Subject Code : 24MCA122 Subject Name : Python Programming

BY
R.PADMAJA,
ASSISTANT PROFESSOR,
MCA DEPARTMENT,
SITAMS

UNIT – III

UNIT IV

1. What is Object Oriented Programming
2. Encapsulation
3. Polymorphism
4. Inheritance
5. Object Oriented Design with UML
6. Computational Problem Solving
7. Vehicle Rental Agency Problem

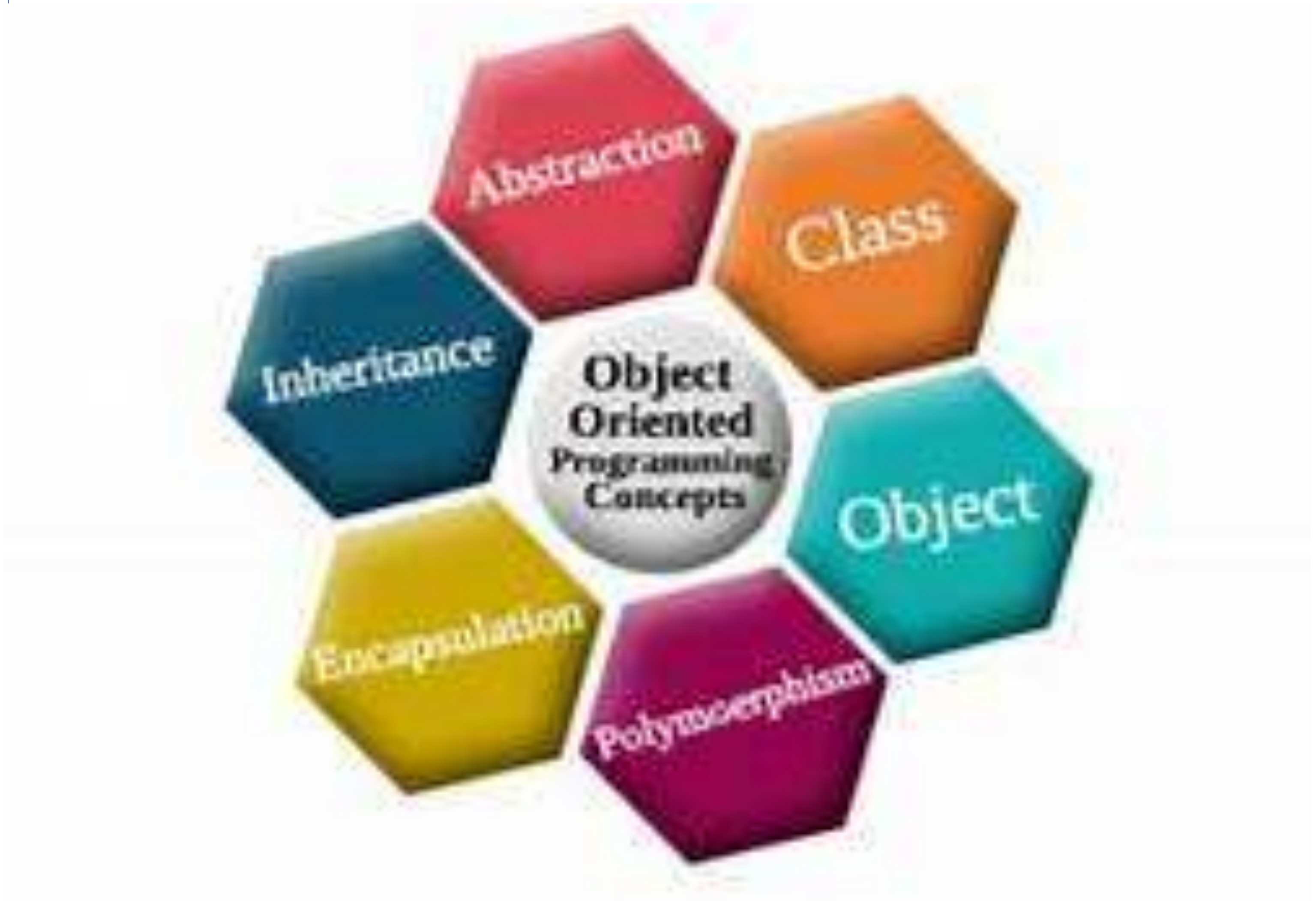
1. What is Object Oriented Programming

- ❖ Like other general-purpose programming languages, Python is also an object-oriented language since its beginning.
- ❖ It allows us to develop applications using an Object-Oriented approach.
- ❖ An object-oriented paradigm is
  - ✓ The method of structuring the program
  - ✓ to design the program using classes and objects.
  - ✓ is a widespread technique to solve the problem by creating **objects.**
- ❖ A Procedure Oriented paradigm is
  - ✓ To design programs using functions (0r)
  - ✓ It is a technique to solve problems by creating **Functions**

1. What is Object Oriented Programming

Procedural Oriented Programming	**Object Oriented Programming**
In procedural programming, the program is divided into small parts called ***functions***.	In object-oriented programming, the program is divided into small parts called ***objects***.
Procedural programming follows a ***top-down approach***.	Object-oriented programming follows a ***bottom-up approach***.
Adding new data and functions is not easy.	Adding new data and function is easy.
Procedural programming does not have any proper way of hiding data so it is ***less secure***.	Object-oriented programming provides data hiding so it is ***more secure***.
In procedural programming, **the function** is more important than the data.	In object-oriented programming**, data is** more important than function.
Procedural programming is used for designing **medium-sized programs.**	Object-oriented programming is used for designing **large and complex programs.**
Code reusability absent in procedural programming,	**Code reusability** present in object-oriented programming.
Examples: C, FORTRAN, Pascal, Basic, etc.	**Examples:** C++, Java, Python, C#, etc.

1. Object Oriented Programming Concepts



1. What is Object Oriented Programming

Object : An Entity that has a **state and Behaviour** associated with it .

Example 1 : **Smith is an entity that has state and behavior**

- ✓ State = RollNo, Name, age, branch etc.
- ✓ Behavior = Writing Internal Exam, External exam, attendance etc..

Example 2: **German Sheperd is an entity that has state and behavior .**

- ✓ State = age, breed, color etc..
- ✓ behavior = eating, walking, sleeping, braking

.

1. What is Object Oriented Programming

Class : Collection of similar Objects

Example 1: collection of smith, Jones, King are called a class "Student" because all such objects share same state and behavior

Example 2: collection of German Sheperd, Labrador,poodle etc.. called a class "Dog" because all such objects share same state and behavior

Inheritance : An Object Inheriting the properties of its parent Object .

Example : child inherits all properties from their parents

1. What is Object Oriented Programming

Polymorphism : It refers to the use of a single type entity (method, operator or object) to represent different types in different scenarios

Example : Mixer gives several different forms of output

Abstraction : Hiding Internal details from the user

Example : Hiding internal details of car from driver

1. What is Object Oriented Programming

Class

- ❖ A class is a collection of similar objects.
- ❖ A class contains the blueprints or the prototype from which the objects are being created.
- ❖ Class defines **data** and **methods** (which manipulate the data)
  - ✓ **Attributes or Properties** of object will become data or variables of a class
  - ✓ **Behavior or actions** of the object will become methods of the class

Object

- ❖ instance of a class

Inheritance

- ❖ Mechanism of deriving a **new class** from **existing class** such that new class acquires all the properties of existing class.

