OOPs — JAVA

Abstract Data Types → ADT is a mathematical model for a certain class of data structures that specifies the behavior of the data and the operations that can be performed on it, without specifying how the operations are implem. ADTs focus on the what (specification) rather than the how (implementation).

Key Characteristics of ADTs

① Encapsulation of Data :-
- the internal representation of data is hidden from the user.
- The user interacts with the data only through a predefined set of operations.

② Abstraction:
- Abstracts away implementation details.
- only the essential features and operations are exposed to the user.

③ Well-defined operations
- Each ADT specifies a set of operations that can be performed, such as adding, deleting, or modifying data.

Samyak

④ Idependence of Implementation → The same ADT can have multiple implementations.

ex) Stacks can be implemented using arrays or linked lists.

- User does not need to know how the operations are implemented.

Examples of ADTs:

① List ② Stack ③ Queue ④ Set

⑤ Map (or Dictionary)

Components of an ADT →

① Concrete State Space

- Represents the actual data stored in the ADT.
- ex) in STACK, the State Space might include the array/list used to store elements and the index of the top element.

② Concrete Invariant

- A condition that must always hold true for the data.

ex) in a Stack, the "top" index must always be within the bounds of the array.

Samyak

③ Abstraction Function :-

- Maps the concrete representation of the ADT to its abstract value.
- For ex, in a stack, the abstraction function maps the array and top index to the logical view of a stack.

Specification of ADTs

- The Specification of an ADT defines what it does, rather than how it does it. It describes the behaviour, operations, and constraints of the ADT in a clear and abstract way, without revealing the implementation details.

Samyak

How to implement an ADT ?

Step-1) Understand the Specification

- Behaviour
- Constraints
- Performance / Requirements

Step-2) Choose a Data Specification

- Select an appropriate concrete data structure to represent the ADT.

ex) Stack → can be implemented using

- Arrays
- Linked lists

- Queue → can be implemented using
  - Arrays (Circular Queue)
  - Linked Lists
- Set → can be implemented using
  - Hash Tables
  - BSTs
- Maps → "
  - Hash Maps
  - R-B Trees

Step-3 Implement the ADT Operations.

For each operation in the ADT specification

- Translate the Abstract behaviour into code
- Ensure the constraints are respected.
- optimize it.

Ex) Implementing the STACK ADT.

Behaviour → LIFO

Operations → push, pop, peek, isEmpty, size

Implementation using an Array:=>

```
class ArrayStack<T> {
    private T[] stack;
    private int top;
    privat static final int capacity=10;
    public ArrayStack() {
        stack=(T[]) new Object[capacity];
        top=-1;
    }
```

```
public void push (T element) {
    if ( top == Stack.length-1) {
            expandCapacity();
    }
    Stack.[++top] = element;
}

public T pop() {
    if (isEmpty()) {
        throw new RuntimeException
        ("Stack is empty!");
    }
    T result = Stack[top];
    Stack[top--] = null;
    return result;
}

public T peek() {
    if (isEmpty()) {
        throw new RuntimeException
        ("Stack is empty!");
    }
    return Stack[top];
}

public boolean isEmpty() {
    return top == -1;
}

public int size() {
    return (top+1);
}
```

Samyak

```
private void expandCapacity() {
    T[] larger = (T[]) new Object
                 [stack.length * 2];
    System.arraycopy(stack, 0, larger,
                     0, stack.length);
    stack = larger;
}
}
```

Step-4 Ensure Encapsulation

- Hide the internal implementation
- Use interfaces (if applicable)

Step-5 Optimization

Samyak

Concrete State Space

- It defines the set of all possible states of the ADTs implementation. It describes the internal representation of the ADT in terms of the data structures and variables used in the implementation
- It is specific to the chosen data structure
- It includes all possible values of the internal variables used to represent the ADT.

• Example STACK ADT

- Abstract View → A stack is a collection

of elements that follow LIFO.

- Concrete Implementation
  - D.S → An array stack[] of fixed size.
  - Concrete State Space
    - the array stack[]
    - An integer top representing the index of the top element.

Here, Concrete State space is all possible combinations of values in stack[] and top.

