Unit-3

**Class and Object**

**Class:** Class is a fundamental concept of object-oriented programming (OOP). A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.

A class in Java can contain:

* **Declarations**
  + A class is declared using the class keyword followed by the class name. For example:  
    public class Student  
      {  
        //Data Member  
        //Member Function  
      }

* **Fields**
  + Fields can be of any data type, including primitive types, reference types, or other classes. Example. int a,b;
* **Methods**
  + Classes contain methods, which represent the behavior of the class.
  + Example:   
    public void Add(int x, int b)

    {

    }

* **Constructors**
  + Constructors are special methods used for initializing objects.

  public class Student

  {

    public Student()

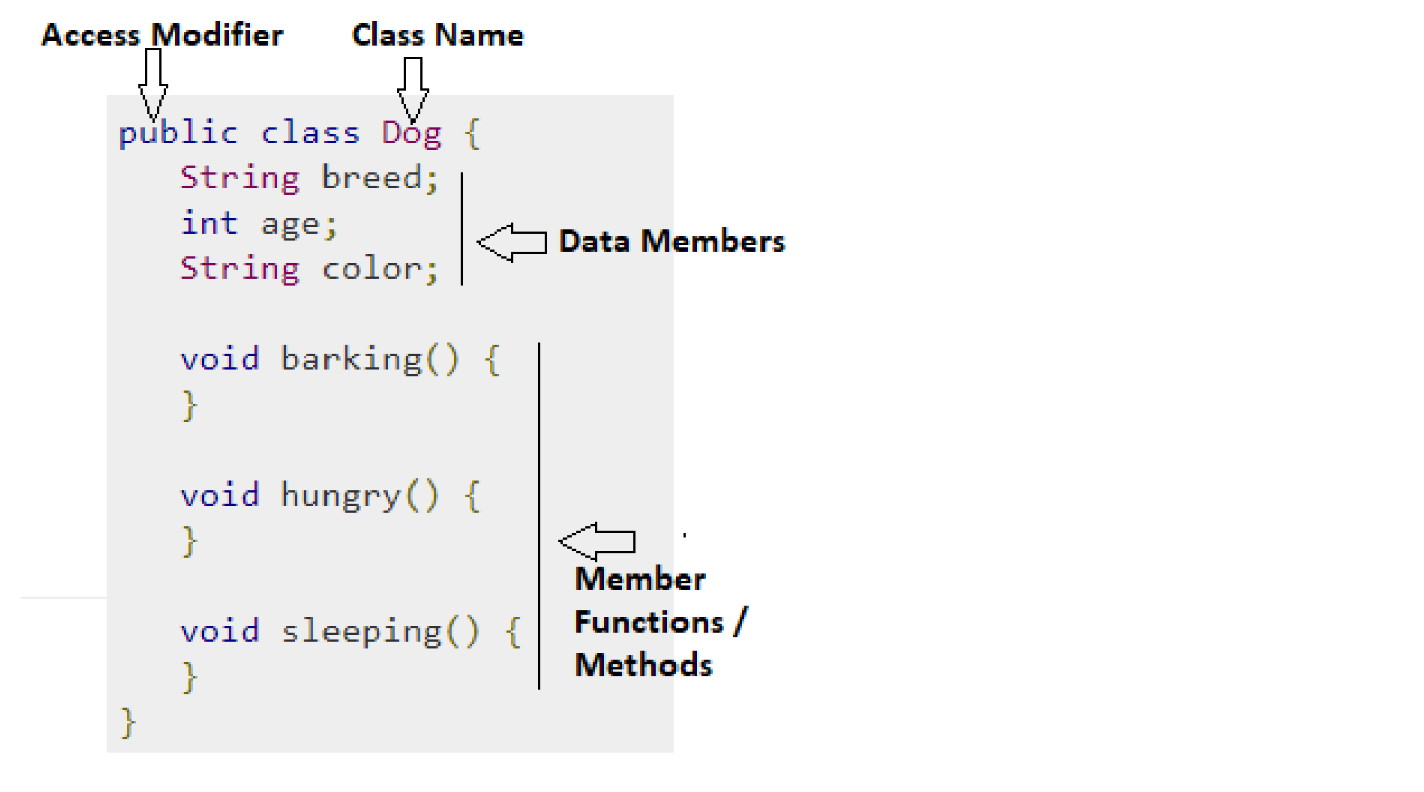
    {

    }

  }

* **Access Modifier**
  + Classes, fields, and methods can have access modifiers such as **public**, **private**, **protected**, or **package-private (default).**
* **Class Body:** class body is surrounded by {}

General Form of Class:



**Object**: An entity that has **state** and **behavior** is known as an **object**.

An object has three characteristics:

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

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

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.

