Object Oriented Programming (OOP) Concepts

Objects, Classes and Encapsulation

JULY,2023

expleo

Contents

- Objects
- Classes
- Encapsulation
- Inheritance
- Abstraction
- Polymorphism

Classes and Objects

Objects: Recap

- **Object** is the primary unit of Object-Oriented Programming.
- It represents the **real-life entities**. **Example**: pen, chair, table, computer, watch, etc.,
- It can be **physical or logical.**
- An **object** has **three characteristics**:
  1. **State:** represents **data or value** stored in an object.
  2. **Behavior:** represents the **behavior or functionality** of an object. This function is used to manipulate the data and interact with other objects
  3. **Identity:** It gives unique name to an object. **Each object** is identified in Java by **unique memory location**.

Objects

State: colour, size, weight, brand

Behaviour: Change channel, Manage Volume

State: Name, colour, Breed, Type

Behaviour: Barking, Fetching, Wagging the tail

State: Name, black hair, black eyes, heighht

Behaviour: eat, study, play, sleep

Class: Recap

- Class is a **template or blueprint** of the objects.
- It defines the **state** (variables) and **behaviour** (methods) **common to all objects** of a certain kind.
- A class is a **logical entity** and describes the object's **properties** and **behaviours.**
- It is used to **create object instances.**

Object and Class-Recap

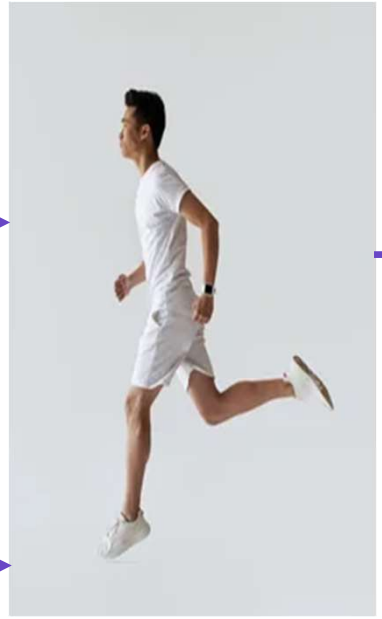
Class – Sportsman

States

- Name
- Sport
- Gender
- Height
- Weight
- rank

Behaviors

- Profiling
- BMI



Objects

Name: Cristiano Ronaldo
Sport: Football
Gender: Male
Height: 1.87m
Weight: 83kg
Rank: 6

Mr. Cristiano Ronaldo is a Football player with the international rank 6.

BMI is 23.7

Name: Rafael Nadal
Sport: Tennis
Gender: Male
Height: 1.85m
Weight: 85kg
Rank: 4

Mr. Rafael Nadal is a Tennis player with the international rank 4.

BMI is 24.6

Name: Mirabai Chanu
Sport: Weightlifting
Gender: Female
Height: 1.50m
Weight: 49kg
Rank: 2

Ms. Mirabai Chanu is a Weightlifting player with the international rank 2.

BMI is 21.8

Object – Instance of a class: Recap

Name: Cristiano Ronaldo
Sport: Football
Gender: Male
Height: 1.87m
Weight: 83kg
Rank: 6

Name: Rafael Nadal
Sport: Tennis
Gender: Male
Height: 1.85m
Weight: 85kg
Rank: 4

Name: Mirabai Chanu
Sport: Weightlifting
Gender: Female
Height: 1.50m
Weight: 49kg
Rank: 2

Instances of Sportsman class

Sports person 1

Sports person 2

Sports person 3

States

Name
Sport
Gender
Height
Weight
rank

Behaviors

Profiling()
BMI()

Sportsman

Class Declaration

Syntax:

Keyword

```
Access specifier class Class_name{
    Fields
      +
    Behaviours

}
```

Example:

Class Declaration

```
public class Theatre {
    int theatreId = 4523;
    String theatreName = "INOX";
    String theatreLocation = "Kochi";

    public void displayTheatre(){
        System.out.println("Theatre ID : "+theatreId);
        System.out.println("Theatre Name : "+theatreName);
        System.out.println("Theatre Location : "+theatreLocation);
    }
}
```

Fields/ Properties / Attributes/ States

Operations/ Behaviours/ Methods

Access Specifiers

- **Access specifiers** defines the **scope and accessibility** of **data and method** in class.
- There are **three** Java access modifiers:

1. **public**: accessible in all class in your application.
2. **protected:** accessible within the class in which it is defined and in its subclass(es).
3. **private:** accessible only within the class in which it is defined.
4. **default** (declared/defined **without using any modifier**) : accessible within **same class** and package within which its class is defined.

