UNIT-4 Objects and Classes

Defining classes for objects, Constructors, accessing objects via reference variable, using classes from the java library, static variables, constants and methods, visibility modifiers and Data field encapsulation, passing objects to methods, array of objects, immutable objects and classes, scope of variable and the this reference.

1. What is the difference between OOP and procedural oriented language?

Object Oriented Programming	Procedural Oriented Programming
Object-oriented programming models software design around **data or objects** rather than functions and logic.	It is a programming language that is derived from structure programming and based upon the concept of calling **procedures**.
In object oriented programming, data is more important than function.	In procedural programming, function is more important than data.
In OOP, objects can move and communicate with each other via member functions.	In procedural programming, data moves freely within the system from one function to another.
Object oriented programming provides data hiding so it is ***more secure***.	Procedural programming does not have any proper way for hiding data so it is ***less secure***.
Object oriented programming have access specifiers like ***private, public, protected,default***.	There is no access specifier in procedural programming.
In OOP, there is a concept of function overloading and operator overloading.	Overloading is not possible in procedural programming.
It is appropriate for complex problems.	It is not appropriate for complex problems.
The examples of object-oriented programming are-*.NET, C#, Python, Java, VB.NET, and C++*.	Examples of Procedural programming include *C, Fortran, Pascal, and VB.*

bottom up approch

top -down approch

2. Explain types of variables and scope of instance, class and local variable with the help of example

Types of Variables in Java

1. Local Variables
2. Instance Variables
3. Static Variables / Class level Variables

1. Local Variables

A variable defined within a block or method or constructor is called a local variable.

- These variables are created when the block is entered, or the function is called and destroyed after exiting from the block or when the call returns from the function.
- The scope of these variables exists only within the block in which the variable is declared. i.e., we can access these variables only within that block.
- **Initialization** of the local variable is **mandatory** before using it in the defined scope.

INITIALIZE LOCAL VARIABLE - Java assigns no default value to a local variable inside a method

```
public class Main {
   public static void main(String[] args) {
    int x;
    System.out.println(x);
   }
}
Main.java:6: error: variable x might not have been initialized
    System.out.println(x);
```

2. Instance Variables

Instance variables are **non-static** variables and are declared in a class outside any method, constructor, or block.

- As instance variables are **declared in a class**, these variables are created when an object of the class is created and destroyed when the object is destroyed.
- Unlike local variables, we may use **access specifiers** for instance variables. If we do not specify any access specifier, then the default access specifier will be used.
- Initialization of Instance Variable is **not mandatory**. Its default value depends on data type.
- Instance Variable can be accessed only by **creating objects**.

3. Static Variables

Static variables are also known as **Class variables**.

- These variables are declared similarly as instance variables. The difference is that static variables are declared using the static keyword within a class outside any method constructor or block.
- Unlike instance variables, we can only have **one copy of a static variable per class** irrespective of how many objects we create.
- Static variables are created at the start of program execution and destroyed automatically when execution ends.
- **Initialization** of Static Variables is **not mandatory**.
- If we access the static variable like the Instance variable (through an object), the compiler will show the warning message, which won't halt the program. The compiler will replace the object name with the class name automatically.
- If we access the static variable without the class name, the compiler will automatically append the class name.

Differences between the Instance variable Vs. the Static variables

- Each object will have its copy of the instance variable, whereas We can only have one copy of a static variable per class irrespective of how many objects we create.
- Changes made in an instance variable using one object will not be reflected in other objects as each object has its own copy of the instance variable.
- In the case of static, changes will be reflected in other objects as static variables are common to all objects of a class.

- **We can access instance variables through object references, and Static Variables can be accessed directly using the class name.**

Example: Local, Static and Instance Variables

```
class StaticAndInstanceVar
{
     // Static variable
     static int a;
     // Instance variable
     int b;
      public static void main(String[] args) {
            int x=1; //local variable
      }
}
```

INITIALIZE INSTANCE VARIABLE - Default Values for instance Data Fields

EX-1

```
public class Main {
  int x;

  public static void main(String[] args) {
    Main myObj = new Main();
    System.out.println(myObj.x); // prints 0 for int type
  }
}
```

EX-2

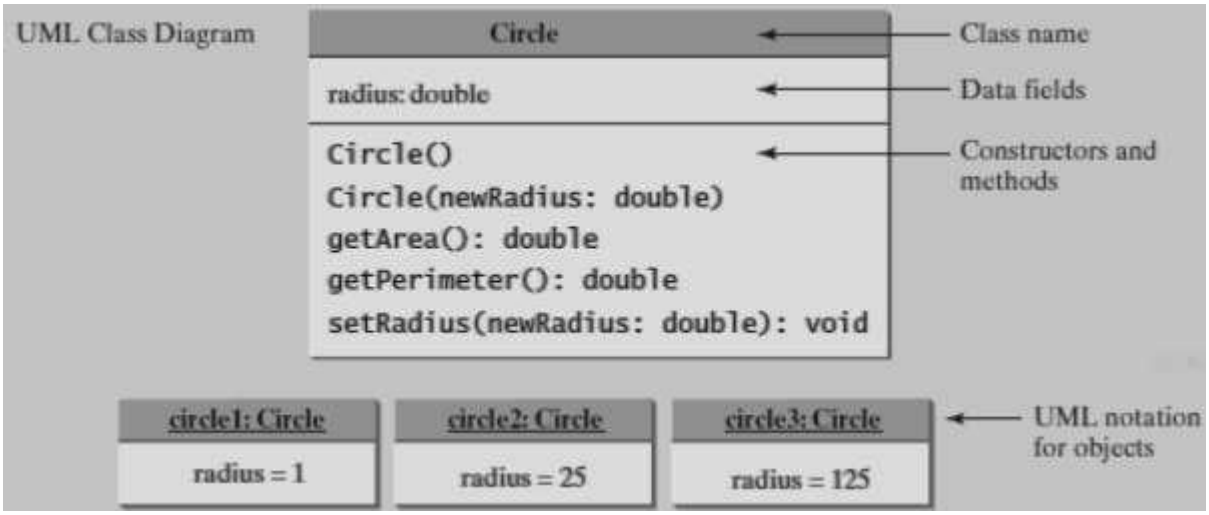
```
class Student {
      String name; // name has the default value null
      int age; // age has the default value 0
      boolean isScienceMajor; // isScienceMajor has default value false
      char gender; // gender has default value '\u0000'
}
```

3. Explain class and object with respect to java

- **Class** is a template, blueprint, or contract that defines what an object's data fields and methods will be.
- A class defines the properties and behaviors for objects.
- An **object** is an instance of a class.
- We can create many instances of a class.
- An object represents an entity in the real world that can be distinctly identified.
  For example, a **student**.
- An **object** has a unique identity, state, and behavior.
  - The **state** of an object (also known as its properties or attributes) is represented by data fields with their current values.
    A **circle** object, for example, has a data field **radius**.
  - The **behavior** of an object (also known as its actions) is defined by methods.

A **circle** object can invoke **setRadius()** method to change its radius.

- A class provides methods of a special type, known as **constructors**, which are invoked to create a new object are designed to initialize the data fields of objects.



4. How to access object via reference variable? Explain with example.

- An object's data and methods can be accessed through the dot (.) operator via the object's reference variable.

```
Circle myCircle = new Circle();
```

- The variable `myCircle` holds a reference to a Circle object `new Circle()`.

Accessing an Object's Data and Methods

- After an object is created, its data can be accessed and its methods can be invoked using the *dot operator* (**.**), also known as the *object member access operator*:

`myCircle.radius` references the data field radius in `myCircle`
`myCircle.getArea()` invokes the `getArea()` method on `myCircle`

- `System.out.println("Area is " + new Circle(5).getArea());`

creates a Circle object and invokes its `getArea()` method to return its area.
An object created in this way is known as **an anonymous object**.

5. Program: Create *SimpleCircle* class to find area of circle

```
public class TestCircle {
  public static void main(String[] args) {
     // Create a new SimpleCircle() object that is referred by circle1 reference variable
     SimpleCircle circle1 = new SimpleCircle();
     System.out.println("The area of the circle of radius " +
     circle1.radius + " is " + circle1.getArea());
```

```
        // Modify circle radius
        circle1.radius = 100;
        System.out.println("The area of the circle of radius " +
        circle1.radius + " is " + circle1.getArea());
      }
  }
class SimpleCircle {
        double radius=1; // Data Field (Instance Variable)

        /** instance method that Returns the area of this circle */
        double getArea() {
           return radius * radius * Math.PI;
        }
  }
```

- **Output:**

  The area of the circle of radius 1.0 is 3.141592653589793
  The area of the circle of radius 100.0 is 31415.926535897932

6. Define object oriented concepts OR List out and explain three main principles of object-oriented programming?

- **Object** means a real-world entity such as a pen, chair, table, computer, watch, etc.
- **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects.
- It simplifies software development and maintenance by providing some concepts:

1. **Object**

- Any entity that has state and behaviour is known as an object.
- An Object can be defined as an instance of a class.
- An object contains an address and takes up some space in memory. A class is a template or blueprint from which objects are created. So, an object is the instance (result) of a class.

