    Example(int i, int j){
        System.out.print("parameterized constructor");
        System.out.println(" with Two parameters");
    }
    /* Parameterized constructor with
     * three integer arguments
     */
    Example(int i, int j, int k){
        System.out.print("parameterized constructor");
        System.out.println(" with Three parameters");
    }
```

# Method Overloading



```
class room{
  float l,b;
  room(float x, float y)     // constuctor 1
  {(l=x;
    b=y;)}
  room(float x)                    // constructor 2
  {l=b=x;}
  int area()
  {return(l*b);}}

 room obj1=new room(10.5,5.5);  // constructor 1 gets invoked
 room obj2= new room(5.5);    // constructor 2 gets invoked
```

# Static Members



- The static keyword in Java is mainly used for memory management.
- The static keyword in Java is used to share the same variable or method of a given class. The users can apply static keywords with variables, methods, blocks, and nested classes.
- The static keyword belongs to the class than an instance of the class.
- The static keyword is a non-access modifier in Java that is applicable for the following:
  - Blocks
  - Variables
  - Methods
  - Classes

# Static Members



- Syntax:

**static** data type variable_name;

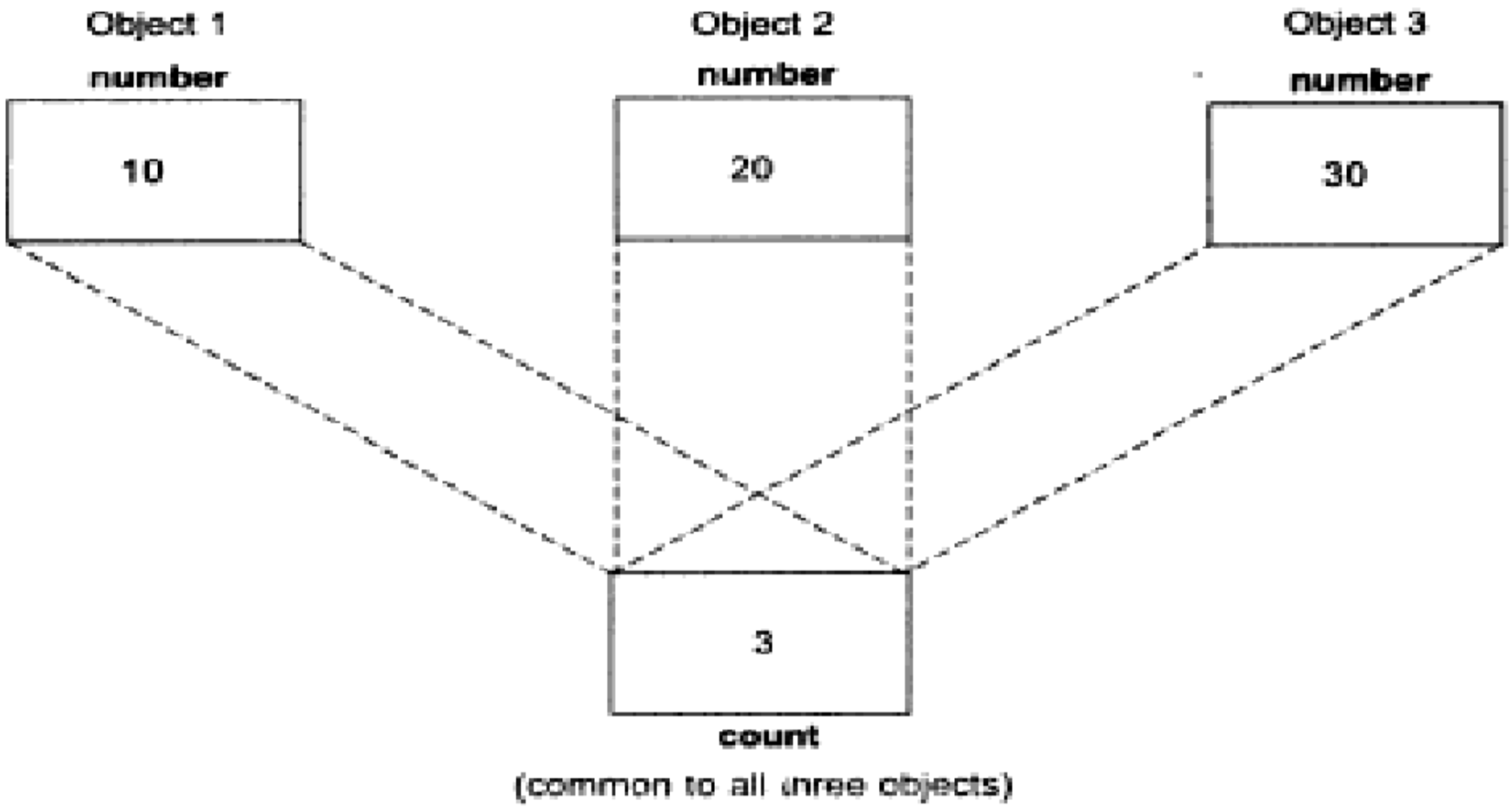
- Ex: static int count;

  static float max(int x, int y);

- Java creates only one copy for a static variable
- Purpose:
  - ✓ **To keep count of how many objects of class have been created**

# Sharing of static data member





# Static Members

Pune Institute of Computer Technology, Pune

When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object.

For example, in the java program, we are accessing static method m1() without creating any object of the Test class.

