Chapter 11 Relationships in OOP

When we design a software using Object-oriented Programming (OOP) methodology, we will try to break our software into classes and create proper relationships between those classes so that all of them can work together.

It is not necessary for every class to be related to all other classes, but a class usually has a relationship with at least one other class.

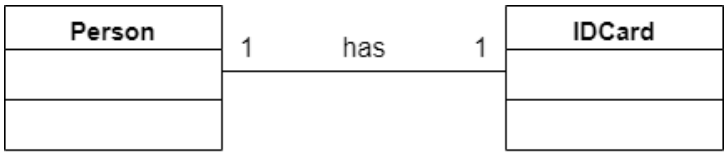
Generally, there are three types of relationships in OOP:

- Association
- Dependency
- Inheritance

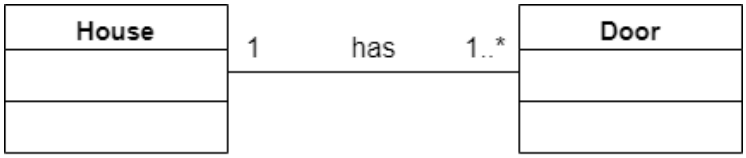
1. Association

In OOP, two classes can associate with each other by one-to-one, one-to-many, many-to-one, or many-to-many.

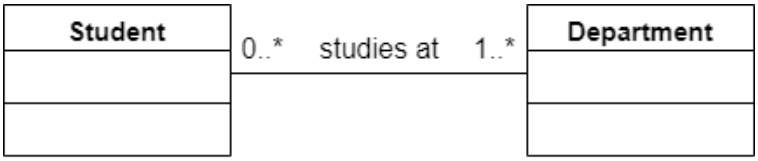
The relationship between `Person` and `IDCard` is one-to-one.



The relationship between `House` and `Door` is one-to-many.



The relationship between `Student` and `Department` is many-to-many.



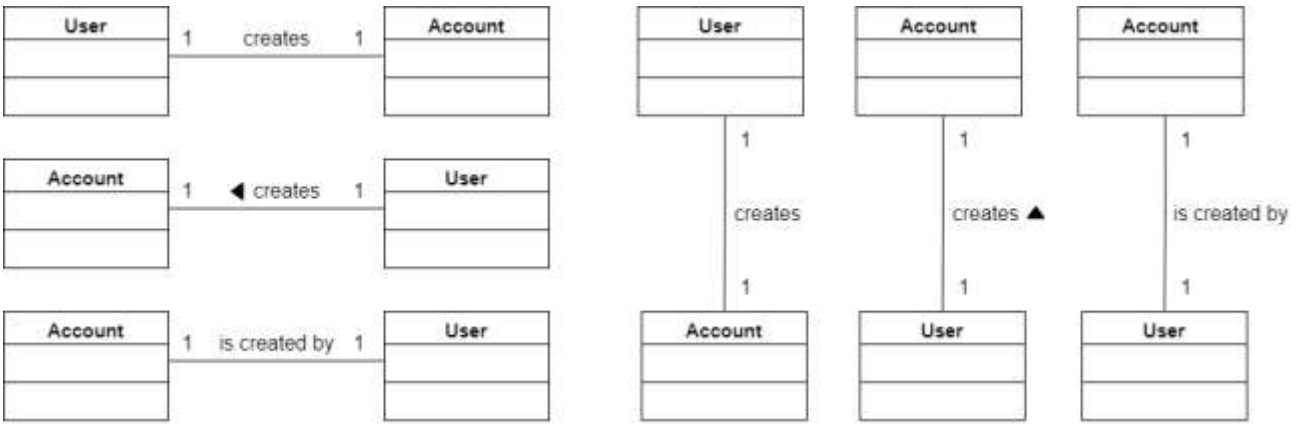
1.1 Multiplicity

A multiplicity is an indication of the **minimum** and the **maximum** allowable number of objects that can participate in a given relationship. It is an essential concept in OOP and database design, representing relationships between objects or entities.

