OOD (Object Oriented Design)

SOLID Principles (Object-Oriented Design)

Definition:
SOLID is a set of **5 object-oriented design principles** that help in writing **clean, maintainable, scalable, and flexible code.**
SOLID = **S O L I D**

Why SOLID is Needed / Purpose
Without SOLID:
- Code becomes tightly coupled
- Hard to modify or test
- Small change breaks many classes

With SOLID:
- Easy maintenance
- Better readability
- Easier testing
- Industry-standard design

S – Single Responsibility Principle (SRP)

Definition
A class should have **only one reason to change.**
Key Point
One class = one responsibility
Bad Example

```
class Employee {
    void calculateSalary() {}
    void saveToDatabase() {}
}
```

Good Example

```
class SalaryCalculator {}
class EmployeeRepository {}
```

Real-Life Example
One person cannot be **teacher + accountant + driver** at same time efficiently.

O – Open/Closed Principle (OCP)

Definition
Software entities should be **open for extension but closed for modification.**
Key Point
Add new behavior without changing existing code.
Example

```
interface Shape {
    double area();
}
class Circle implements Shape {}
class Rectangle implements Shape {}
```

Real-Life Example
Mobile apps add features via **updates**, not rewriting whole app.

L – Liskov Substitution Principle (LSP)

Definition
A child class should be **usable in place of its parent class** without breaking the program.

Key Point
Subclass must not change parent behavior.
Bad Example
Square extends Rectangle (changes width/height logic)
Real-Life Example
If a **car** is a **vehicle**, it must behave like a vehicle.

I – Interface Segregation Principle (ISP)
Definition
Clients should not be forced to implement **unused methods**.
Key Point
Many small interfaces > one big interface
Bad Example

```
interface Worker {
    void work();
    void eat();
}
```

Good Example

```
interface Workable { void work(); }
interface Eatable { void eat(); }
```

Real-Life Example
A robot works but does not eat.

D – Dependency Inversion Principle (DIP)
Definition
High-level modules should not depend on low-level modules; both should depend on **abstractions**.
Key Point
Depend on interfaces, not concrete classes.
Example

```
class Car {
    Engine engine;
    Car(Engine engine) {
        this.engine = engine;
    }
}
```

Real-Life Example
Switch works with **electricity**, not a specific power plant.

Advantages of SOLID

- Loose coupling
- High cohesion
- Easy testing
- Scalable design
- Used in Spring, Hibernate

Common Interview Questions (Cognizant Level)
Q1. What does SOLID stand for?
A. Five OOP design principles.
Q2. Which principle is most violated in beginners' code?
A. SRP.
Q3. Which principle is used in Spring Dependency Injection?
A. DIP.

Q4. Difference between SRP and ISP?
A. SRP → class responsibility
ISP → interface responsibility
Q5. Which principle improves extensibility?
A. OCP.

One-Line Summary (Quick Revision)
SOLID principles help in designing flexible, maintainable, and scalable object-oriented software.

Singleton Design Pattern

Definition:
Singleton Pattern ensures that **only one object (instance)** of a class is created and provides a **global access point** to that instance.

Why it is Needed / Purpose
Singleton is used when:
- Only one object is required for the entire application
- Shared resources must be controlled

Examples:
- Database connection
- Logger
- Configuration settings

Core Idea (Key Rules)
1. **Private constructor** – prevents object creation from outside
2. **Static instance variable** – stores single object
3. **Public static method** – returns the same instance

How It Works (Step-by-Step)
1. Constructor is made private
2. Class creates one static object
3. Every call returns the same object reference

Real-Life Example (Easy Analogy)
- **Government Aadhaar Server** → Only one central system
- **Printer in office** → One shared printer for all users

Basic Singleton Implementation (Eager Initialization)
```
class Singleton {
    private static Singleton instance = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return instance;
    }
}
```

Lazy Initialization (Most Asked)
```
class Singleton {
```

```
    private static Singleton instance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

Thread-Safe Singleton

