# UNIT–1 : Introduction and Basic Structural Modeling (PYQ Solution)

AKTU · Object Oriented System Design with C++ (BCS054) · Solutions by **Chirag Jain**, Ycotes

## SECTION A Answers (≈70 Words Each)

---

### Q1. State the principles of modeling. (Jan 2026)

**A model is a simplified representation of a system. The four principles are: (1) the choice of model decides how a problem is attacked; (2) every model can be expressed at different levels of precision; (3) the best models are connected to reality; (4) no single model is enough, so a set of nearly independent models is used to view the system.**

---

### Q2. Mention any two advantages of object-oriented approach. (Jan 2026)

1. **Reusability: classes and inheritance let existing code be reused, reducing development time and cost.**
2. **Maintainability and modularity: data and functions are bundled in objects, so changes stay local and the system is easier to extend and debug. Encapsulation also gives better data security.**

---

### Q3. Define data encapsulation. Give example. (2023)

**Data encapsulation is the bundling of data members and the functions that operate on them into one unit (a class), while restricting direct access to the data. Access is only through public methods. Example: class `Account` keeps `balance` private and exposes `deposit()` and `getBalance()`.**

---

### Q4. Define generosity. (2023)

**Generosity (genericity) is the facility to define a class or function with a type as a parameter, so one definition works for many data types. In C++ it is**

**implemented with templates, e.g. `template<class T> T max(T a, T b)`. It promotes reuse and type safety.**

---

### Q5. Differentiate between link and association. (2022)

| Link | Association |
|---|---|
| A connection between two **objects** | A description of a group of links between **classes** |
| Instance-level, exists at run time | Class-level, shown in class diagram |
| Example: Ravi works for Infosys | Example: Person works for Company |

---

### Q6. What is information hiding? (2022)

**Information hiding means hiding the internal implementation details of a class and revealing only a public interface. Private data and internal methods are inaccessible to outside code. It protects data from accidental change and lets the implementation change without affecting users of the class.**

---

### Q7. Define Object identity with example. (2021)

**Object identity is the property by which every object has its own unique existence, independent of the values of its attributes. Two objects with identical attribute values are still distinct. Example: two `Student` objects both named Amit with roll no. 5 are two different objects, each with a separate address or object ID.**

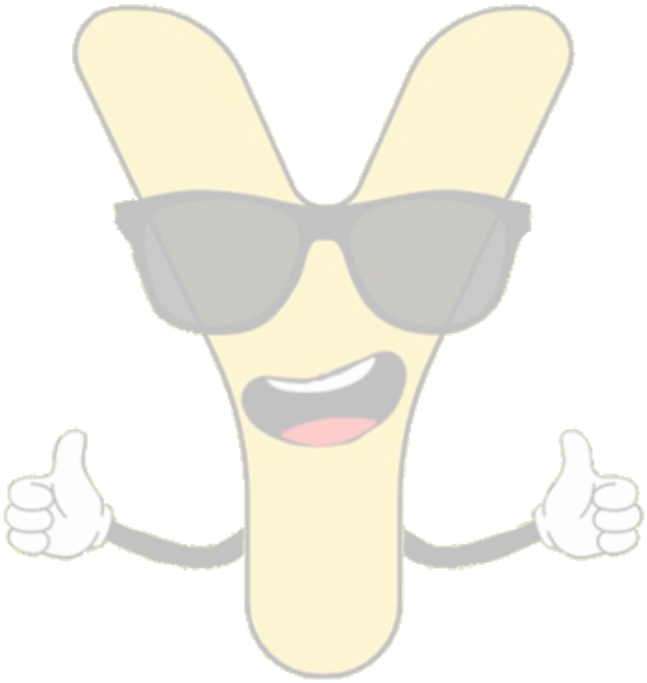
---

### Q8. List the features of Object-oriented paradigms. (2021)

- **Objects and classes**
- **Encapsulation and data abstraction**
- **Information hiding**
- **Inheritance (reuse and hierarchy)**
- **Polymorphism (one interface, many forms)**
- **Message passing and dynamic binding**

- **Object identity and genericity**

---



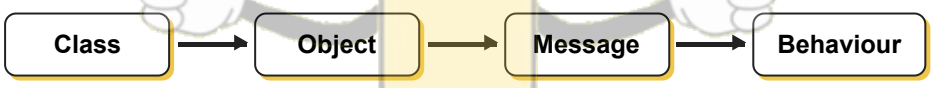
# SECTION B Answers (≈250 Words Each)

### Q1. What do you understand by Object-Oriented Technology? Explain the concept of object orientation. Discuss object identity, information hiding, polymorphism and generosity with suitable examples, and the pros and cons of the technology. (Jan 2026, 2021)

**Object orientation** is a way of building software as a collection of **objects** that combine data (attributes) and behaviour (operations) and communicate by passing messages. Objects are grouped into **classes**, related through inheritance, association and aggregation.