Concrete Invariant

↳ Set of conditions that must always hold true for the internal representation of the ADT. It ensures that the concrete state space correctly represents valid abstract states of the ADT.

- They ensure the implementation adheres to the ADTs rules and constraints.
- They define what makes a concrete state "valid".

ex) Stack ADT.

Here, concrete invariants are :-

① $0 \leq top < capacity$ (The top index must always be within valid bounds of the array)

② Elements in stack[0] to stack[top] are the valid elements of the stack.

③ top = -1 ; means Stack is empty.

③ Abstraction Function

The abstraction function maps a concrete state (Implementation) to its corresponding abstract state (specification). It provides a way to interpret the internal representation of the ADT as the abstract data it is meant to represent.

- It is a conceptual bridge between the abstract and concrete representations.
- It describes how the internal data structures and variables correspond to the ADTs abstract behaviour.
- It is used to verify that the implementation behaves as specified.

Eg) Stack ADT,

- Concrete representation
  - ↳ Array Stack[]
  - ↳ Integer top.
- Abstract representation
  - A sequence of elements, where the last element is the top of the stack.

- Abstraction Function →
- Maps the (Stack[], top) with [Stack[0], Stack[1] . . . Stack[top])

- If top = -1, the Stack is empty, corresponding to the abstract state [ ].

Features of OOP

① Encapsulation → It is the process of bundling the data (fields) and the methods (functions) that operate on the data into a single unit, known as a class. It restricts direct access to some of the object's components, ensuring that data is not accessed or modified arbitrarily.

- Provides data hiding by making fields private and exposing them via public methods (getters and setters).
- Protects the integrity of the data by controlling access.
- Helps achieve modularity and separation of concerns.

ex)

```
class BankAccount {
    private double balance;

    Public BankAccount(double initialBalance)
    {
        this.balance = initialBalance;
    }
    Public double getBalance(){
        return balance;
    }
```

```
public void deposit(double amount){
    if(amount>0){
        balance = balance + amount;
    }
}
```

Here the "balance" is private and cannot be accessed directly, ensuring controlled access through methods.

Object Identity

↳ It refers to the unique identity of an Object that distinguishes it from other objects, even if the objects have the same data or state.

ex)

```
class Car{
    String model;
    String color;
    public Car(String model, String color){
        this.model = model;
        this.color = color;
    }
    public static void main(String[] args){
        Car car1 = new Car("X", "Red");
        Car car2 = new Car("X", "Red");
        print(car1 == car2);
```

Samyak

print(car1.equals(car2));

3

3
→ False
→ False

As, car1, car2 are different.
Also, default equals methods compares reference, not data.

Polymorphism - But Not Inheritance

- Polymorphism in OOP, that allows a single function, method, or operator to work in different ways based on the context.
It means "many forms". It allows objects of different classes to be treated as objects of a common superclass.

ⓐ Compile Time Polymorphism (STATIC)
↳ Acheived through method overloading and operator overloading.

ⓑ Runtime Polymorphism (DYNAMIC)
↳ Acheived through method overriding.

→ While Inheritance is a common way to acheive polymorphism (via method overriding in subclasses), but polymorphism can also be acheived through

interfaces, abstract classes, or delegation.

Polymorphism using Interfaces

```
interface Animal {
    void makesound();
}
class Dog implements Animal {
    public void makesound() {
        Sout("woof");
    }
}
class cat implements Animal {
    Public makesound() {
        void
        Sout("Meow");
    }
}
public class Main {
    public static void main(String args[]){
        Animal myDog = new Dog();
        Polymorphism
        Animal mycat = new Cat();
        myDog.makeSound();
    }
}
```

here Dog, Cat implement the Animal interface, but they don't share any inheritance relationship.

We achieve polymorphism because both classes have a makesound() method, but their behaviour is different.

Polymorphism using Abstract Classes.

- An abstract class is a class that cannot be instantiated on its own and may contain abstract methods (methods without a body) that must be implemented by subclasses.

Samyak

ex)

```
abstract class Animal {
        abstract void makesound();
}
class Dog extends Animal {
        void makesound() {
                Sout("woof");
        }
}
Public class Main {
        PSVM {
                Animal myDog = new Dog();
                myDog.makesound();
        }
}
```

- Abstract classes allow polymorphism without requiring inheritance from a concrete class, especially when the abstract class provides a base implementation of some methods.

- Polymorphism without inheritance promotes loose coupling between classes, making our code more modular and maintainable.

Inheritance in OO Design

→ It allows one class (called the child class or subclass) to acquire the properties (fields) and behaviors (methods) of another class (parent or superclass). It represents an "is-a" relationship where the subclass is a specialized version of the superclass.

Types of Inheritance

1. Single Inheritance
   → A subclass inherits from one superclass.

ex)

```
class Animal {
    void eat() {
        Sout("Eating");
    }
}
class Dog extends Animal {
    void bark() {
        Sout("Woof");
    }
}
Public class Main {
    PSVM {
        Dog dog = new Dog();
        dog.eat();
        dog.bark();
    }
}
```

---

② Multi-Level Inheritance

↳ Class inherits from a class, which itself inherits from another class.

ex)

```
class Animal {
    void eat() {
        Sout("Eating");
    }
}
class Mammal extends Animal {
    void walk() {
        Sout("Walk");
    }
}
```

```
Class Dog extends Mammal {
        void bark() {
            sout("Barking");
        }
}
Public class Main {
    PSVm {
        Dog dog = new Dog();
        dog.eat();
        dog.walk();
        dog.bark();
    }
}
```

---

③ Heirarchial Inheritance

→ Multiple subclasses inherit from a single superclass.

---

④ Multiple Inheritance (via Interfaces in JAVA).

→

ex)

```
interface Flyable {
    void fly();
}
interface Animal {
    void eat();
}
class Bird implements Flyable, Animal {
    Public void fly() {
        sout("FLY");
    }
```

```
    public void eat() {
        Sout(" Eats");
    }
}
public class Main {
    PSVM {
        Bird b = new Bird();
        b.fly();
        b.eat();
    }
}
```

Method Overriding in Inheritance

ex)

```
class Animal {
    void sound() {
        Sout("Some Sound");
    }
}
class Dog extends Animal {
    @Override
    void sound() {
        Sout("bark");
    }
}
public class Main {
    PSVM {
        Animal a = new Dog();
        a.sound();
    }
}
```

bark

↳ Bcoz of Runtime Polymorphism

Samyak

Using the Super Keyword

→ Super is used to

ⓐ Access a superclass's constructor.

ⓑ Access a superclass's methods or fields.

ex)

```
class Animal {
    String name;
    Animal (String name) {
        this.name = name;
    }
    void displayName() {
        Sout ("Animal " + name);
    }
}
class Dog extends Animal {
    Dog(String name) {
        super(name);
    }
    @Override
    void displayName() {
        super.displayName();
        Sout ("This is a dog");
    }
}
Public class Main {
    Psvm {
        Dog dog = new Dog("Buddy");
        dog.displayName();
    }
}
```

Samyak

Super Keyword → It is a keyword in Java and is a reference variable that is used to access members (methods or constructors) of the immediate parent class. It is primarily used in inheritance to call the parent class's constructor or methods and differentiate them from the subclass's members when they are overridden or shadowed.

Limitations of Inheritance

① Tight Coupling → making changes in the superclass affects all the subclasses.

② Inheritance vs Composition → Sometimes, composition ("has-a" relationship) is more flexible than Inheritance ("is-a" relationship)

Design Patterns

↳ They are proven, reusable solutions to common problems that occur in software design. They are like blueprints for solving specific design issues in a structured and efficient way.

They promote :→

① Reusability ② Maintainability
③ Communication ④ Efficiency

→ ~~Introduced~~ Popularized by the GIOF → Gang of Four.

Samyak

Classifications

(a) Creational Patterns

↳ They deal with object creation mechanisms. They abstract the instantiation process, making a system independent of how objects are created, composed, and represented.

ex) Singleton, Factory Method, Abstract Factory, Builder, Prototype etc.