```
// Java program to demonstrate that a static member
// can be accessed before instantiating a class

class Test
{
        // static method
        static void m1()
        {
                System.out.println("from m1");
        }

        public static void main(String[] args)
        {
                // calling m1 without creating
                // any object of class Test
                m1();
        }
}
```

```
Class MathOperation
{
   static float mul(float x,float y)
  {
          return x*y;
   }
  static float divide(float x,float y)
 {
          return x/y;
  }
}
```

```
Class MathApplication
{
 public void static main(String args[])
  {
          float a,b;
          a=MathOperation.mul(4.0,5.0);
          b=MathOepration.divide(a,2.0);
          System.out.println(“ b=”+b);
   }
}
```

# Limitations

Pune Institute of Computer Technology, Pune

Static Methods have several restrictions :-

- They can only call other static methods.
- They can only access static data.
- They can not refer to this or super in any way.

# Exercise 1

Pune Institute of Computer Technology, Pune

```
// 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);
  }
}
```

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

# Exercise 2

```
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);
  }
}
```

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

# *this* keyword



- The **this** keyword refers to the current object in a method or constructor.
- The most common use of this keyword is to eliminate the confusion between class attributes and parameters with the same name (because a class attribute is shadowed by a method or constructor parameter).
- this can also be used to:
  - Invoke current class constructor
  - Invoke current class method
  - Return the current class object

# Instance Variable Hiding



- It is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes.
- Interestingly, you can have local variables, including formal parameters to methods, which overlap with the names of the class' instance variables.
- However, when a local variable has the same name as an instance variable, the local variable hides the instance variable.

# this: To refer current class instance variable



```
class Student
 {
    int rollno;
    String name;
    float fee;
   Student(int rollno,String name,float fee)
   {
    rollno=rollno;
    name=name;
    fee=fee;
     }
   public void display()
   {
              System.out.println(rollno+" "+name+" "+fee);
       }
}
```

```
class Test
 {
    public static void main(String args[])
   {
            Student s1=new Student(111,"ankit",5000f);
            Student s2=new Student(112,"sumit",6000f);

     s1.display();
     s2.display();
    }
}
```

# this: To refer current class instance variable



```
class Student
{
   int rollno;
   String name;
   float fee;

   Student(int rollno,String name,float fee)
   {
      this.rollno=rollno;
      this.name=name;
      this.fee=fee;
   }

   public void display()
   {
      System.out.println(rollno+" "+name+" "+fee);
   }
}
```

```
class Test
 {
    public static void main(String args[])
    {
       Student s1=new Student(111,"ankit",5000f);
       Student s2=new Student(112,"sumit",6000f);
      s1.display();
      s2.display();
       }
}
```

# this: To invoke current class method

Pune Institute of Computer Technology, Pune

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

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

# this() : To invoke current class constructor



```
class A
{
   A()
    {
      System.out.println("hello a");
     }

   A(int x)
   {
     this();
     System.out.println(x);
     }
}
```

```
class Test
{
     public static void main(String args[])
   {
        A a=new A(10);
     }
}
```

# Calling parameterized constructor from default constructor

Pune Institute of Computer Technology, Pune

```
class A{
        A()
        {
           this(5);
           System.out.println("hello a");
        }

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

# Question

Pune Institute of Computer Technology, Pune

```
class MyClass{
	MyClass(){
		System.out.print("one");
	}
	public void myMethod(){
		this();
		System.out.print("two");
	}
}

public class TestClass{
	public static void main(String args[]){
		MyClass obj = new MyClass();
		obj.myMethod();
	}
}
```

# *final* keyword



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

# JAVA final variable



If you make any variable as final, you cannot change the value of final variable.