- **Object identity:** each object has a unique identity independent of its attribute values. Two `Car` objects with the same colour and model are still two objects.
- **Information hiding:** internal data is private; only a public interface is exposed. `Account.balance` is private; `deposit()` is public.
- **Polymorphism:** one operation behaves differently for different classes. `area()` of `Circle` and `Rectangle` use different formulas.
- **Generosity (genericity):** classes or functions parameterised by type, e.g. `template<class T> class Stack`.

```
class Shape { public: virtual double area() = 0; };
class Circle : public Shape { double r; public: double area() { return 3.14*r*r; } };
```




*Objects interact by passing messages*

#### Advantages

- Reuse through inheritance and templates
- Easier maintenance and extension (modularity)
- Closer to the real world, so analysis and design map naturally
- Data security through encapsulation

#### Disadvantages

- Steeper learning curve and careful design needed
- Larger programs and some run-time overhead (dynamic binding)
- Poor class design leads to deep hierarchies and tight coupling
- Not suited to very small or purely computational tasks

Example: in a banking system, `Savings` and `Current` accounts inherit from `Account`, sharing code while overriding `withdraw()`.

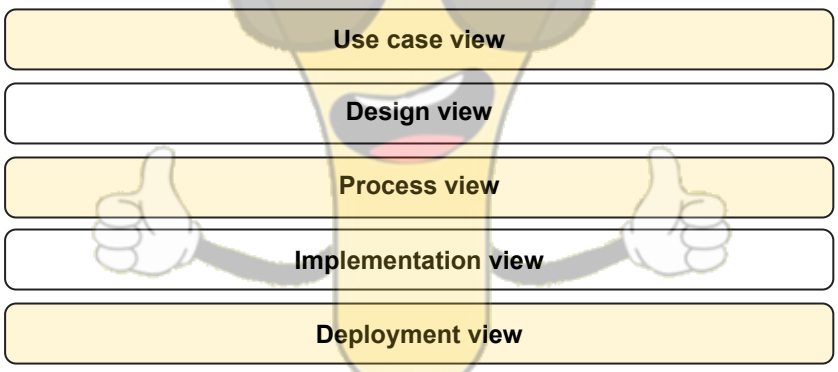
In conclusion, object-oriented technology gives reusable, maintainable software at the cost of design effort.

---

### Q2. Explain the architecture of UML. (2023)

The **Unified Modeling Language (UML)** architecture is described through five complementary **views** of a system.

- **Use case view:** the system's external behaviour as seen by users; the base for all other views.
- **Design view:** classes, interfaces and collaborations that form the solution (structure and behaviour).
- **Process view:** threads and processes, concurrency and synchronisation.
- **Implementation view:** components, files and releases that make up the physical system.
- **Deployment view:** nodes (hardware) on which components are deployed.




*UML architectural views*

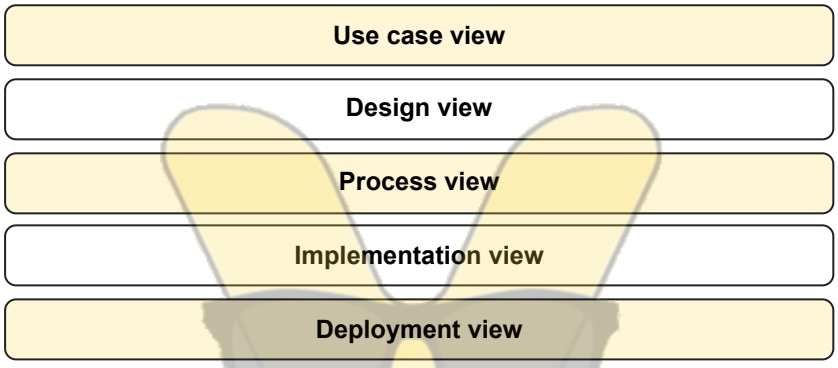
Each view uses static diagrams (class, object, component, deployment) and dynamic diagrams (sequence, collaboration, state, activity). The use case view drives the other four, and together they give a complete picture of the system.