Exactly one	1	Department — has — 1 Boss	A department has one and only one boss.
Zero or more	0..*	Employee — has — 0..* Child	An employee has zero to many children.
One or more	1..*	Boss — is responsible for — 1..* Employee	A boss is responsible for one or more employees.
Zero or one	0..1	Employee — is married to — 0..1 Spouse	An employee can be married to zero or one spouse.
Specified range	2..4	Employee — takes — 2..4 Vacation	An employee can take from two to four vacations each year.
Multiple, disjoint ranges	1..3,5	Employee — is a member of — 1..3,5 Committee	An employee is a member of one to three or five committees.

1.2 Reading Direction

Relationships should be read from left to right and top to bottom.



Practice

Draw the class diagrams showing the relationships between classes below:

1. Person and Facebook Account
2. Employee and Company
3. Library and Book
4. Bank and Account
5. Teacher and Subject
6. Employee and Employee (Unary Relationship)

Download drawio

Exercises

I. Draw the class diagrams showing the relationships between classes below:

1. Student and Course
2. Homework and Course
3. Person and House
4. Person and ID card
5. Person and Person
6. Student and Account
7. Hotel and Room
8. Guest and Booking
9. Truck and Driver
10. Doctor and Hospital
11. Doctor and Prescription
12. Prescription and Medicine
13. Doctor and Patient
14. Character and Game
15. University and Parking Lot

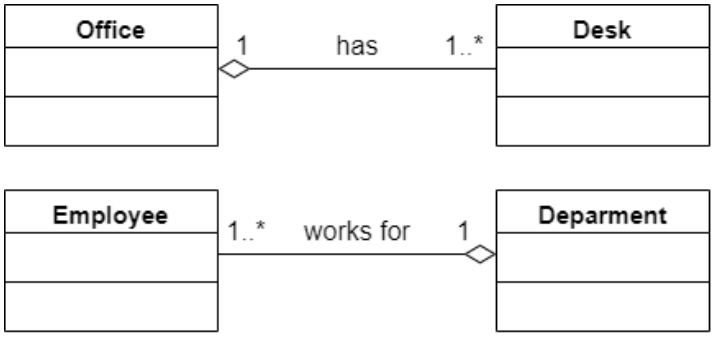
Note that you are not allowed to move any class to left or right!

Association relationship can be further classified into **Aggregation** and **Composition**.

1.3 Aggregation

In this relationship, one object of a class (parent) contains one or many objects of another class (child). In aggregation, a child can exist independent of the parent.

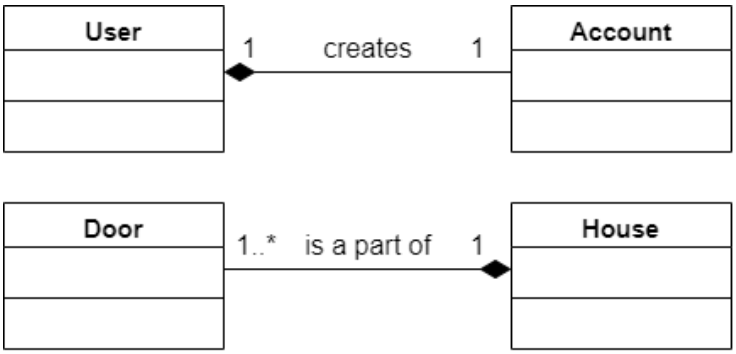
Let's see some examples below:



1.4 Composition

In this relationship, one object of a class (parent) contains one or many objects of another class (child). In aggregation, a child **cannot exist without its parent**.

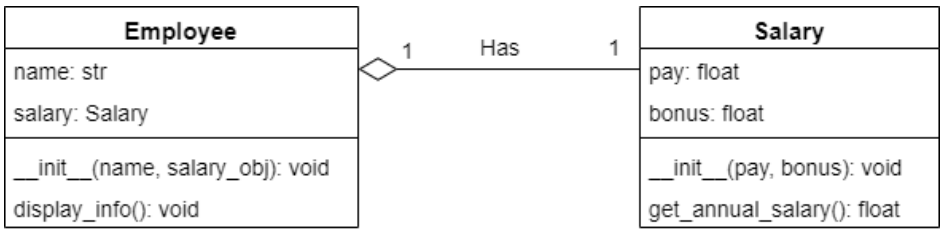
Let's see some examples below:



1.6 Implementing Association Relationships

To implement an association relationship, a class must have another class as any of it attribute.

