Q.1 What do you mean by an object? Explain. Discuss the various characteristics of objects with suitable example.

Answer: Objects are key to understanding object-oriented technology. Look around right now and you'll find many examples of real-world objects: your dog, your desk, your television set, your bicycle.

Real-world objects share two characteristics: They all have state and behavior. Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail). Bicycles also have state (current gear, current pedal cadence, current speed) and behavior (changing gear, changing pedal cadence, applying brakes). Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming.

Software objects are conceptually similar to real-world objects: they too consist of state and related behavior. An object stores its state in fields (variables in some programming languages) and exposes its behavior through methods (functions in some programming languages). Methods operate on an object's internal state and serve as the primary mechanism for object-to-object communication.
Q.2 What is the difference between Procedure based programming language and Object oriented programming language.

Answer:

### Difference Between Procedure Oriented Programming (POP) & Object Oriented Programming (OOP)

<table>
<thead>
<tr>
<th>Procedure Oriented Programming</th>
<th>Object Oriented Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Divided Into</strong></td>
<td>In POP, program is divided into small parts called <strong>functions</strong>.</td>
</tr>
<tr>
<td><strong>Importance</strong></td>
<td>In OOP, program is divided into parts called <strong>objects</strong>.</td>
</tr>
<tr>
<td><strong>Approach</strong></td>
<td>POP follows <strong>Top Down approach</strong>.</td>
</tr>
<tr>
<td><strong>Access Specifiers</strong></td>
<td>POP follows <strong>Bottom Up approach</strong>.</td>
</tr>
<tr>
<td><strong>Data Moving</strong></td>
<td>OOP has access specifiers named Public, Private, Protected, etc.</td>
</tr>
<tr>
<td><strong>Expansion</strong></td>
<td>POP does not have any access specifier.</td>
</tr>
<tr>
<td><strong>Data Access</strong></td>
<td>In OOP, data can move freely from function to function in the system.</td>
</tr>
<tr>
<td><strong>Data Hiding</strong></td>
<td>In POP, Data can move freely from function to function in the system.</td>
</tr>
<tr>
<td><strong>Overloading</strong></td>
<td>In POP, Overloading is not possible.</td>
</tr>
<tr>
<td><strong>Examples</strong></td>
<td>In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.</td>
</tr>
</tbody>
</table>

Q.3 What do you mean by polymorphism. Explain with an example?

Polymorphism is another important OOP concept. Polymorphism means the ability to take more than one form. By providing the interface mechanism, Java allows to fully utilize the one interface, multiple methods aspect of polymorphism. For example, a dog's sense of smell() is polymorphic which has many forms such as If the dog smells() a cat, it will bark and run after it. If the dog smells() its food, it will salivate and run to its bowl. The same sense of smell() in both situations doing different job depending on what is being smelled(), ie. the type of data operated upon by the dog's nose. This polymorphism is achieved in java in 2 ways ie. by defining the interfaces or by overriding the methods in the subclass.

Q.4 Give the conceptual model of UML.

To understand the UML, you need to form a **conceptual model of the language**, and this requires learning three major elements: the UML's basic building blocks, the rules that dictate how those building blocks may be put together, and some common mechanisms that apply throughout the UML.
Basic Building Block of UML

1. Things
2. Relationships
3. Diagrams

Things in the UML

There are four kinds of things in the UML:

1. Structural things
2. Behavioral things
3. Grouping things
4. Annotational things

Structural things are the nouns of UML models. These are the mostly static parts of a model, representing elements that are either conceptual or physical. In all, there are seven kinds of structural things.

1. Classes
2. Interfaces
3. Collaborations
4. Use cases
5. Active classes
6. Components
7. Nodes

Behavioral Things are the dynamic parts of UML models. These are the verbs of a model, representing behavior over time and space. In all, there are two primary kinds of behavioral things.

1. Interaction
2. State machine

Grouping Things:

1. are the organizational parts of UML models. These are the boxes into which a model can be decomposed
2. There is one primary kind of grouping thing, namely, packages.

Annotational things are the explanatory parts of UML models. These are the comments you may apply to describe about any element in a model.

Relationships in the UML: There are four kinds of relationships in the UML:

1. Dependency
2. Association
3. Generalization
4. Realization
Diagrams in the UML

Diagram is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships). In theory, a diagram may contain any combination of things and relationships.