```
class Singleton {
    private static Singleton instance;

    private Singleton() {}

    public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

Best Practice (Double-Checked Locking)

```
class Singleton {
    private static volatile Singleton instance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

Advantages

- Controlled access to single instance
- Saves memory
- Useful for shared resources

Disadvantages / Limitations

- Difficult to test (mocking issues)
- Global state can cause hidden dependencies
- Breaks SOLID (SRP & DIP) if overused

Singleton vs Normal Class

Feature	Singleton	Normal Class
Instances	One only	Multiple
Constructor	Private	Public
Use case	Shared resource	General objects

Common Interview Questions (Cognizant Level)

Q1. What is Singleton Pattern?
A. Ensures only one instance of a class.
Q2. Why constructor is private in Singleton?
A. To prevent object creation using new.
Q3. Is Singleton thread-safe by default?
A. No.
Q4. How to make Singleton thread-safe?
A. Synchronization or double-checked locking.
Q5. Where is Singleton used in real projects?
A. Logging, database connections, configuration.

One-Line Summary (Quick Revision)

Singleton Pattern restricts a class to only one object and provides a global access point to it.

Factory Design Pattern

Definition:
Factory Pattern is a **creational design pattern** that provides an **object-creation method** without exposing the creation logic and returns objects using a **common interface or parent class**.
Client asks for object, factory decides which object to create.

Why it is Needed / Purpose

Without Factory:
- Client uses new everywhere
- Code becomes tightly coupled
- Hard to add new object types

With Factory:
- Loose coupling
- Centralized object creation
- Easy to extend

Core Idea (Pattern Rules)

1. Create a **common interface / abstract class**
2. Implement multiple concrete classes
3. Create a **Factory class** to return objects
4. Client uses factory, not new

How It Works (Flow Pattern)

Client → Factory → Concrete Object
Client doesn’t know which class is created.

Real-Life Example (Easy Analogy)

- **Restaurant** → You order food, kitchen decides how to prepare it
- **Mobile Factory** → Factory produces Android or iPhone based on request

---

Simple Factory Example (Core Interview Part)

Step 1: Common Interface

```
interface Notification {
    void notifyUser();
}
```

Step 2: Implementations

```
class EmailNotification implements Notification {
    public void notifyUser() {
        System.out.println("Email sent");
    }
}

class SMSNotification implements Notification {
    public void notifyUser() {
        System.out.println("SMS sent");
    }
}
```

Step 3: Factory Class

```
class NotificationFactory {
    public static Notification createNotification(String type) {
        if (type.equalsIgnoreCase("EMAIL"))
            return new EmailNotification();
        else if (type.equalsIgnoreCase("SMS"))
            return new SMSNotification();
        return null;
    }
}
```

Step 4: Client Code

```
Notification n =
    NotificationFactory.createNotification("EMAIL");
n.notifyUser();
```

Advantages

- Loose coupling
- Centralized creation logic
- Easy to add new classes
- Follows **Open/Closed Principle**

Disadvantages / Limitations

- Extra classes increase complexity
- Factory may grow large if many types added

Factory vs Singleton

Feature	Factory	Singleton
Purpose	Create objects	Restrict object
Objects	Multiple types	Single instance
Focus	Creation logic	Instance control

Factory vs Direct Object Creation

Direct (new)	Factory
Tight coupling	Loose coupling
Hard to extend	Easy to extend
Client controls creation	Factory controls creation

Real-Time Usage

- Spring BeanFactory
- Logger creation
- Database connection types
- Payment gateways

Common Interview Questions (Cognizant Level)

Q1. What problem does Factory Pattern solve?
A. Removes tight coupling from object creation.
Q2. Which SOLID principle does Factory follow?
A. Open/Closed Principle.
Q3. Can Factory return different objects?
A. Yes, based on input.
Q4. Is Factory Pattern creational?
A. Yes.
Q5. Where is Factory used in Spring?
A. BeanFactory and ApplicationContext.

One-Line Summary (Quick Revision)

Factory Pattern creates objects without exposing creation logic and returns them via a common interface.

Builder Design Pattern

Definition

Builder Pattern is a **creational design pattern** used to **construct complex objects step by step**, without exposing the construction logic.

- It separates **object creation** from **object representation**.

Why it is Needed / Purpose

Problem with constructors:

- Too many parameters
- Confusing order
- Hard to read and maintain

Builder Pattern helps to:

- Create objects in a clean way
- Avoid constructor overloading
- Improve readability

When to Use Builder Pattern

- Object has **many optional parameters**
- Immutable objects needed
- Complex object creation

Examples:

- User profile

- Configuration objects
- HTTP request builders

Core Idea (Pattern Rules)

1. Create a **static inner Builder class**
2. Builder sets values step by step
3. build() method returns final object
4. Object is usually immutable

How It Works (Flow Pattern)

Client → Builder → build() → Object

Real-Life Example (Easy Analogy)

- **Ordering Burger** → Choose bun, cheese, toppings step by step
- **House Construction** → Build room by room

Builder Pattern Example (Most Important)

Step 1: Main Class