Example 01: Aggregation relationship in a one-to-one relationship.



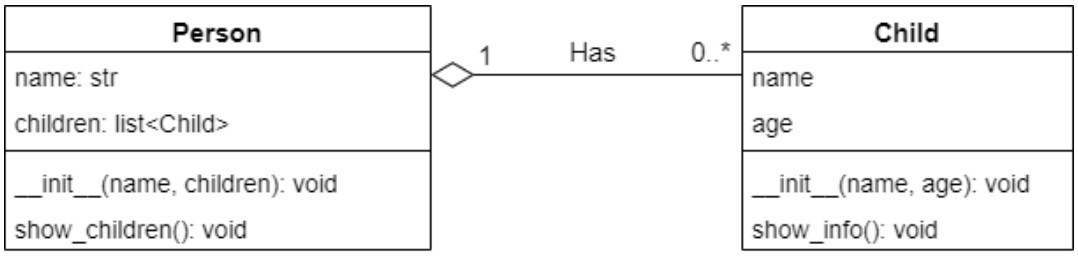

```
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

    def display_info(self):
        print(self.name, "gets", salary.get_annual_salary(), "USD/year")

class Salary:
    def __init__(self, pay, bonus):
        self.pay = pay
        self.bonus = bonus

    def get_annual_salary(self):
        return self.pay * 12 + self.bonus

salary = Salary(600, 50)
employee = Employee("John", salary)
employee.display_info()
```

Output:

```
John gets 7250 USD/year
```

In the aggregation relationship, child object **can exist without its parent object**. In the above example, we do not create an object of the **`Salary`** class inside the **`Employee`** class. Instead, we create the object of the **`Salary`** class **outside** and passing it as a parameter of the **`Employee`** class. So, the life time of the **`Salary`** object does not depend on the life time of the **`Employee`** object.

Example 02: Aggregation relationship in a one-to-many relationship.

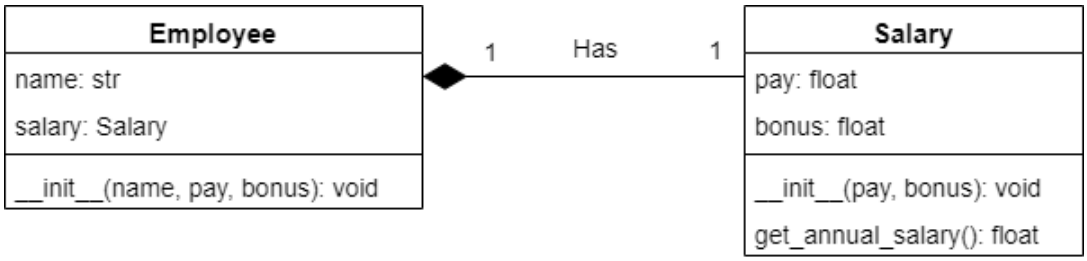



```python
class Person:
    def __init__(self, name, children = []):
        self.name = name
        self.children = children

    def show_children(self):
        print("Here are the children of", self.name, ":")
        for child in self.children:
            child.show_info()

class Child:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def show_info(self):
        print("Child Name: ", self.name, ", Age", self.age)

parent = Person("John")
parent.children.append(Child("Jack", 20))
parent.children.append(Child("Lucy", 22))
parent.show_children()
```

Output:

```
Here are the children of John :
Child Name:  Jack , Age 20
Child Name:  Lucy , Age 22
```

Example 03: Composition relationship in a one-to-one relationship.