---

# SECTION C Answers (≈500 Words Each)

## Q1. Explain the UML architecture and its relationship with object-oriented analysis and design. (Jan 2026)

The **Unified Modeling Language (UML)** is a standard graphical language for visualising, specifying, constructing and documenting software systems. Its architecture is viewed through five views: **use case, design, process, implementation and deployment**. The use case view captures user requirements and drives the other four.




*Five views of UML architecture*

### Relationship with OOAD

- **Object-oriented analysis (OOA)** finds the problem-domain objects and requirements. UML supports it with use case diagrams, class diagrams (domain model), activity and sequence diagrams.
- **Object-oriented design (OOD)** refines the model into a solution. UML supports it with detailed class diagrams, interaction diagrams, state machines, component and deployment diagrams.
- UML is only a **notation**; it does not prescribe a process. Processes such as the Unified Process use it in iterative phases: inception, elaboration, construction, transition.

| Phase | UML artefact |
|---|---|
| Requirements | Use case diagram |
| Analysis | Class, activity, sequence diagrams |
| Design | Refined class, state, interaction diagrams |
| Implementation | Component diagram |
| Deployment | Deployment diagram |

**Benefits:** a common language for analysts, designers and clients; the same model moves smoothly from analysis to design to code; tools can generate code from models.

**Example:** for a library system, use cases (Issue Book, Return Book) are found in analysis; classes `Book`, `Member`, `Loan` are refined with attributes and operations in design; the sequence diagram shows how objects collaborate to issue a book.

In conclusion, UML provides the vocabulary and architecture through which OOAD models are expressed, communicated and refined.

---

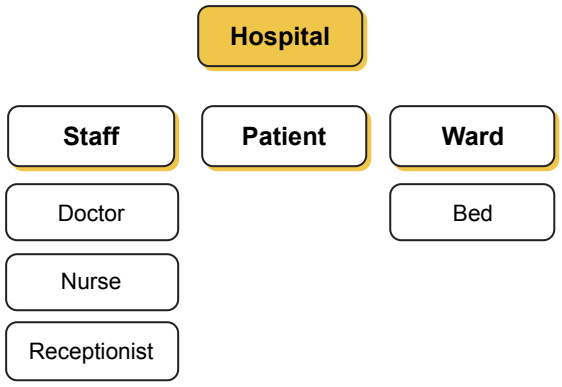
### Q2. Explain how a hospital management system can be designed using object-oriented modeling. Highlight advantages over structured modeling. (Jan 2026)

**Step 1: Requirements and use cases.** Actors: Patient, Doctor, Receptionist, Pharmacist, Admin. Use cases: Register Patient, Book Appointment, Admit Patient, Prescribe Medicine, Generate Bill, Manage Staff.

**Step 2: Identify classes** (from nouns in the requirements).

| Class | Attributes | Operations |
|---|---|---|
| Patient | id, name, age, history | register(), viewReport() |
| Doctor | id, name, speciality | diagnose(), prescribe() |
| Appointment | date, time, status | book(), cancel() |
| Ward/Bed | wardNo, bedNo, occupied | allocate(), release() |
| Bill | billNo, amount | calculate(), pay() |
| Medicine | code, name, stock | issue() |

**Step 3: Relationships.** Patient **books** Appointment with Doctor (association, 1 to many); Hospital **has** Wards (aggregation); Ward **contains** Beds (composition); `Doctor` and `Nurse` generalise `Staff` (inheritance).




*Main classes of the hospital system*

**Step 4: Behaviour.** A sequence diagram for Book Appointment: Receptionist → Patient record → Doctor schedule → Appointment → Bill. A state diagram for Bed: Free → Reserved → Occupied → Free.

**Step 5: Architecture.** Component diagram (UI, business logic, database) and deployment diagram (client PCs, application server, database server).

**Advantages over structured modeling**

- Models map naturally to real entities (patient, doctor), while structured models split data and processes.
- Changes are local: a new staff type is a new subclass.
- Inheritance and polymorphism give reuse (common Staff behaviour).
- Encapsulation protects sensitive patient data.
- One model flows from analysis to design to code; DFD-to-structure-chart conversion is error-prone.
- Better suited to large evolving systems.