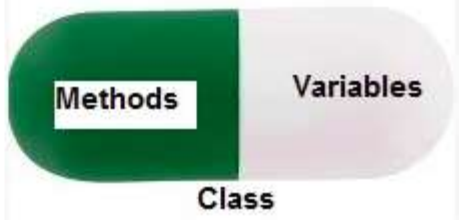
Polymorphism

- ❖ **same entity can perform different operations in different scenarios**.
- ❖ Polymorphism is implemented **through Overloading and overridng**

1. What is Object Oriented Programming

Encapsulation

- Encapsulation refers to the bundling of attributes and methods inside a single class.
- It prevents outer classes from accessing and changing attributes and methods of a class.
- This also helps to achieve **data hiding**.
- A class is an example of encapsulation as it encapsulates all the data that is member functions, variables, etc.



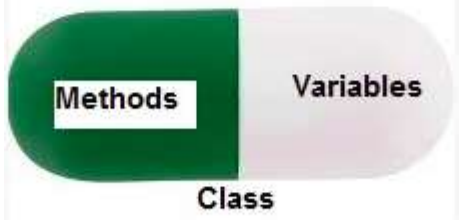
Data Abstraction

- It hides unnecessary code details from the user. Also, when we do not want to give out sensitive parts of our code implementation and this is where data abstraction came.
- Data Abstraction in Python can be achieved by creating abstract classes.

2. Encapsulation

Encapsulation

- ❖ Encapsulation refers to the bundling of attributes and methods inside a single class.
- ❖ It prevents outer classes from accessing and changing attributes and methods of a class.
- ❖ This also helps to achieve **data hiding**.
- ❖ A class is an example of encapsulation as it encapsulates all the data that is member functions, variables, etc.



2. Encapsulation

- Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section etc.
- The finance section handles all the financial transactions and keeps records of all the data related to finance.
- Similarly, the sales section handles all the sales-related activities and keeps records of all the sales.
- Now there may arise a situation when due to some reason an official from the finance section needs all the data about sales in a particular month.
- In this case, he is not allowed to directly access the data of the sales section.
- He will first have to contact some other officer in the sales section and then request him to give the particular data.
- This is what encapsulation is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name “sales section”.
- Using encapsulation also hides the data. In this example, the data of the sections like sales, finance, or accounts are hidden from any other section.

2. Encapsulation

```
# Define the Person class
class Person:

   # Constructor to initialize object
attributes
   def __init__(self, name, age):
      self.name = name
      self.age = age

   # Method to get the person's name
   def get_name(self):
      return self.name

   # Method to get the person's age
   def get_age(self):
      return self.age

   # Method to update the person's age
   def update_age(self, new_age):
      self.age = new_age
```

```
# Main program
# Create Person objects (instances)
person1 = Person("Alice", 25)
person2 = Person("Bob", 30)

# Access object attributes using methods
print(person1.get_name(),"is", person1.get_ag
e()," years old.")
print(person2.get_name(),"
is", person2.get_age()," years old.")

# Update age
person1.update_age(26)

# Check the updated age
print(person1.get_name(),"is
now",person1.get_age())
```

OUTPUT

```
Alice is 25  years old.
Bob  is 30  years old.
Alice is now 26
```

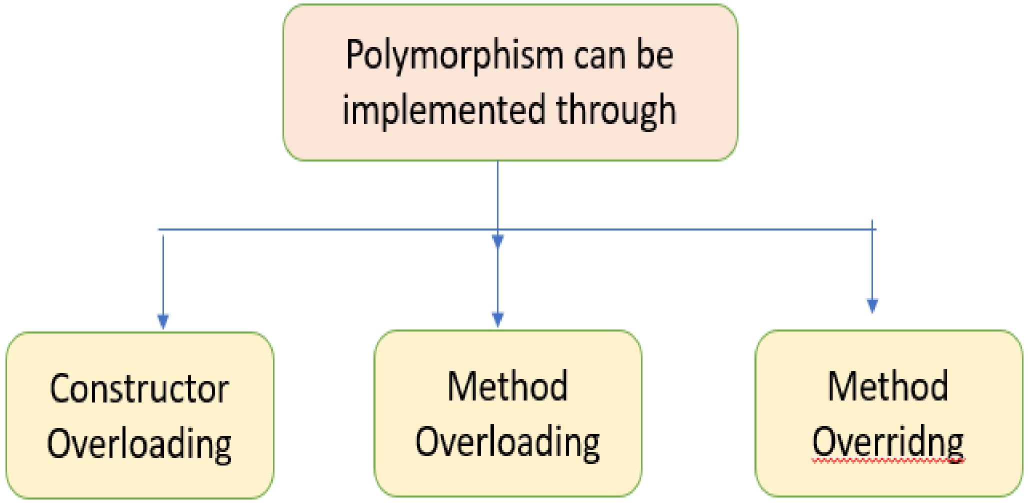
2. Encapsulation

```
class Rectangle:
    def __init__(self,l,b):
        self.l=l
        self.b=b
    def area(self):
        return self.l*self.b


r1=Rectangle(2,3)
print('Area of first Rectangle',r1.area())
r2 = Rectangle(4,5)
print('Area of second rectangle',r2.area())
```

OUTPUT

```
Area of first Rectangle 6
Area of second rectangle 20
```

3. Polymorphism



3. 1 Constructor Overloading

CONSTRUCTOR OVERLOADING

- ❖ In Python, **constructor overloading** refers to the ability to **define multiple constructors** (initializers) for a class **with different sets of parameters**.
- ❖ It allows you to create objects of a class in various ways, depending on the parameters provided.
- ❖ Unlike some other programming languages, Python does not support **explicit constructor overloading like method overloading** (where you can define multiple methods with the same name but different parameter lists).
- ❖ Instead, ***Python uses a different approach to achieve constructor overloading by utilizing default values and optional arguments***.

3. 1 Constructor Overloading

```python
class Const:
    def __init__(self,a=0,b=0,c=0):
        self.a=a
        self.b=b
        self.c=c
    def sum(self):
        return self.a+self.b+self.c
s1 = Const(3,4)
sum2 = s1.sum()
print('sum of 2 numbers',sum2)
s2 = Const(2,3,4)
sum3 = s2.sum()
print('sum of 3 numbers',sum3)
s3 = Const()
sum0 = s3.sum()
print("sum of 0 numbers is ",sum0)
```

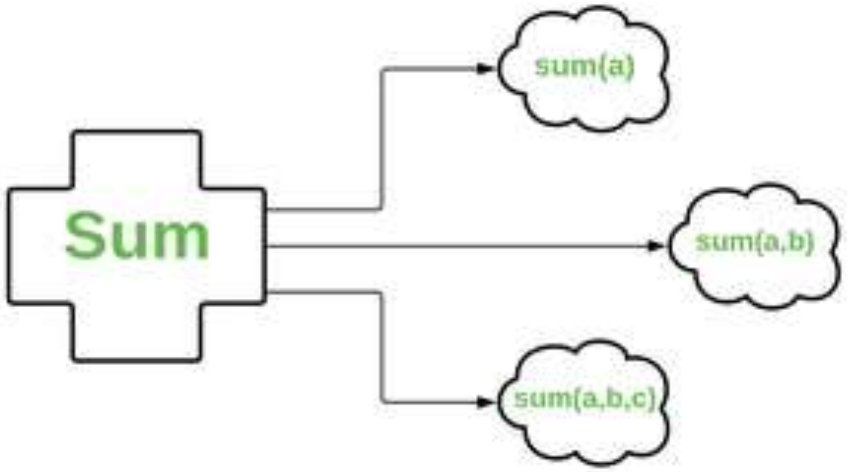
OUTPUT

```
sum of 2 numbers 7
sum of 3 numbers 9
sum of 0 numbers is  0
```

3.2 Method Overloading

METHOD OVERLOADING

- In Python, **method overloading** refers to the ability to **define multiple methods in a class with the same name but different parameter lists**.
- Each method performs a different operation based on the number or type of arguments passed to it. .