**Syntax:**

Student st=new Student();

Below Example shows implementation of Class and Object

public class App {

    public static void main(String[] args) throws Exception {

      Student st=new Student();

      st.DisplayName();

     }

  }

  class Student

  {

   public void DisplayName()

   {

    System.out.println("Sunil Chaudhary");

   }

  }

|  |  |
| --- | --- |
| **Abstraction** | **Encapsulation** |
| Abstraction is a general concept formed by extracting common features from specific example or The act of withdrawing or removing something **unnecessary** | Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both **safe from outside interference** and **misuse** |
| You can use abstraction using **Interface** and **Abstract** class | You can implement encapsulation using **Access Modifiers**(public, protected and private) |
| Abstraction solves the problem in **Design** level | Encapsulation solves the problem in **Implementation** level |
| Hiding implementation using abstract class and interface | Encapsulation hiding data using getters and setters |

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

Abstraction means to show **What** part of functionality.

**For example,** if you have a class representing a car, the user of that class might only need to know how to **start** the car, **stop** the car, and perhaps how to **accelerate** and **brake**. They don't need to know the details of how the engine works or how the transmission shifts gears.

**This is typically achieved using abstract classes and interfaces.**

**Using Abstract Class:**

public class App {

    public static void main(String[] args) throws Exception {

        Circle c = new Circle();

        c.draw(); // Output: Drawing Circle

     }

  }

abstract class Shape {

    abstract void draw(); // Abstract method

}

class Circle extends Shape {

    void draw() {

        System.out.println("Drawing Circle");

    }

}

class Rectangle extends Shape {

    void draw() {

        System.out.println("Drawing Rectangle");

    }

}

**Using Interface**

public class App {

    public static void main(String[] args) throws Exception {

        Circle circle = new Circle();

        circle.draw(); // Output: Drawing Circle

     }

  }

interface Drawable {

    void draw();

}

class Circle implements Drawable {

    public void draw() {

        System.out.println("Drawing Circle");

    }

}

class Rectangle implements Drawable {

    public void draw() {

        System.out.println("Drawing Rectangle");

    }

}

**Encapsulation:** **Encapsulation** is one of the fundamental concepts of Object Oriented Programming (OOP) paradigm. It is the process of wrapping the data stored in the member variables of a class with its member functions.

It is done in such a way that the data is hidden to everything outside the class scope, and can only be accessed and modified through its own member functions.

**How to achieve Encapsulation:**

* Declaring the class variables as **private** so that they are inaccessible from outside the scope of the class.
* Designing **getter** and **setter** methods for the class and using them accordingly.

**Why do we need Encapsulation:**

* It helps you in achieving loose coupling.
* Encapsulation makes the application simple and easy to debug.
* Allows the programmer to control the data accessibility of a class.

**Advantages of Encapsulation:**

* Cleaner, more organized and less complex code.
* More flexible code as can modify a unit independently without changing any other unit.
* Makes the code more secure.
* The code can be maintained at any point without breaking the classes that use the code.

LAB: Write a java program to achieve encapsulation using private access modifier.

**Example: using private access modifier**

public class App {

    private int length;

    private int breadth;

    public App(int l, int b)

    {

        this.length=l;

        this.breadth=b;

    }

    public void Area()

    {

        System.out.println(length\*breadth);

    }

    public static void main(String[] args) throws Exception {

        App ap=new App(2,3);

        ap.Area();

    }

}

Output:6

LAB: Write a java program to achieve encapsulation using getter and setter.

**Example:** using **getter** and **setter**

public class App {

    private String author;

    private String title;

    public String getAuthor() {

        return author;

      }

      public void setAuthor(String a) {

        this.author = a;

      }

      public String getTitle() {

        return title;

      }

      public void setTitle(String t) {

        this.title = t;

      }

    public static void main(String[] args) throws Exception {

       App a=new App();

       a.setAuthor("Sunil Chaudhary");

       a.setTitle("MR.");

       System.out.println(a.getTitle()+" "+a.getAuthor());

    }

}

Example:

public class App {

    public static void main(String[] args) throws Exception {

      Rectangle rect=new Rectangle();

      rect.setLength(5);

        rect.setBreadth(5);