Note:

- For **classes** and **interface**, you can use either **public or default**
- For **attributes**, **methods** and **constructors**, you can use the one of the following: **Private, Protected, Public**

Access specifiers : In a Nutshell

Access Modifier	Within Class	Within Package	Outside package by subclass only	Outside package
Public	Yes	Yes	Yes	Yes
Private	Yes	No	No	No
Protected	Yes	Yes	Yes	No
Default	Yes	Yes	No	No

Object Creation

- When an object of a class is created, the class is said to be **instantiated**. All the instances can have the same attributes and the functions of the class.
- The memory allocation of **each object is unique** , which means the **value initialized to each object is distinct**.
- A single class may have any number of instances.

Syntax:

```
ClassName objectName = new ClassName();
```

- The new keyword **creates (instantiates) a new instance**. It instantiates a class by allocating **memory for a new object.**

Object Creation

```
/*  This Example demonstrates  how to create the Object for Theatre class */
public class Theatre {
    int theatreId = 4523;
    String theatreName = "INOX";
    int totalTheatreScreens = 12;
    String theatreLocation = "Kochi";
    public void getTheatreDetails(){          //Displaying Theatre details
        System.out.println("Theatre ID : "+theatreId);
        System.out.println("Theatre Name : "+theatreName);
        System.out.println("Total Theatre Screens : "+totalTheatreScreens);
        System.out.println("Theatre Location : "+theatreLocation);
```

```
public static void main(String[] args) {
        //Declare and instantiate
        Theatre T1 = new Theatre();
        //Declare the reference
        Theatre T2;
        //Then instantiate
        T2 = new Theatre();
    }
}
```

Note:

- A **declaration** only create **reference variable.**
- Allocate **memory** to object only at that **time of instantiation**.

Accessing Class Members

- After creation of object to access class members using **dot operator (.)**

```
/*  This Example demonstrates  how to create the Object for Theatre class */
public class Theatre {
   int theatreId = 4523;
   String theatreName = "INOX";
   int totalTheatreScreens;
   String theatreLocation = "Kochi";
   public void getTheatreDetails(){          //Displaying Theatre details
       System.out.println("Theatre ID : "+theatreId);
       System.out.println("Theatre Name : "+theatreName);
       System.out.println("Theatre Location : "+theatreLocation);
   }
```

```
public static void main(String[] args) {
        //Declare and instantiate
        Theatre T1 = new Theatre();
        //Fields are accessed
        T1.totalTheatreScreens=12;
        //Methods are called
        T1.getTheatreDetails();
  }
}
```

Accessing Class Members: Example #1

```
/**This Example demonstrates  how to access class members using object. */
public class Employee {//Create Employee class
        int empId;
       String empName;

       void setEmployeeDetail(int id,String name) {
            empId = id;
            empName = name;
       }
      void getEmployeeDetail () {
            System.out.println("Employee id : "+empId);
            System.out.println("Employee name : "+empName);
      }
}
```

Accessing Class Members: Example #1

```
public class EmployeeMain {
    public static void main(String[] args) {
        Employee Emp1 = new Employee();       //First Object Creation
        Employee Emp2 = new Employee();       //Second Object Creation
        Emp1.setEmployeeDetail(1001, "RAM");
        Emp2.setEmployeeDetail(1002, "RAJ");
        Emp1.getEmployeeDetail();
        Emp2.getEmployeeDetail();
    }
}
```

Output

```
Employee id : 1001
Employee name : RAM
Employee id : 1002
Employee name : RAJ
```