  State: represents the data (value) of an object.

  Behaviour: represents the behaviour (functionality) of an object such as deposit, withdrawal, etc.

2. **Class**

- A class is a group of objects which have common properties.
- Collection of objects is called class.
- It is a template or blueprint from which objects are created.
- It is a logical entity.
- It can't be physical.

  A class in Java can contain: Field, Method, Constructor, blocks, Nested class…

  Syntax to declare a class:

```
class <class_name>{
   Data fields section;
   Constructors section;
```

```
    methods section;
}
```

3. Abstraction

- Abstraction is a process of **hiding** the implementation details and showing only functionality to the user.
- It focuses on reducing design complexity.
- It can be achieved by keeping main() method implementation to test utility class in separate class
- Another way, it shows only essential things to the user and hides the internal details.
- In Java, we use ***abstract*** class and ***interface*** to achieve abstraction.

4. Encapsulation

- Encapsulation is used to hide the values or state of an bject inside a class, preventing direct access to them by clients in a way that could expose hidden implementation details or violate any restriction on an object.

- Encapsulation in Java is a process of wrapping code and data together into a single unit, for example, a capsule which is mixed of several medicines.
- We can create a fully encapsulated class in Java by making all the **data members** of the class **private**.
- Now we can use **setter** and **getter** methods to set and get the data in it.

5. Inheritance

- When one class (subclass) acquires all the properties and behaviours of its parent class (superclass), it is known as inheritance.
- It provides code reusability.
- It is used to achieve runtime polymorphism.

6. Polymorphism:

- Polymorphism in Java is **a concept by which we can perform a single action in different ways**.
- Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.
- If one task is performed in different ways, it is known as polymorphism.
- A real-life example of polymorphism, a person at the same time can have different characteristics. Like a man at the same time is a person and an employee.
- In Java, we use method overloading and method overriding to achieve polymorphism.

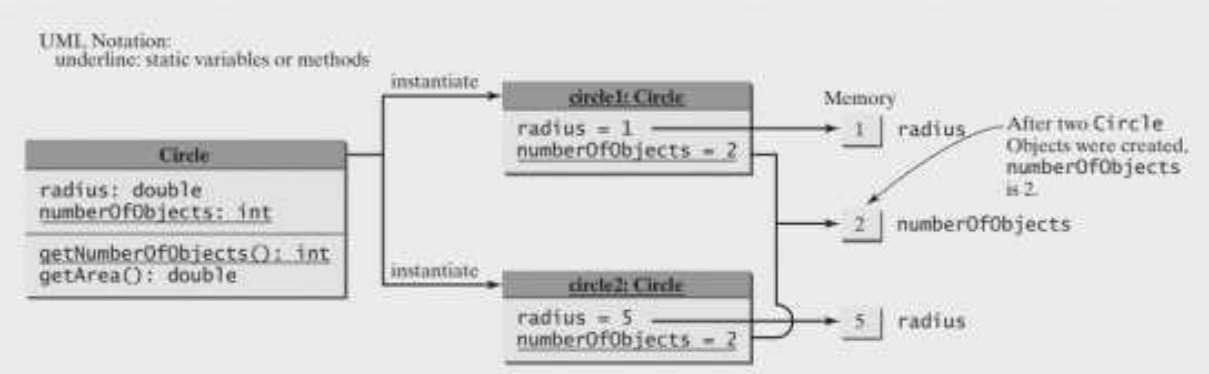
7. What is the purpose of 'static' keyword? Write a program to demonstrate the use of static members of the class OR Explain static variable and static method with example.

- The static keyword in Java is used for memory management mainly.
- The static keyword belongs to the class than an instance of the class.
- **'static'** keyword is used for variable, method, block and nested class(inner class)
- **A static variable**, also known as class variable **is shared** by all objects of the class.

- Static variables store values for the variables in a **common** memory location.
- **A static method** can be called without creating an instance of the class.
- A static method cannot access instance members of the class directly.
- For example, suppose that you create the following objects*:*

```
Circle circle1 = new Circle();
Circle circle2 = new Circle(5);
```

- The ***radius*** in ***circle1*** is independent of the ***radius*** in ***circle2*** and is stored in a different memory location. Changes made to circle1's radius do not affect circle2's radius, and vice versa.