        System.out.println(rect.getLength()\*rect.getBreadth());

     }

  }

  class Rectangle

  {

    private int length;

    private int breadth;

    public int getLength() {

      return length;

    }

    public void setLength(int length) {

      this.length = length;

    }

    public int getBreadth() {

      return breadth;

    }

    public void setBreadth(int breadth) {

      this.breadth = breadth;

    }

  }

**Output: MR. Sunil Chaudhary**

**Constructor:**

A constructor is a block of codes similar to the method. It is called when an instance of the [class](https://www.javatpoint.com/object-and-class-in-java) is created.   
Constructor name must be the same as its class name  
A Constructor must have no explicit return type

There are three types of constructor in java.

* Default Constructor
* No-Args constructor
* Parameterized constructor

**Default Constructor:**

If we do not create any constructor, the Java compiler automatically creates a no-arg constructor during the execution of the program.

This constructor is called the default constructor.

public class App {

    int a;

    boolean b;

    public static void main(String[] args) throws Exception {

        App ap=new App();

        System.out.println(ap.a);

        System.out.println(ap.b);

     }

  }

**No-Args Constructor:**

constructor may or may not have any parameters (arguments).

If a constructor does not accept any parameters, it is known as a no-argument

public class App {

    public static void main(String[] args) throws Exception {

        Rectangle rect=new Rectangle();

        rect.Add();

     }

  }

  class Rectangle

  {

    int a=0;

    int b=0;

    public Rectangle()

    {

        a=5;

        b=6;

    }

    public void Add()

    {

        System.out.println(a+b);

    }

  }

**Parameterized Constructor:**

A Java constructor can also accept one or more parameters. Such constructors are known as parameterized constructors.

public class App {

    public static void main(String[] args) throws Exception {

        Rectangle rect=new Rectangle(5,6);

        rect.Add();

     }

  }

  class Rectangle

  {

    int first=0;

    int second=0;

    public Rectangle(int x, int y)

    {

        first=x;

        second=y;

    }

    public void Add()

    {

        System.out.println(first+second);