Accessing Class Members: Example#2

```
/* This Example demonstrates  how to access class members of Theatre Class*/
public class Theatre {
    int theatreId;
    String theatreName;
    int totalTheatreScreens;
    String theatreLocation;

    public void setTheatreDetails(int id, String name, int count,String address) {
        theatreId = id;
        theatreName = name;
        totalTheatreScreens = count;
        theatreLocation = address;
    }
```

Accessing Class Members: Example#2

```
    public void getTheatreDetails(){
        System.out.println("------Theatre Detail------");
        System.out.println("Theatre ID : "+theatreId);
        System.out.println("Theatre Name : "+theatreName);
        System.out.println("Total Theatre Halls : "+totalTheatreScreens);
        System.out.println("Theatre Location : "+theatreLocation);
        System.out.println("------------------------");
    }
    public static void main(String[] args) {
        Theatre T = new Theatre();
        T.setTheatreDetails(4523, "INOX", 10, "Kochi");
        T.getTheatreDetails();
    }
}
```

Output

```
------Theatre Detail------
Theatre ID : 4523
Theatre Name : INOX
Total Theatre Halls : 10
Theatre Location : Kochi
------------------------
```

Object Creation: More Details

Multiple Object Creation

- To create multiple objects like multiple variable, declare in the same declaration.
- **Example:** Theatre T1 = new Theatre(), Theatre T = new Theatre(); **//two objects in single creation**

Anonymous object

- It means **nameless object**. An object which has **no reference** is known as an **anonymous object**.
- It can be used **at the time of object creation** only. If you have to use an **object only once**, an **anonymous object** is a **good approach**.
- **Syntax**: new className().methodName()
- **Example**: new Theatre().setTheatreDetails(1001,”INOX”,10,”Kochi”);

Object Creation: More Details

Array of Objects:

- Like array of primitive types, the **array of objects** to store the **location of reference variables** of the object.
- **Syntax:** Class obj[]= new Class[array_length]
- **Example**: Theatre obj[]=new Theatre[10]; //Create 10 Objects

Array of Objects: Example #1

```
/** This example demonstrates the array of objects.  */
public class EmloyeeArrayObject {
    int empId;
    String empName;
    void setEmployeeDetail(int id,String name){
        empId = id;
        empName = name;
    }
     void getEmployeeDetail (){
        System.out.println("Employee id : "+empId);
        System.out.println("Employee name : "+empName);
    }
}
```

Array of Objects: Example #1

```
public class EmloyeeArrayObject {
    public static void main(String[] args) {
        Employee Emp[] = new Employee[2];
        for(int i=0;i<2;i++) {
            Emp[i] = new Employee();
    }
    System.out.println("-----Employee 1 Detail-----");
    Emp[0].setEmployeeDetail(1001, "AMUDHAN");
    Emp[0].getEmployeeDetail();
    System.out.println("-----Employee 2 Detail-----");
    Emp[1].setEmployeeDetail(1002, "RAJ");
    Emp[1].getEmployeeDetail();
    }
}
```

Output

```
-----Employee 1 Detail-----
Employee id : 1001
Employee name : AMUDHAN
-----Employee 2 Detail-----
Employee id : 1002
Employee name : RAJ
```

Constructors

- In Java, **Constructor** is a special method that is **executed automatically** whenever an **instance of object** is created. It is used **to initialize the object** and **give initial values** for object attributes.

Rules for Constructor:

- Constructor has **same name** as **class name**.
- Constructor **do not have return value** but can take **arguments**.
- Constructor cannot be **abstract, static, final, and synchronized. (Will discuss Later)**

Constructors

- There are **two types** of constructors in Java:

  1. No-argument/default constructor
  2. Parameterized constructor

Default Constructor

- A **constructor** that has **no parameters** is known as **default constructor**. It is either **user defined** or **compiler defined constructor**.
- If we don't create any constructor in a class, then **compiler creates default constructor** for the class and **assign default values** to attributes .
- The **user defined** default constructor provides user given initial values to attributes of the class for **each instantiation**.