For this reason, the UML includes nine such diagrams:

1. Class diagram
2. Object diagram
3. Use case diagram
4. Sequence diagram
5. Collaboration diagram
6. Statechart diagram
7. Activity diagram
8. Component diagram
9. Deployment diagram

Rules of the UML

The UML has semantic rules for:

1. Names What you can call things, relationships, and diagrams
2. Scope The context that gives specific meaning to a name
3. Visibility How those names can be seen and used by others
4. Integrity How things properly and consistently relate to one another
5. Execution What it means to run or simulate a dynamic model

Common Mechanisms in the UML

UML is made simpler by the presence of four common mechanisms that apply consistently throughout the language.

1. Specifications
2. Adornments
3. Common divisions
4. Extensibility mechanisms

Extensibility Mechanisms

The UML's extensibility mechanisms include:

1. Stereotypes
2. Tagged values
3. Constraints
Q.1 Write short notes on architectural modeling with suitable example and diagrams.

Answer:- Architectural modeling involves Component and Deployment. A component typically represents the physical packaging of otherwise logical elements, such as classes, interfaces, and collaborations. Good components define crisp abstractions with well-defined interfaces, making it possible to easily replace older components with newer, compatible ones. These logical and physical views are both necessary. If you are building a disposable building for which the cost of scrap and rework is essentially zero (for example, if you are building a doghouse), you can probably go straight to the physical building without doing any logical modeling. If, on the other hand, you are building something enduring for which the cost of change or failure is high, then building both logical and physical models is the pragmatic thing to do to manage risk.

It's the same thing when building a software-intensive system. You do logical modeling to visualize, specify, and document your decisions about the vocabulary of your domain and the structural and behavioral way those things collaborate. You do physical modeling to construct the executable system. Whereas these logical things live in the conceptual world, the physical things live in the world of bits; that is, they ultimately reside on physical nodes and can be executed directly or can, in some indirect manner, participate in an executing system.

In the UML, all physical things are modeled as components. A component is a physical thing that conforms to and realizes a set of interfaces. Interfaces therefore bridge your logical and physical models. For example, you may specify an interface for a class in a logical model, and that same interface will carry over to some physical component that realizes it.

The UML provides a graphical representation of a component, as Figure shows

Nodes, just like components, live in the material world and are an important building block in modeling the physical aspects of a system. A node is a physical element that exists at runtime and represents a computational resource, generally having at least some memory and, often, processing capability. You use nodes to model the topology of the hardware on which your system executes. A node typically represents a processor or a device on which components may be deployed. Good nodes crisply represent the vocabulary of the hardware in your solution domain.

The components you develop or reuse as part of a software-intensive system must be deployed on some set of hardware in order to execute. This is in effect what a software-intensive system is all about; such a system encompasses both software and hardware. When you architect a software-intensive system, you have to consider both its logical and physical dimensions. On the logical side, you'll find things such as classes, interfaces, collaborations, interactions, and state machines. On the physical side, you'll find components
(which represent the physical packaging of these logical things) and nodes (which represent the hardware on which these components are deployed and execute). The UML provides a graphical representation of node, as Figure shows.

Q.2 Define state. Give state diagram for phone line.

Answer:- A state is a condition or situation in the life of an object during which it satisfies some condition, performs some activity, or waits for some event.

Statechart diagrams are one of the five diagrams in the UML for modeling the dynamic aspects of systems. A statechart diagram shows a state machine. An activity diagram is a special case of a statechart diagram in which all or most of the states are activity states and all or most of the transitions are triggered by completion of activities in the source state. Thus, both activity and statechart diagrams are useful in modeling the lifetime of an object. However, whereas an activity diagram shows flow of control from activity to activity, a statechart diagram shows flow of control from state to state.

You use statechart diagrams to model the dynamic aspects of a system. For the most part, this involves modeling the behavior of reactive objects. A reactive object is one whose behavior is best characterized by its response to events dispatched from outside its context. A reactive object has a clear lifetime whose current behavior is affected by its past. Statechart diagrams may be attached to classes, use cases, or entire systems in order to visualize, specify, construct, and document the dynamics of an individual object.

Statechart diagrams are not only important for modeling the dynamic aspects of a system, but also for constructing executable systems through forward and reverse engineering.

In the UML, you model the event-ordered behavior of an object by using statechart diagrams. As Figure shows, a statechart diagram is simply a presentation of a state machine, emphasizing the flow of control from state to state.
Q.3 Draw Use case diagram for a Restaurant System

Answer:

Q.4 What are packages and why they are used?