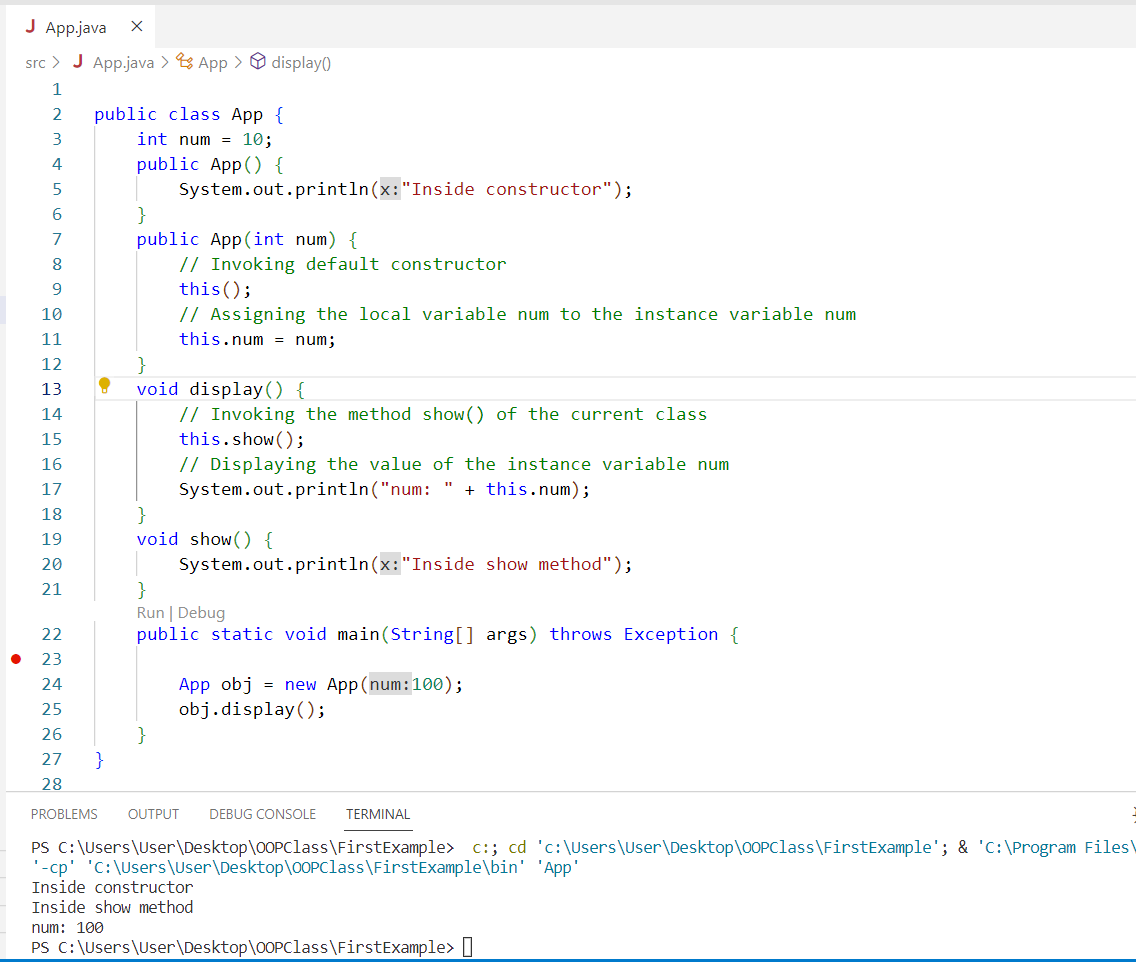
    }

  }

**“this” keyword**

* It can be used to call current class methods and fields, to pass an instance of the current class as a parameter,
* To differentiate between the local variable (variable that is declared inside the body of a method) and instance variables (variable is defined without the STATIC keyword, but as outside of a method declaration).
* To Invoke Default Constructor
* Using “this” reference can improve code readability and reduce naming conflicts.
* To Invoking **method** of **Current Class**

**Lab.Write a java program to demonstrate “this” keyword**

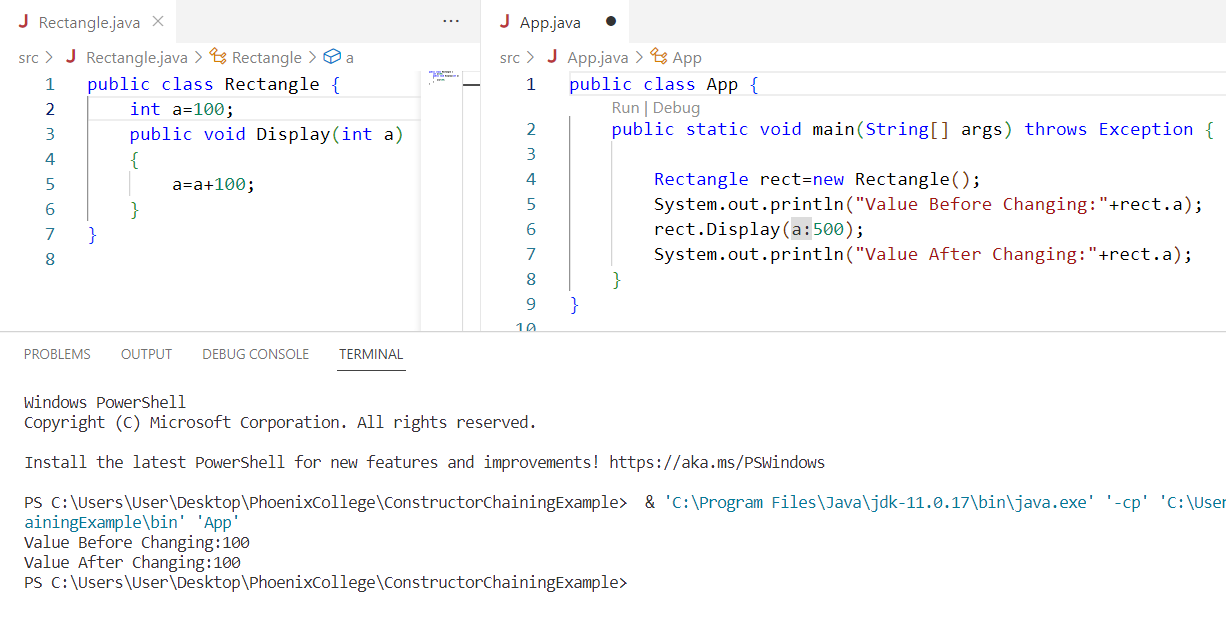
****

## Method (Pass by Value and Pass By Reference)

## When a primitive type is passed to a method ,it is done by use of call-by-value . Objects are implicitly passed by use of call-by-reference