3.2 Method Overloading

```python
class Meth:
    def sum(self,x=0,y=0,z=0):
        return x+y+z
s1=Meth()
so2=s1.sum(2,3)
so3=s1.sum(1,2,3)
print("sum of 2 numbers is ",so2)
print("sum of 3 numbers is ",so3)
```

OUTPUT

```
sum of 2 numbers is  5
sum of 3 numbers is  6
```

3.2 Method Overloading

```
class methOverloading:
    def add(self, a, b):
        return a + b

    def add(self, a, b, c):
        return a + b + c

    def add(self, *args):
        return sum(args)


# Creating an object of MyClass
obj = methOverloading()

# Calling the overloaded add method
print(obj.add(2, 3))                 # Output: 5
print(obj.add(2, 3, 4))              # Output: 9
print(obj.add(2, 3, 4, 5, 6))        # Output: 20
```

4. Inheritance

- ❖ **Inheritance** is a fundamental concept in object-oriented programming (OOP) that allows **a class (called the "child" or "derived" class) to inherit properties and behaviors from another class (called the "parent" or "base" class).**
- ❖ **The child class can extend or modify the functionality of the parent class while reusing its existing features.**
- ❖ In Python, inheritance is supported, and it plays a significant role in building complex and organized code structures

4.1 Types of Inheritance

In Python, there are several types of inheritance that allow classes to inherit properties and behaviors from other classes in different ways. The common types of inheritance in Python are:

1) Single Inheritance:

Single inheritance involves a class inheriting from only one base class. It is the most common type of inheritance and represents a simple hierarchy of classes.

```
class ParentClass:
    pass

class ChildClass(ParentClass):
    pass
```

4.1 Types of Inheritance

4) Hierarchical Inheritance:

Hierarchical inheritance involves multiple classes inheriting from the same base class. It creates a hierarchy of classes derived from a common base.

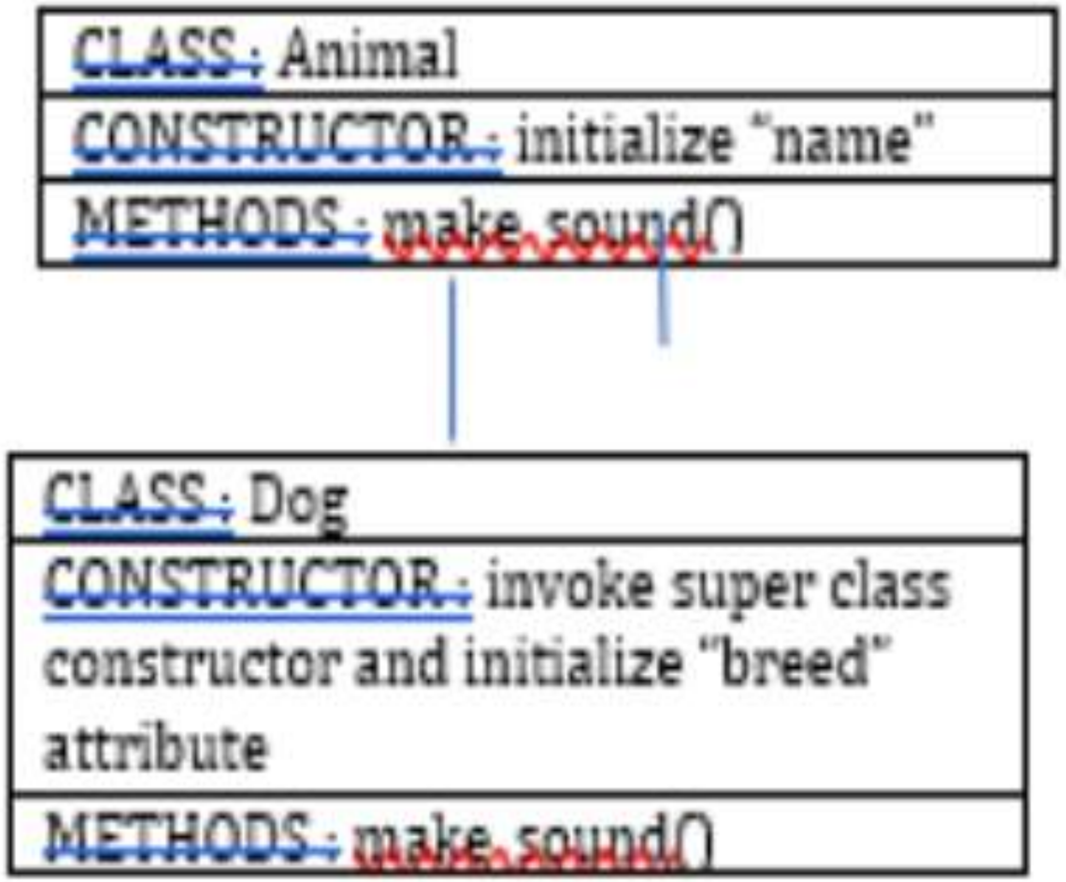
```
class ParentClass:
    pass

class ChildClass1(ParentClass):
    pass

class ChildClass2(ParentClass):
    pass
```

5) Hybrid Inheritance:

Hybrid Inheritance (Mixing Multiple and Multilevel Inheritance): Hybrid inheritance is a combination of multiple inheritance and multilevel inheritance.

```
class GrandparentClass:
    pass

class ParentClass1(GrandparentClass):
    pass

class ParentClass2(GrandparentClass):
    pass

class ChildClass(ParentClass1, ParentClass2):
    pass
```

4.2 Single Inheritance


CLASS : Animal
CONSTRUCTOR : initialize "name"
METHODS : make_sound()

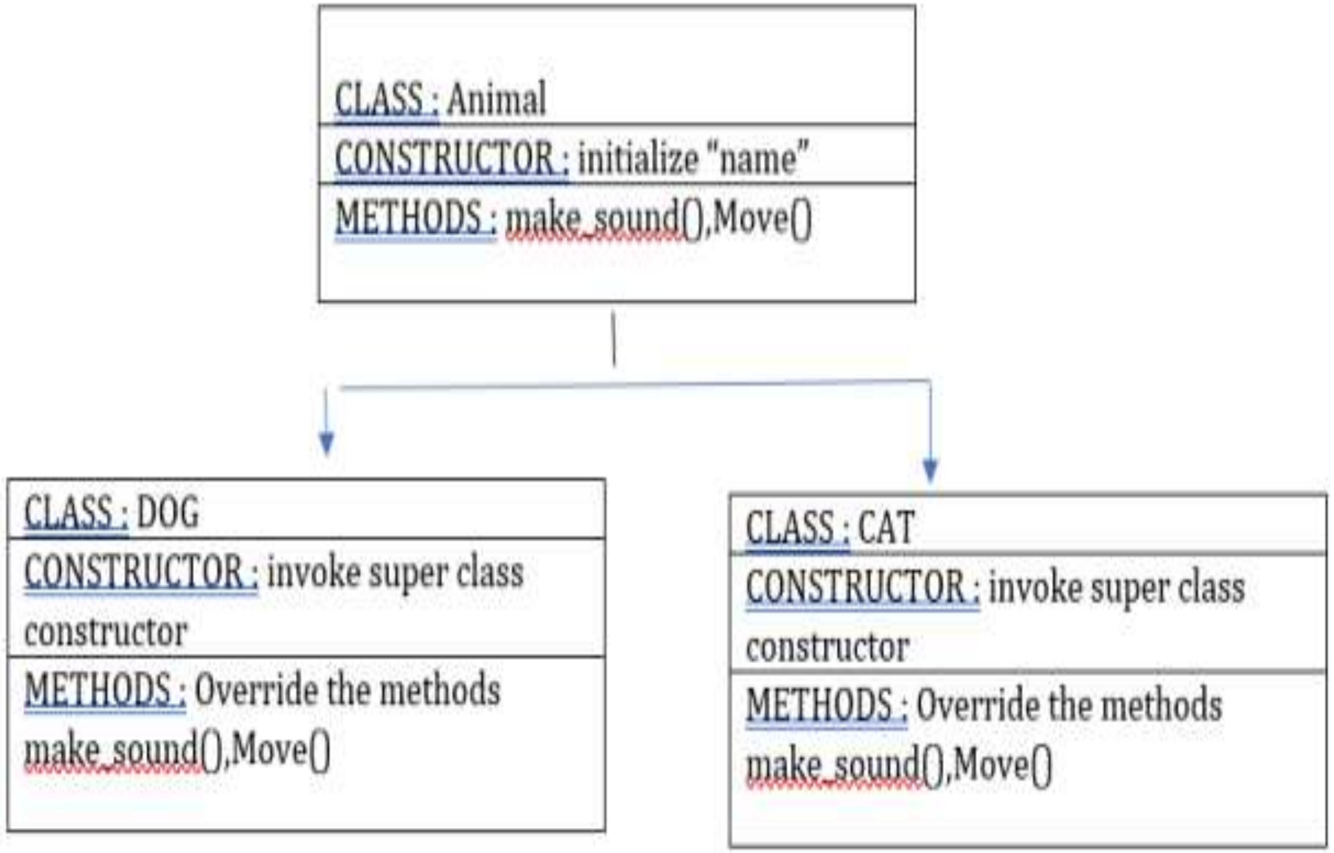
CLASS : Dog
CONSTRUCTOR : invoke super class constructor and initialize "breed" attribute
METHODS : make_sound()

4.2 Single Inheritance

```
# Base class
class Animal:
    def __init__(self, name):
        self.name = name

    def make_sound(self):
        pass
# Derived class inheriting from the Animal class
class Dog(Animal):
    def __init__(self, name, breed):
        # Call the constructor of the base class using the super()
function
        super().__init__(name)
        self.breed = breed
    def make_sound(self):
        return "Woof!"
# Create instances of the derived classes
dog = Dog("Buddy", "Golden Retriever")
# Accessing attributes and methods of the base class through the
derived classes
print(dog.name ," is a", dog.breed ," and says ", dog.make_sound())
```

```
Buddy  is a Golden Retriever  and says  Woof!
```

4.3 Hierarchical Inheritance

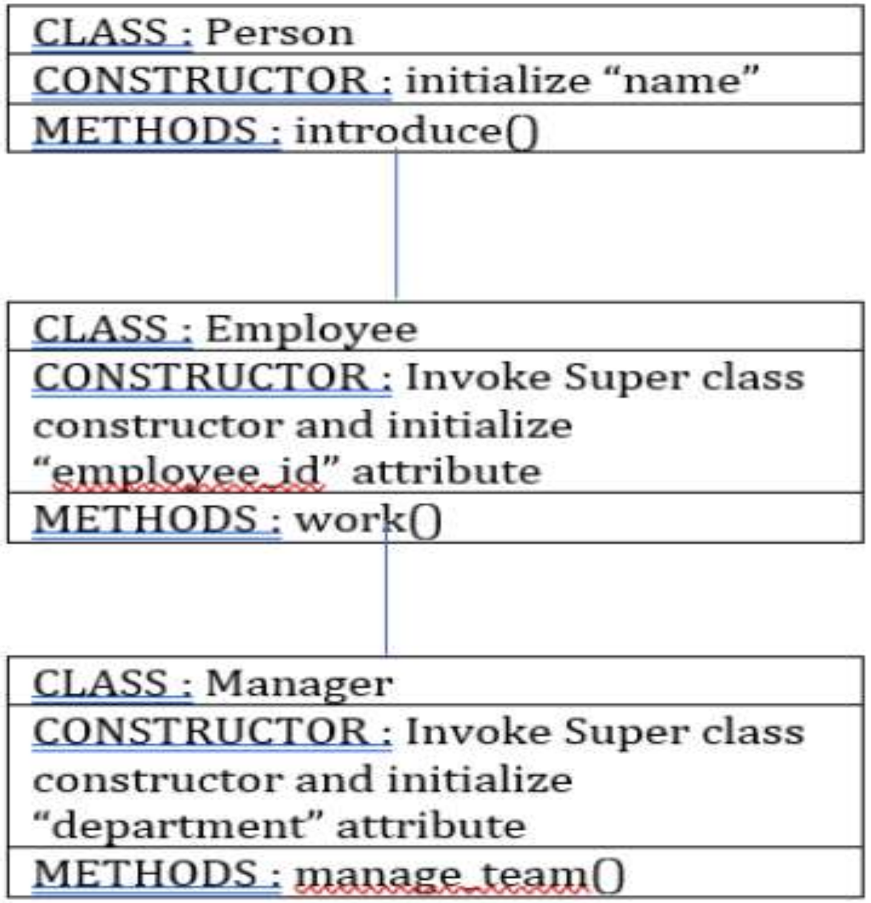

CLASS : Animal
CONSTRUCTOR : initialize "name"
METHODS : make_sound(),Move()
CLASS : DOG
CONSTRUCTOR : invoke super class constructor
METHODS : Override the methods make_sound(),Move()
CLASS : CAT
CONSTRUCTOR : invoke super class constructor
METHODS : Override the methods make_sound(),Move()

4.3 Hierarchical Inheritance

```python
class Animal:
    def __init__(self, name):
        self.name = name

    def make_sound(self):
        pass

    def move(self):
        print(f"{self.name} is moving.")

class Dog(Animal):
    def make_sound(self):
        print("Woof!")

class Cat(Animal):
    def make_sound(self):
        print("Meow!")
```

```python
# Creating objects of the derived classes
dog = Dog("Buddy")
cat = Cat("Whiskers")

# Calling the methods
dog.make_sound()  # Output: Woof!
dog.move()        # Output: Buddy is moving.

cat.make_sound()  # Output: Meow!
cat.move()        # Output: Whiskers is moving.
```

4.3 Multi level Inheritance



CLASS : Person
CONSTRUCTOR : initialize “name”
METHODS : introduce()

CLASS : Employee
CONSTRUCTOR : Invoke Super class constructor and initialize “employee_id” attribute
METHODS : work()

CLASS : Manager
CONSTRUCTOR : Invoke Super class constructor and initialize “department” attribute
METHODS : manage_team()

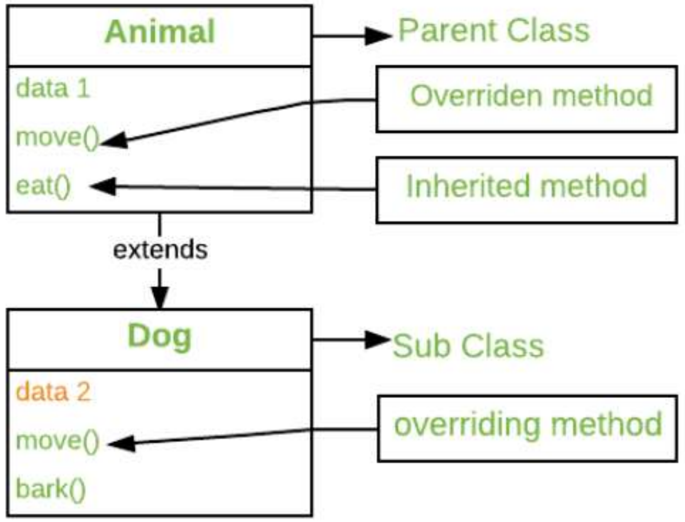
```python
# Base class
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def introduce(self):
        return  "Hi  my name is ", self.name , " and I am ", self.age ," years old."
# Derived class inheriting from the Person class
class Employee(Person):
    def __init__(self, name, age, employee_id):
        super().__init__(name, age)
        self.employee_id = employee_id
    def work(self):
        return "I am an employee working with ID ", self.employee_id
# Further derived class inheriting from the Employee class
class Manager(Employee):
    def __init__(self, name, age, employee_id, department):
        super().__init__(name, age, employee_id)
        self.department = department
    def manage_team(self):
        return "I am managing a team in the department of ", self.department
# Create an instance of the Manager class
manager = Manager("John Doe", 35, "12345", "Sales")
# Accessing attributes and methods through multilevel inheritance
print(manager.introduce())    # Calls the introduce() method in the Manager class
print(manager.work())         # Calls the work() method in the Employee class
print(manager.manage_team())  # Calls the manage_team() method in the Manager class
```

```
('Hi  my name is ', 'John Doe', ' and I am ', 35, ' years old.')
('I am an employee working with ID ', '12345')
('I am managing a team in the department of ', 'Sales')
```

4.4 Method Overriding

Method Overriding

When a method in a subclass has the same name, same parameters or signature and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to **override** the method in the super-class.



- ❖ The version of a method that is executed will be determined by the object that is used to invoke it.
- ❖ If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed

4.4 Method Overriding

❖ method overriding allows you to change or extend the behavior of the inherited method to suit the needs of the subclass.

Method Overriding

```
class Animal:
    def makeSound(self):
        print("Animal makes a sound");

class Dog(Animal):
    #Override
    def makeSound(self):
        print("Dog barks");
d = Dog()
d.makeSound()
```

Output

```
Dog barks
```

5 Object Oriented Design using UML

- **Object-oriented design (OOD)** is a **process of designing software systems using the principles of object-oriented programming.**
- **Unified Modeling Language (UML)** is a **standard visual representation used to model and visualize the various aspects of object-oriented systems**.
- UML provides a set of diagrams to describe different perspectives of a software system, aiding in the understanding, communication, and design of complex systems.
- **There are several types of UML diagrams used in object-oriented design. Here are some of the most commonly used ones:**

1. **Class Diagram:**
   - ✓ Represents the static structure of the system.
   - ✓ Shows classes, their attributes, methods, and relationships between classes.
   - ✓ It is used to visualize the class hierarchy and associations between classes.
2. **Object Diagram:**
   - ✓ Represents a snapshot of instances of classes and their relationships at a particular time.
   - ✓ Shows objects and links between objects with their current data values.

5 Object Oriented Design using UML

3. **Use Case Diagram:**
   - ✓ Represents the functionalities or use cases of the system from the user's perspective.
   - ✓ Shows actors (users or external systems) and their interactions with the system through use cases.
4. **Sequence Diagram:**
   - ✓ Represents the interactions between objects in a particular scenario or use case.
   - ✓ Shows the flow of messages between objects over time, indicating the order of execution.
5. **State Diagram:**
   - ✓ Represents the states and transitions of a single object or a class.
   - ✓ Shows how an object changes its state in response to events.

5 Object Oriented Design using UML

6. **Package Diagram:**
   - ✓ Represents the organization and dependencies between packages or modules in a system.
7. **Component Diagram:**
   - ✓ Represents the physical components and their relationships in the system.
   - ✓ Used to visualize the architecture and distribution of the system.
8. **Deployment Diagram:**
   - ✓ Represents the physical deployment of software components on hardware nodes.
   - ✓ Shows how software artifacts are distributed across different machines.

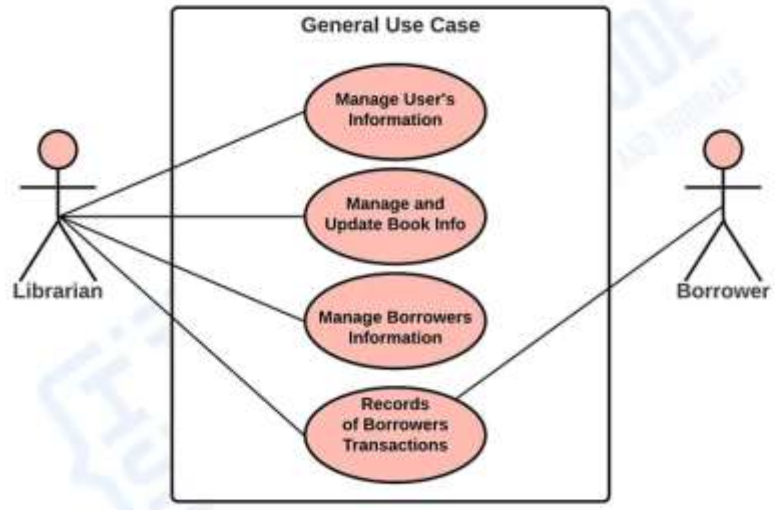
5 Object Oriented Design using UML

UML Diagrams for Library Management Systems

Use Case Diagram

Use Case Diagram of Library Management System contains the main **use cases and users** in the system.