Singleton Pattern :→

```
class Singleton {
    private static Singleton instance;
    private Singleton() {
    }
    Public static Singleton getInstance() {
        if(instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

(b) Structural Patterns

→ They focus on the composition of classes and objects. They define how classes and objects can be

combined to form larger structures.

→ Goal is to simplify relationships between entities.

ex) Adapter, Bridge, Composite, Decorator, Facade, Proxy, Flyweight.

© Behavioral Patterns

→ Behavioral Patterns focus on communication and interaction between objects. They help define how objects collaborate to perform tasks.

Goal is to simplify communication between objects and make them loosely coupled.

ex) Observer, Strategy, Command, Iterator, Mediator, State, Template Method, Chain of Responsibility.

The Iterator Design Pattern

→ The Iterator Pattern is a behavioral design pattern that provides a way to sequentially access elements of a collection (like lists, arrays, or other aggregates) without exposing its underlying representation.

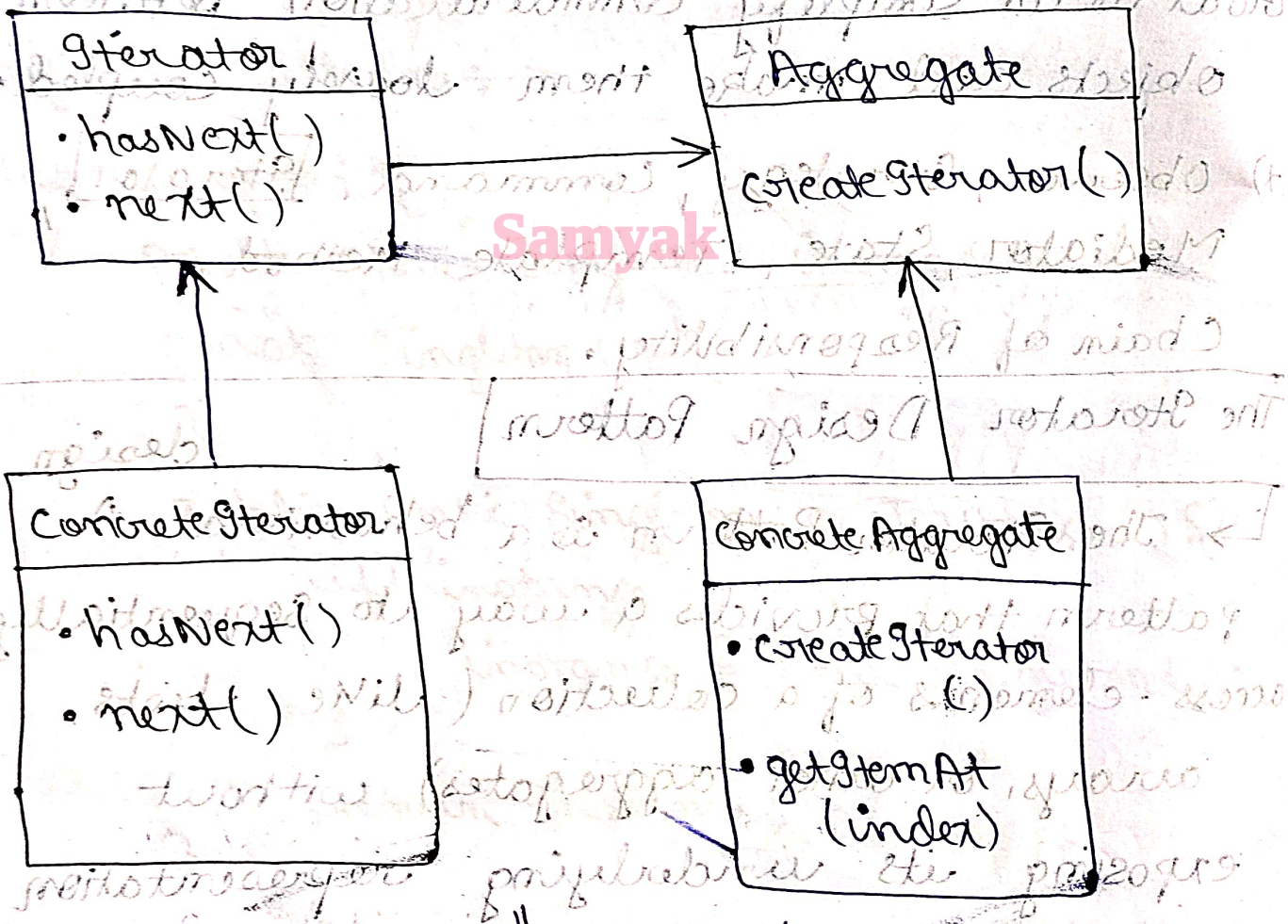
It decouples the logic for traversing the collection from the actual collection structure, making it easier to iterate over various data structures in a uniform way.

The pattern consists of two main parts →

① Iterator → An object responsible for accessing and traversing elements in the collection.

② Aggregate (collection) → The data structure (like a list or array) that holds the elements and provides a method to get an iterator.

Class Diagram



Samyak

→ Java provides the "Iterator" interface in the java.util package. Collections like ArrayList, HashSet provide built-in iterators.

Code

