OBJECT ORIENTED PROGRAMMING SYSTEMS IN JAVA

Difference between Procedural programming and OOPs?

Procedural Programming	Oops
Procedural Programming is based on functions.	Object-oriented programming is based on real-world objects.
It shows the data to the entire program.	It encapsulates the data.
It does not have a scope for code reuse.	It provides more scope for code reuse.
It follows the concept of top-down programming.	It follows a bottom-up programming paradigm.
The nature of the language is complicated.	It is less complicated in nature, so it is easier to modify, extend and maintain.

1. **Object-Oriented Programming:**
   - OOPS is a methodology to design a program using classes and objects.
   - It simplifies the software development and maintenance by providing some concepts defined below:

2. **Class:**
   - class is a collection of objects with similar attributes.
   - It's a blueprint or template from which objects are made.
   - Class is the only logical representation of the data.

- The class does not occupy any memory space till the time an object is instantiated.
- For example, Human being is a class. The body parts of a human being are its properties, and the actions performed by the body parts are known as functions.

3. **Object:**
   - **Object** is a run-time entity.
   - It is an instance of the class.
   - It could be either physical or logical
   - An object can represent a person, place or any other item.
   - An object can operate on both data members and member functions.

```
public class Q1_Class_Object {
    Run | Debug
    public static void main(String args[]) {
        Student s1 = new Student();   //new dynamically allocates memory
        s1.name = "Aman";
        s1.age = 24;
        s1.getInfo();
           }
}

class Student {
    String name;
    int age;
    public void getInfo() {
        System.out.println("The name of this Student is " + this.name);
        System.out.println("The age of this Student is " + this.age);
    }
 }
```

```
The name of this Student is Aman
The age of this Student is 24
```

- **Note :** When an object is created **using** a **new keyword**, then space is allocated for the variable in a heap,
- and the starting address is stored in the stack memory.

4. **'this' keyword:**
   - 'this' keyword in Java that refers to the current instance of the class. In OOPS it is used to:
     - pass the current object as a parameter to another

method
    - refer to the current class instance variable
    - this() : to invoke current class constructor
- The super keyword is used to access hidden fields and overridden methods or attributes of the parent class.
- Following are the cases when this keyword can be used:
    - Accessing data members of parent class when the member names of the class and its child subclasses are same.
    - To call the default and parameterized constructor of the parent class inside the child class.
    - Accessing the parent class methods when the child classes have overridden them.

5. **Constructor**:
    - Constructor is a special method which is **invoked automatically** at the time of object creation.
    - It is used to **initialize the data members** of new objects generally.
    - Necessary condition:
        i. Constructors have the **same name** as class.
        ii. Constructors don't have a **return type**. (Not even void)
        iii. Constructor is only **called once**, at object creation.
        iv. First line of the constructor must be this() / super() . If not present compiler will add super() by default
        v. We can use only one (out of this() and super) but not both simultaneously.
        vi. Constructor overloading is allowed

There can be **three types** of constructors in Java.

1. Non-Parameterized constructor :
    - A constructor which has no argument is known as non-parameterized constructor(or no-argument constructor).
    - If we don't create one then it is created by default by Java.

```java
public class Q2_Non_Para_Constructor {
    Run | Debug
    public static void main(String[] args) {
        Student sidd=new Student();
        sidd.name="Siddhesh";
        sidd.roll=32;
        sidd.printInfo();
    }
}
//NON-PARAMETERISED CONSTRUCTOR
class Student{
    String name;
    int roll;
    public void printInfo(){
        System.out.println("Name: "+this.name);
        System.out.println("Roll: "+this.roll);
    }
    Student(){
        System.out.println(x:"Constructor is called first");
    }
}
```

```
Constructor is called first
Name: Siddhesh
Roll: 32
```