To declare a static variable or define a static method, put the modifier **static** in the variable or method declaration.

Constants in a class are shared by all objects of the class. Thus, constants should be declared as **final static**. For example, the constant **PI** in the **Math** class is defined as:

```
final static double PI = 3.14159265358979323846;
```

1) Java static variable:

- If you declare any variable as static, it is known as a static variable.
- The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
- The static variable gets memory only once in the class area at the time of class loading.

2) Java static method:

- If you apply static keyword with any method, it is known as static method.
- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.

Example of static variable and method: Program to create the new circle class, named **Circle** having a static method **getObjects()** that keeps track of the number of objects created for the mentioned class.

```
class Circle {
      double radius;
      static int Objects = 0;    //static variable-shareb by all objects

     /** Construct a circle with radius 1 */
```

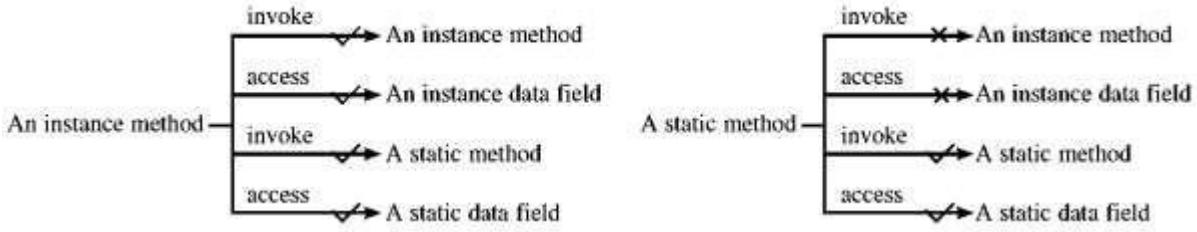
```
    Circle() {
    radius = 1;
    Objects++;
    }

    /** Static method to Return static variable-Objects */
     static int getObjects() {
          return Objects;
         }
    }

public class TestCircle {

  public static void main(String[] args)
  {

    System.out.println("The number of Circle objects is " +
                        Circle.Objects);

    Circle c1 = new Circle();

    // Use classname to access static variable name
     System.out.println("The number of Circle objects is after
                         creating c1:"+ Circle.getObjects());

    Circle c2 = new Circle();
    System.out.println("The number of Circle objects is after
                         creating c2: " + Circle.getObjects());
  }
}
```

Output:

The number of Circle objects is 0
The number of Circle objects is after creating c1:1
The number of Circle objects is after creating c2: 2

NOTE:

- Use **ClassName.methodName(arguments)** to invoke a static method and **ClassName.staticVariable** to access a static variable. This improves readability, because this makes the static method and data easy to spot.
- static method is used to manipulate static variable



8. What is the use of static import? Explain it giving an example.

- static import allows to access the static members of a class without class qualifications. For Example, to access the static methods you need to call the using class name:
  `Math.sqrt(25);`
- But, using static import you can access the static methods directly.

- **Example**

```
import static java.lang.Math.*;
public class Sample{
   public static void main(String args[]){
      System.out.println(sqrt(25));
   }
}
```

- **Output**

```
5.0
```

9. Differentiate between constructor and method of a class.

Constructors	Methods
Constructor is used to create and initialize an Object.	Method is used to execute certain statements.
A constructor is invoked implicitly by the System.	A method is to be invoked during program code.
A constructor is invoked when new keyword is used to create an object.	A method is invoked when it is called.
A constructor cannot have any return type.	A method can have a return type.
A constructor must have same name as that of the class.	A method name can not be same as class name.
A constructor cannot be inherited by a subclass.	A method is inherited by a subclass.

```
public class JavaTester {
      JavaTester(){
      num = 3;
      System.out.println("Constructor invoked.");
   }
   public void init(){
      System.out.println("Method invoked.");
```

```
    }
    public static void main(String args[]) {
        JavaTester tester = new JavaTester(); //constructor
        tester.init(); //method
    }
}
```

- **Output**

Constructor invoked.
Method invoked.

10. What is a constructor in JAVA? How many types of constructors are there in JAVA? Explain with examples.

- In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling the constructor, memory for the object is allocated in the memory.
- It is a special type of method which is used to initialize the object.
- Every time an object is created using the ***new()*** keyword, at least one constructor is called. It calls a **default constructor** if there is no explicit constructor available in the class. In such case, Java compiler provides a default constructor by default.