Default Constructor: Example #1

```
/**
  * This example demonstrates compiler defined default constructor.
*/

class Employee {  //define Employee class
     int empId;
     String empName;

     void getEmployeeDetails () {
        System.out.println("The default Initial value of employee id is: "+empId);
        System.out.println("The default Initial value of employee name is: "+empName);
    }
}
```

Default Constructor: Example #1

```
class EmployeeMain  {          //Main Class
    public static void main (String[] args)
    {
            Employee emp= new Employee();
            // Default constructor initialize default values to the objects
           emp.getEmployeeDetails();
    }
}
```

Output

The default Initial value of employee id is: 0

The default Initial value of employee name is: null

Default Constructor: Example #2

```
/**
  * This example demonstrates user defined default constructor.
*/

class Employee {      //define Employee class
    int empId;
    String empName;
     //User defined default constructor

    Employee() {
        empId=1111;  //Initial value of employee id
        empName="AAA-BBB";  //Initial value of employee name
    }
}
```

Default Constructor: Example #2

```
 void getEmployeeDetails () {
        System.out.println("The Initial value of employee id is: "+empId);
        System.out.println("The Initial value of employee name is: "+empName);
    }
}
class EmployeeMain {    //Main Class
    public static void main (String[] args) {
    // this invoke compiler defined default constructor.
        Employee emp= new Employee();
    // Display assigned initial values
        emp.display();
     }
}
```

Output

The Initial value of employee id is: 111

The Initial value of employee name is: AAA-BBB

Parameterized Constructor

- A **constructor** that accepts arguments are known as **parameterized constructor**. It is use to **initialize with your own values**.
- In **parameterized constructor,** we must provide **initial values** of objects **as arguments** to the constructors.

Parameterized Constructor: Example #1

```
/**
  * This example demonstrates parameterized constructor.
*/

class Employee { //define Employee class
    int empId;
    String empName;


     //Parameterized constructor
     Employee(int id, String name){
        empId=id;  //Assign Initial employee id
        empName=name;  //Assign Initial employee name
    }
}
```

Parameterized Constructor: Example#1

```
 void getEmployeeDetails (){
       System.out.println("User given initial employee id is: "+empId);
       System.out.println("User given initial employee name is: "+empName);
   }
}
class EmployeeMain {  //Main Class
   public static void main (String[] args) {
          // Pass arguments to the constructor
          Employee emp= new Employee(1003,"Peter");
          //Display employee initial values
           emp.display();
    }
}
```

Output

User given Initial value of employee id is: 103

User given Initial value of employee name is: Peter

Constructor Overloading

- **Constructor overloading: More than one constructor** with different parameter lists depending on the application. Each constructors functions in its **own distinct way**.
- **Example:**

```
Employee() { ….}  //Default Constructor

Employee(int empId) {…..}  //One Parameter Constructor

Employee(String name, String designation) {….} // Two parameter Constructor
```

- They are differentiated by the Java compiler by the **number of arguments**, **order of arguments** listed and the **types of each arguments**.

Constructor Overloading: Example #1

```
/**
  * This example demonstrates the constructor overloading */
class Employee {  //define Employee class
      int empId;
     String empName;
     Employee(){     //Default Constructor
           empId=0;
           empName="AAA-BBB";
    }
    Employee(int id, String name){    //Parameterized constructor
           empId=1111;  //Assign Initial employee id
           empName=name;  //Assign Initial employee name
    }
```

Constructor Overloading: Example #1

```
  void getEmployeeDetails (){
        System.out.println("Employee id is: "+empId);
        System.out.println("Employee name is:"+empName);
   }
}
class EmployeeMain {  //Main Class
    public static void main (String[] args) {
          Employee emp0=new Employee();  //Default Constructor
          Employee emp1= new Employee(1001,"Ram Kumar");  //Parameterized Constructor
          //Display employee initial values
          emp0.getEmployeeDetails();
          emp1.getEmployeedisplay();
       }
}
```

Output

Employee id : 1002

Employee name :Guru

Employee id : 1003

Employee name :Peter

'this' Keyword

- **'this'** is a reference variable that refers to the **current object**.