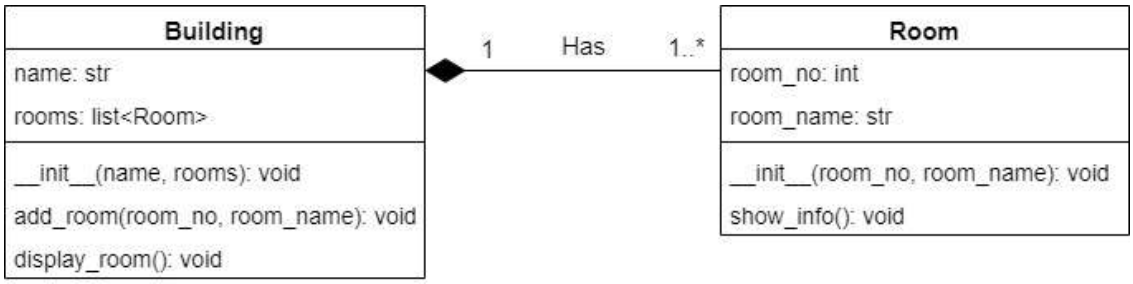

```
class Employee:
    def __init__(self, name, pay, bonus):
        self.name = name
        self.salary = Salary(pay, bonus)

class Salary:
    def __init__(self, pay, bonus):
        self.pay = pay
        self.bonus = bonus

    def get_annual_salary(self):
        return self.pay * 12 + self.bonus

employee = Employee("John", 600, 50)
salary = employee.salary.get_annual_salary()
print(employee.name, "gets", salary, "USD/year")
```

Output:

```
John gets 7250 USD/year
```

In the above example, we get the same output as we got before using the Aggregation concept. But the difference is that here we create the object of the **`Salary`** class **inside** the **`Employee`** class, rather than that we are creating an object of the **`Salary`** class outside and passing it as a parameter of **`Employee`** class. So, the life time of the **`Salary`** object depends on the life time of the **`Employee`** object.

Example 04: Composition relationship in a one-to-many relationship.




```
class Building:
    def __init__(self, name):
        self.name = name
        self.rooms = []

    def add_room(self, room_no, room_name):

        room = Room(room_no, room_name)

        self.rooms.append(room)

    def display_rooms(self):
        print("Here are the rooms in", self.name, "building:")
        for room in self.rooms:
            room.show_info()

class Room:
    def __init__(self, room_no, room_name):
        self.room_no = room_no
        self.room_name = room_name

    def show_info(self):
        print("Room No:", self.room_no, ", Name", self.room_name)

building = Building("STEM")
building.add_room(101, "STEM101")
building.add_room(102, "STEM102")
building.display_rooms()
```

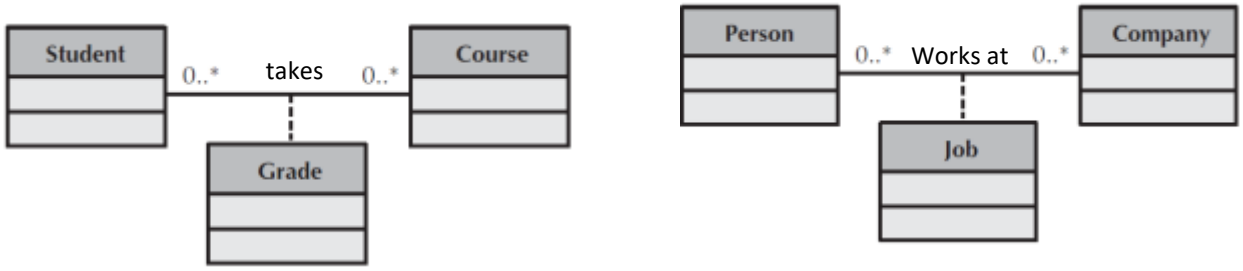
Output:

```
Here are the rooms int STEM building:
Room No: 101 , Name STEM101
Room No: 102 , Name STEM102
```