- There are some rules defined for the constructor.
  1. Constructor name **must** be the same as its class name
  2. A Constructor must have **no** explicit return type
  3. A Java constructor **cannot** be abstract, static, final, and synchronized
  4. Constructors are invoked using the **new** operator when an object is created.

- There are two types of constructors in Java: **no-arg constructor, and parameterized constructor.**

1. No-arg constructor (Non Parameterized)

A constructor is called a "**No-arg constructor**" when it doesn't have any parameter.

- Syntax of no-arg constructor :

```
<class_name>()
{
}
```

- Example of no-arg constructor :

```
class Bike
{
      //creating a no-arg constructor
       Bike()
        {
          System.out.println("Bike is created ");
        }
       public static void main(String args[])
```

```
        {
           //calling a default constructor
            Bike b=new Bike();
         }
}
```

Output : Bike is created

2. **Parameterized constructor:** A constructor which has a specific number of parameters is called a parameterized constructor.

```
public class Main{
    public static void main(String[] args){
       //creating objects and passing values
         Student s1 = new Student(1,"Jay");
         Student s2 = new Student(2,"Hardik");
   //calling method to display the values of object
         s1.display();
         s2.display();
    }
}
class Student
{
    int id;
    String name;
    //creating a parameterized constructor
    Student(int i,String n)
      {
        id = i;
        name = n;
      }
   //method to display the values
   public void display()
     {
       System.out.println(id+" "+name);
     }
}
```

Output : 1 Jay
2 Hardik

11. What do you mean by Overloading? Explain constructor overloading with suitable example.

- Constructor overloading in Java refers to the use of more than one constructor in an instance class. However, each overloaded constructor must have different signatures.
- Each constructor is differentiated by the compiler by the number of parameters in the list and their types.

Program of constructor overloading:

```
class Student
{
    int id;
    String name;
    int age;
    //creating two arg constructor
    Student(int i,String n)
    {
        id = i;
        name = n;
    }
    //creating three arg constructor
    Student(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[])
    {
    Student s1 = new Student5(1,"Aryan");
    Student s2 = new Student5(2,"Aditya",21);
    s1.display();
    s2.display();
    }
}
```

Output: 1 Aryan
2 Aditya 21

12. Why we generally declare constructor as public member.

- In most cases, the constructor should be public.
- However, if you want to prohibit the user from creating an instance of a class, use a private constructor.
- For example, there is no reason to create an instance from the Math class, because all of its data fields and methods are static. To prevent the user from creating objects from the Math class.

- The public constructor means it can be accessible outside the class.
- The other class can also access them easily
- We make the constructor as public to initialize the class anywhere in the program .

- Constructors can be public, private, protected or default (no access modifier at all).

- Making something private doesn't mean nobody can access it. It just means that nobody outside the class can access it. So private constructor is useful too.
- One of the use of private constructor is to serve *singleton* classes. A singleton class is one which limits the number of objects creation to one. Using private constructor we can ensure that no more than one object can be created at a time.

13. Why there is no destructor in java.

- **There is no concept of destructor in Java**. In place of the destructor, Java provides the ***garbage collector*** that works the same as the destructor. The garbage collector is a program (thread) that runs on the JVM. It automatically deletes the unused objects (objects that are no longer used) and free-up the memory. The programmer has no need to manage memory, manually. It can be error-prone, vulnerable, and may lead to a memory leak.

SHALLOW COPY & DEEP COPY Constructor

14. Using classes from the java library OR Explain Random and Date class with the help of a program.

- The Java API contains a rich set of classes for developing Java programs.
  - ❖ **The Date Class**
  - A Date object represents a specific date and time.

java.util.Date	
+Date()	Constructs a Date object for the current time.
+Date(elapseTime: long)	Constructs a Date object for a given time in milliseconds elapsed since January 1, 1970, GMT.
+toString(): String	Returns a string representing the date and time.
+getTime(): long	Returns the number of milliseconds since January 1, 1970, GMT.
+setTime(elapseTime: long): void	Sets a new elapse time in the object.

```
java.util.Date date = new java.util.Date();
System.out.println("The elapsed time since Jan 1, 1970 is " +
date.getTime() + " milliseconds");
System.out.println(date.toString());
```

```
The elapsed time since Jan 1, 1970 is 1324903419651 milliseconds
Mon Dec 26 07:43:39 EST 2022
```

  - ❖ **The Random Class**
  - Another way to generate random numbers is to use the *java.util.Random* class, as shown in Figure.
  - A **Random** object can be used to generate random values.

java.util.Random	
+Random()	Constructs a Random object with the current time as its seed.
+Random(seed: long)	Constructs a Random object with a specified seed.
+nextInt(): int	Returns a random int value.
+nextInt(n: int): int	Returns a random int value between 0 and n (excluding n).
+nextLong(): long	Returns a random long value.
+nextDouble(): double	Returns a random double value between 0.0 and 1.0 (excluding 1.0).
+nextFloat(): float	Returns a random float value between 0.0F and 1.0F (excluding 1.0F).
+nextBoolean(): boolean	Returns a random boolean value.