```
import java.util.ArrayList;
import java.util.Iterator;
public class Main {
    psvm {
    ArrayList<String> books = new ArrayList
                              <>();
    books.add("B1");
    books.add("B2");
    Iterator<String> iterator = books.iterator();
    while (iterator.hasNext()) {
        sout(iterator.next());
    }
}
}
```

Output
B1
B2

Without using built-in Iterator

```
class BookCollection {
    private String[] books;
    private int size;
    public BookCollection(int capacity) {
        books = new String[capacity];
        size = 0;
    }
    public void addBook(String book) {
        if(size < books.length) {
            books[size++] = book;
        }
    }
```

```
    public BookIterator iterator() {
        return new BookIterator
                (books, size);
    }
}
class BookIterator {
    private String[] books;
    private int position;
    private int size;
    public BookIterator(String[] books,
                        int size) {
        this.books = books;
        this.position = 0;
        this.size = size;
    }
    public boolean hasNext() {
        return position < size;
    }
    public String next() {
        return books[position++];
    }
}
public class Main {
    psvm {
        BookCollection c = new BookCollection(5);
        c.addBook("B1");
        c.addBook("B2");
        BookIterator i = c.i();
        while(i.hasNext()) {
            sout(i.next());
        }
    }
}
```

Samyak

Output
B1
B2

Iterable	Iterator
ⓐ Represents a collection that can be iterated. ⓑ java.lang.Iterable. ⓒ iterator() ⓓ Represents a collection ex) ArrayList	ⓐ Objects used to traverse the elements of a collection. ⓑ java.util.Iterator. ⓒ hasNext(), next(), remove() ⓓ Represents the mechanism to traverse a collection

Static Polymorphism	Dynamic Polymorphism
ⓐ Resolved at compile-time ⓑ Achieved through method overloading / operator overloading. ⓒ Early binding ⓓ Same method names with different parameter lists ⓔ Less flexible	ⓐ Resolved at runtime. ⓑ Achieved through method overriding. ⓒ Late binding ⓓ Same method name with the same parameter list in a subclass. ⓔ More Flexible

Samyak

⑦ Inheritance is not necessary	⑦ Inheritance is mandatory
⑧ Faster Performance	⑧ Slightly Slower

Model-View-Controller MVC

→ It is a design pattern that divides an application into 3 interconnected components
→ Model, View and Controller.

Purpose

- Each component has a distinct role, reducing the complexity of managing the application.
- Easier to scale applications by working on individual components independently.
- Testability → Simplifies testing as the business logic (Model) is separate from the user interface (View).

Key Components

① Model →
- Represents the data and the business logic of the application.

- Responsible for fetching, processing and updating data.
- Contains no information about the UI.
- Notifies the View of changes so the UI can update itself.

ex)
```
Public class Student {
    private String name;
    private int rollNO;
    public Student(String name,
                   int rollNO) {
        this.name = name;
        this.rollNO = rollNO;
    }
    Public String getName() {
        return name;
    }
    Public void setName(String
                        name) {
        this.name = name;
    }
}
```

② View →

- Represents the UI.
- Displays data from the Model to the user.
- Contains no logic for processing data; it simply renders the data.

provided by the Model.

- updates dynamically when the Model changes.

ex)