In the UML, the package is a general purpose mechanism for organizing modeling elements into groups. You use packages to arrange your modeling elements into larger chunks that you can manipulate as a group. You can control the visibility of these elements so that some things are visible outside the package while others are hidden. You can also use packages to present different views of your system's architecture. Well-designed packages group elements that are semantically close and that tend to change together. Well-structured packages are therefore loosely coupled and very cohesive, with tightly controlled access to the package's contents.

The UML provides a graphical representation of package, as Figure shows.

![Use case diagram for a Restaurant System](image)
Part-3

Q.1 (a) Explain the principles of modeling.

Answer: The use of modeling has a rich history in all the engineering disciplines. That experience suggests four basic principles of modeling.

First, The choice of what models to create has a profound influence on how a problem is attacked and how a solution is shaped.

Second, Every model may be expressed at different levels of precision.

Third, The best models are connected to reality.

Fourth, No single model is sufficient. Every nontrivial system is best approached through a small set of nearly independent models.

(b) Write the names of any 4 UML diagrams and explain their use.

Answer: A diagram is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships). You draw diagrams to visualize a system from different perspectives, so a diagram is a projection into a system.

Diagrams:
1. Class diagrams address the static design view of a system.
2. Use case diagrams address the static use case view of a system. These diagrams are especially important in organizing and modeling the behaviors of a system.
3. Interaction diagrams address the dynamic view of a system. A sequence diagram is an interaction diagram that emphasizes the time-ordering of messages.
4. A collaboration diagram is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages. Sequence diagrams and collaboration diagrams are isomorphic, meaning that you can take one and transform it into the other.

Q.2 (a) Discuss the concept of encapsulation with suitable example.

Answer: Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper. Access to the code and data inside the wrapper is tightly controlled through a well-defined interface.

To relate this to the real world, consider the automatic transmission on an automobile. It encapsulates hundreds of bits of information about your engine, such as how much you are accelerating, the pitch of the surface you are on, and the position of the shift lever. You, as the user, have only one method of affecting this complex encapsulation: by moving the gear-
shift lever. You can't affect the transmission by using the turn signal or windshield wipers, for example. Thus, the gear-shift lever is a well-defined (indeed, unique) interface to the transmission. Further, what occurs inside the transmission does not affect objects outside the transmission. For example, shifting gears does not turn on the headlights! In Java, the basis of encapsulation is the class.

(b) What do you mean by generalization and inheritance? Use some examples to explain.

Answer:

**Generalization**

A *generalization* is a relationship between a general thing (called the superclass or parent) and a more specific kind of that thing (called the subclass or child). Generalization is sometimes called an "is-a-kind-of" relationship: one thing (like the class BayWindow) is a kind of a more general thing (for example, the class Window). Generalization means that objects of the child may be used anywhere the parent may appear, but not the reverse.

Graphically, generalization is rendered as a solid directed line with a large open arrowhead, pointing to the parent, as shown in Figure

![Generalization Diagram](image)

**Inheritance**

*Inheritance* is the process by which one object acquires the properties of another object. This is important because it supports the concept of hierarchical classification.

Generalization is a relationship that uses the concept of Inheritance as Parent Child relationship.

Most of the time, you'll find single inheritance sufficient. A class that has exactly one parent is said to use single inheritance. There are times, however, when multiple inheritance is better, and you can model those relationships, as well, in the UML.
Q.3 (a) Define Link and Association. Discuss the role of link and association in object modelling with suitable example.

A link is a semantic connection among objects. In general, a link is an instance of an association. As Figure shows, wherever a class has an association to another class, there may be a link between the instances of the two classes; wherever there is a link between two objects, one object can send a message to the other object.

![Diagram showing the relationship between Person and Company classes with links and associations](image)

A link specifies a path along which one object can dispatch a message to another (or the same) object. Most of the time, it is sufficient to specify that such a path exists. If you need to be more precise about how that path exists, you can adorn the appropriate end of the link with any of the following standard stereotypes. As an instance of an association, a link may be rendered with most of the adornments appropriate to associations, such as a name, association role name, navigation, and aggregation. Multiplicity, however, does not apply to links, since they are instances of an association.

(b) What do you understand by basic Behavioral Modeling mainly Use case Diagram?

Use case diagrams are one of the five diagrams in the UML for modeling the dynamic aspects of systems (activity diagrams, statechart diagrams, sequence diagrams, and collaboration diagrams are four other kinds of diagrams in the UML for modeling the dynamic aspects of systems). Use case diagrams are central to modeling the behavior of a system, a subsystem, or a class. Each one shows a set of use cases and actors and their relationships.