- For example, the following code creates two **Random** objects with the same seed **3**.

```
Random random1 = new Random(3);
System.out.print("From random1: ");
for (int i = 0; i < 10; i++)
System.out.print(random1.nextInt(1000) + " ");

Random random2 = new Random(3);
System.out.print("\nFrom random2: ");
for (int i = 0; i < 10; i++)
System.out.print(random2.nextInt(1000) + " ");
```

- The code generates the **same sequence** of random **int** values:

```
From random1: 734 660 210 581 128 202 549 564 459 961
From random2: 734 660 210 581 128 202 549 564 459 961
```

14. SHALLOW COPY & DEEP COPY Constructor

In shallow copy, only fields of primitive data type are copied while the objects references are not copied. Deep copy involves the copy of primitive data type as well as object references.

Deep Copy

```
public class Person {
        int x,y;
        Person(int x, int y)
         {
             this.x = x;
             this.y = y;
         }
         Person(Person p)
         {
             this.x = p.x;
             this.y = p.y;
         }

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

```
            Person p1 = new Person(1,2);
            Person p2 =  new Person(p1);

            System.out.println(p1.x + " " + p1.y); // prints "1 2"
            System.out.println(p2.x + " " + p2.y); // prints "1 2"

            p2.x = 3;
            p2.y = 4;

            System.out.println(p1.x + " " + p1.y); // prints "1 2"
            System.out.println(p2.x + " " + p2.y); // prints "3 4"
        }
    }
OUTPUT:
1 2
1 2
1 2
3 4
```

Shallow Copy

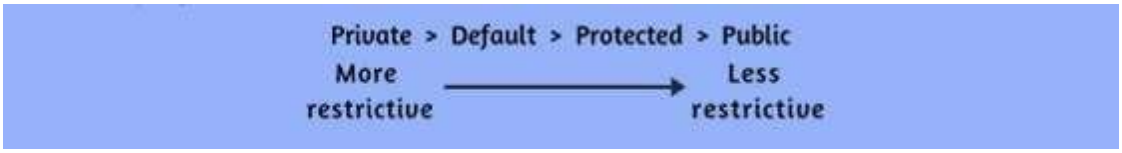
```
public class Person {

        int x,y;

        Person(int x, int y)
        {
            this.x = x;
            this.y = y;
        }

        public static void main(String[] args)
        {
            Person p1 = new Person(1,2);
            Person p2 =  p1;

            System.out.println(p1.x + " " + p1.y); // prints "1 2"
            System.out.println(p2.x + " " + p2.y); // prints "1 2"

            p2.x = 3;
            p2.y = 4;

            System.out.println(p1.x + " " + p1.y); // prints "1 2"
            System.out.println(p2.x + " " + p2.y); // prints "3 4"
        }
    }
OUTPUT:
1 2
1 2
3 4
3 4
```

Unit-4: Objects and Classes

15. Explain different Visibility modifiers.

- Visibility modifiers define the visibility of the class.
- Four access modifiers in java include **default, public, private and protected**.
- If no keyword is mentioned then that is **default** access modifier. It is called package-private also
- **public** visibility modifier for classes, methods, and data members to denote that they can be accessed from any other classes.
- The **private** modifier makes methods and data fields accessible only from within its own class.
- The **protected** access modifier can be applied on the data member, method and constructor.
- **private** and **protected** keywords cannot be used for classes and interfaces

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y



1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.

```
class A{
private int data=50;
private void msg()
{
   System.out.println("Hello java");
}
}

public class Simple{
 public static void main(String args[]){
   A obj=new A();
   System.out.println(obj.data);//Compile Time Error
   obj.msg();//Compile Time Error
   }
}
```

2. **Protected :** The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.