In conclusion, object-oriented modeling gives a modular, reusable and maintainable hospital system.

---

## Q3. Explain the principles and importance of modelling. (2023)

A **model** is a simplified view of a system that omits irrelevant detail. We model because we cannot understand a complex system in its entirety.

**Principles**

1. The choice of model strongly influences how the problem is attacked and the solution shaped.
2. Every model may be expressed at different levels of precision.
3. The best models are connected to reality.
4. No single model suffices; a small set of nearly independent models is needed.

**Importance**

- **Visualisation:** shows the system as it is or as we want it.
- **Specification:** states structure and behaviour precisely.
- **Construction:** guides code writing and can generate code.
- **Documentation:** records decisions for later maintenance.
- **Risk reduction:** errors are caught cheaply before coding; a building is planned on blueprints before it is built.
- **Communication:** a shared language between customers, analysts and developers.
- **Managing complexity** through abstraction and decomposition.

| Without models | With models |
|---|---|
| Errors found late | Errors found early |
| Poor team communication | Common understanding |
| Hard to maintain | Documented design |

Example: an architect builds a small model before a skyscraper; a software team draws class and sequence diagrams before an online shopping system. In conclusion, modelling is essential for managing complexity and producing correct software.

---

### Q4. What is UML? List all building blocks of UML. Explain all types of things used in UML. Discuss the conceptual model of UML in detail. (2023, 2022)

**UML (Unified Modeling Language)** is a standard visual language for specifying, visualising, constructing and documenting software systems.

**Conceptual model of UML** has three parts: building blocks, rules, and common mechanisms.




*Conceptual model of UML*

**1. Building blocks:** things, relationships and diagrams.

**Things**

- **Structural things** (nouns, static parts): class, interface, collaboration, use case, active class, component, node.
- **Behavioural things** (verbs, dynamic parts): interaction (messages between objects) and state machine.
- **Grouping things:** package, used to organise elements.
- **Annotational things:** note, a comment attached to elements.

**Relationships:** dependency (uses), association (structural link), generalisation (is-a), realisation (implements an interface).

**Diagrams:** structural (class, object, component, deployment, package) and behavioural (use case, sequence, collaboration/communication, state, activity).

**2. Rules:** how elements are put together well-formed: names, scope, visibility, integrity and execution.

**3. Common mechanisms**

- **Specifications:** textual details behind the graphical notation.
- **Adornments:** extra markings such as visibility or multiplicity.
- **Common divisions:** class versus object, interface versus implementation.
- **Extensibility:** stereotypes, tagged values and constraints.

| Building block | Examples |
|---|---|
| Things | Class, use case, state machine, package, note |
| Relationships | Dependency, association, generalisation, realisation |
| Diagrams | Class, sequence, activity |

In conclusion, the conceptual model defines the vocabulary, rules and mechanisms that make UML models consistent and extensible.

---

## Q5. Demonstrate the different relationships used in class diagram with their notations with the help of a neat class diagram. Explain

### generalization, aggregation and association in detail. (2023, 2022)

A class diagram shows classes and their relationships.

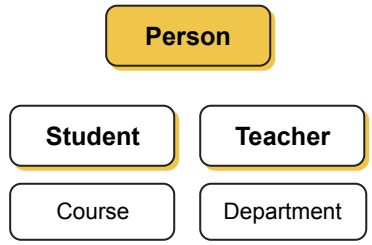
| Relationship | Meaning | Notation |
|---|---|---|
| **Association** | structural link between classes | solid line, optional arrow, name and multiplicity |
| **Aggregation** | whole–part, part can exist alone ("has-a") | line with hollow diamond at whole |
| **Composition** | strong whole–part, part dies with whole | line with filled diamond |
| **Generalization** | is-a, child inherits parent | solid line with hollow triangle at parent |
| **Dependency** | one class uses another temporarily | dashed arrow |
| **Realization** | class implements interface | dashed line with hollow triangle |

**Association:** a relationship among objects of two classes, e.g. `Teacher teaches Student`. It has a name, roles and multiplicity (1 to 0..*). It may be unidirectional or bidirectional; a link is an instance of an association.

**Aggregation:** a special association expressing "whole–part". `Department` has `Professors`; deleting the department does not delete the professors. Composition is the stronger form: `House` and `Room`.

**Generalization:** relates a general class (superclass) to a specific class (subclass). The subclass inherits attributes and operations and may add or override. `Vehicle` is generalised from `Car` and `Truck`.

```
      Person
        ^   (generalization)
    Student ---------- Course
              enrols   (association, 1..* )
College <>---- Department   (aggregation)
Department <#>-- Lab        (composition)
```




*Generalization and association in a college model*

Brought to you By Ycotes : https://theycotes.com

In conclusion, choosing the right relationship records how classes collaborate and how long their instances live.

---

### Q6. Define the term multiplicity and quantification with suitable examples. (2022)

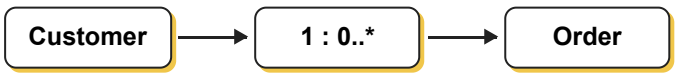
**Multiplicity** specifies how many instances of one class may be related to one instance of another class in an association. It is written at the association end.

| Notation | Meaning |
|---|---|
| 1 | exactly one |
| 0..1 | zero or one (optional) |
| * or 0..* | zero or many |
| 1..* | one or many |
| m..n | between m and n |

#### Examples

- `Customer 1 ---- 0..* Order`: a customer places zero or more orders; each order belongs to exactly one customer.
- `Person 0..1 ---- 1 Passport`: a person may hold at most one passport.
- `Student * ---- * Course`: many-to-many; usually resolved with an association class `Enrollment`.

**Quantification** (qualification) restricts the objects at the far end of an association by using a **qualifier**, reducing multiplicity from many to one. Example: `Bank --- Account` with qualifier `accountNo`: given a bank and an account number, there is exactly one account, so multiplicity changes from 1..* to 0..1.




*Multiplicity of Customer and Order*

Multiplicity states the cardinality constraints in the model; qualification makes lookups precise and reduces ambiguity.

---

### Q7. What do you understand by a static member function of a class? Discuss their characteristics. Give an example where you can justify the

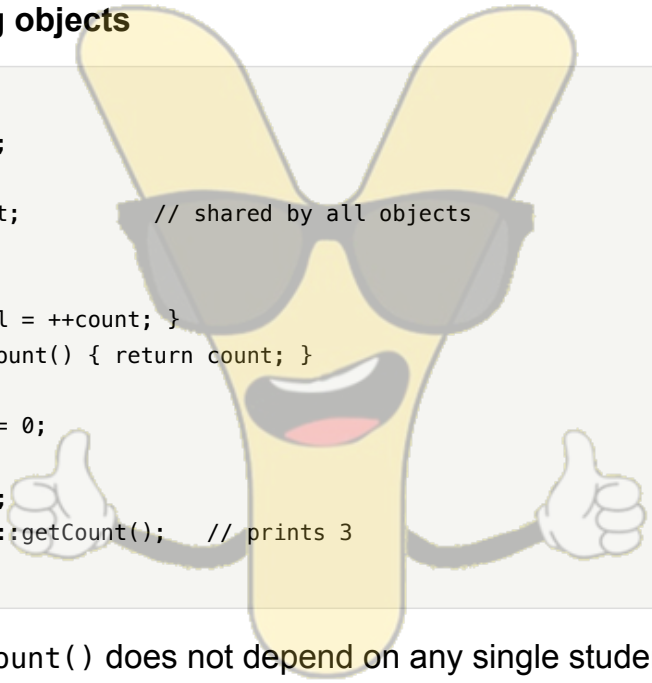
### use of static member functions. (2021)

A **static member function** belongs to the class as a whole, not to any particular object. It is declared with the keyword `static`.

#### Characteristics

- Can be called with the class name: `Class::fn()`, without any object.
- Has no `this` pointer.
- Can access only **static data members** and other static member functions directly; non-static members need an object.
- Cannot be `virtual` or `const`.
- One copy is shared by all objects.
- Often used with static data members to keep class-wide state.

#### Example: counting objects



```
#include <iostream>
using namespace std;
class Student {
    static int count;          // shared by all objects
    int roll;
public:
    Student() { roll = ++count; }
    static int getCount() { return count; }
};
int Student::count = 0;
int main() {
    Student a, b, c;
    cout << Student::getCount();   // prints 3
}
```

**Justification:** `getCount()` does not depend on any single student object; it reports information about the entire class. Making it static allows calling it even before any object exists, and it cannot accidentally modify per-object data. Other uses: factory functions, utility functions (`Math::sqrt`) and singleton access.

In conclusion, static member functions provide class-level behaviour that is independent of object state.

---