LIBRARY MANAGEMENT SYSTEM

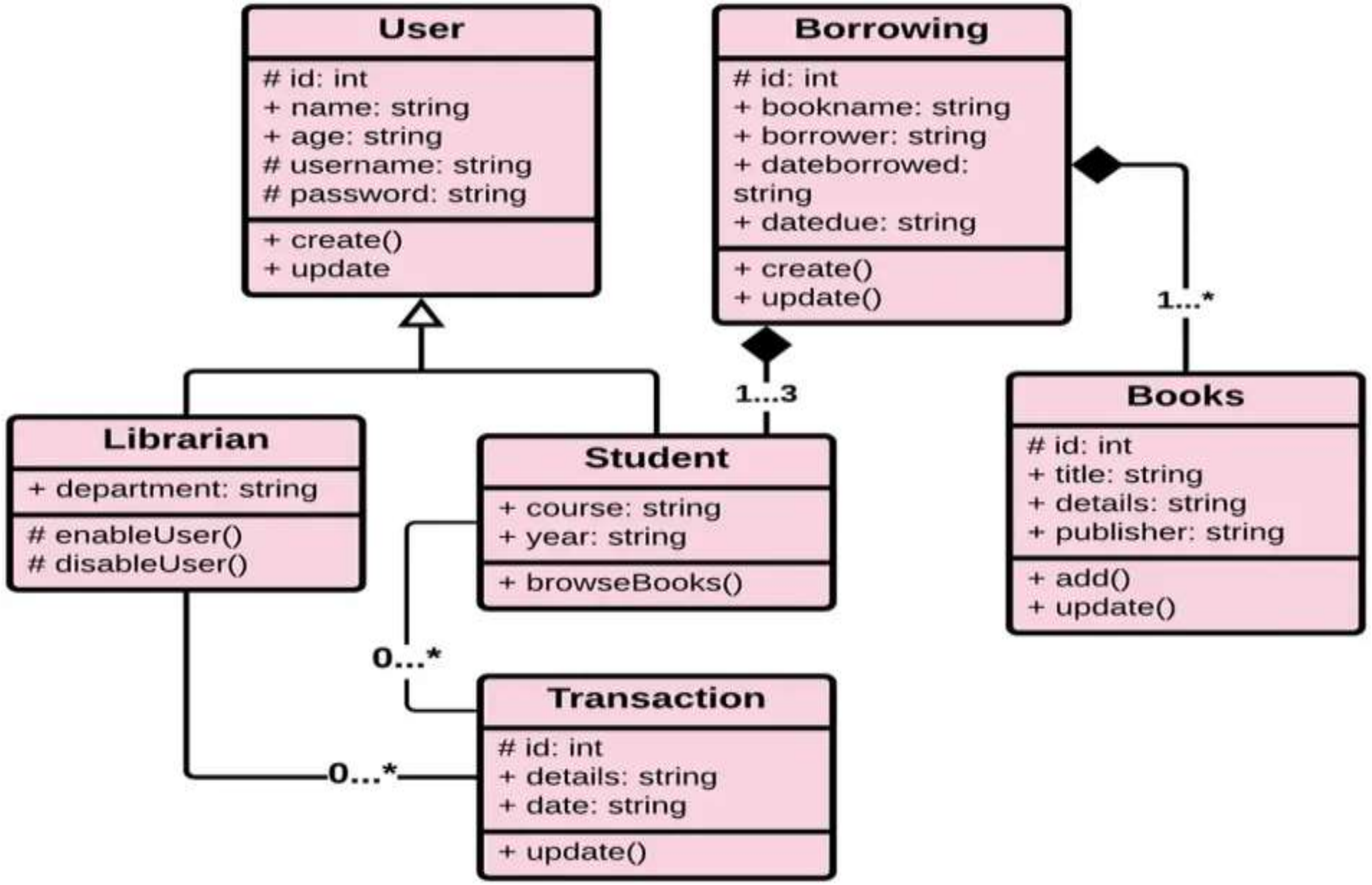

General Use Case
Manage User's Information
Manage and Update Book Info
Manage Borrowers Information
Records of Borrowers Transactions
Librarian
Borrower

USE CASE DIAGRAM

5 Object Oriented Design using UML

Class Diagram

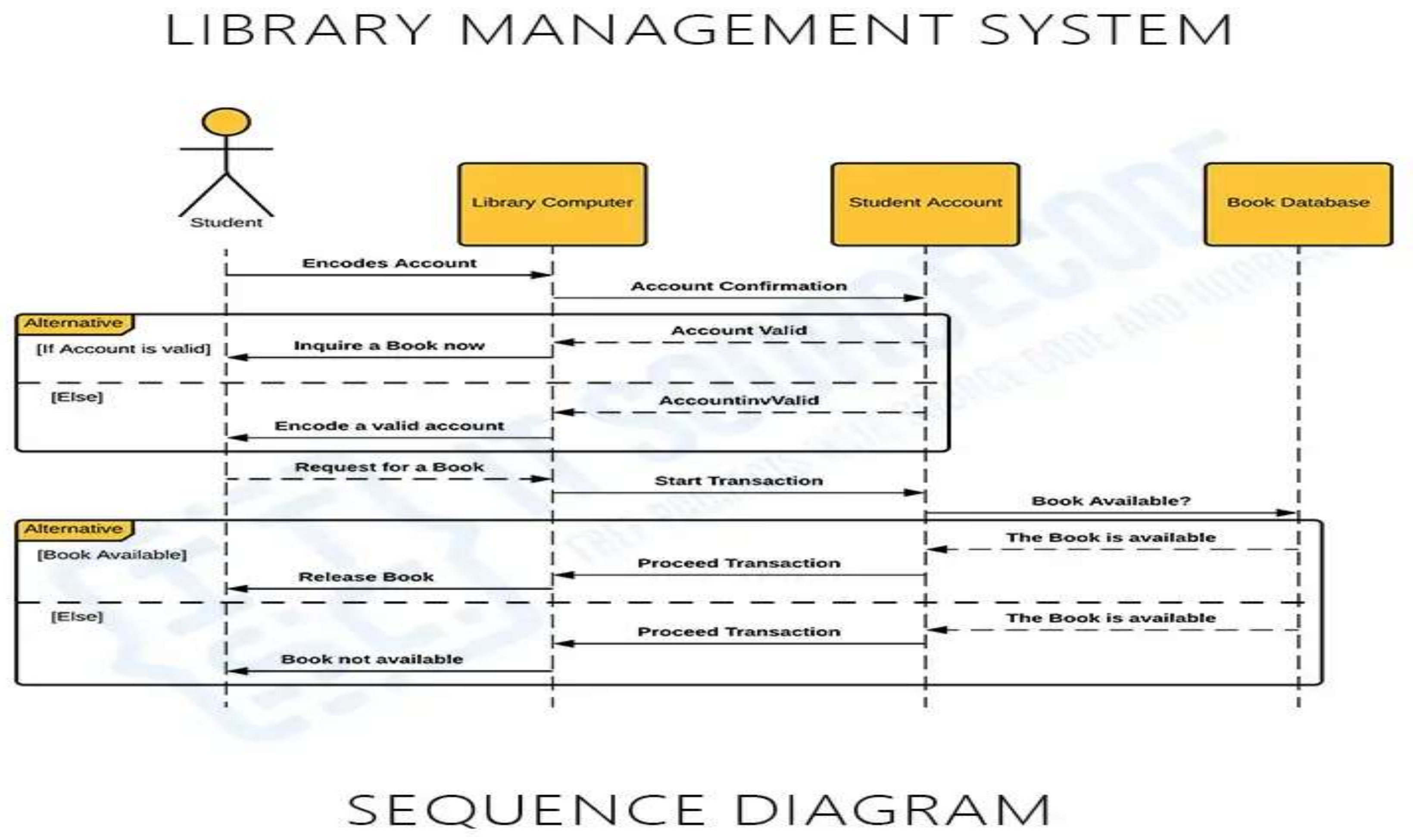
It is used to visualize the class hierarchy and associations between classes of a system