```
//save by A.java
package pack;
public class A{
protected void msg()
{
  System.out.println("Hello Java");
}
}
//save by B.java
package mypack;
import pack.*;

class B extends A{
  public static void main(String args[]){
   B obj = new B();
   obj.msg();
  }
}
Output : Hello Java
```

3. **Public :** The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

```
//save by A.java
package pack;
public class A{
public void msg()
  {
     System.out.println("Hello Java");
   }
}

//save by B.java
package mypack;
import pack.*;
class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

Output: `Hello Java`

4. **Default**: When no access modifier is specified for a class, method, or data member – It is said to be having the **default** access modifier by default.

- The data members, class or methods which are not declared using any access modifiers i.e. having default access modifier are accessible **only within the same package (package private)**.
- In this example, we will create two packages and the classes in the packages will be having the default access modifiers and we will try to access a class from one package from a class of the second package.

```
// Java program to illustrate error while using class from different package with default
// modifier
package p2;
import p1.*;
// This class of package p2 is having default access modifier
class Class2
{
     public static void main(String args[])
     {
          // Accessing Class1 from package p1
          Class1 obj = new Class1();
          obj.display();
     }
}
```

Output:

Compile time error

16. Data field encapsulation

- Making data fields private protects data and makes the class easy to maintain.
- To prevent direct modifications of data fields, you should declare the data fields private, using the **private** modifier. This is known as *data field encapsulation*.
- **Encapsulation** is useful in **hiding** the **data**(*instance variables*) of a class from an *illegal direct access*.
- To make a private data field accessible, provide a **getter** method to return its value. To enable a private
- data field to be updated, provide a **setter** method to set a new value. A getter method is also referred to as an accessor and a setter to a mutator.

```
class cat{
    private int weight = 30;
    public int get(){
        return weight;
    }
    public void set(int w){
        weight = w;
    }
}

public class Main{
    public static void main(String args[]){
        cat c = new cat();
    //  System.out.println(c.weight);
        System.out.println(c.get());    // 30
        c.set(50);
        System.out.println(c.get());    // 50
    }
}
```

17. How to pass object to a method? Explain with example

- Passing an object to a method is to pass the reference of the object.
- You can pass objects to methods. Like passing an array, passing an object is actually passing the reference of the object.

Write a program to demonstrate passing an object to a method to check if two objects are equal or not.

```java
// Java program to demonstrate objects passing to methods.
class ObjectPass
{
    int a, b;
    ObjectPass(int i, int j)
    {
        a = i;
        b = j;
    }
    // return true if o is equal to the invoking object
    boolean equalTo(ObjectPass o)
    {
        return (o.a == a && o.b == b);
    }
}
// Driver class
public class Test
{
    public static void main(String args[])
    {
        ObjectPass ob1 = new ObjectPass(100, 200);
        ObjectPass ob2 = new ObjectPass(100, 200);
        ObjectPass ob3 = new ObjectPass(1, 1);

        System.out.println("ob1 == ob2: " + ob1.equalTo(ob2));
        System.out.println("ob1 == ob3: " + ob1.equalTo(ob3));
    }
}
```

Output:

obj1 == obj2 true

obj1 == obj3 false

18. Explain array of objects with the help of a program.

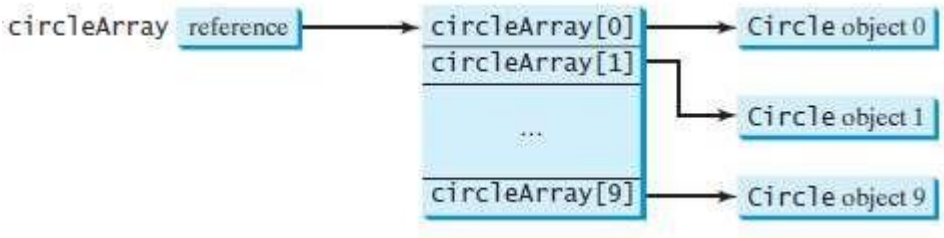
- *An array can hold objects as well as primitive type values.*
- For example, the following statement declares and creates an array of ten **Circle** objects:

```
Circle[] circleArray = new Circle[10];
```

- To initialize **circleArray**, you can use a **for** loop like this one:

```
for (int i = 0; i < circleArray.length; i++) {
     circleArray[i] = new Circle();
}
```

- An array of objects is actually an *array of reference variables*.
- So, invoking `circleArray[1].getArea()` **involves two levels** of referencing, as shown in Figure. `circleArray` references the entire array; `circleArray[1]` references a `Circle` object.



Note
When an array of objects is created using the new operator, each element in the array is a reference variable with a default value of null.

