Student Name: ___________________________ **Roll No:** ________________ **Section:** ______________

Lab Series No. 11.

Lab 11 –Introduction to Object Oriented Programming.

Lab Objectives:

1. Introduction to Class and Class Diagram
    a. Notation of Class Diagram
    b. Attributes
    c. Operations
    d. Visibility
2. Relationship
3. Case Study

1. Introduction to Class and Class Diagram

• Class diagrams are visual representations of the static structure and composition of a particular system using the conventions set by the Unified Modeling Language (UML).
• System designers use class diagrams as a way of simplifying how objects in a system interact with each other.
• Using class diagrams, it is easier to describe all the classes, packages, and interfaces that constitute a system and how these components are interrelated.
• Since class diagrams are used for many different purposes, such as making stakeholders aware of requirements to highlighting your detailed design, you need to apply a different style in each Circumstance.

Example

• Simple class diagram may be used to show how an organization such as a convenient store chain is set up.
• Precisely detailed class diagrams can readily be used as the primary reference for translating the designed system into a programming code.

Notation of Class Diagram

1. Class

• An object is any person, place, thing, concept, event, screen, or report applicable to your system.
Objects both know things (they have attributes) and they do things (they have methods).
• A class is a representation of an object and, in many ways, it is simply a template from which objects are created.

Student Name: ___________________________ Roll No: ________________ Section: ______________

• Classes form the main building blocks of an object-oriented application.

Example

Although thousands of students attend the university, you would only model one class, called Student, which would represent the entire collection of students.

2. **Attributes**

An attribute of a class represents a characteristic of a class that is of interest for the user.

The full format of the attribute text notation is:
Visibility name: type multiplicity = default [property-string]

3. **Operations**

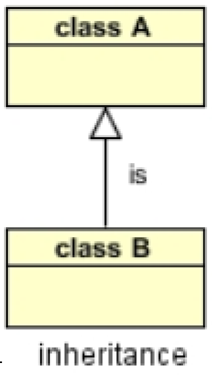
A UML operation is a declaration, with a name, parameters, return type, exceptions list, and possibly a set of constraints of pre and post conditions. But, it isn't an implementation – rather, methods are implementation.

4. **Visibility**:
Use visibility markers to signify who can access the information contained within a class.

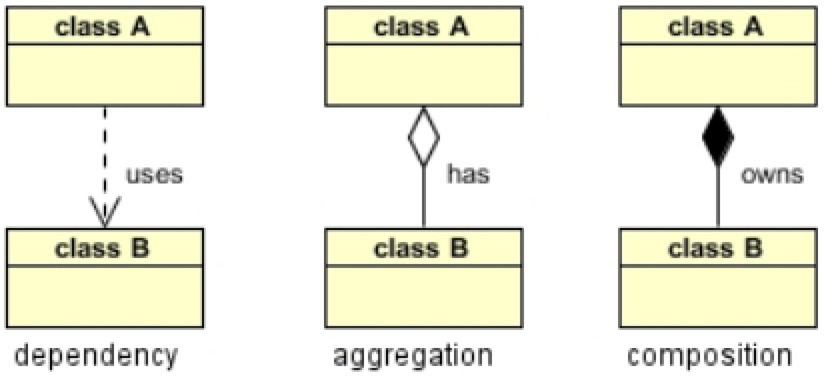
- Public +
- Private -
- Protected #

2. Relationship

Dependency: class A uses class B
Aggregation: class A has a class B
Composition: class A owns a class B
Inheritance: class B is a Class A (or class A is extended by class B)



Student Name: ______________________ Roll No: ____________ Section: ___________



1. Association

An association is a "using" relationship between two or more objects in which the objects have their own life time and there is no owner.

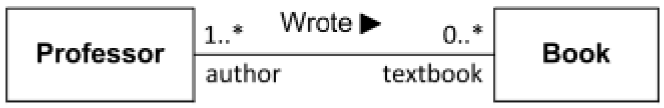
For **Example**: A patient may visit one or many doctors and same way, a doctor can be associated with multiple patients. If a patient dies, existence of doctor will not be vanished and similarly if doctor dies patient will remain patient.

Association is represented as thin line connecting two classes. Association can be unidirectional (shown by arrow at one end) or bidirectional (shown by arrow at both end) or without arrow.

Multiplicity defines how many instances can be associated at any given moment**.**

0..1	No instances or one instance	A flight seat can have no or one passenger only
1	Exactly one instance	A class can have zero or more students
0..* or *	Zero or more instances	A class can have zero or more students.
1..*	One or more instances (at least one)	A flight can have one or more passenger

Example:



Association Wrote between Professor and Book with association ends author and textbook.