(It will be constant).

```
class Bike
  {
     final int speedlimit=90;  //final variable
     void run()
     {
      speedlimit=400;
      }
    public static void main(String args[])
    {
        Bike obj=new  Bike();
         obj.run();
     }
 }
```

**Output: Compile time error**

# JAVA final method



If you make any method as final, you cannot override it.

```
class Bike
{
  final void run()
  {
    System.out.println("Bike");
  }
}
```

```
class Honda extends Bike
 {
      void run()
       {
         System.out.println("Honda");
       }

    public static void main(String args[])
     {
       Honda honda= new Honda();
       honda.run();
      }
}
```

# Can final method be inherited?

Pune Institute of Computer Technology, Pune

Yes, final method is inherited but you cannot override it.

```
class Bike
 {
  final void run()
    {
    System.out.println("running...");
     }
}
```

```
class Honda extends Bike
  {
     public static void main(String args[])
     {
        Honda H= new Honda()
                H.run();
     }
}
```

# final parameter



If you declare any parameter as final, you cannot change the value of it.

```
class Bike
  {
    int cube(final int n)
     {
       n=n+2;//can't be changed as n is final
       n*n*n;
     }

   public static void main(String args[])
     {
      Bike b=new Bike();
      b.cube(5);
    }
  }
```

# Question

```
class A{
        public static void method(int i){
                System.out.print("Method 1");
        }

        public static int method(String str){
                System.out.print("Method 2");
                return 0;
        }
}

public class Test{

        public static void main(String args[]){
                A.method(5);
        }
}
```

# Question

```
public class Profile {
        private Profile(int w) { // line 1
                System.out.print(w);
        }
        public static Profile() { // line 5
                System.out.print (10);
        }
        public static void main(String args[]) {
                Profile obj = new Profile(50);
        }
}
```

# Question

```
public class Test {
        Test() {    }  // line 1
        static void Test() {  this(); } // line 2
        public static void main(String[] args) { // line 3
                Test(); // line 4
        }
}
```

# Question

```
public class Profile {
        private Profile(int w) { // line 1
                System.out.print(w);
        }
        public final Profile() { // line 5
                System.out.print(10);
        }
        public static void main(String args[]) {
                Profile obj = new Profile(50);
        }
}
```

# Garbage Collection (Object Destruction)



- In java, garbage means unreferenced objects.
- When object is created using the dynamic operator new, memory is allocated for it from heap region.
- So, to destroy the object, free() function is used in C language and delete() in C++. But, in java it is performed automatically.
- Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects or unreachable objects.
- Unreachable Objects: Object said to be unreachable if it does not contain any reference to it.

# Garbage Collection in JAVA



- Java garbage collection is an automatic process.
- Automatic garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects.
- An in-use object, or a referenced object, means that some part of your program still maintains a pointer to that object.
- An unused or unreferenced object is no longer referenced by any part of your program. So, the memory used by an unreferenced object can be reclaimed.
- The programmer does not need to mark objects to be deleted explicitly. The garbage collection implementation lives in the JVM.

# Activities in JAVA Garbage Collection



Two types of garbage collection activity usually happen in Java.

- **Minor or incremental Garbage Collection:** It is said to have occurred when unreachable objects in the young generation heap memory are removed.
- **Major or Full Garbage Collection:** It is said to have occurred when the objects that survived the minor garbage collection are copied into the old generation or permanent generation heap memory are removed. When compared to the young generation, garbage collection happens less frequently in the old generation.

# Advantages

- It makes java memory efficient because garbage collector removes the unreferenced objects from heap memory.
- It is automatically done by the garbage collector(a part of JVM) so we don't need to make extra efforts.