Write a program to demonstrate use of array of objects to display marks of JAVA, CO and OS subjects for n students.

```
OUTPUT

JAVA CO OS

12   23 18

20   25 19

13   14 15
```

```java
import java.util.*;
class Student {
   int java;
   int co;
   int os;
}

public class ArrayObjects{
  public static void main(String[] args) {
        Student[] studentArray = new Student[3];
        Scanner sc = new Scanner(System.in);
        for ( int i=0; i<studentArray.length; i++) {
        studentArray[i]=new Student();
       }

    for ( Student s : studentArray ) {
    s.java = sc.nextInt(); // sc is a Scanner object
    s.co = sc.nextInt();
    s.os = sc.nextInt();
    }

    System.out.println("JAVA CO OS");
    for ( Student x : studentArray ) {
      System.out.println(x.java + "   " + x.co + " " + x.os + " ");
    }
  }
}
```

19. Explain the concept of immutable object with example.

- *You can define immutable classes to create immutable objects. The contents of immutable objects cannot be changed.*
- Immutable class means that once an object is created, we cannot change its content. In Java, all the wrapper classes (like Integer, Boolean, Byte, Short) and String class is immutable. We can create our own immutable class as well.

- For a class to be immutable, it must meet the following requirements:
  - ■ All data fields must be private and final.
  - ■ There can't be any mutator methods (setters) for data fields.
  - ■ No accessor methods(getters) can return a reference to a data field that is mutable.

If a class contains only private data fields and no setter methods, is the class immutable?
Not necessarily. To be immutable, the class must also contain **no getter** methods that would return a reference to a mutable data field object.

```
// class is declared final
final class Immutable {
      // private class members
      private String name;
      Immutable(String name) {
            // class members are initialized using constructor
            this.name = name;
      }
      // getter method returns an immutable data field
      public String getName() {
            return name;
      }
}
class Main {
  public static void main(String[] args) {
     // create object of Immutable
     Immutable obj = new Immutable("newLJ");
     System.out.println("Name: " + obj.getName());
  }
}
```

Output: newLJ

- A **mutable** object can be changed after it's created, and an **immutable** object can't.
- Any time you change a string (e.g.: tacking on an extra character, making it lowercase, swapping two characters), you're actually creating a new and separate copy
- Immutability is mainly used for thread-safety and hashing techniques.

20. Explain keyword this.

- ***this*** keyword can be used to refer current class instance variable.
- it refers to the object itself.
- **It is used to reference hidden data fields or invoke an overloaded constructor of the same class.**
- If the instance variables and local variable (parameter) name is same, ***this*** keyword resolves the problem of ambiguity.

❖ ***this* keyword to invoke hidden instance variables**

```
class Student
{
    int rno;
    String name;

     //rno and name are used for local and instance variable names
    Student(int rno,String name)
    {
      this.rno=rno;    //assigning local variable to instance
      this.name=name;
    }
    void display()
    {
       System.out.println(rno+" "+name+);
    }
}
class TestThis
{
  public static void main(String args[])
  {
     Student s1=new Student(1,"Aditya");
     Student s2=new Student(2,"Aryan");
     s1.display();
     s2.display();
  }
}
```

Output: 1 Aditya
2 Aryan

❖ ***Using this to Invoke a Constructor***

- The **this** keyword can be used to invoke another constructor of the same class.
- For example,you can rewrite the **Circle** class as follows:

```
public class Circle {
  private double radius;

  public Circle(double radius) {
    this.radius = radius;
  }

  public Circle() {
    this(1.0);
  }

  ...
}
```

The `this` keyword is used to reference the hidden data field radius of the object being constructed.

The `this` keyword is used to invoke another constructor.

The line `this(1.0)` in the second constructor invokes the first constructor with a `double` value argument.

NOTE:
Java requires that the this(arg-list) statement appear first in the constructor before any other executable statements.

If a class has multiple constructors, it is better to implement them using this(arg-list) as much as possible. In general, a constructor with no or fewer arguments can invoke a constructor with more arguments using this(arg-list). This syntax often simplifies coding and makes the class easier to read and to maintain.

21. Scope of Variable

- The scope of instance and static variables is the entire class, regardless of where the variables are declared.

```
public class Circle {
  public double findArea() {
    return radius * radius * Math.PI;
  }

  private double radius = 1;
}
```

(a) The variable `radius` and method `findArea()` can be declared in any order.

```
public class F {
  private int i ;
  private int j = i + 1;
}
```

(b) `i` has to be declared before `j` because `j`'s initial value is dependent on `i`.