2. Parameterized constructor :

- Constructor which has parameters is called a parameterized constructor.
- It is used to provide different values to distinct objects.

```java
public class Q3_Para_Constructor {
    Run | Debug
    public static void main(String[] args) {
        Student sidd=new Student(name:"Siddhesh",roll:32);
        sidd.name="Siddhesh";
        sidd.roll=32;
        sidd.printInfo();
    }
}
//PARAMETERISED CONSTRUCTOR
class Student{
    String name;
    int roll;
    public void printInfo(){
        System.out.println(this.name);
        System.out.println(this.roll);
    }
    Student(String name,int roll){
        this.name=name;
        this.roll=roll;
    }
}
```

```
Siddhesh
32
```

3. Copy Constructor :

- A Copy constructor is an **overloaded** constructor used to declare and initialize an object from another object.

```java
public class Q4_Copy_Constructor {
    Run | Debug
    public static void main(String[] args) {
        Student sidd=new Student();
        sidd.name="Siddhesh";
        sidd.roll=32;
        sidd.printInfo();

        Student ram=new Student(sidd);
        ram.printInfo();
    }

}
//COPY CONSTRUCTOR
class Student{
    String name;
    int roll;
    public void printInfo(){
        System.out.println(this.name);
        System.out.println(this.roll);
    }
    Student(Student stud){
        this.name= stud.name;
        this.roll=stud.roll;
    }
    Student(){
    }
}
```

```
Siddhesh
32
Siddhesh
32
```

- **Note :** Unlike languages like C++, Java has no Destructor.
- Instead, Java has an **efficient garbage collector** that deallocates memory automatically.
- There is only a user defined copy constructor in Java(C++ has a default one too).

4. Constructor Overloading:

```java
public class Q4_Constructor_Overloading {
    Run | Debug
    public static void main(String[] args) {
        Student sidd=new Student();
        sidd.printInfo();
        Student ram=new Student(name:"Ram");
        ram.printInfo();
        Student shyam=new Student(name:"Siddhesh",roll:32);
        shyam.printInfo();
    }
}
class Student{
    String name;
    int roll;
    public void printInfo(){
        System.out.println(this.name);
        System.out.println(this.roll);
    }
    Student(){
        System.out.println(x:"Constructor is called");
    }
    Student(String name){
        this.name=name;
    }
    Student(String name,int roll){
        this.name=name;
        this.roll=roll;
    }
}
```

```
Constructor is called
null
0
Ram
0
Siddhesh
32
```

1. Polymorphism

- Polymorphism allows us to do a single operation in multiple
- ways.

- With polymorphism, each of these classes will have different underlying data.
- Precisely, (Greek words) Poly means 'many' and morphism means 'forms'.
- Polymorphism allows us to perform a single action in different ways.
- In other words, polymorphism allows you to define one interface and have multiple implementations.

Types of Polymorphism

1. Compile Time Polymorphism (Static)
2. Runtime Polymorphism (Dynamic)

1) **Compile Time Polymorphism** (static polymorphism)
   - The polymorphism which is implemented at the compile time is known as compile-time polymorphism.
   - Example - **Method Overloading**

Method Overloading:

- Method overloading is a technique which allows you to have more than one function with the **same function name** but with **different functionality**.
- Compiler is responsible to perform method resolution based on refefrence type.
- Method overloading can be possible by changes in:

1. The **type of the arguments.**
2. The **number of arguments.**

```java
public class Q7_MethodOverloading {
    Run | Debug
    public static void main(String[] args) {
        Student s1= new Student();
        s1.name="Siddhesh";
        s1.roll=32;
        s1.div='A';

        s1.print(s1.name);
        s1.print(s1.roll);
        s1.print(s1.div);
        s1.print(s1.name,s1.roll);
    }
}
class Student{
    String name;
    int roll;
    char div;

    public void print(char div) { System.out.println(this.div); }
    public void print(String name) {
        System.out.println(this.name);
    }
    public void print(int roll) {
        System.out.println(this.roll);
    }

    public void print(String name,int roll) {
        System.out.println(this.name+" "+this.roll);
    }
}
```

```
Siddhesh
32
A
Siddhesh 32
```

2) **Runtime Polymorphism** :

- Runtime polymorphism is also known as **dynamic polymorphism.**
- Function **overriding** is an example of runtime polymorphism.
- Function overriding means when the **child class contains** the **method** which is already **present in the parent** class.