```
class User {
    private final String name;
    private final int age;
    private final String email;

    private User(Builder builder) {
        this.name = builder.name;
        this.age = builder.age;
        this.email = builder.email;
    }
```

Step 2: Builder Class

```
    static class Builder {
        private String name;
        private int age;
        private String email;

        Builder setName(String name) {
            this.name = name;
            return this;
        }

        Builder setAge(int age) {
            this.age = age;
            return this;
        }

        Builder setEmail(String email) {
            this.email = email;
            return this;
        }

        User build() {
            return new User(this);
        }
    }
}
```

Step 3: Client Code

```
User user = new User.Builder()
                .setName("Ali")
                .setAge(22)
                .setEmail("ali@gmail.com")
                .build();
```

Advantages

- Clean and readable object creation
- Handles optional parameters well
- Supports immutability
- Avoids constructor confusion

Disadvantages / Limitations

- More code
- Not suitable for simple objects

Builder vs Factory

Feature	Builder	Factory
Purpose	Build complex object	Create object types
Parameters	Many optional	Few
Process	Step-by-step	One-step

Builder vs Constructor

Constructor	Builder
Hard to read	Easy to read
Parameter order matters	Order doesn't matter
Not flexible	Flexible

Real-Time Usage

- StringBuilder
- Lombok @Builder
- HTTP request creation
- Spring configuration

Common Interview Questions (Cognizant Level)

Q1. Why Builder Pattern is better than constructor?
A. Readability and flexibility.

Q2. Is Builder Pattern immutable?
A. Yes, usually.

Q3. Which design pattern is used in StringBuilder?
A. Builder Pattern.

Q4. When should you avoid Builder?
A. When object is simple.

Q5. Is Builder a creational pattern?
A. Yes.

One-Line Summary (Quick Revision)

Builder Pattern constructs complex objects step by step with clean, readable, and flexible code.

Behavioral Design Patterns

Definition

Behavioral Design Patterns focus on **how objects interact and communicate** with each other and **how responsibilities are distributed** among them.

- They mainly deal with **object behavior and communication.**

Why Behavioral Patterns are Needed / Purpose

Without behavioral patterns:

- Tight coupling between objects
- Hard-to-change behavior
- Complex conditional logic (if-else)

With behavioral patterns:

- Flexible communication
- Easy behavior change
- Clean responsibility separation

Common Behavioral Design Patterns

1. **Strategy**
2. **Observer**
3. **Command**
4. **Iterator**
5. **State**
6. **Template Method**
7. **Chain of Responsibility**
8. **Mediator**
9. **Memento**
10. **Visitor**

*(For interviews, **Strategy**, **Observer**, **Command** are most important)*

---

1. **Strategy Pattern**

   Definition

   Defines a **family of algorithms**, encapsulates them, and makes them **interchangeable at runtime.**

   Example

   Payment methods: Card, UPI, Cash

   Key Benefit

   - Avoids if-else chains

```
interface Payment {
    void pay();
}
class CardPayment implements Payment {}
class UpiPayment implements Payment {}
```

---

2. **Observer Pattern**

   Definition

   Creates a **one-to-many dependency** so that when one object changes state, all dependents are notified.

   Example

   YouTube subscribers get notified on new video

   Key Benefit

   - Loose coupling