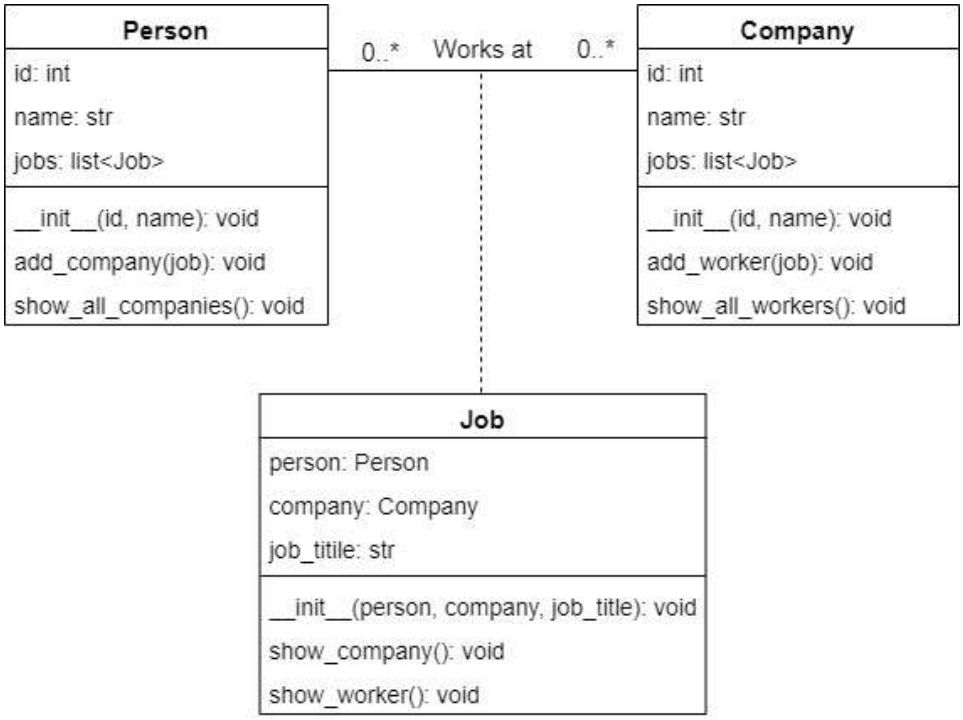
1.5 Association Classes

When two classes have a many-to-many relationship, or when a relationship itself has associated properties, in these cases, an association class is formed, which can have its own attributes and methods.

Let's look at some examples:



Example 05: Association relationship in a many-to-many relationship.




```python
class Person:
    def __init__(self, id, name):
        self.id = id
        self.name = name
        self.jobs = []

    def add_company(self, job):
        self.jobs.append(job)

    def show_all_companies(self):
        for job in self.jobs:
            job.show_company()
```

```python
class Company:
    def __init__(self, id, name):
        self.id = id
        self.name = name
        self.jobs = []

    def add_worker(self, job):
        self.jobs.append(job)

    def show_all_workers(self):
        for job in self.jobs:
            job.show_worker()
class Job:
    def __init__(self, person, company, job_title):
        self.person = person
        self.company = company
        self.job_title = job_title

        person.add_company(self)
        company.add_worker(self)

    def show_company(self):
        print(self.company.name, "as", self.job_title)

    def show_worker(self):
        print(self.person.name, "works as", self.job_title)

# Create persons
john = Person(1, "John")
lucy = Person(2, "Lucy")

# Create companies
google = Company(1, "Google")
amazon = Company(2, "Amazon")

# Create jobs
Job(john, amazon, "Data Analyst")
Job(john, google, "Data Engineer")
Job(lucy, google, "Database Designer")

# Display the companies john works at
print("John works at:")
john.show_all_companies()

# Display the workers who work at Google
print("\nGoogle's workers:")
google.show_all_workers()
```

Output

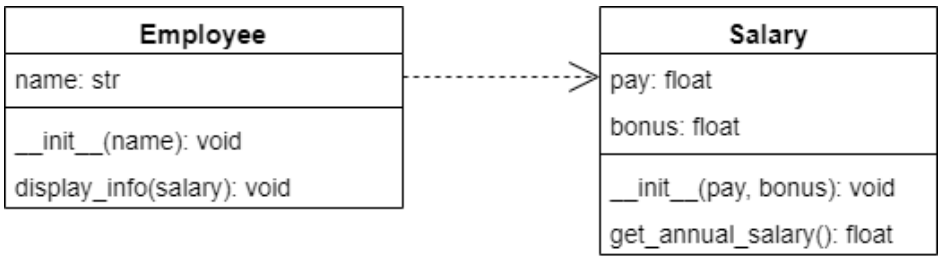
```
John works at:
Amazon as Data Analyst
Google as Data Engineer

Google's workers:
John works as Data Engineer
Lucy works as Database Designer
```

2. Dependency

When a class (client) uses another class (supplier) for its full implementation, the relationship between them is called a **Dependency**.