- Hence, **the child class overrides the method of the parent class.**
- In case of function overriding, parent and child classes both contain the same function with a different definition.
- Method overriding not possible for private or final methods.
- Necessary condition:
    a. Method signature (Method name+ no./order of argument) must be same.
    b. Return type must be same until 1.4 V . After 1.5V, covariant return type is allowed
    c. Child class method scope must not be reduced
    d. Method resolution is taken care by JVM according to run time object

Property	Overloading	Overriding
1. Method names	Must be same	Must be same
2. Argument Types	Must be different (at least order)	Must be same (Including order)
3. Method signatures	Must be different	Must be same.
4. Return Type	No Restrictions	Must be same but this rule is applicable until 1.4version only. From 1.5v onwards co-variant return types are allowed.
5.private,static & final methods	Can be overloaded	Cannot be overridden
6. Access modifiers	No Restrictions	We can't reduce scope of Access modifier but we can increase.
7. throws clause	No Restrictions	If child class method throws any checked exception compulsory parent class method should throw the same checked exception are its parent otherwise we will get compile time error but there are no restrictions for Unchecked Exceptions.
8. Method Resolution	Always takes care by compiler based on reference type.	Always takes care by JVM based on Runtime object.
9.Also Known as	Compile time polymorphism or static polymorphism or early binding.	Runtime polymorphism, dynamic polymorphism or late binding.

3) **Method hiding**: It is similar to method overriding except both parent class method and child class method are static.

```java
public class Q7_Method_Overriding {
    Run | Debug
    public static void main(String[] args) {
        Triangle t1=new Triangle();
        Shape s1= new Shape();
        s1.area();  //area() of Shape class is called
        t1.area();//area() of Triangle class is called
    }
}
class Shape{
    public void area(){
        System.out.println(x:"Displays Area ");
    }
}
class Triangle extends Shape {
    public void area() {
        System.out.println(x:"Displays Triangle Area ");
    }
}
```

```
Displays Area
Displays Triangle Area
```

2. **Inheritance**
   - Inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically.
   - In such a way, you can **reuse, extend or modify** the attributes and behaviors which are defined in other classes.
   - In Java, the class which inherits the members of another class is called derived class
   - and the class whose members are inherited is called base class.
   - It is called as IS A Relationship.
   - Advntage: Code Reusability

   Types of Inheritance :

   a. Single inheritance :
      - When one class inherits another class, it is known as single level inheritance

```java
public class Q6_SingleInheritance {
    Run | Debug
    public static void main(String[] args) {
        Triangle t1=new Triangle();
        t1.area();    //area() of Shape class is called
        t1.area(base:10,height:5);
    }
}

class Shape{
    public void area(){
        System.out.println(x:"Displays Area ");
    }
}
class Triangle extends Shape {
    public void area(float base,float height) {
        float totalArea=0.5f*base*height;
        System.out.println(totalArea);
    }
}
```

```
Displays Area
25.0
```

b. Hierarchical inheritance :

- Hierarchical inheritance is defined as the process of deriving more than one class from a base class.

```java
public class Q6_HeirarchicalInheritance {
    Run | Debug
    public static void main(String[] args) {
        Triangle t1=new Triangle();
        t1.area();    //area() of Shape class is called
        t1.area(base:10,height:5);
        Circle c1=new Circle();
        c1.area();    //area() of Shape class is called
        c1.area(radius:7);
    }
}

class Shape{
    public void area(){
        System.out.println(x:"Displays Area ");
    }
}
class Triangle extends Shape {
    public void area(float base,float height) {
        float totalArea=0.5f*base*height;
        System.out.println(totalArea);
    }
}
class Circle extends Shape {
    public void area(float radius) {
        float totalArea=3.14f*radius*radius;
        System.out.println(totalArea);
    }
}
```

```
Displays Area
25.0
Displays Area
153.86002
```

c. <u>Multilevel inheritance</u> :