5 Object Oriented Design using UML

Sequence or Interaction Diagram

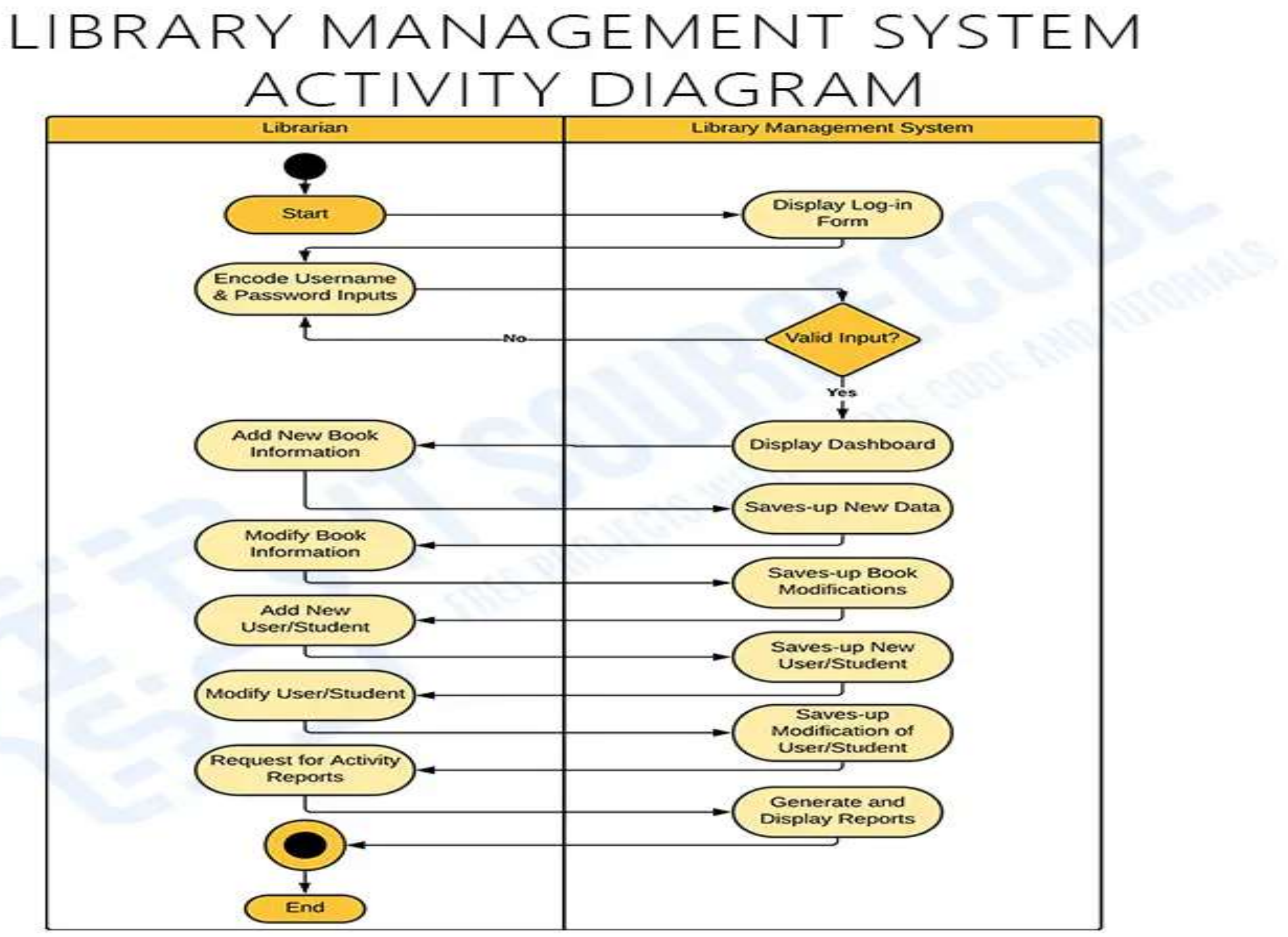
Represents the interactions between objects in a particular scenario or use case.



5 Object Oriented Design using UML

Activity Diagram

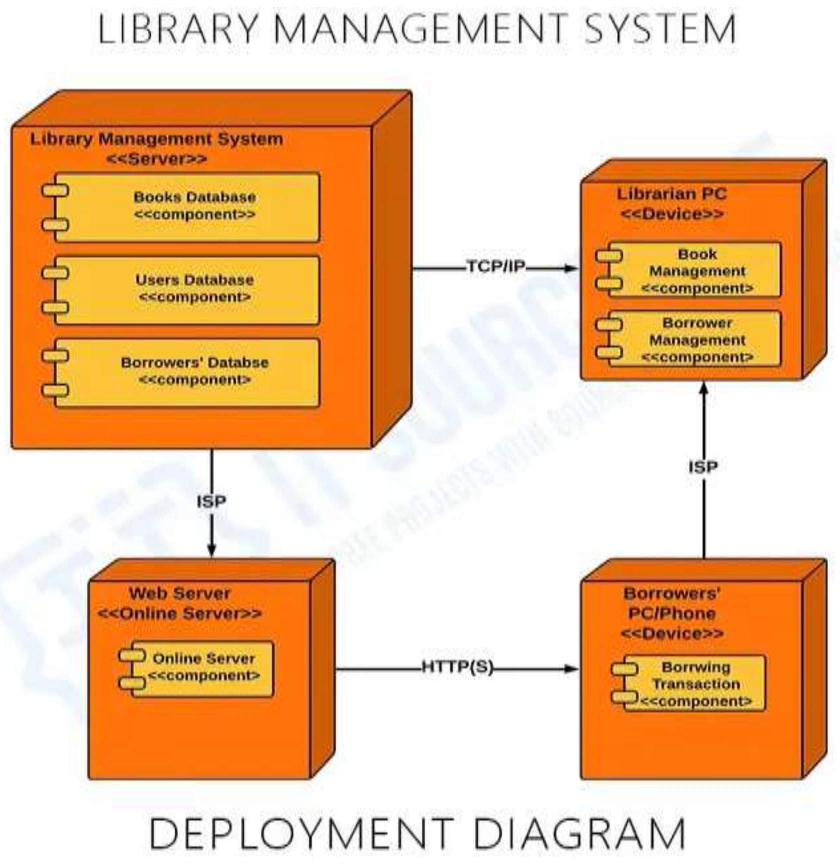
Activity Diagram to illustrate the activities between user and the system


LIBRARY MANAGEMENT SYSTEM ACTIVITY DIAGRAM
Librarian
Library Management System
Start
Display Log-in Form
Encode Username & Password Inputs
Valid Input?
No
Yes
Display Dashboard
Add New Book Information
Saves-up New Data
Modify Book Information
Saves-up Book Modifications
Add New User/Student
Saves-up New User/Student
Modify User/Student
Saves-up Modification of User/Student
Request for Activity Reports
Generate and Display Reports
End

5 Object Oriented Design using UML

Deployment Diagram

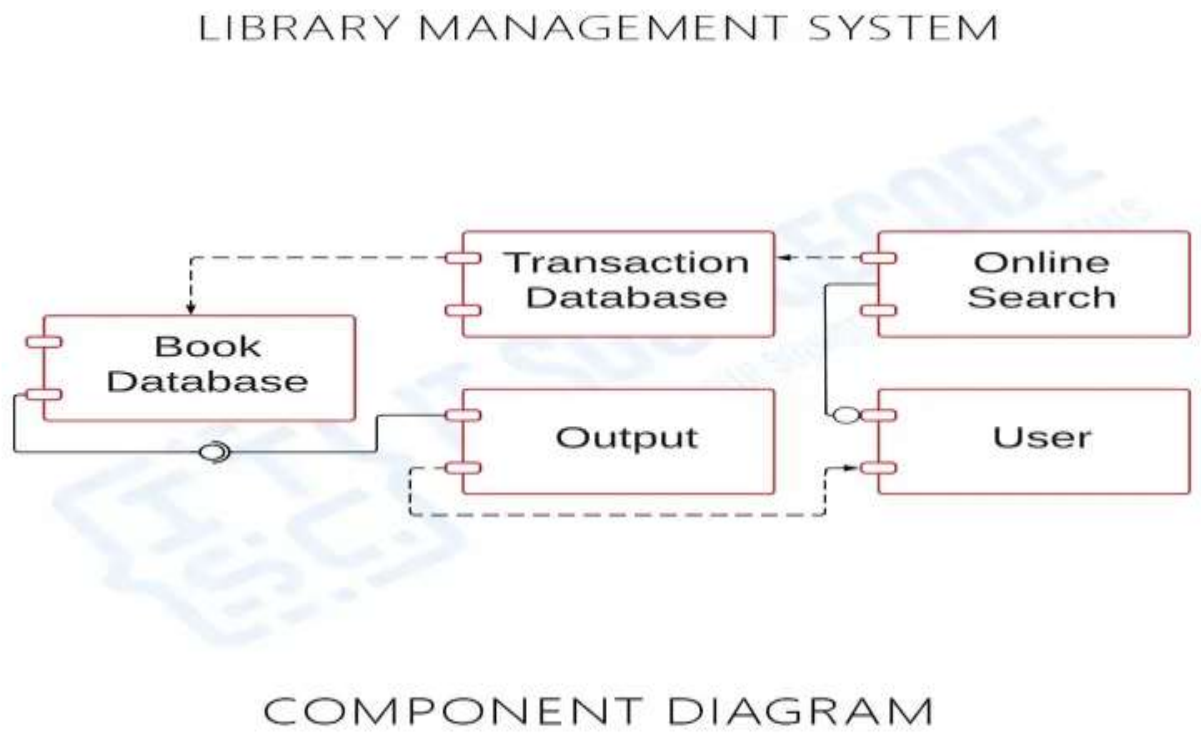
Represents the physical deployment of software components on hardware nodes.


LIBRARY MANAGEMENT SYSTEM
Library Management System <<Server>>
Books Database <<component>>
Users Database <<component>
Borrowers' Databse <<component>
TCP/IP
Librarian PC <<Device>>
Book Management <<component>
Borrower Management <<component>
ISP
ISP
Web Server <<Online Server>>
Online Server <<component>
HTTP(S)
Borrowers' PC/Phone <<Device>>
Borrwing Transaction <<component>
DEPLOYMENT DIAGRAM

5 Object Oriented Design using UML

Component Diagram

- ✓ Used to visualize the architecture and distribution of the system.



6 Computational Problem Solving

- Computational problem solving is the process of using computational tools, algorithms, and techniques to solve problems efficiently and effectively.
- It involves breaking down complex problems into smaller, more manageable tasks and then employing computational methods to analyze and find solutions.
- This approach is widely used in various fields, including computer science, engineering, mathematics, physics, and many other disciplines.
- The steps involved in computational problem solving typically include:

1) Problem Understanding: Clearly defining the problem and understanding its requirements, constraints, and objectives. It is essential to have a thorough understanding of the problem's context before attempting to solve it computationally.
2) Algorithm Design: Designing a step-by-step procedure or algorithm that outlines the solution to the problem. The algorithm should be precise, unambiguous, and capable of handling various input scenarios.

6. Computational Problem Solving

3) Data Representation: Identifying the data required for the problem and determining the appropriate data structures to represent and manipulate that data. Choosing the right data structures can significantly impact the efficiency of the solution.
4) Coding: Translating the algorithm into a specific programming language and writing the code that implements the solution. This step involves using the syntax and features of the chosen programming language effectively.
5) Testing: Thoroughly testing the code with different test cases to ensure that it produces the correct output for various inputs. Testing helps identify and fix any errors or bugs in the implementation.
6) Optimization: Analyzing the algorithm and code to find ways to improve efficiency and reduce computational complexity. Optimization may involve reducing time and space complexity to make the solution more scalable and faster.
7) Execution and Evaluation: Running the code on real data or inputs to obtain the solution and evaluating the results to determine its accuracy and effectiveness in solving the problem.
8) Iteration: If necessary, revisiting the problem-solving process to make improvements or adjustments based on feedback or new requirements.

7. Vehicle Rental Agency Problem

- The vehicle rental agency problem involves designing a program that simulates the operations of a **vehicle rental agency.**
- The agency typically offers **various types of vehicles, such as cars, bikes, motorcycles, vans, etc., for rent to customers on a short-term basis.**
- The primary goal of the vehicle rental agency is **to manage the rental and return processes efficiently.**
- Customers can rent a vehicle for a specific duration and then return it when they are done using it.
- During the rental period, the vehicle should be marked as "rented" and unavailable for other customers to rent.
- Once returned, the vehicle becomes available for other customers again.

Key features of the vehicle rental agency problem include:

1) **Vehicle Types**: The agency offers a fleet of different types of vehicles, each with its own attributes (e.g., make, model, year, capacity, etc.).
2) **Rental Process**: Customers can request to rent a specific vehicle based on their requirements. The rental agency needs to keep track of the availability status of each vehicle and allow customers to rent only those that are currently available.

7. Vehicle Rental Agency Problem

3) **Return Process**: After the rental period is over, customers return the vehicle to the agency. The agency then updates the vehicle's status to "available" for other customers to rent.

4) **Inventory Management**: The agency needs to maintain an inventory of available vehicles, ensuring that the number of available vehicles is updated correctly after each rental and return.

5) **Rental Duration and Pricing**: The rental agency may have different pricing models based on the type of vehicle and the duration of the rental. Calculating the rental cost and handling payments is an essential aspect of the problem.

6) **Vehicle Inspection**: The agency may need to inspect vehicles for damages or issues before renting them out again to ensure customer safety and satisfaction.

7) **Customer Management**: The agency may keep records of customer information, rental history, and other details for better customer service.

7. Vehicle Rental Agency Problem

```
class Vehicle:
    def __init__(self, make, model, year, is_available=True):
        self.make = make
        self.model = model
        self.year = year
        self.is_available = is_available
class RentalAgency:
    def __init__(self):
        self.vehicles = []
    def add_vehicle(self, vehicle):
        self.vehicles.append(vehicle)
```

7. Vehicle Rental Agency Problem

```
def display_available_vehicles(self):
        available_vehicles=[]
        for i in self.vehicles:
            if i.is_available ==True:
                available_vehicles.append(i)
        if len(available_vehicles) >0:
            for k in available_vehicles:
                print(f"Available Vehicle is  {k.make} of  k.model} {k.year} ")
        else:
            print("No vehicles are currently available.")
```

7. Vehicle Rental Agency Problem

```
def rent_vehicle(self, make, model, year):
        for i in self.vehicles:
            if i.make == make and i.model == model and i.year ==  year and i.is_available:
                i.is_available = False
                print(f"You have successfully rented the {year} {make} {model}.")
                return
        print(f"Sorry, the {year} {make} {model} is not available for rent.")
def return_vehicle(self, make, model, year):
        for k in self.vehicles:
            if k.make == make and k.model == model and k.year == year and not k.is_available:
                k.is_available = True
                print(f"You have successfully returned the {year} {make} {model}.")
                return
        print("Invalid vehicle details. Please check and try again.")
```

7. Vehicle Rental Agency Problem

```python
agency = RentalAgency()
vehicle1 = Vehicle("Toyota", "Corolla", 2020)
vehicle2 = Vehicle("Honda", "Civic", 2019)
vehicle3 = Vehicle("Ford", "F-150", 2021)
agency.add_vehicle(vehicle1)
agency.add_vehicle(vehicle2)
agency.add_vehicle(vehicle3)
agency.display_available_vehicles()
agency.rent_vehicle("Toyota", "Corolla", 2020)
agency.rent_vehicle("Ford", "F-150", 2021)
agency.display_available_vehicles()
agency.return_vehicle("Toyota", "Corolla", 2020)
agency.display_available_vehicles()
```

END OF THE UNIT - IV