In this relationship, the client class uses the supplier class as its method parameter, method return type, or a local variable for any of its method.

Example 06: In this example, the `Employee` class uses the `Salary` class as its method parameter.

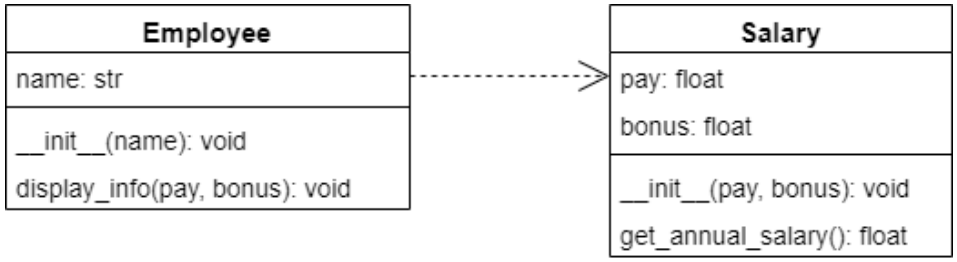



```
class Employee:
    def __init__(self, name):
        self.name = name

    def display_info(self, salary):

        print(self.name, "gets", salary.get_annual_salary(), "USD/year")

class Salary:
    def __init__(self, pay, bonus):
        self.pay = pay
        self.bonus = bonus

    def get_annual_salary(self):
        return self.pay * 12 + self.bonus
employee = Employee("John")
salary = Salary(600, 50)
employee.display_info(salary)
```

Output:

```
John gets 7250 USD/year
```

Example 07: In this example, the **`Employee`** class uses a **`Salary`** class as a local variable for its method.

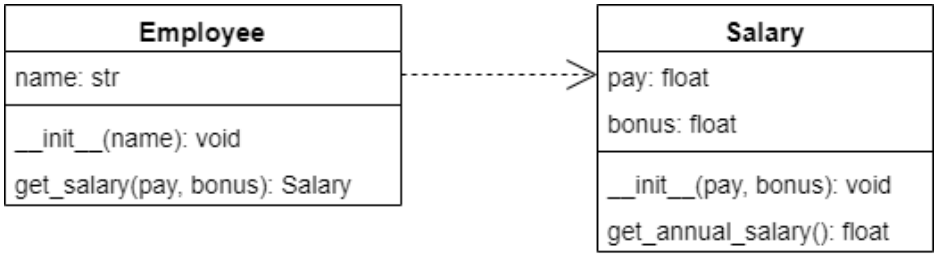



```
class Employee:
    def __init__(self, name):
        self.name = name

    def display_info(self, pay, bonus):

        salary = Salary(pay, bonus)

        print(self.name, "gets", salary.get_annual_salary(), "USD/year")

class Salary:
    def __init__(self, pay, bonus):
        self.pay = pay
        self.bonus = bonus

    def get_annual_salary(self):
        return self.pay * 12 + self.bonus

employee = Employee("John")
employee.display_info(600, 50)
```

Output:

```
John gets 7250 USD/year
```

Example 08: In this example, the **`Employee`** class uses the **`Salary`** class as its method return type.




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

    def get_salary(self, pay, bonus):

        return Salary(pay, bonus)

class Salary:
    def __init__(self, pay, bonus):
        self.pay = pay
        self.bonus = bonus

    def get_annual_salary(self):
        return self.pay * 12 + self.bonus

employee = Employee("John")
salary = employee.get_salary(600, 50)

print(employee.name, "gets", salary.get_annual_salary(), "USD/year")
```

Output:

```
John gets 7250 USD/year
```

3. Inheritance

In OOP, one class is allowed to inherit the features (attributes and methods) of another class. This is called **inheritance**. Inheritance provides **reusability** of a code, i.e., when we want to create a new class and there is already a class that includes some of the code that we want, we let our new class inherit the existing class. By doing this, we are reusing the attributes and methods of the existing class.

In inheritance, there are two kinds of classes:

- **Parent class** (aka. superclass or base class): The class whose features are inherited.
- **Child class** (aka. subclass or derived class): The class that inherits another class. The child class can add its own attributes and methods in addition to the parent class's attributes and methods. Also, a child class can also provide its specific implementation to the methods of the parent class.

Syntax: Here's the syntax of the inheritance:

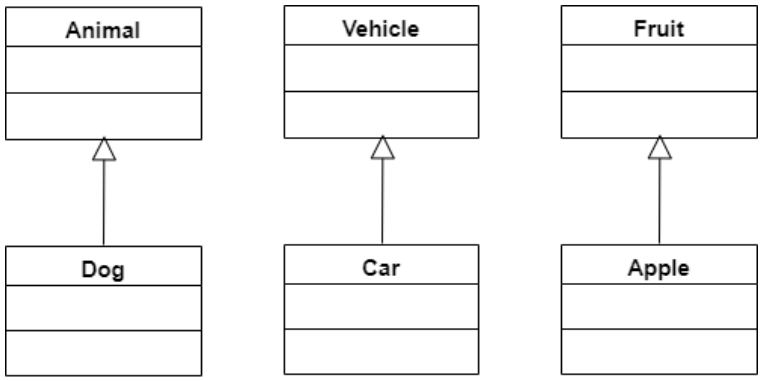
```
# define a parent class
class ParentClass:
    # attributes
    # methods

# define a child class
class ChildClass(ParentClass): # inheritance
    # attributes
    # methods
```

We use inheritance only if there exists an **is-a relationship** between two classes. For example,

`Dog` **is an** `Animal`, so `Dog` can inherit `Animal`

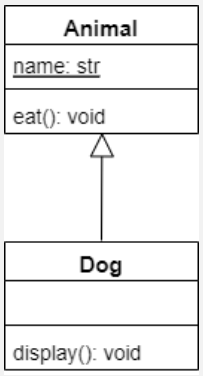
`Car` **is a** `Vehicle`, so `Car` can inherit `Vehicle`

`Apple` **is a** `Fruit`, so `Apple` can inherits `Fruit`



Example 09: Using Inheritance

```
class Animal:
    name = ""

    def eat(self):
        print("The dog is eating")

class Dog(Animal):  # inherit from Animal

    # new method in the child class
    def display(self):
        # access an attribute of the parent class
        print("Its name is", self.name)

# create an object of the child class
dog1 = Dog()

# access an attribute and a method of the parent class
dog1.name = "Lucky"
dog1.eat()

# call a method of the child class
dog1.display()
```



Output:

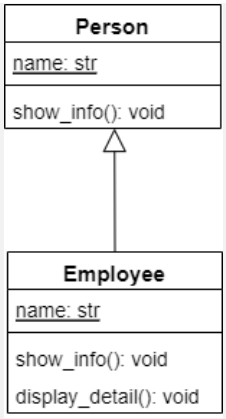
```
The dog is eating
Its name is Lucky
```

Note: In class diagrams, classes variables are underlined.

3.1 Using `super()` Function

The `super()` function returns a temporary object of the parent class that allows us to call a parent class method inside a child class method.

Example 10: Using `super()` function to access to parent class's members.




```
# Parent class
class Person:
    name = "Jonh"

    def show_info(self):
        print("This is a person")

# Child class
class Employee(Person):
    name = "Jackie"

    def show_info(self):
        print("This is an employee")

    def display_detail(self):
        super().show_info()  # access a method of the parent class
        print("Person name:", super().name)  # access an attribute of the
                                             # parent class
        self.show_info()
        print("Employee name:", self.name)

employee = Employee()
employee.display_detail()
```

Output:

```
This is a person
Person name: Jonh
This is an employee
Employee name: Jackie
```

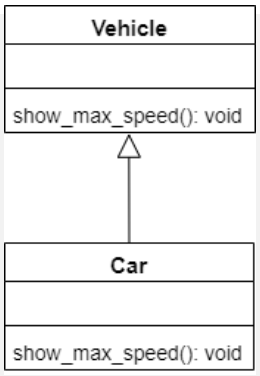
3.2 Method Overriding

In inheritance, all members available in the parent class are by default available in the child class. If the child class does not satisfy with parent class implementation, then the child class is allowed to redefine that method by extending additional methods in the child class. This concept is called **method overriding**.

When a child class method has the same name, same parameters, and same return type as a method in its parent class, then the method in the child is said to override the method in the parent class.

Example 11: We create two classes named Vehicle (Parent class) and Car (Child class). The class Car extends from the class Vehicle so, all members of the parent class are available in the child class. In addition to that, the child class redefined the method **`show_max_speed`**().