Usage of 'this' Keyword

- It can be used **to refer instance variable** of current class and return the current class instance.
- It can be used **to invoke or initiate** current class **constructor.** It means **constructor chaining** is possible. **For example**, Call default constructor from parameterized

Note:

- this keyword cannot be used outside a class

'this' Keyword :Example #1

```
/**
  * This example illustrates the problem
  * if we dont use 'this' keyword.
*/

public class Employee {
  int empId;        // instance variable
  String empName;  //instance variable
  Employee(String empName, int empId )   {
      empName = empName;
      empId = empId;
  }
  void display()   {
      System.out.println(name+ " \t"+ empId);
  }
}
```

Both local and instance variables are same

Note:

- **'this'** keyword resolves the ambiguity between instance and local variable.
- This example illustrates the problem, if we don't use 'this' keyword.

'this' Keyword :Example #1

```
//Main Class
class EmployeeMain {
    public static void main (String[] args) {
          Employee emp = new Employee("Manas Kumar",29);
         emp.display();
     }
}
```

'this' Keyword :Example #2

```
/**
  * This program illustrates the use of 'this' keyword.
*/
public class Employee {
  int empId;
  String empName;
  Employee(String empName, int empId )  {
      this.empName = empName;
      this.empId = empId;
  }
  void display()  {
      System.out.println(name+ " \t"+ empId);
  }
}
```

Here this.empId and empName are instance variables

Note:

- **'this'** keyword resolves the ambiguity between instance and local variables.

'this' Keyword :Example #2

```
//Main Class
class EmployeeMain {
    public static void main (String[] args) {
          Employee emp = new Employee("Manas Kumar",29);
         emp.display();
     }
}
```

Output

Employee id : 29

Employee name : Manas Kumar

Static Members

- The **static** keyword in Java is used for **managing memory efficiently** with the class instances.
- We can apply **static** keyword with **variables, methods, blocks** and **nested classes**.
- The **static** keyword belongs to the class rather than each instance of the class. The **static member** is common to the class and it is **same for all the instances** created for that class. For **example**, the company name of employees, college name of students, bank name of account holders, etc.
- **Static members** can be accessed with objects of a class.
- Static members can be **accessed before any objects** of its class are created.

Static Block and Static Variables

Static Block

- **Static block** mainly used for to **initialize the static variables**.
- It is **executed** at the time of **class is loaded** in the memory.
- In case of **multiple static blocks**, which will execute in the **same order**.

Static Variable

- **Static variable** also called as **class variable**.
- It is **common to all the instance** of the class.
- **Memory allocation** for static variables are **happens when the class is loaded** in the memory.

Static Members : Static Block

```
/**
  * This example demonstrates static block */
class Employee{  //Main Class
   static int empId;
   static String empName;
   static{
      System.out.println("Static Block 1");
      empId = 1001;
      empName = "Alex";
  }
static{
      System.out.println("Static Block 2");
      empId = 1002;
      empName = "Peter";
  }
```

Static Members : Static Block

```
public static void main(String args[])
 {
    System.out.println("Employee Id : "+empId);
    System.out.println("Employee Name : "+empName);
  }
}
```

Note:

- **'static block'** is used to initialize the static data member.
- It is executed only once when the class gets loaded.

Output

```
Static Block 1
Static Block 2
Employee Id : 1002
Employee Name : Peter
```

Static Members : Static Variable

```
/**
  * This example demonstrates static variable.
*/

class Employee   { //define Employee class
    int empId;
    String empName;
    static String companyName=“ABC Solutions”; //static variable
     //Parameterized constructor
     Employee(int id, String name) {
        empId=id;  //Assign Initial employee id
        empName=name;  //Assign Initial employee name
    }
}
```

Static Members : Static Variable

```
 void display (){    //Employee Details
       System.out.println("Company Name : "+companyName);  //common to all employee
       System.out.println("Employee Id : "+empId);
       System.out.println("Employee Name  : "+empName);
} }
class EmployeeMain {//Main Class
   public static void main (String[] args) {
    // Pass arguments to the constructor
       Employee emp1= new Employee(1001,"Ram Kumar");
       Employee emp2= new Employee(1002,"Raj Kumar");
  //Display employee details
       emp1.display();
       emp2.display();
 } }
```