```
interface Observer {
    void update();
}
```

3. **Command Pattern**
   Definition
   Encapsulates a **request as an object,** allowing parameterization and queuing.
   Example
   Remote control buttons
   Key Benefit
   Decouples sender from receiver

---

4. **Iterator Pattern**
   Definition
   Provides a way to **access elements of a collection sequentially** without exposing internal structure.
   Example
   Java Iterator

---

5. **State Pattern**
   Definition
   Allows an object to **change its behavior when its internal state changes.**
   Example
   ATM machine states

---

6. **Template Method Pattern**
   Definition
   Defines the **skeleton of an algorithm** but lets subclasses define specific steps.
   Example
   Game level logic

Advantages of Behavioral Patterns

- Loose coupling
- Easy extension
- Better code readability
- Clean separation of responsibilities

Behavioral vs Creational vs Structural

Type	Focus
Creational	Object creation
Structural	Class relationships
Behavioral	Object interaction

Common Interview Questions (Cognizant Level)

Q1. What are Behavioral Patterns?
A. Patterns that manage object behavior and interaction.
Q2. Most used behavioral pattern?
A. Strategy and Observer.
Q3. Which pattern avoids if-else logic?
A. Strategy Pattern.
Q4. Observer real-time example?
A. Event listeners, notifications.
Q5. Which pattern Java Iterator uses?
A. Iterator Pattern.

One-Line Summary (Quick Revision)
Behavioral Design Patterns define how objects communicate and manage behavior efficiently.

Structural Design Patterns

Definition
Structural Design Patterns deal with **how classes and objects are composed** to form **larger, flexible, and efficient structures**.

- Focus is on **class relationships and structure**, not creation or behavior.

Why Structural Patterns are Needed / Purpose
Without structural patterns:

- Tight coupling between classes
- Difficult to extend existing code
- Complex class hierarchies

With structural patterns:

- Reusability
- Flexibility
- Clean object composition

Common Structural Design Patterns

1. **Adapter**
2. **Decorator**
3. **Facade**
4. **Composite**
5. **Proxy**
6. **Bridge**
7. **Flyweight**

(For interviews: Adapter, Decorator, Facade, Proxy are most important)

---

1. **Adapter Pattern**
   Definition
   Converts the **interface of a class into another interface** that a client expects.
   Example
   Charger adapter (3-pin to 2-pin)
   Key Benefit
   - Makes incompatible interfaces work together

   class Adapter implements Target {}

---

2. **Decorator Pattern**
   Definition
   Adds **new behavior to an object dynamically** without modifying its structure.
   Example
   Adding cheese or toppings to pizza
   Key Benefit
   - Avoids subclass explosion

   class CheeseDecorator extends PizzaDecorator {}

---

3. **Facade Pattern**
   Definition
   Provides a **simplified interface** to a complex subsystem.

Example
Using a TV remote instead of operating internal circuits
Key Benefit
- Hides complexity

class BankingFacade {}

---

4. **Composite Pattern**
   Definition
   Treats **individual objects and groups of objects uniformly.**
   Example
   File system (file & folder)
   Key Benefit
   Hierarchical structure

---

5. **Proxy Pattern**
   Definition
   Provides a **placeholder or control object** to manage access to another object.
   Example
   ATM card (proxy to bank account)
   Key Benefit
   - Access control / lazy loading

   class ProxyService implements Service {}

---

6. **Bridge Pattern**
   Definition
   Separates **abstraction from implementation** so both can vary independently.
   Example
   Remote and TV brands

---

7. **Flyweight Pattern**
   Definition
   Reduces memory usage by **sharing common objects.**
   Example
   String pool in Java

Structural vs Creational vs Behavioral

Pattern Type	Focus
Creational	Object creation
Structural	Class composition
Behavioral	Object interaction

Advantages of Structural Patterns
- Reduced complexity
- Better code reuse
- Loose coupling
- Improved maintainability

Common Interview Questions (Cognizant Level)
Q1. What are Structural Design Patterns?
A. Patterns that define class and object structure.
Q2. Which pattern hides complexity?
A. Facade Pattern.
Q3. Which pattern adds behavior dynamically?
A. Decorator Pattern.

Q4. Which pattern converts one interface to another?
A. Adapter Pattern.
Q5. Which pattern controls access to object?
A. Proxy Pattern.

One-Line Summary (Quick Revision)
Structural Design Patterns define how classes and objects are composed to build flexible software structures.

Introduction to UML (Unified Modeling Language)

Definition
UML (Unified Modeling Language) is a **standard visual language** used to **design, visualize, and document** software systems before actual coding.

- It represents **how a system is structured and how it behaves** using diagrams.

Why UML is Needed / Purpose
UML helps to:

- Understand system requirements clearly
- Communicate ideas between developers, testers, and clients
- Reduce errors before coding
- Plan large systems efficiently

Where UML is Used

- Software design phase
- System documentation
- Enterprise applications
- Object-Oriented systems

Core Building Blocks of UML

1. **Things** – classes, objects, components
2. **Relationships** – association, inheritance, dependency
3. **Diagrams** – visual representation

Types of UML Diagrams (Very Important)
UML diagrams are mainly divided into **two categories**:

Structural Diagrams *(What the system IS)*

1. **Class Diagram** ⭐⭐⭐
2. Object Diagram
3. Component Diagram
4. Deployment Diagram
5. Package Diagram

Most important for interviews: **Class Diagram**

Behavioral Diagrams *(What the system DOES)*

1. **Use Case Diagram** ⭐⭐⭐
2. Sequence Diagram ⭐⭐
3. Activity Diagram
4. State Diagram

Most important for interviews: **Use Case & Sequence**

Class Diagram

Purpose

Shows **classes, attributes, methods, and relationships.**

Structure