2. Aggregation

Aggregation is a special form of association. It is also a relationship between two classes like association, however, it's a **directional** association, which means it is strictly a **one way association, means**

unidirectional association. It represents a **Has-A** relationship.

For Example: Consider two classes Student class and Address class. Each student must have an address so the relationship between student and address is a Has-A relationship. But if you consider its vice versa then it would not make sense as an Address doesn't need to have a Student necessarily.
NOTE: Unarguably, Address is an attribute of a student, but here in this example I am breaking address into several fields i.e city, province and country. This is the reason for making address a class**.**

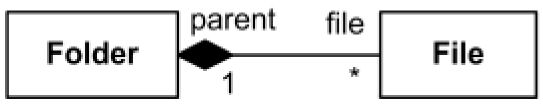
3. Composition

Composition is a special case of aggregation. In a more specific manner, a restricted aggregation is called composition. When an object contains the other object, if the contained object cannot exist without the existence of container object, then it is called composition.

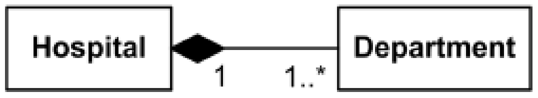
For **Example**: Consider the same scenario with some modifications. In this scenario, a student has address and each student has different address (Please keep sibling relationship argument apart). So, when a student record is added his house number and street number will be entered. And if I delete the record of a particular student, then his/her record will be of no use.

filled black diamond at the aggregate (whole) end.

Example:



Folder could contain many files, while each File has exactly one Folder parent.
If Folder is deleted, all contained Files are deleted as well.



Hospital has 1 or more Departments, and
each Department belongs to exactly one Hospital.
If Hospital is closed, so are all of its Departments.

4. Generalization

In object oriented programming, the concept of IS-A is a totally based on Inheritance, which can be of two types Class Inheritance or Interface Inheritance. It is just like saying "A is a B type of thing". For example, Apple is a Fruit, Car is a Vehicle etc. Inheritance is uni-directional. For example House is a Building. But Building is not a House.

It is key point to note that you can easily identify the IS-A relationship. Wherever you see an extends keyword or implements keyword in a class declaration, then this class is said to have IS-A relationship.

Student Name: ___________________________ **Roll No:** ________________ **Section:** ______________

Case Study1

Library Management System

Problem Statement:

The case study titled Library Management System is library management software for the purpose of monitoring and controlling the transactions in a library. This case study on the library management system gives us the complete information about the library and the daily transactions done in a Library. We need to maintain the record of new s and retrieve the details of books available in the library which mainly focuses on basic operations in a library like adding new member, new books, and up new information, searching books and members and facility to borrow and return books. It features a familiar and well thought-out, an attractive user interface, combined with strong searching, insertion and reporting capabilities. The report generation facility of library system helps to get a good idea of which are ths borrowed by the members, makes users possible to generate hard copy.

The following are the brief description on the functions achieved through this case study:

End-Users

• Librarian: To maintain and update the records and also to cater the needs of the users.

- Reader: Need books to read and also places various requests to the librarian.
- Vendor: To provide and meet the requirement of the prescribed books.

Class Diagram

Classes identified:

Library

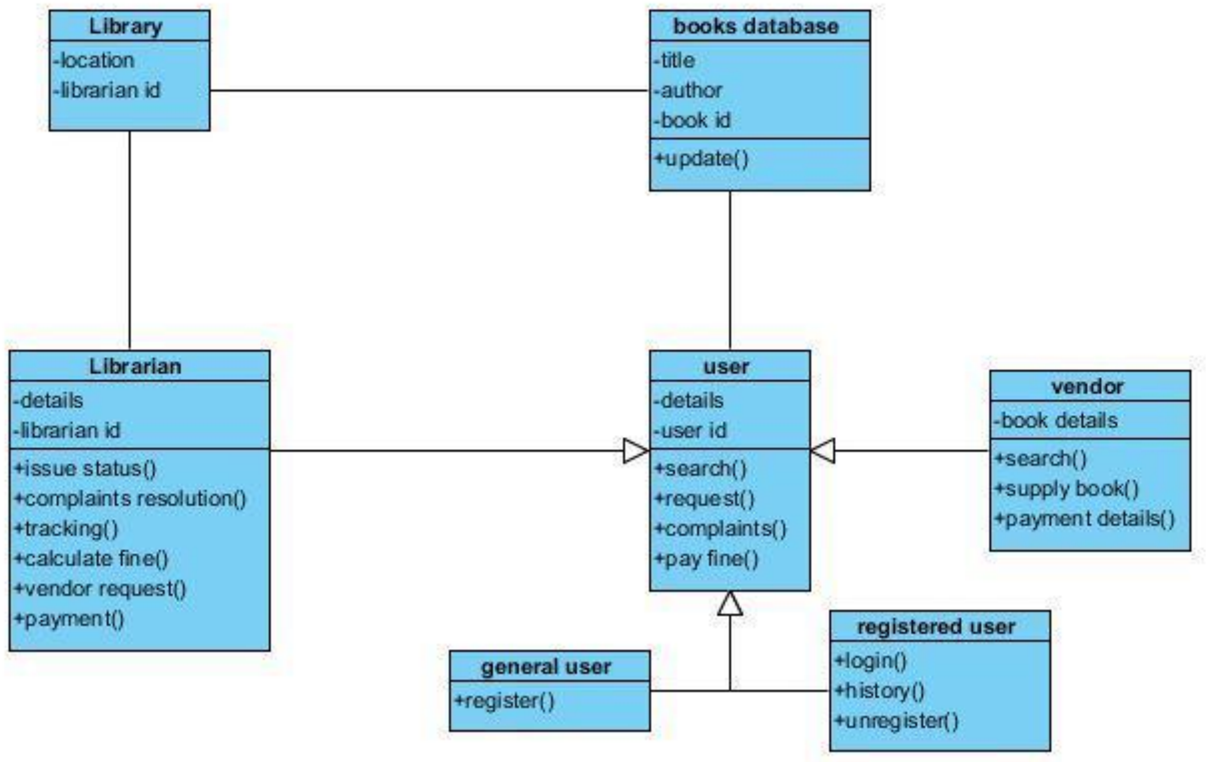
Librarian

Books Database

User

Vendor

Student Name: ___________________________ **Roll No:** ________________ **Section:** _____________



Use-case Diagram

Actors vs Use Cases:

Librarian
•Issue a book
•Update and maintain records
•Request the vendor for a book
•Track complaints

User
•Register
•Login
•Search a book
•Request for isse
•View history
•Request to the Librarian
•Unregister

Student Name: ___________________________ Roll No: ________________ Section: ______________

Books Database
•Update records
•Show books status

Vendors
•Provide books to the library
•Payment acknowledgement

1. The _init_() Method

The __init__() method is profound for two reasons. Initialization is the first big step in an object's life; every object must be initialized properly to work properly. The second reason is that the argument values for __init__() can take on many forms.

Because there are so many ways to provide argument values to __init__(), there is a vast array of use cases for object creation. We take a look at several of them. We want to maximize clarity, so we need to define an initialization that properly characterizes the problem domain.

Before we can get to the __init__() method, however, we need to take a look at the implicit class hierarchy in Python, glancing, briefly, at the class named object. This will set the stage for comparing default behavior with the different kinds of behavior we want from our own classes.

In this example, we take a look at different forms of initialization for simple objects (for example, playing cards). After this, we can take a look at more complex objects, such as hands that involve collections and players that involve strategies and states.

Python is a multi-paradigm programming language. Meaning, it supports different programming approach.

One of the popular approach to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).

An object has two characteristics:

- attributes
- behavior

Let's take an example:

Student Name: ___________________________ Roll No: ________________ Section: ______________

Parrot is an object,

- name, age, color are attributes
- singing, dancing are behavior

The concept of OOP in Python focuses on creating reusable code. This concept is also known as DRY (Don't Repeat Yourself).

In Python, the concept of OOP follows some basic principles:

1	***Inheritance***	A process of using details from a new class without modifying existing class.
2	***Encapsulation***	Hiding the private details of a class from other objects.
3	***Polymorphism***	A concept of using common operation in different ways for different data input.

Python Object Inheritance

Inheritance is the process by which one class takes on the attributes and methods of another. Newly formed classes are called child classes, and the classes that child classes are derived from are called parent classes.

It's important to note that child classes override or extend the functionality (e.g., attributes and behaviors) of parent classes. In other words, child classes inherit all of the parent's attributes and behaviors but can also specify different behavior to follow. The most basic type of class is an object, which generally all other classes inherit as their parent.