Note:

- Here, companyName Static variable is common for all objects created for that class.
- All instances of the class share the same static variable. **static block** and **static variables** are executed in order they are present in the program

Static Members : Static Variable

Output

```
Company Name : ABC Solutions
Employee Id : 1001
Employee Name  : Ram Kumar
Company Name : ABC Solutions
Employee Id : 1002
Employee Name  : Raj Kumar
```

Static Methods

- A **static method** belongs to the **class and common for all the object** of a class.
- It can be invoked **without object** of class or using **class name**.

Restrictions:

- Can access only other static methods and variables.
- Cannot refer to **this or super** keyword.

Static Methods : Example

```
/**
  * This example demonstrates static method.
*/

class Employee  {  //define Employee class
     int empId;
    String empName;
    static String companyName = “ABC Solutions”; //static variable
     //Parameterized constructor
     Employee(int id, String name){
        empId = id;  //Assign Initial employee id
        empName = name;  //Assign Initial employee name
    }
}
```

Static Methods : Example

```
//Change company name
static void getCompany(){
        companyName = “XYZ Private Ltd”; //Access static data
}
//Display Employee Details
void display (){
        System.out.println(“Company Name : ”+companyName);  //common to all employee
        System.out.println(“Employee Id : ”+empId);
        System.out.println(“Employee Name  : ”+empName);
       }
}
```

Static Methods : Example

```
class EmployeeMain {//Main Class
    public static void main (String[] args) {
        // Pass arguments to the constructor
        Employee emp1= new Employee(1001,"Ram Kumar");
        Employee emp2= new Employee(1002,"Raj Kumar");
        //Display employee details
        emp1.display();
        emp2.display();
        Employee.getCompany();  //Access static method
        //Display employee details after change company
        emp1.display();
        emp2.display();
    }
}
```

Output

```
Company Name : ABC Solutions
Employee Id : 1001
Employee Name  : Ram Kumar
Company Name : ABC Solutions
Employee Id : 1002
Employee Name  : Raj Kumar
Company Name : XYZ Private Ltd
Employee Id : 1001
Employee Name  : Ram Kumar
Company Name : XYZ Private Ltd
Employee Id : 1002
Employee Name  : Raj Kumar
```

Static Members : Static Method

```
/ * This example demonstrates static members created for the Theatre Screen class */
public class TheatreScreen {
   private static int totalseats = 20 ;        //static variable
   TheatreScreen(){
              System.out.println(“Current Seat Availability : "+totalseats);
   }
   public void DisplayTheatreScreen() {
              System.out.println(“Current Seat Availability : "+totalseats);
   }
   public static void BookTicket(int nooftickets) {    // static method
              System.out.println("No. of Seats Booked : "+nooftickets);
```

Static Members : Static Method

```
            totalseats -= nooftickets;
    }
    public static void main(String args[]) {
            System.out.println("Screen Ticket Availability Status:");
            TheatreScreen TS1 = new TheatreScreen();
            TS1.BookTicket(4);
            TheatreScreen TS2 = new TheatreScreen();
            TS2.BookTicket(5);
            TS2.DisplayTheatreScreen();
    }
}
```

Output

```
Screen Ticket Availability Status:
Current Seat Availability: 20
No. of Seats Booked : 4
Current Seat Availability : 16
No. of Seats Booked : 5
Current Seat Availability : 11
```

Encapsulation

Introduction

- **Encapsulation** is a another important concept of **Object Oriented Programming.**
- It is the process of **binding the data and behaviour** into a single unit called **class**.

Need of Encapsulation:

- In this world, many data are **sensitive, confidential and personal**. Hence privacy is an important threat with respect to every data.
- **Data privacy** is achieved with the help of **encapsulation concept** in Java.