You apply use case diagrams to model the use case view of a system. For the most part, this involves modeling the context of a system, subsystem, or class, or modeling the requirements of the behavior of these elements.

Use case diagrams are also important for testing executable systems through forward engineering and for comprehending executable systems through reverse engineering. As Figure 17-1 shows, you can provide a use case diagram to model the behavior of that box• which most people would call a cellular phone.
Use case diagrams commonly contain:
- Use cases
- Actors
- Dependency, generalization, and association relationships

Like all other diagrams, use case diagrams may contain notes and constraints. Use case diagrams may also contain packages, which are used to group elements of your model into larger chunks.

When you model the static use case view of a system, you'll typically apply use case diagrams in one of two ways.
1. To model the context of a system
Modeling the context of a system involves drawing a line around the whole system and asserting which actors lie outside the system and interact with it. Here, you'll apply use case diagrams to specify the actors and the meaning of their roles.

2. To model the requirements of a system
Modeling the requirements of a system involves specifying what that system should do (from a point of view of outside the system), independent of how that system should do it. Here, you'll apply use case diagrams to specify the desired behavior of the system. In this manner, a use case diagram lets you view the whole system as a black box; you can see what's outside the system and you can see how that system reacts to the things outside, but you can't see how that system works on the inside.

Q.4 Explain common modeling techniques of a class diagram.

Class diagrams are the most common diagram found in modeling object-oriented systems. A class diagram shows a set of classes, interfaces, and collaborations and their relationships. You use class diagrams to model the static design view of a system. For the most part, this involves modeling the vocabulary of the system, modeling collaborations, or modeling schemas.
Common Modeling Techniques

1. Modeling Simple Collaborations

To model a collaboration,
· Identify the mechanism you'd like to model. A mechanism represents some function or behavior of the part of the system you are modeling that results from the interaction of a society of classes, interfaces, and other things.
· For each mechanism, identify the classes, interfaces, and other collaborations that participate in this collaboration. Identify the relationships among these things, as well.
· Use scenarios to walk through these things. Along the way, you'll discover parts of your model that were missing and parts that were just plain semantically wrong.
· Be sure to populate these elements with their contents. For classes, start with getting a good balance of responsibilities. Then, over time, turn these into concrete attributes and operations.

2. Modeling Logical Database Schema

To model a schema,
· Identify those classes in your model whose state must transcend the lifetime of their applications.
· Create a class diagram that contains these classes and mark them as persistent (a standard tagged value). You can define your own set of tagged values to address database-specific details.
· Expand the structural details of these classes. In general, this means specifying the details of their attributes and focusing on the associations and their cardinalities that structure these classes.
· Watch for common patterns that complicate physical database design, such as cyclic associations, one-to-one associations, and n-ary associations. Where necessary, create intermediate abstractions to simplify your logical structure.
· Consider also the behavior of these classes by expanding operations that are important for data access and data integrity. In general, to provide a better separation of concerns, business rules concerned with the manipulation of sets of these objects should be encapsulated in a layer above these persistent classes.
· Where possible, use tools to help you transform your logical design into a physical design.

3. Forward And Reverse Engineering

To forward engineer a class diagram,
· Identify the rules for mapping to your implementation language or languages of choice. This is something you'll want to do for your project or your organization as a whole.
· Depending on the semantics of the languages you choose, you may have to constrain your use of certain UML features. For example, the UML permits you to model multiple inheritance, but Smalltalk permits only single inheritance. You can either choose to prohibit developers from modeling with multiple inheritance (which makes your models language-
dependent) or develop idioms that transform these richer features into the implementation language (which makes the mapping more complex).

- Use tagged values to specify your target language. You can do this at the level of individual classes if you need precise control. You can also do so at a higher level, such as with collaborations or packages.
- Use tools to forward engineer your models.

To reverse engineer a class diagram,

- Identify the rules for mapping from your implementation language or languages of choice. This is something you'll want to do for your project or your organization as a whole.
- Using a tool, point to the code you'd like to reverse engineer. Use your tool to generate a new model or modify an existing one that was previously forward engineered.
- Using your tool, create a class diagram by querying the model. For example, you might start with one or more classes, then expand the diagram by following specific relationships or other neighboring classes. Expose or hide details of the contents of this class diagram as necessary to communicate your intent.