When you define a new class, Python 3 it implicitly uses object as the parent class. So the following two definitions are equivalent:

Student Name: ____________________________ **Roll No:** ________________ **Section:** ______________

Exercise 1: Write a class of Dog, each dog must be of species type mammal. Each dog has its name and age. The class can have method for description () and sound () which dog produces. Create an object and perform some operations.

```
class Dog:

    # Class Attribute
    species = 'mammal'

    # Initializer / Instance Attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # instance method
    def description(self):
        return "{} is {} years old".format(self.name, self.age)

    # instance method
    def speak(self, sound):
        return "{} says {}".format(self.name, sound)

# Instantiate the Dog object
razer = Dog("Razer", 6)

# call our instance methods
print(razer.description())
print(razer.speak("Woof Woof"))
```

Student Name: ___________________________ Roll No: ________________ Section: ______________

Exercise 2: Write a class of Dog, each dog must be of species type mammal. Each dog has its name and age. The class can have method for description () and sound () which dog produces. Now this time you need to create two sub classes of Dogs one is Bull Dog and other is Russell Terrier Create few objects and perform some operations including the inheritance.

```
# Parent class
class Dog:

    # Class attribute
    species = 'mammal'

    # Initializer / Instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # instance method
    def description(self):
        return "{} is {} years old".format(self.name, self.age)

    # instance method
    def speak(self, sound):
        return "{} says {}".format(self.name, sound)


# Child class (inherits from Dog class)
class RussellTerrier(Dog):
    def run(self, speed):
        return "{} runs {}".format(self.name, speed)


# Child class (inherits from Dog class)
class Bulldog(Dog):
    def run(self, speed):
        return "{} runs {}".format(self.name, speed)


# Child classes inherit attributes and
# behaviors from the parent class
thunder = Bulldog("Thunder", 9)
print(thunder.description())

# Child classes have specific attributes
# and behaviors as well
print(thunder.run("slowly"))

spinter = Bulldog("Spinter", 12)
print(spinter.description())
print(spinter.run("fast"))

roger = RussellTerrier("Roger", 5)
```

```
print(roger.description())
print(roger.run("quickly"))
```

Exercise 3: Extending question number 2, now we need to check that either the different dog classes and their objects link with each other or not. In this case we need to create a method to find either it's an instance of each other objects or not.

```
# Parent class
class Dog:

    # Class attribute
    species = 'mammal'

    # Initializer / Instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # instance method
    def description(self):
        return "{} is {} years old".format(self.name, self.age)

    # instance method
    def speak(self, sound):
        return "{} says {}".format(self.name, sound)


# Child class (inherits from Dog() class)
class RussellTerrier(Dog):
    def run(self, speed):
        return "{} runs {}".format(self.name, speed)


# Child class (inherits from Dog() class)
class Bulldog(Dog):
    def run(self, speed):
        return "{} runs {}".format(self.name, speed)


# Child classes inherit attributes and
# behaviors from the parent class
thunder = Bulldog("Thunder", 9)
print(thunder.description())

# Child classes have specific attributes
# and behaviors as well
print(thunder.run("slowly"))

# Is thunder an instance of Dog()?
print(isinstance(thunder, Dog))

# Is thunder_kid an instance of Dog()?
```

```
thunder_kid = Dog("ThunderKid", 2)
print(isinstance(thunder, Dog))

# Is Kate an instance of Bulldog()
Kate = RussellTerrier("Kate", 4)
print(isinstance(Kate, Dog))

# Is thunder_kid and instance of kate?
print(isinstance(thunder_kid, Kate))
print("Thanks for understanding the concept of OOPs")
```

NOTE:

Make sense? Both thunder_kid and Kate are instances of the Dog() class, while Spinter is not an instance of the Bulldog() class. Then as a sanity check, we tested if kate is an instance of thunder_kid, which is impossible since thunder_kid is an instance of a class rather than a class itself—hence the reason for the TypeError.

Student Name: ___________________________ **Roll No:** ________________ **Section:** ______________

Programming Exercise

Task 1: Discuss in detail what you understand by Classes, Objects and find the relation of Instances which may have in some cases and for other it will not, for all the exercises and tasks starting from task 3.

Task 2: Create UML diagrams for all the exercises and Tasks starting from task 3. Use dia software to create classes and UML diagram.

Task 3: Create class and sub classes for different types of residential houses. Each house object has different parameters such as number of location of the house, rooms, parking available or not. Price of the house.

Task 4: Create class and sub classes for differ types of vehicles for Toyota Motors. Each car object belongs to some particular model, color, price, type of car such as saloon, luxury or sports. They can be registered in different years and made in different years.

Task 5: Create class and sub classes for different types of plants. Some of the plants have fruits that we can eat and some of them are poisonous. Some are local and some are imported from other countries. Some of the fruits are sweet whereas some are sour. Create parameters according to your own visualizations.

Task 6: Create a simple class of student, Teacher, with some attributes and methods.

Task 7: Create a class plant with some attributes and methods.