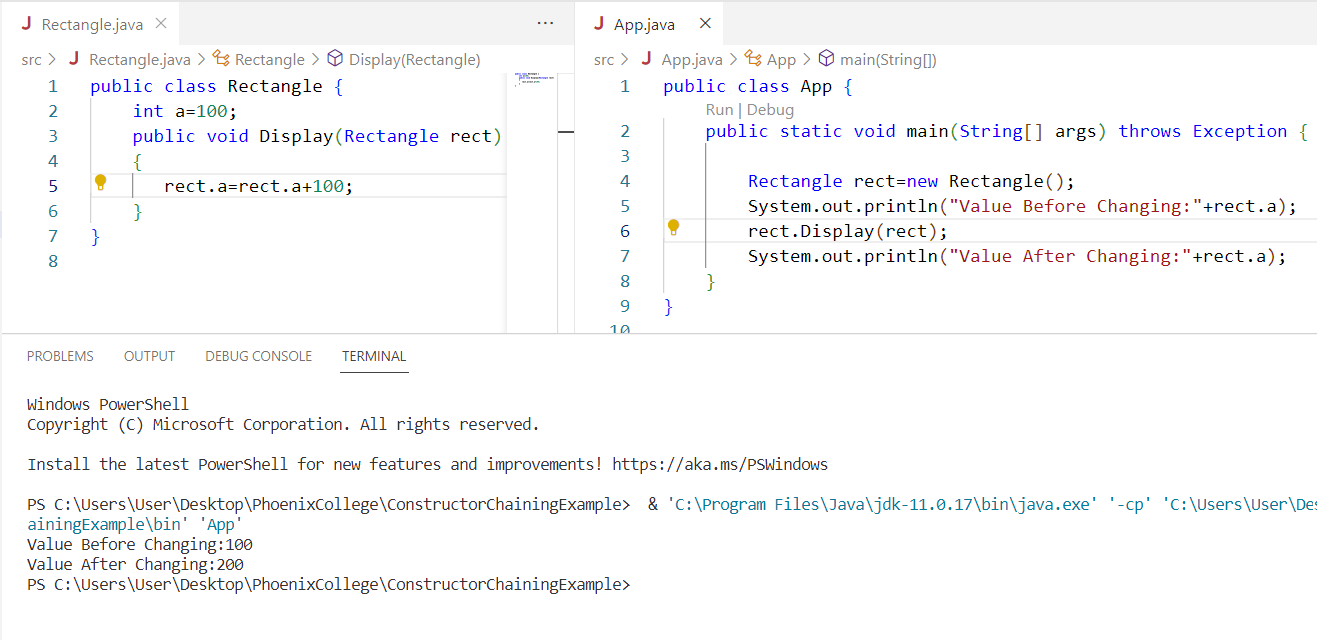
## Pass by Value in Java passing the parameters by values does not affect the original variable.

## Here we have initialized a variable ‘a’ with some value and used the pass-by-value technique to show how the value of the variable remains unchanged.