```
----------------
| Class Name   |
----------------
| variables    |
----------------
| methods()    |
----------------
```

Example

```
Student
- id
- name
+ getName()
```

Use Case Diagram

Purpose

Shows **user interactions** with the system.

Components

- Actor (User)
- Use case (Action)
- System boundary

Example

Student → Login → View Result

Sequence Diagram

Purpose

Shows **interaction between objects over time.**

Key Elements

- Lifeline
- Messages
- Activation box

Activity Diagram

Purpose

Shows **workflow or process flow.**

Example

Login → Validate → Success / Failure

UML Relationships (Very Important)

Relationship	Meaning
Association	Uses
Aggregation	Has-a (weak)
Composition	Has-a (strong)
Inheritance	Is-a
Dependency	Temporary use

Real-Life Example

- **Blueprint of a building** before construction
- **Flowchart** before writing code

Advantages of UML

- Clear visualization
- Better communication
- Saves development time
- Reduces misunderstandings

Limitations of UML

- Time-consuming for small projects
- Requires practice to master
- Over-design risk

Common Interview Questions (Cognizant Level)

Q1. What is UML?
A. A visual modeling language for software design.
Q2. Why UML is used before coding?
A. To plan and visualize the system.
Q3. Most important UML diagram?
A. Class Diagram.
Q4. Difference between Use Case and Sequence diagram?
A. Use Case → user view
Sequence → object interaction
Q5. UML used only in Java?
A. No, language-independent.

One-Line Summary (Quick Revision)

UML is a standard modeling language used to visualize, design, and document software systems.

UML Diagram Types

Definition

UML Diagram Types are different kinds of diagrams used to **visually represent the structure and behavior** of a software system.

- Each diagram answers a **specific question** about the system.

Why UML Diagrams are Needed

UML diagrams help to:

- Understand system design clearly
- Communicate ideas easily
- Reduce development errors
- Document the system

Classification of UML Diagrams

UML diagrams are broadly divided into **two main categories**:

A. Structural Diagrams *(What the system IS)*

These diagrams show the **static structure** of the system.

Structural UML Diagrams List

1. **Class Diagram** ⭐⭐⭐
2. Object Diagram

3. Component Diagram
4. Deployment Diagram
5. Package Diagram
6. Composite Structure Diagram

Class Diagram (Most Important)
Purpose:
Shows classes, attributes, methods, and relationships.
Used for:
Designing OOP systems

Object Diagram
Purpose:
Shows objects and their values at a specific time.
Used for:
Debugging and understanding runtime objects

Component Diagram
Purpose:
Shows software components and dependencies.
Used for:
Microservices & large applications

Deployment Diagram
Purpose:
Shows physical deployment of software on hardware.
Used for:
Cloud and server architecture

Package Diagram
Purpose:
Shows package organization and dependencies.
Used for:
Project structure planning

B. Behavioral Diagrams *(What the system DOES)*
These diagrams show **dynamic behavior** of the system.

Behavioral UML Diagrams List
1. **Use Case Diagram** ⭐⭐⭐
2. **Sequence Diagram** ⭐⭐⭐
3. Activity Diagram
4. State Machine Diagram
5. Communication Diagram
6. Timing Diagram
7. Interaction Overview Diagram

Use Case Diagram (Most Important)
Purpose:
Shows user interaction with the system.
Used for:
Requirement gathering

Sequence Diagram
Purpose:
Shows message flow between objects over time.
Used for:
Understanding method calls

Activity Diagram
Purpose:
Shows workflow or business process.
Used for:
Process modeling

State Machine Diagram
Purpose:
Shows state changes of an object.
Used for:
ATM, vending machines

Communication Diagram
Purpose:
Shows object interactions focusing on relationships.

Quick Comparison Table

Diagram	Focus
Class	Structure
Use Case	User view
Sequence	Message flow
Activity	Workflow
Deployment	Hardware

Real-Life Example

- **Class Diagram** → Blueprint of a building
- **Sequence Diagram** → Step-by-step process
- **Deployment Diagram** → Where system is installed

Common Interview Questions (Cognizant Level)
Q1. How many UML diagram types are there?
A. 14 (UML 2.x)
Q2. Most important UML diagrams for interviews?
A. Class, Use Case, Sequence.
Q3. Which diagram shows system structure?
A. Structural diagrams.
Q4. Which diagram shows interaction over time?
A. Sequence diagram.
Q5. UML language dependent?
A. No.

One-Line Summary (Quick Revision)
UML diagram types are used to visualize both the structure and behavior of a software system.

UML Diagram Elements

Definition
UML Diagram Elements are the **basic building blocks** used to create UML diagrams.
They represent **things, relationships, and rules** inside a system.

Why UML Elements are Important
They help to:

- Represent system clearly
- Maintain standard notation
- Avoid confusion in design
- Communicate ideas effectively

Main Categories of UML Elements
UML elements are divided into **4 major groups**:

1. **Structural Elements**
2. **Behavioral Elements**
3. **Grouping Elements**
4. **Annotational Elements**

A. Structural Elements *(Static parts of system)*
Class
Represents a blueprint of objects.
Notation:

```
----------------
| Class Name   |
----------------
| attributes   |
----------------
| methods()    |
----------------
```

Example: Student, Employee

Object
Instance of a class at runtime.
Notation:
student1 : Student

Interface
Defines a contract without implementation.
Notation:
<<interface>> Payment

Component
Represents a modular part of system.
Example: Login Module, Payment Module

Node
Represents physical hardware or server.
Example: Web Server, Database Server

B. Behavioral Elements *(Dynamic parts of system)*

Use Case

Represents a functionality provided by system.

Example: Login, Register, Pay Bill

Interaction

Shows communication between objects.

Used in: Sequence Diagram

State

Represents condition/state of an object.

Example: Logged In, Logged Out

C. Grouping Elements

Package

Groups related UML elements.

Example: com.bank.user

D. Annotational Elements

Note

Used to add explanation or comments.

Notation:

```
-----------------
|  Note text    |
-----------------
```

UML Relationships (Very Important)

a) Association

Basic relationship (uses).

Example: Teacher – Student

---

b) Aggregation

Weak “has-a” relationship.

Symbol: ◇

Example: Department has Employees

---

c) Composition

Strong “has-a” relationship.

Symbol: ◆

Example: House has Rooms

---

d) Inheritance (Generalization)

“Is-a” relationship.

Symbol: ▲

Example: Dog → Animal

---

e) Dependency

Temporary usage relationship.

Example: Method parameter

Visibility Symbols

Symbol	Meaning
+	Public
-	Private
#	Protected
~	Default

Real-Life Analogy

- **Class** → Blueprint
- **Object** → Actual house
- **Package** → Apartment building
- **Note** → Sticky note

Common Interview Questions (Cognizant Level)

Q1. What are UML elements?
A. Building blocks of UML diagrams.
Q2. Which element groups other elements?
A. Package.
Q3. Difference between aggregation and composition?
A. Composition is strong ownership.
Q4. What symbol represents inheritance?
A. Hollow triangle.
Q5. What does a note represent in UML?
A. Explanation or comment.

One-Line Summary (Quick Revision)

UML diagram elements are the basic components used to model the structure and behavior of a system.