```
Public class StudentView {
    Public void printDetails( String
        StudentName, int rollNo) {
        Sout(" Student: ");
        Sout(" Name : " + StudenName);
        Sout(" Roll: " + rollNo);
    }
}
```

Samyak

③ Controller →

- Acts as an intermediary between the Model and the View.
- Handles user input, processes it (using the Model), and updates the View Accordingly.
- Contains the logic for application behavior.

ex)

```
public class StudentController {
    private Student model;
    private StudentView view;
```

```
Public StudentController(Student model,
                         StudentView view) {
    this.model = model;
    this.view = view;
}
Public void SetStudentName(String name) {
    model.setName(name);
}
Public String getStudentName() {
    return model.getName();
}
Public int getStudentRollNo() {
    return model.getRollNo();
}
Public void updateView() {
    view.printStudentDetails
        (model.getName(),
         model.getRollNo());
}
```

Samyak

How MVC pattern Works?

① User Interaction → The user interacts with the application via the View. (ex. clicking a button, entering text)

② Controller Handles Input → The controller processes

user input and updates the Model accordingly.

③ Model Updates
Data → The Model updates its State and notifies the View of any changes.

④ View Renders
Updated Data → The View queries the Model for updated data and re-renders the UI.

Complete Example of MVC

```
Public class MVCPattern {
    PSVm {
        Student model = retrieveStudent
                        FromDatabase();
        StudentView view = new StudentView();
        StudentController c = new StudentController
                              (model, view);
        c.updateView();
        c.setStudentName("Samyak");
        c.updateView();
    }
    private static Student retrieveStudentFrom
                   Database() {
        return new Student("Mishra",
    }
}
```

Samyak

Output

Student :

Name : ~~Xxxx~~ Mishra

RollNO : 100

Student

Name : Samyak

RollNO : 100

Revise MVC From Organizer for Questions.

Commands as Methods and as Objects

Commands as Methods

- A method is a function defined within a class that encapsulates a specific operation or behavior.
- When commands are implemented as methods, they are directly tied to the class they belong to.
- Commands are tightly coupled with the object they act upon.
- They operate on the data (fields) of the class they belong to.
- The behavior is fixed and directly implemented in the method.

Samyak

ex)

```
Class Light {
    public void turnOn() {
        Sout("ON");
    }
    public void turnoff() {
        Sout("OFF");
    }
}
Public class Main {
    PSVm {
        Light l = new light();
        l.turnON();
        l.turnOFF();
    }
}
```

command as a method.

Command as Objects

- A command object encapsulates a request or action as a standalone object.
- This object contains all the information needed to execute the command, such as the target object and any parameters.
- Commands are decoupled from the object they act upon.
- They can be stored, passed as arguments, or executed later.

- Useful in scenarios where actions need to be queued, logged, or executed in a transactional way.

ex)

```
interface Command {
        void execute();
}
class TurnOnLightCommand implements
                                Command {
        private Light light;
        public TurnOnLightCommand(Light
                                light){
            this.light = light;
        }
        @Override
        public void execute() {
            light.turnOn();
        }
}
class TurnOffLightCommand implements
                                Command {
        private Light light;
        public TurnOffLightCommand(Light
                                light) {
            this.light = light;
        }
        @
        public void execute() {
            light.turnOff();
        }
}
```

```
class Light {
    public void turnOn() {
        System.out.
            println("ON");
    }
    public void turnoff() {
        sout("OFF");
    }
}
class RemoteControl {
    private Command command;
    public void setCommand(
            Command command) {
        this.command = command;
    }
    public void pressButton() {
        command.execute();
    }
}
public class Main {
    PSVM {
    Light light = new Light();
    Command turnon = new
        TurnonLightCommand(light);
    Command turnoff = new
        TurnoffLightCommand(light);
    RemoteControl remote
        = new RemoteControl();
```

Samyak

```
remote.setCommand(turnOn);
remote.pressButton();
remote.setCommand(turnOff);
remote.pressButton();
}
}
```

Commands as Methods	Commands as Objects
(a) Tightly coupled to the object.	(a) Decoupled from the object
(b) Limited Reusability	(b) High Reusability
(c) Simple operations	(c) complex scenarios like queues or logs.
(d) Low complexity	(d) Higher complexity

Memory Management in JAVA → From organizer and PDF

Samyak