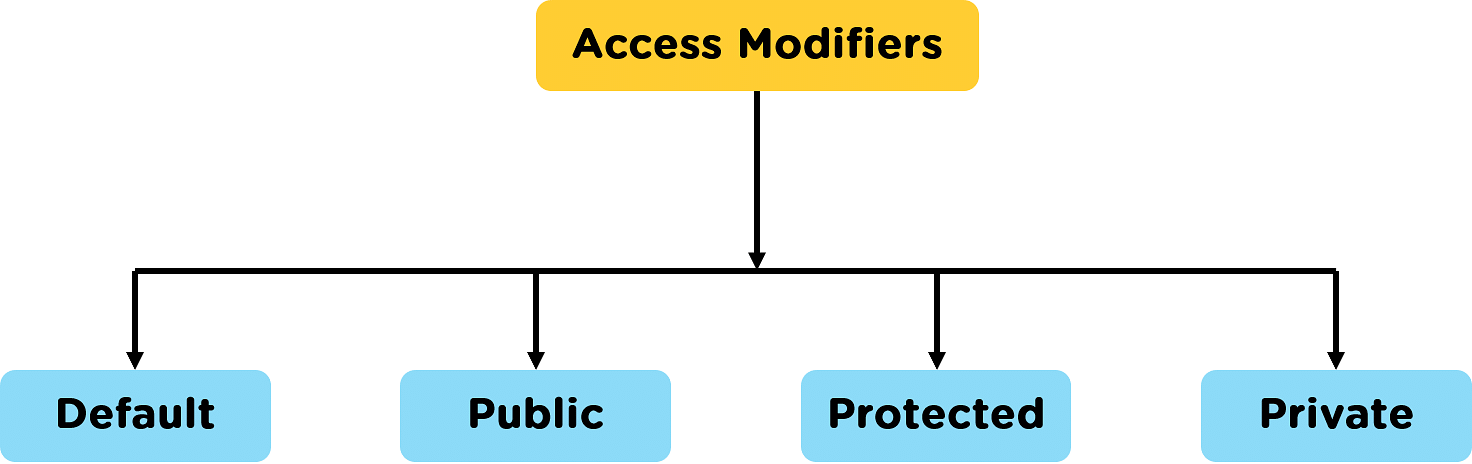
****

## Pass by Reference in Java

As you can see, when we have passed the object reference as a value instead of a value, the original value of the variable ‘a’ is changed to 200. This is because of the changes in the called

method.

## Access Modifiers Access modifiers specify the availability of a parent class. In real-time coding, we cannot allow child classes to have access to all of the other classes. The availability of a data member, method, or constructor can be described in four ways.



|  |  |
| --- | --- |
| Modifier | Description |
| Default | declarations are visible only within the package (package private) |
| Private | declarations are visible within the class only(data member,Function Member) |
| Protected | declarations are visible within the package or all subclasses  (data member,Function Member) |
| Public | declarations are visible everywhere |

**Default Access Modifier:**

package defaultPackage;

class Logger {

void message(){

System.out.println("This is a message");

}

}

Here, the **Logger** class has the **default** access modifier. And the class is visible to all the classes that belong to the **defaultPackage** package. However, if we try to use the **Logger** class in another class outside of **defaultPackage**, we will get a compilation error.

**Private Access Modifier**  
When **variables** and **methods** are declared **private**, they cannot be accessed outside of the class. For example,

class Data {

// private variable

private String name;

}

public class Main {

public static void main(String[] main){

// create an object of Data

Data d = new Data();

// access private variable and field from another class

d.name = "Programiz";

}

}

In the above example, we have declared a **private** variable named name. When we run the program, we will get the following error:  
The error is generated because we are trying to access the **private** variable of the Data class from the **Main class**. In this case, we can use the **getters** and **setters** method. For example,

class Data {

private String name;

// getter method

public String getName() {

return this.name;

}

// setter method

public void setName(String name) {

this.name= name;

}

}

public class Main {

public static void main(String[] main){

Data d = new Data();

// access the private variable using the getter and setter

d.setName("Programiz");

System.out.println(d.getName());

}

}

**Protected Access Modifier**  
When **methods** and **data** members are declared **protected**, we can access them within the same **package** as well as from **subclasses**. For example,

class Animal {

// protected method

protected void display() {

System.out.println("I am an animal");

}

}

class Dog extends Animal {

public static void main(String[] args) {

// create an object of Dog class

Dog dog = new Dog();

// access protected method

dog.display();

}

}

**Public Access Modifier**  
When **methods**, **variables**, **classes**, and so on are declared **public**, then we can access them from anywhere. The public access modifier has no scope **restriction**. For example,

// Animal.java file

// public class

public class Animal {

// public variable

public int legCount;

// public method

public void display() {

System.out.println("I am an animal.");

System.out.println("I have " + legCount + " legs.");

}

}

// Main.java

public class Main {

public static void main( String[] args ) {

// accessing the public class

Animal animal = new Animal();

// accessing the public variable

animal.legCount = 4;

// accessing the public method

animal.display();

}

}

**Polymorphism:**

**Polymorphism in Java** is a concept by which we can perform a *single action in different ways*(one name multiple forms). Polymorphism is derived from 2 Greek words: **poly** and **morphs**. The word "poly" means **many** and "morphs" means **forms**. So polymorphism means many forms.

There are two types of polymorphism in Java:

* compile-time polymorphism(method overloading)
* runtime polymorphism. (Method Overriding)

**Method Overloading:**

public class App {

    public static void main(String[] args) throws Exception {

      Calculator cal=new Calculator();

      System.out.println(cal.add(2, 3));

      System.out.println(cal.add(4, 5, 6));

     }

  }

  class Calculator {

    // Method to add two integers

    public int add(int a, int b) {

        return a + b;

    }

    // Method to add three integers

    public int add(int a, int b, int c)

    {

        return a + b + c;

    }

}

Method Overriding:

