Assignment 8

1. Class and Object

In Object-oriented programming, an entity that has state and behaviour is known as an Object. Objects are the instances of a class that are created to use the attributes and method of a class.

An object has three characteristics:

State which represents the data [value] of an object.

Behaviour which represents the functionality of an object.

Identity: An object identity is typically implemented via a unique ID.

For example, Smartphone is an Object. Its name is iPhone 14, state - color is black, behaviour - The smartphone can make calls, send text messages, browse the internet and take photos.

A class in Java is a set of objects which shares common characteristics / behaviour and common properties / attributes. It is a user-defined blueprint or template from which objects are created.

For example, Student is a class while particular student named Saikiran is an object.

A class in Java can contain Data members, Method, Constructor, Nested class and Interface declaration:

```
access_modifier class <class_name>
{
 data member;
 method;
 constructor;
 nested class;
 interface;
}
```

2. Class members.

Class members refer to the variables and methods that are defined within a class. These members are the building blocks that define the structure

and behaviour of objects instantiated from the class. There are two main types of class members: fields (or attributes) and methods.

Fields represent the state of data associated with objects of the class. They define the characteristics or properties of objects. For example, in a class representing a smartphone, fields could include attributes such as brand, model, year, color and storage capacity. Each instance of the Smartphone class (each Smartphone object) would have its own values for these fields, allowing them to represent different smartphones with unique characteristics.

Methods, on the other hand, define the behaviour or actions that objects of the class can perform. They encapsulate the operations that objects can execute. Continuing with the Smartphone class, methods could include behaviours such as making a call, sending a text message, browsing the internet, taking a photo and playing music.

These methods would contain the instructions or algorithms for performing these actions, allowing smartphone objects to interact with their users and environment.

```
Public class Smartphone {
// Fields
private String brand;
Private String model;
Private int year;
Private String color;
Private int storageCapacity;

// Methods
Public void makeCall (String phoneNumber) {
 // Code to initiate a call to the specified phone number

}
 Public Void sendMessage (String recipient, String message) {
 // Code to send a text message to the specified
   recipient with the given message
 }
```

```
Public void browseInternet (String Website) {
  // Code to open a web browser and navigate to the
     specified website
}
Public void takePhoto[]{
  // Code to activate the camera and capture a photo
 }
}
```

3. Encapsulation, Information hiding

Java Encapsulation is a way of hiding the implementation details of a class from outside access and only exposing a public interface that can be used to interact with the class.

Encapsulation is achieved by declaring the instance variables of a class as private, which means they can only be accessed within the class. To allow outside access to the instance variables, public methods called getters and setters are defined, which are used to retrieve and modify the values of the instance variables. By using getters and setters, the class can enforce its own data validation

rules and ensure that its internal state remains consistent.

Information hiding is a technique for reducing the dependencies between modules. The intended client is provided with all the information needed to use the module correctly and with nothing more. The client uses only the (publicly) available information. Information hiding also helps to reduce the system complexity to increase the robustness by limiting the interdependencies between software components. Information hiding is achieved by using the private access specifier.

We can implement an information-hiding mechanism by making class attributes inaccessible from the outside. Also provide getter and/or setter methods for attributes to be readable or updatable by other classes.

Access specifiers define how the members functions and variables can be accessed from outside the class. So, there are three access specifiers available.

within a class that are stated as follows:

Private members / methods: Functions and Variables declared as private can be accessed only within the same class, and they cannot be accessed outside the class they are declared.

Public members / methods: Functions and Variables declared under public can be accessed from anywhere.

Protected members / methods: Function and variables declared as protected cannot be accessed outside the class except a child class. This specifier is generally used in inheritance.

4. Generalization

Generalization is the process of taking out common properties and functionalities from two or more classes and combining them together into another class which acts as the parent class of those classes or what we may say the generalized class of those specialized classes. All the subclasses are a type of superclass. So we can say that subclass "is-A" superclass. Therefore, Generalization is termed as

'is-A relationship':

Let's consider a scenario where we have two classes - SmartphoneUser and TabletUser. Both classes have common properties like name, age and gender. To avoid redundancy, we create a parent class called DeviceUser, which contains these common properties and then extend SmartphoneUser and TabletUser from it.

The DeviceUser class serves as the parent class containing common properties like name, age and gender. Both SmartphoneUser and TabletUser classes extend DeviceUser and inherit these properties. Additionally, each subclass contains its own specific properties and methods, such as phoneNumber and email for SmartphoneUser and tabletModel and WifiNetwork for TabletUser. This demonstrates the extension provided by generalization, where common attributes and behaviours are inherited by child classes., allowing for code reuse and organization.

5. Composition and aggregation

The composition is a design technique in Java to implement a 'has-a' relationship. Java Inheritance is used for code reuse purposes and the same we can do by using composition. The composition is achieved by using an instance variable that refers to other objects. If an object contains the other and the contained object cannot exist without the existence of that object, then it is called composition. In more specific words composition is a way of describing reference between two or more classes using instance variable and an instance should be created before it is used.

In the Smartphone class, there is a composition relationship with the Battery and Camera classes. Each Smartphone object contains a Battery object and a Camera Object, and they cannot exist independently of the Smartphone. If a Smartphone object is destroyed, its associated Battery and

Camera objects will also be destroyed. This relationship demonstrates composition, where the Smartphone class is composed of the Battery and Camera classes.

An aggregation is a relationship between two classes where one class contains an instance of another class. For example, when an object A contains a reference to another object B or we can say Object A has a HAS-A relationship with Object B, then it is termed as Aggregation in Java Programming. Aggregation in Java helps in reusing the code. Object B can have utility methods and which can be utilized by multiple objects. Whichever class has Object B then it can utilize its methods.

Example, the Smartphone class has aggregation relationships with the Battery and Processor classes. Each Smartphone object contains instances of the Battery and Processor classes, but they can exist independently of the Smartphone. If a Smartphone object is destroyed, its associated Battery and Processor objects remain unaffected.

This relationship demonstrates aggregation, where the Smartphone class is composed of the Battery and Processor classes.

7. Static Method matching

A static method is a method that belongs to a class, but it does not belong to an instance of that class and they are referenced by the class name itself or reference to the Object of that class. In the static method, the method can only access only static data members and static methods of another class or the same class but cannot access non-static methods and variables.

The static method uses compile-time or early binding the static method cannot be overridden because of early binding. In the static method, less memory is used for execution because memory allocation happens only once because the static keyword fixed a particular memory for that method in ram.

Example. We have a Smartphone class with a static method PrintInfo(). We also have two subclasses, Android and IOS, which override the PrintInfo() method. However, since static methods cannot be overridden in Java, the PrintInfo() method called on the reference variables androidPhone and iosPhone is resolved based on their declared type (Smartphone). As a result both calls output "This is a smartphone", demonstrating static method matching in Java.

```
Public class Main {
    Public static void main (String[] args) {
     Smartphone androidphone = new Android[];
     Smartphone iosPhone = new IOS[];
     // Calls the static method from the Smartphone class
     androidPhone.printInfo[]; // Output: This is a smartphone
    // Calls the static method from the Smartphone class
     iosphone.printInfo[]; // Output: This is a smartphone

    }
```

8. Dynamic binding

Dynamic binding is also referred to as a run-time polymorphism. In this type of binding, the functionality of the method call is not decided at compile-time. In other words, it is not possible to decide which piece of code will be executed as a result of a method call at compile-time.

Characteristics:

Linking - Linking between method call and method implementation is resolved at run time.

Resolve mechanism - Dynamic binding uses object type to resolve binding.

Example - Method overriding is the example of Dynamic binding

Type of Methods - Virtual methods use dynamic binding.

Example, We have a superclass Phone with a method makeCall(), and a subclass Smartphone that overrides the makeCall() method and adds an additional method openAppstore(). When we create a Smartphone Object and assign it to a reference variable

of type Phone, dynamic binding allows the makeCall() method to be resolved at runtime to the overridden implementation in the Smartphone class. However, since the reference variable is of type Phone, we cannot directly access the openAppStore() method. We can still access it by casting the reference variable to Smartphone, demonstrating dynamic binding in action.

```
Class Phone {
    Void makeCall() {//
    }
}
Class Smartphone extends Phone {
    @Override
    Void makeCall() {
    //
    }
    Void openAppStore() {
    //
    }
}
Public class Main {
    Public Static void main (String[] args) {
```

```
Phone myPhone = new Smartphone(); // Creating a Smartphone
                Object with a phone reference.
// Dynamic binding - the actual method called depends
on the runtime type of the object myPhone.makeCall();
// Output: Making a call from a smartphone
// Since myPhone is of type Phone, it cannot directly
   access the openAppStore() method.
// myPhone.openAppStore(); // This would result in compilation
                              error
    }
}
```

9. Polymorphism

The word polymorphism means having many forms. Polymorphism is considered one of the important features of Object-Oriented Programming. 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.

Following are the characteristics of Polymorphism in Java:

- The functionality of a method behaves differently in different scenarios.
- The behaviour of a method depends on the data provided.
- It allows the same name for a member or method in a class with different types.
- Polymorphism supports implicit type conversion.

There are two different types of Polymorphism in Java. They are:

Compile - Time Polymorphism: The overloading method resolution takes place in the compilation stage. Method overloading is the process in which the class has two or more methods with the same name.

Run - Time Polymorphism: Run - Time Polymorphism is a procedure where the program execution takes place during Run-Time. Here, the resolution of an overriding happens in the execution stage.

Method Overriding is a procedure in which the compiler can allow a child class to implement a specific method already provided in the parent class.

Operator Overriding a procedure where we can define an operator in both parent and child classes with the same signature, but the with different operational capability. Java does not allow operator overriding to avoid ambiguites.

Example, both Smartphone and BasicPhone are subclasses of Phone. They override the call() method inherited from the Phone class with their own implementations. At runtime, when we call the call() method using references of type Phone, the JVM determines which version of the method to execute based on the actual type of the object. This demonstrates runtime Polymorphism, where the appropriate method implementation is selected dynamically based on the Object's type at runtime.

```
class Phone {
    public void call() {
    System.out.println("calling...");
    }
}
```

```
class Smartphone extends Phone {
  @Override
  Public Void call() {
  System.out.println("making a call from a smartphone...");
  }
}
class BasicPhone extends Phone {
  @Override
  Public void call(){
   System.out.println("making a call from a basic phone...");
  }
}
Public class Polymorphism Example {
 Public static void main (String[] args] {
  Phone phone1 = new Smartphone();
  Phone phone2 = new BasicPhone();
  phone1.call(); // Output: Making a call from a smartphone...
  phone2.call();// output: Making a call from a basic phone...
  }
}
```

10. Deep copy, shallow copy

Shallow copy: When we do a copy of some entity to create two or more than two entities as well, then we can say we have done a shallow copy. In shallow copy, new memory allocation never happens for the other entities, and the only reference is copied to the other entities.

- Whenever we use default implementation of clone method we get shallow copy of object means it creates new instance and copies all the field of object to that new instance and returns it as object type, we need to explicitly cast it back to our original object. This is shallow copy of the object.
- clone() method of the object class support shallow copy of the object. If the object contains primitive as well as non primitive or reference type variable in shallow copy, the cloned object also refers to the same object to which the original object refers as only the object references gets copied and not the referred objects themselves.

Deep copy: When we do a copy of some entity to make two or more than two entities such that changes in one entity are not reflected in the other entities, then we can say we have done a deep copy. In the deep copy, a new memory allocation happens for the other entities, and reference is not copied to the other entities. Each entity has its own independent reference.

A lazy copy can be defined as a combination of both shallow copy and deep copy. The mechanism follows a simple approach - at the initial state, shallow copy approach is used. A counter is also used to keep a track on how many objects share the data. When the program wants to modify the original object, it checks whether the object is shared or not. If the object is shared, then the deep copy mechanism is initiated.

In shallow copy, only fields of primitive data type are copied while the object references.

There is no hard and fast rule as to when to do a deep copy. Lazy copy is combination of both of these approaches.

11. Fat interface

Interface Segregation Principle says that the interface should not be bloated with methods that implementing classes don't require. For such interfaces, also called "fat interfaces", implementing classes are unnecessarily forced to provide implementations [dummy/empty] even for those methods that they don't need. In addition, the implementing classes are subject to change when the interface changes.

A fat interface increases coupling between classes and makes the code harder to maintain and understand. In the interface segregation principle, the word interface does not necessarily mean a Java interface. The interface segregation principle also applies for abstract classes or in fact, any public method that our own class depends upon.

Let's consider a fat interface called Smartphone Functionality that contains various methods related to smartphone features. In this example, We'll include methods for making calls, sending messages, accessing the internet and taking photos.

```
interface SmartphoneFunctionality {
    Void makeCall(String phoneNumber);
    Void sendMessage(String phoneNumber, String message);
    Void accessInternet(String url);
    Void takePhoto();
    // Other smartphone functionalities...
}
Class Smartphone implements SmartphoneFunctionality {
    @Override
    Public Void makeCall(String phoneNumber) { // Implementation }
    @Override
    Public Void sendMessage(String phoneNumber, String message) {
        // Implementation
    }
    @Override
    Public Void accessInternet(String url) {
        // Implementation
    }
```

```
@Override
Public void takePhoto() {
  // Implementation
}
// Other methods required by Smartphone
}
```

In this example, Smartphone functionality interface defines a set of methods that cover various smartphone functionalities. However, a class like Smartphone that implements this interface may not necessarily need all these methods. This can lead to a bloated / FAT interface and implementation class. It's better to have more focused interfaces that define specific sets of functionalities.

12. Open-Closed Principle:

The Principle states that software entities like class, modules, functions, etc. should be able to extend a class behaviour without modifying it. This Principle seperates the existing code from modified mode to provide better stability, maintainability and minimizes the changes in the code.

This means that a class should be flexible enough to accomodate changes when business requirements change without breaking the existing code. the dependent classes do not have to change.

By adhering to this principle, developers can ensure better stability, maintainability, and minimize the need for changes in the codebase when new requirements arise.

One approach to implementing the OCP is through inheritance. In this approach, a superclass is created with a baseline set of functionality, which remains closed for modifications. If there is a need to extend or modify this functionality, developers can subclass the superclass, add new features, and override existing methods as needed. This approach allows the superclass behaviour to remain intact, and clients interacting with it do not need to be updated. However, it may introduce tight coupling between the superclass and its subclasses.

Another approach is to use interfaces and abstract methods. In this approach, interfaces and are defined with a set of methods, serving as a contract for implementing classes. The interfaces remain closed for modifications, and different implementations can be created based on business requirements. This promotes loose coupling between classes, as implementation classes are independent of each other. This approach allows for greater flexibility and scalability, as new implementations can be added without affecting existing code.

Let's consider a class called SmartphoneApp that represents various applications installed on a smartphone. Each application has specific functionality, such as a calculator, messaging app and camera app. Following the OCP, we design the SmartphoneApp class to be open for extension but closed for modification.

how the Open-closed Principle can be applied to the Smartphone App class:

- We create a base class called SmartphoneApp that defines common behaviours and attributes for all smartphone applications. This class serves as the foundation for all specific application classes.
- To ensure that the SmartphoneApp class is closed for modification, we define abstract methods or interfaces for specific functionalities that can vary among applications. For example, we may have an abstract method run() that represents the main functionality of each application.
- New applications can be added to the smartphone without modifying the SmartphoneApp class. Instead, we extend the SmartphoneApp class to create new subclasses for each application. Each subclass implements the abstract methods or interfaces to provide specific functionality.
- Implementation: We can have subclasses like CalculatorApp, MessagingApp, and CameraApp that extend the SmartphoneApp class. Each subclass

implements the run() method according to its specific functionality. If we need to add a new application in the future, we can create a new subclass without modifying the existing SmartphoneApp class.

13. Dynamic linking and static linking

Static linking means link or resolve all unresolved symbols and assign relocatable addresses before loading the final object file into the main memory, i.e. all linking is done before loading the program in the main memory.

Disadvantage: Every program generated must contain copies of exactly the same common system library functions. In terms of both physical memory and disk-space usage, it is much more efficient to load the system libraries into memory only once.

Dynamic linking means some unresolved symbol that is not solved at linking time, it will be solved after loading the object file into

main memory, i.e, linking will be done at the time of execution when programs are in main memory. Dynamic linking reduces the size of executable files and promotes code reuse, as multiple programs can access the same shared libraries.

Advantage: Memory requirements of the program are reduced. A DLL is loaded into memory only once, whereas more than one application may use a single DLL at the moment, thus saving memory space. Application support and maintainance costs are also lowered.

14. Fragile base class problem

Inheritance is one of the fundamental principles in an object-oriented paradigm, bringing a lot of values in software design and implementation. However, there are situations where even the correct use of inheritance breaks the implementations.

The improper design of the parent class can lead subclasses to use the superclass in unexpected ways.

This often leads to broken code, even when the IS-A criterion is met. This architectural problem is known as the fragile base class problem in object-oriented programming systems.

The obvious reason for the this problem is that the developer of a base class has no idea of the subclass design. When the modify the base class, the subclass implementation could potentially break.

To avoid / reduce the Fragile Base Class Problem, developers should follow best practices such as minimizing the scope of changes to base classes, providing clear documentation for class interfaces, and using design patterns like the Open-Closed Principle to ensure that classes are open for extension but closed for modification. Additionally, through testing and version control practices can help detect and manage changes that may introduce fragility to the base class hierarchy.

6. Dynamic Allocation

Dynamic memory allocation in Java means that memory is allocated to Java objects during the run time or the execution time. i.e objects are created using the new keyword, and memory for these objects is allocated on the heap. It is contrary to static memory allocation. The dynamic memory allocation takes place in the heap space. The heap space is where new objects are always created and their references are stored in the stack memory.

The Memory allocation in Java is divided into parts, namely Heap, Stack, Code and Static. Heap Memory can accessible from the complicated memory management technique, including the Young Generation, Old or Tenured Generation and Permanent Generation. In heap memory, when it gets full, it returns java.lang.OutOf-MemoryError. The access in this memory is comparatively slower than that of the Stack memory.

Let's consider a smartphone example, we can dynamically allocate memory for smartphone objects based on user input.

```
Public class Smartphone {
    Private String brand;
    Private String model;
    Private double price;
    Public Smartphone (String brand, String model, double price) {
        this.brand = brand;
        this.model = model;
        this.price = price;
    }
    public void displayInfo () {
        // displays brand, model and price
    }
    Public static void main (String [] args) {
        // Read the values by using scanner
        // Dynamic allocation of smartphone object
        Smartphone smartphone = new Smartphone (brand, model, price);
        // Display smartphone information
        smartphone.displayInfo ();
    }
```

The Smartphone class represents Smartphones with attributes such as brand, model and price. In the main method, memory for a Smartphone object is dynamically allocated using the new keyword based on user input for brand, model, and price. The display Info method then displays the information of the dynamically allocated Smartphone object.