Introduction

Achieve encapsulation in Java:

- Encapsulation can be **achieved** by declaring data in the **class members as private**.
- Provide **public setter and getter methods** to modify and view the private class members. (**Good Practice**)

Note:

- Encapsulation **prevents the private class members** being accessed by **external classes and methods**. Therefore, it is also known as **data hiding.**

Setter and getter Methods

Setter and getter Method: It is used to **update and retrieve the value of variables**.

Rules for setter Method:

- It should be **public**, if it needs to be accessed from outside.
- The **return-type** should be **void**.
- The setter method should be prefixed with **set**.
- It should **take some argument** i.e. it should not be **no-argument** method.

Rules for getter Method:

- It should be **public**, if it needs to be accessed from outside
- The **return-type** should **not be void** i.e. according to our requirement we have to give return-type.
- The getter method should be prefixed with **get**.
- It **should not** take **any argument**.

Advantages

- Use only a **setter or getter method**, you implicitly achieve member of the class that becomes **read-only or write-only option.**
- It gives you **command over the data**. **For Example**, You can write the logic inside the setter method if you need to set the value of empId based on some criteria.
- It is a way to **achieve data hiding** in Java because other class will not be able to access the data through the private data members.
- The encapsulate class is **easy to test**. So, it is **better for unit testing.**
- The standard **IDE's** are providing the **facility to generate** the getters and setters method it **helps** to create **encapsulate class easy and fast**.

Example

```
/**
  * This example demonstrates encapsulation features using getter and setter methods
*/

class Employee {      //define Employee class
    //private data members
    private int _empId;
    private String _empName;
     //public getter and setter methods
    //Set employee id
    public void setId(int id) {
        _empId=id;
    }
}
```

Example

```
//Set employee name
 public void setName(String name) {
        _empName=name;
    }
   //Get employee id
   public int getId() {
        return _empId;
    }
    //Get employee name
    public String getName(){
        return _empName;
    }
}
```

Example

```
class EmployeeMain   {    //Main Class
    public static void main (String[] args)    {
        Employee emp= new Employee(); //create instance of Employee class
       //setting values through setter methods
        emp.setId(1001);
        emp.setName(“Ram Kumar”);
       //getting values through getter methods
        System.out.println(“Employee Id: “+emp.getId());
        System.out.println(“Employee Name: “+emp.getName());
    }
}
```

Output

Employee Id : 1001

Employee Name Ram Kumar

Quiz

1. X is a keyword that denotes member variable or method can be accessed, without requiring an instantiation of the class to which it belongs. X is ____

a) This

b) static

c) volatile

d) public

e) None of the above

b) static

Quiz

2. In case the programmer does not provide a constructor for a class, Java compiler will

a) Throw error

b)Create Default constructor

c) Throw run time exception

d) Create new object

e) None of the above

b) Create Default constructor

Quiz

3. _____ provides objects with the ability to hide their internal characteristics and behaviour.

a) Encapsulation

b) Abstraction

c) Polymorphism

d) Inheritance

e) None of the above

a) Encapsulation

Quiz

4. In a class, an attribute needs to be accessed from any class in that application. What should be the access specifiers for that attribute

a) protected

b) private

c) public

d) default

e) None of the above

c) public

Quiz

5. Given a class with the name Trainee. Which of the following instantiates an object for this class?

a) Trainee t;

b) Trainee()

c) Trainee t=new Trainee()

d) All the options

e) None of the above

c) Trainee t=new Trainee()

Quiz

6. Assume, a class Person. Identify the correct signature for the constructor

a) void Person()

b) private void Person()

c) public Person()

d) public void Person()

e) None of the above

c) public Person()

Quiz

7. Which of the following keywords acts as a reference variable to the current object?

a) this

b) reference

c) static

d) public

e) None of the above

a) this

Quiz

8. The keyword used to create a new object in Java is ___.

a) class

b) java

c) new

d) create

e) None of the above

c) new

Quiz

9. In a .java file, how many numbers of public class allowed?

a) 1

b) 2

c) 3

d) Any number

a) 1

Quiz

10. How many maximum numbers of objects can be created from a single Class in Java?

a) 32

b) 64

c) 256

d) None of these Above

d) None of these Above