- Multilevel inheritance is a process of deriving a class from another derived class.

```java
public class Q6_MultilevelInheritance {
    Run | Debug
    public static void main(String[] args) {
        EquilateralTriangle t2= new EquilateralTriangle();
        t2.area();
        t2.area(base:10,height:10);
        t2.sidesInfo();
    }
}
class Shape{
    public void area(){
        System.out.println(x:"Displays Area ");
    }
}
class Triangle extends Shape {
    public void area(float base,float height) {
        float totalArea=0.5f*base*height;
        System.out.println(totalArea);
    }
}
class EquilateralTriangle extends Triangle {
    public void sidesInfo() {
        System.out.println(x:"All sides are equal");
    }
}
```

```
Displays Area
50.0
All sides are equal
```

d. Hybrid inheritance :

- Hybrid inheritance is a combination of simple, multiple inheritance and hierarchical inheritance.

3. Encapsulation

- Encapsulation is the process of combining data and functions into a single unit called class.
- In Encapsulation, the data is not accessed directly; it is accessed through the functions present inside the class. In simpler words, attributes of the class are kept private and public getter and setter methods are provided to manipulate these attributes.

- Thus, encapsulation makes the concept of data hiding possible.

4. **Data hiding:**
   - It is used as a security such that no internal data will be accessed without authentication
   - An unauthorised user will not get access to internal data.
   - **Getters and Setters :** `The private` variables can only be accessed within the same class (an outside class has no access to it). However, it is possible to access them if we provide public **get** and **set** methods.

```
public class Q5_Encapsulation {
    Run | Debug
    public static void main(String[] args) {
        Account acc1=new Account();
        acc1.name="Sid";
        acc1.setPassword("12485pass:");
        System.out.println(acc1.getPassword()); //12485
    }
}
class Account{
    String name;
    private String password;
    public String getPassword() {
        return this.password;
    }
    public void setPassword(String pass) {
        this.password = pass;
    }
}
```

5. **Abstraction**
   - We try to obtain an **abstract view**, model or structure of a real life problem, and reduce its unnecessary details.
   - In simple terms, it is hiding the unnecessary details & showing only the essential parts/functionalities to the user.
   - **Data binding : Binding** refers to the link between method call and method definition.

Abstraction is achieved in 2 ways :

1) Abstract class
2) Interfaces (Pure Abstraction)

1) **Abstract Class**

- It has only declaration and no implementation
- Child class is responsible to provide implementation
- If a class contain at least one abstract method then the class is declared as abstract.
- We can't create object of abstract class.
- Abstract Class can have constructors
- Abstract Class can have abstract as well as non-abstract methods
- It can have final methods which will force the subclass not to change the body of the method

```java
public class Q8_Abstraction {
    Run | Debug
    public static void main(String[] args) {
        Horse h=new Horse();
        h.eat();
        h. noOfLegs();
//          Animal a=new Animal(); //Cannot create animal object
    }
}
abstract class Animal
{
    Animal(){
        System.out.println(x:"Animal is created first");
    }
    abstract void noOfLegs();
    public void eat(){
        System.out.println(x:"Animal eats");
    }
}
class Horse extends Animal
{
    Horse(){
        System.out.println(x:"Horse is created after animal");
    }
    public void noOfLegs (){
        System.out.println(x:"Horse has 4 legs");
    }
}
```

```
Animal is created first
Horse is created after animal
Animal eats
Horse has 4 legs
```

2) Interfaces

- All the fields in interfaces are(fps) public, static and final by default.
- All methods are (pa)public & abstract by default.
- A class that implements an interface must implement all the methods declared in the interface.

- We cant declare an interface private or protected.(If the members of the interface are private you cannot provide implementation to the methods or, cannot access the fields of it in the implementing class.

  In general, the protected members can be accessed in the same class or, the class inheriting it. But, we do not inherit an interface we will implement it.)
- Interfaces support the functionality of multiple inheritance.

```java
public class Q9_Interface {
    Run | Debug
    public static void main(String[] args) {
        Horse horse = new Horse();
        horse.walk();
        horse.eatGrass();
    }
}
interface Animal {
    int eyes=2;
    void walk();//by default public and abstract
}
interface Herbivore{
    void eatGrass();
}
class Horse implements Animal,Herbivore {
    public void walk() {
        System.out.println(x:"Horse walks on 4 legs");
    }
    public void eatGrass() {
        System.out.println(x:"Eats grass");
    }
}
```

Interface	Abstract Class
1. If we don't know anything about implementation just we have requirement specification then we should go for interface.	1. If we are talking about implementation but not completely (partial implementation) then we should go for Abstract class.
2. Inside Interface every method is always public and abstract whether we are declaring or not. Hence interface is also considered as 100% pure Abstract class.	2. Every method present in Abstract class need not be public and Abstract. In addition to abstract methods we can take concrete methods also.
3. We can't declare interface method with the following modifiers. Public ---> private, protected, Abstract ---> final, static, synchronized, native, strictfp	3. There are no restrictions on Abstract class method modifiers.
4. Every variable present inside interface is always public, static and final whether we are declaring or not.	4. The variables present inside Abstract class need not be public static and final.
5. We can't declare interface variables with the following modifiers. private, protected, transient, volatile.	5. There are no restrictions on Abstract class variable modifiers.
6. For Interface variables compulsory we should perform initialization at the time of declaration otherwise we will get compile time error.	6. For Abstract class variables it is not required to perform initialization at the time of declaration.
7. Inside interface we can't declare instance and static blocks. Otherwise we will get compile time error.	7. Inside Abstract class we can declare instance and static blocks.
8. Inside interface we can't declare constructors.	8. Inside Abstract class we can declare constructor, which will be executed at the time of child object creation.

3) Static Keyword

Static can be :

1. Variable (also known as a class variable)
2. Method (also known as a class method)
3. Block
4. Nested class

```
class Student{
    String name;
    static String school;
    public static void changeSchoolname(){
        school="new School";
    }
}
public class StaticExample {
    public static void main(String[] args) {
        Student tarun=new Student();
        Student sid=new Student();
        tarun.name="Tarun";
        Student.school="Ratnam";
        Student.changeSchoolname();

        System.out.println("Taruns school: "+tarun.school);
        System.out.println("Sids school: "+sid.school);
    }
}
```

1. Package in Java

- Package is a grouping mechanism to group related classes, interfaces and sub-packages into a single unit.
- It is used to avoid name conflict and allows to write a better maintainable code.
- Packages can be built-in or user defined.
- Package Statement must be first statement& then import
- We can import Built-in packages using 2 methods.
  Explicit import import java.util.ArrayList;
  Implicit import import java.util.*;

```
import java.util.Scanner;
import java.io.IOException;
```

2. Access Modifiers in Java

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

➢ **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.

➢ **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.

➢ **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.

➢ **Private -> Class Level Access**

➢ Default –> Package Level Access

➢ Public –> Global Level Access

```
package newpackage;

class Account {
   public String name;
   protected String email;
   private String password;

   public void setPassword(String password) {
       this.password = password;
   }
}
public class Sample {
   public static void main(String args[]) {
       Account a1 = new Account();
```

```
        a1.name = "Apna College";
        a1.setPassword("abcd");
        a1.email = "hello@apnacollege.com";
    }
}
```

What is auto widening and explicit narrowing?

The data is implicitly casted from small sized primitive type to big sized primitive type. This is called auto-widening. i.e The data is automatically casted from byte to short, short to int, int to long, long to float and float to double..

You have to explicitly cast the data from big sized primitive type to small sized primitive type. i.e you have to explicitly convert the data from double to float, float to long, long to int, int to short and short to byte. This is called explicit narrowing.

Java Threads

A thread is a lightweight subprocess, the smallest unit of processing.

Threads are independent.

If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.

Threads allows a program to operate more efficiently by doing multiple things at the same time.

Threads can be used to perform complicated tasks in the background without interrupting the main program.

Creating a Thread

There are two ways to create a thread.

It can be created by extending the `Thread` class and overriding its `run()` method:

```
public class Main extends Thread {

  public void run() {
```

```
    System.out.println("This code is running in a thread");
  }
}
```

Another way to create a thread is to implement the `Runnable` interface:

```
public class Main implements Runnable {
  public void run() {
    System.out.println("This code is running in a thread");
  }
}
```

Life Cycle of a thread

1. ***NEW –*** a newly created thread that has not yet started the execution
2. ***RUNNABLE –*** either running or ready for execution but it's waiting for resource allocation
3. ***BLOCKED –*** waiting to acquire a monitor lock to enter or re-enter a synchronized block/method
4. ***WAITING –*** waiting for some other thread to perform a particular action without any time limit
5. ***TIMED_WAITING –*** waiting for some other thread to perform a specific action for a specified period
6. ***TERMINATED –*** has completed its execution

Errors

There are three types of errors in java.

1) Syntax errors

2) Logical errors

3) Runtime errors- also called Exceptions

Syntax Errors

When compiler finds something wrong with our program,

it throws a syntax error

```
int   a = 9  // No semicolon, syntax errors!
a =   a + 3;
d = 4; // Variable not declared, syntax errors
```

Logical errors

A logical error or a bug occurs when a program

compiles and round but does the wrong thing.

- Message delivered wrongly

- Wrong time of chats being displayed

Incorrect redirects!

Runtime errors

Java may sometimes encounter an error while the program is running.

These are also called Exceptions!

These are encountered due to circumstances like bad input and (or) resource constraints.

Ex: User supplies 'S' + 8 to a program that adds 2 numbers.

Syntax errors and logical errors are encountered by the programmers, whereas Run-time errors are encountered by the users.

Exceptions in Java

An exception is an event that occurs when a program is executed dissented the normal flow of instructions.

There are mainly two types of exceptions in java:

1) Checked exceptions - compile-time exceptions (Handle by the compiler)

2) Unchecked exceptions - Runtime exceptions

Commonly Occurring Exceptions

Following are few commonly occurring exceptions in java:

1) Null pointer exception

2) Arithmetic Exception

3) Array Index out of Bound exception

4) Illegal Argument Exception

5) Number Format Exception

```
public class cwh_80_try {
    public static void main(String[] args) {
        int a = 6000;
        int b = 0;
        try {
            int c = a / b;
            System.out.println("The result is " + c);
        }
        catch(Exception e) {
            System.out.println("We failed to divide. Reason:");
            System.out.println(e);
        }
        catch (ArithmeticException e){
            System.out.println("ArithmeticException occured!");
            System.out.println(e);
        }
        catch (ArrayIndexOutOfBoundsException e){
            System.out.println("ArrayIndexOutOfBoundsException occured!");
            System.out.println(e);
        }

        System.out.println("End of the program");
    }
}
```

this and super keyword in Java

this keyword in Java :

this is a way for us to reference an object of the class which is being created/referenced.

It is used to call the default constructor of the same class.

this keyword eliminates the confusion between the parameters and the class attributes with the same name.

```java
class constr {
    int x;
    public int getX() {
        return x;
    }
    constr(int x) {
        this.x = x;
    }
}
public class ConsExample{
    public static void main(String[] args) {
        constr obj1 = new constr(65);
        System.out.println(obj1.getX()); //65
    }
}
```

Super keyword

A reference variable used to refer immediate parent class object.

It can be used to refer immediate parent class instance variable.

It can be used to invoke the parent class method.

```java
class parentClass{
    parentClass(){
        System.out.println("Super class constructor");
    }
}
class childClass extends parentClass{
    childClass() {
        super();
        System.out.println("I am a constructor");
    }
}
public class SuperExample {
    public static void main(String[] args) {
        parentClass e = new parentClass();
        childClass d = new childClass();
    }
}
//Super class constructor
//Super class constructor
//I am a constructor
```

```java
class MyThread1 extends Thread{
    public void run(){
        int i=0;
        while(i<100000)
            System.out.println("I am coking");
    }
}
```

```
class MyThread2 extends Thread{
    public void run(){
        int i=0;
        while(i<100000)
            System.out.println("I am eating!!!!!!!!!!!!!!");
    }
}
public class ThreadsExample {
    public static void main(String[] args) {
        MyThread1 t1=new MyThread1();
        MyThread2 t2=new MyThread2();
        t1.start();
        t2.start();
    }
}
I am eating!!!!!!!!!!!!!
I am coking
```

```
class MyThreadR1 implements Runnable{
    public void run(){
        int i=0;
        while(i<100000)
            System.out.println("I am coking");
    }
}
class MyThreadR2 implements Runnable{
    public void run(){
        int i=0;
        while(i<100000)
            System.out.println("I am eating!!!!!!!!!!!!!!");
    }
}
public class ThreadRunnable {
    public static void main(String[] args) {
        MyThreadR1 bullet1=new MyThreadR1();
        Thread gun1=new Thread(bullet1);

        MyThreadR2 bullet2=new MyThreadR2();
        Thread gun2=new Thread(bullet2);

        gun1.start();
        gun2.start();
    }
}
```

The significant differences between extending Thread class and implementing Runnable interface:

- When we extend Thread class, we can't extend any other class even we require and When we implement Runnable, we can save a space for our class to extend any other class in future or now.

- When we extend Thread class, each of our thread creates unique object and associate with it. When we implements Runnable, it shares the same object to multiple threads.

57. What is constructor chaining?

Constructor chaining is a method to call one constructor from another concerning a current object reference. It can be done in two ways: –

1. Using the “this” keyword, the reference can be made to the constructor in the current class.
2. To call the constructor from the base class “super” keyword will be used.

58. What is Coupling in OOP, and why is it helpful?

The degree of dependency between the components is called coupling.

Types of Coupling

A. **Tight Coupling** – If the dependency between components is high, these components are called tightly coupled.

Ex: –

Below three Classes are highly dependent on each other hence they are tightly coupled.

```
class P
{
	static int a = Q.j;
}

class Q
{
	static int j = R.method();
}
```

```
class R
{
   public static int method(){
      return 3;
   }
}
```

B. **Loose Coupling** – If the dependency between components is low, it is called loose coupling. Loose coupling is preferred because of the following reasons:-

1. It increases the maintainability of code
2. It provides reusability of code

59. Name the operators that cannot be overloaded

All the operators except the + operator cannot be overloaded.

60. What is Cohesion in OOP?

The modules having well-defined and specific functionality are called cohesion.

Advantages

It improves the maintainability and reusability of code.

62. What are the types of variables in OOP?

Variables are basic units to store data in RAM for Java programs.

Variables should be declared before using them in Java programming. Variable initialization can be static or dynamic. The syntax for variable declaration and static initialization is: –

Types of variables

- **Primitive Variables:** It is used to represent primitive values like int, float, etc.
- Reference Variables: It is used to refer to objects in Java.
- **Instance Variables:** Variables whose value varied from object to object are instance variables. For every object, a separate copy of the instance variable is created. Instance variables are declared within the Class and outside any method/block/constructor
- **Static variables:** For static Variables, a single copy of the variable is created, and that copy is shared between every Class object. The static variable is created during class loading and destroyed at class unloading.
- Static variables can be accessed directly from the static and instance area. We are not required to perform initialization explicitly for static variables, and JVM will provide default values.
- **Local Variables:** Variables declared inside a method or block or constructor are local variables. Hence the scope of local variables is the same as the block's scope in which we declared that variable.

JVM doesn't provide default values, and before using that variable, the initialization should be performed explicitly.

63. What do you understand by Garbage Collection in the OOPs world?

Garbage collection is a memory recovery technique included in programming languages like C# and Java. A GC-enabled programming language contains one or more garbage collectors that automatically free up memory space allocated to objects that are no longer needed by the program.

64. Is it possible to run a Java application without implementing the OOPs concept?

No, since Java programmes are founded on the concept of object-oriented programming models, or OOPs, a Java application cannot be implemented without it.

52. What are pure virtual functions?

A pure virtual function/method is a function whose implementations are not provided in the base class, and only a declaration is provided. The pure virtual function can have its implementation code in the derived class; otherwise, the derived class will also be considered an abstract Class. The Class containing pure virtual functions is abstract.

39. What is Garbage Collection(GC)?

Programming languages like C# and Java include garbage collection (GC) as a memory recovery mechanism. A programming language that supports garbage collection (GC) contains one or more GC engines that automatically release memory space that has been reserved for things the application is no longer using.