# How can an object be unreferenced?



1. By nulling a reference:

Integer i=new Integer(4); //

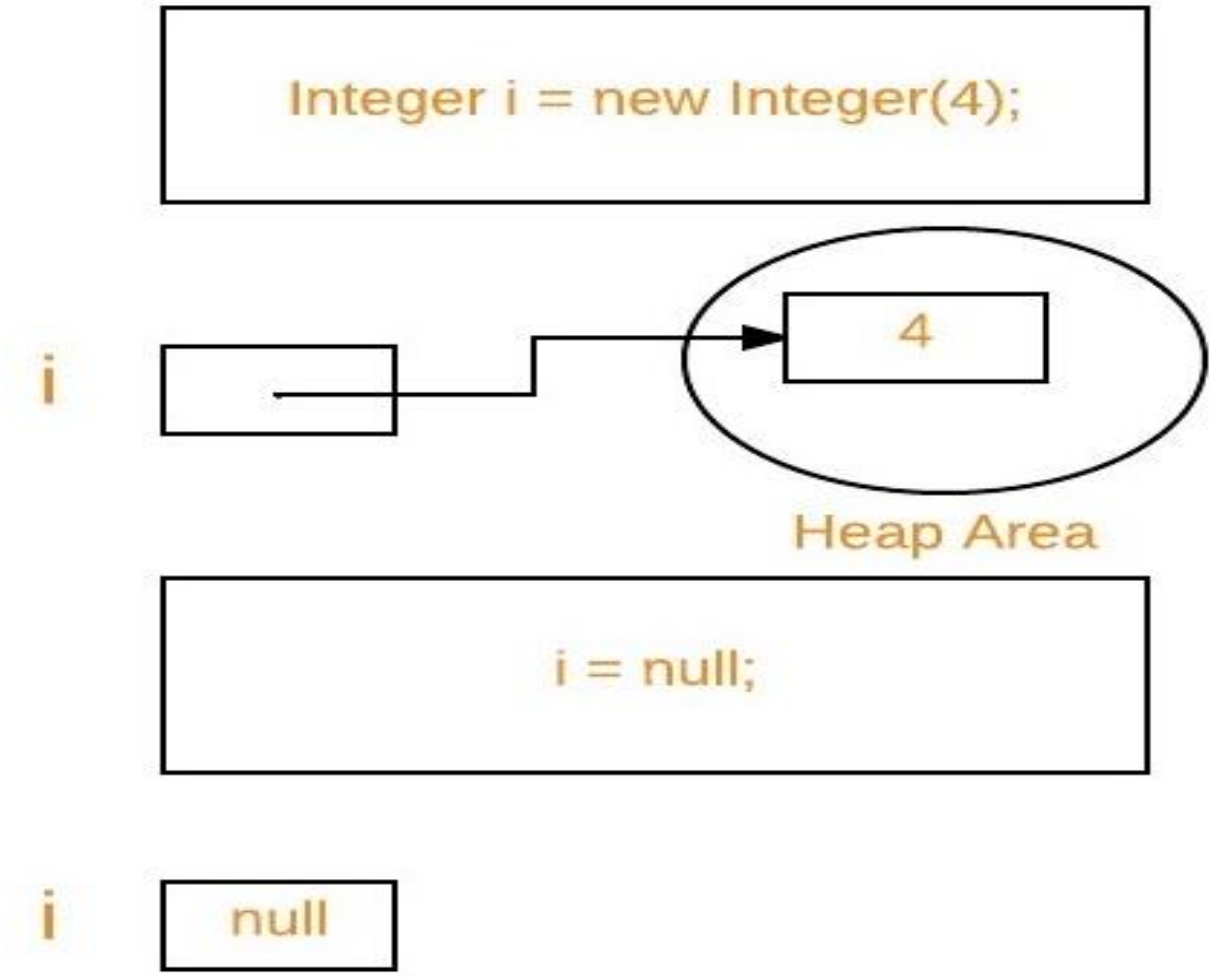
new integer object is

reachable via the reference in

'i'

i=null; // integer object is not

reachable longer



# Eligibility for Garbage Collection



- An object is said to be eligible for GC(garbage collection) if it is unreachable.
- After i = null, integer object 4 in the heap area is suitable for garbage collection in the above image.

# How to make an object eligible for Garbage Collector?



There are generally four ways to make an object eligible for garbage collection.

- Nullifying the reference variable
- Re-assigning the reference variable
- An object created inside the method
- Island of Isolation

# Requesting JVM to run Garbage Collector



- Once an object is made eligible for garbage collection, it may not be destroyed immediately by the garbage collector.
- Whenever JVM runs the Garbage Collector program, then only the object will be destroyed. But when JVM runs Garbage Collector, we can not expect.
- We can also request JVM to run Garbage Collector. There are two ways to do it :
- **Using System.gc() method:** System class contain static method gc() for requesting JVM to run Garbage Collector.
- **Using Runtime.getRuntime().gc() method:** Runtime class allows the application to interface with the JVM in which the application is running. Hence by using its gc() method, we can request JVM to run Garbage Collector.

```
// Java program to demonstrate requesting  JVM to run Garbage Collector


public class Test
{       public static void main(String[] args)
        {               Test t1 = new Test();
                        Test t2 = new Test();
                        t1 = null; // Nullifying the reference variable
                        System.gc();  // requesting JVM for running Garbage Collector
                        t2 = null;  // Nullifying the reference variable
                        Runtime.getRuntime().gc();   // requesting JVM for running Garbage Collector
            }
}
```

# *finalize* method



- In many Java programs memory is not the only resource that is used by object
- Sometimes other resources like files, network connections, database connections are also used.
- When we no longer need the objects that use these resources then the object should release theses resources in disciplined manner.
- If this is not done, then resource leaks will happen and hence waste of resources encounters.

# finalize method



- To avoid this, Java provides a mechanism called finalize() method to give up the resources when object is no longer needed.
- The finalize() method is invoked each time by garbage collector just before reclaiming the object space.
- Thus, the finalize() method is opposite of a constructor.
- So, finalize() method performs operations like closing open files, terminating network connection, terminating database connection and other cleanup work.
- The finalize() method is invoked each time before the object is garbage collected.
- The gc() method is used to invoke the garbage collector to perform cleanup processing.

```
public class TestGarbage
 {
    public void finalize()
          {
               System.out.println("object is garbage collected");
          }
   public static void main(String args[])
      {
           TestGarbage s1=new TestGarbage();
           TestGarbage s2=new TestGarbage();
           s1=null;
           s2=null;
           System.gc();
       }
 }
```

# Recursion in JAVA



- Recursion in java is a process in which a method calls itself continuously.
- A method in java that calls itself is called recursive method.

```
returntype methodname()
{
    //code to be executed
   methodname(); //calling same method
}
```

# Recursion Example

Pune Institute of Computer Technology, Pune

```
public class RecursionExample

{
  static int count=0;

    static void p()
      {
            count++;
            if(count<=5){
              System.out.println("hello "+count);
              p();
           }
       }
  public static void main(String[] args)
     {
            p();
      }
}
```

# Command Line Arguments



- Java command-line argument is an argument passed at the time of running the Java program.
- In the command line, the arguments passed from the console can be received in the java program and they can be used as input.
- The users can pass the arguments during the execution bypassing the command-line arguments inside the main() method.
- We need to pass the arguments as space-separated values.
- We can pass both strings and primitive data types (int, double, float, char, etc) as command-line arguments.
- These arguments convert into a string array and are provided to the main() function as a string array argument.

# Command Line Arguments



- When command-line arguments are supplied to JVM, JVM wraps these and supplies them to args[].
- It can be confirmed that they are wrapped up in an args array by checking the length of args using args.length.
- Internally, JVM wraps up these command-line arguments into the args[ ] array that we pass into the main() function.
- JVM stores the first command-line argument at args[0], the second at args[1], the third at args[2], and so on.

# Parsing



- Used to convert command line arguments which are in string format into a number (0,1,2,3,…N).

- PARSE : It is a method which take a string(input) as an argument and convert in other formats as like:
    1. Integer
    2. Float
    3. Double

- The Types of Parse Methods :
    1. parseInt();
    2. parseDouble();
    3. parseFloat();

# parseInt()



- While operating upon strings, there are times when we need to convert a number represented as a string into an integer type.
- The method generally used to convert String to Integer in Java is *parseInt()*.
- This method belongs to Integer class in java.lang package.
- It takes a valid string as a parameter and parses it into primitive data type int.
- It only accepts String as a parameter and on passing values of any other data type, it produces an error due to incompatible types.

- *public static int parseInt(String s)*
- *public static int parseInt(String s, int radix)*

# parseFloat()



- The parseFloat() method in Float Class is a built in method in Java that returns a new float initialized to the value represented by the specified String, as done by the valueOf method of class Float.

- *public static float parseFloat(String s)*

- Parameters: It accepts a single mandatory parameter s which specifies the string to be parsed.

- Return type: It returns float value represented by the string argument.

- Exception: The function throws two exceptions which are described below:
  - NullPointerException– when the string parsed is null
  - NumberFormatException– when the string parsed does not contain a parsable float

# parseDouble()



- The parseDouble() method of Java Double class is a built in method in Java that returns a new double initialized to the value represented by the specified String, as done by the valueOf method of class Double.
- *public static double parseDouble(String s)*
- Parameters: It accepts a single mandatory parameter s which specifies the string to be parsed.
- Return type: It returns double value represented by the string argument.
- Exception: The function throws two exceptions which are described below:
  - NullPointerException– when the string parsed is null
  - NumberFormatException– when the string parsed does not contain a parsable float

## Without Parse Method

```
public class WithoutParseMethod {

public static void main(String[] args)

{
 System.out.println(args[0]+1);
 System.out.println(args[1]+1);

}

}
```

## With Parse Method

```
public class WithParseMethod {

public static void main(String[] args)
{
   int a= Integer.parseInt(args[0]);
   float b=Float.parseFloat(args[1]);
   double c=Double.parseDouble(args[2]);
   String s= args[3];

System.out.println(a+1);
 System.out.println(b+1);
 System.out.println(c+1);
 System.out.println(s);
}
}
```

# Different Ways of Passing Object as Parameter



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

    Rectangle(int l, int b) {
        length = l;
        width = b;
    }

    void area(Rectangle r1) {
        int areaOfRectangle = r1.length * r1.width;
        System.out.println("Area of Rectangle : "
                                    + areaOfRectangle);
    }
}

class RectangleDemo {
    public static void main(String args[]) {
        Rectangle r1 = new Rectangle(10, 20);
        r1.area(r1);
    }
}
```

# By passing Instance Variables one by one

Pune Institute of Computer Technology, Pune

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

    void area(int length, int width) {
        int areaOfRectangle = length * width;
        System.out.println("Area of Rectangle : "
                + areaOfRectangle);
    }
}

class RectangleDemo {
    public static void main(String args[]) {
        Rectangle r1 = new Rectangle();
        Rectangle r2 = new Rectangle();

        r1.length = 20;
        r1.width = 10;

        r2.area(r1.length, r1.width);
    }
}
```

# Returning the Object from Method



```
class Rectangle {
      int length;
      int breadth;

      Rectangle(int l,int b) {
        length = l;
        breadth = b;
      }

      Rectangle getRectangleObject() {
        Rectangle rect = new Rectangle(10,20);
        return rect;
      }
}
```

```
class RetOb {
      public static void main(String args[]) {
        Rectangle ob1 = new Rectangle(40,50);
        Rectangle ob2;

        ob2 = ob1.getRectangleObject();
        System.out.println("ob1.length : " + ob1.length);
        System.out.println("ob1.breadth: " + ob1.breadth);

        System.out.println("ob2.length : " + ob2.length);
        System.out.println("ob2.breadth: " + ob2.breadth);

      }
}
```

# Final class



- When a class is declared with final keyword, it is called a final class. A final class cannot be extended (inherited).
- There are two uses of a final class:
  - One is to prevent inheritance, as final classes cannot be extended.
  - The other use of final with classes is to create an immutable class like the predefined String class.

# Nesting of Methods



```
class nesting{
  int m,n;
  nesting(int x, int y){
     m=x;
     n=y;}
   int largest(){
        if (m>=n)
             return(m);
               else
             return(n);  }
 void disp(){
  int large = largest();      // calling a method
    System.out.println(large);
}}
```

Method can be called using only its name by another method of same class.

```
class nesting_test
{
  public static void main(String args[]){
          nesting obj= new nesting(10,20)
                  obj.display();}
     }
```

# Important Points



1) Can we give the access modifier to the constructor?
Ans: Yes, default, protected, public, private
But private constructor can not be invoked by the object creation statement.

2) Can we make constructor as a static and final?
Ans: No, can't make constructor as a static and final.

3) Can we access the static fields and methods using this keyword?
Ans:- Yes, but same fields are common for all objects.

4) Can static methods can access non static members of class?
Ans:- No, static methods access only the static members of class.

5) Can final methods can access static members of class?
Ans:- Yes, final methods can access static fields as well as static methods of same class.

# Important Points



6) Can we override the final method?
Ans: No

7) Can we overload the final method?
Ans: Yes, we can overload the final method.

8) Can we overload the static method?
Ans: Yes, we can overload the static method.

9) Can we override the static methods?
Ans: Yes, we can override the static methods.

10) Can we inherit the final class?
Ans:- No, we can not inherit the final class.

# References


- E Balagurusamy, “Programming with JAVA”, Tata McGraw Hill, 6th Edition.
- Herbert Schildt, “Java: The complete reference”, Tata McGraw Hill, 7th Edition.
- https://www.tutorialspoint.com
- https://www.geeksforgeeks.org