```
class Vehicle:
    def show_max_speed(self):
        print("Max speed is 100 Km/Hour")

class Car(Vehicle):
    # overridden the implementation of Vehicle class
    def show_max_speed(self):
        print("Max speed is 200 Km/Hour")

car = Car()
car.show_max_speed()
```



Output:

```
Max speed is 200 Km/Hour
```

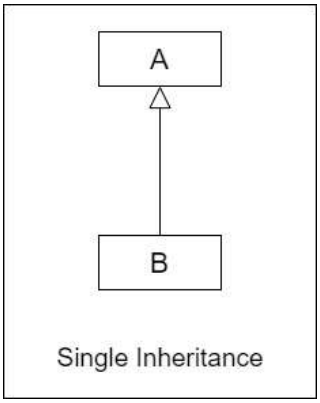
3.3 Types of Inheritances

There are five types of inheritances:

- Single Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Multiple Inheritance
- Hybrid Inheritance

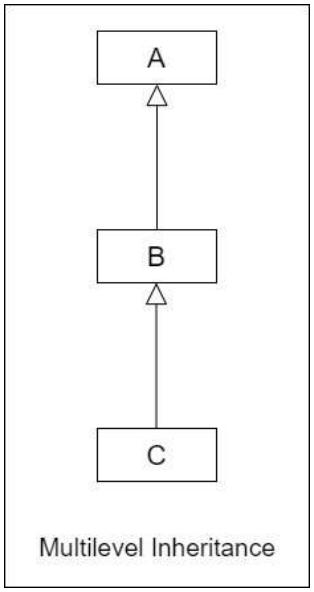
3.3.1 Single Inheritance

In single inheritance, a child class inherits the features of a parent class. In a diagram below, the class A serves as a parent class for the child class B.



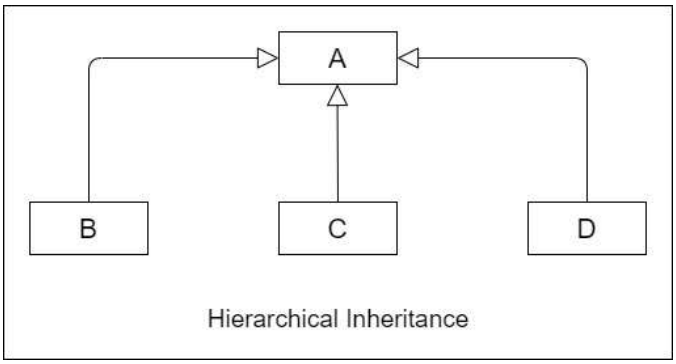
3.2 Multilevel Inheritance

In Multilevel Inheritance, a child class inherits from a parent class and as well as the child class also acts as a parent class to another class. In below diagram, class A serves as a parent class for the child class B, which in turn serves as a parent class for the child class C.



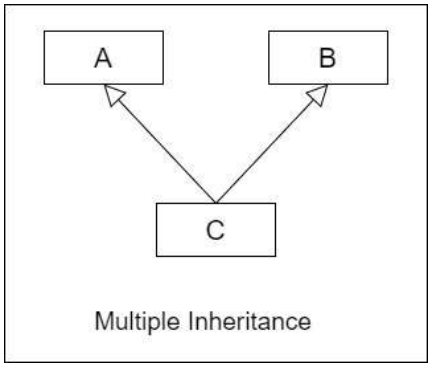
3.3 Hierarchical Inheritance

In Hierarchical Inheritance, one class serves as a parent class for more than one child class. In below diagram, class A serves as a parent class for the child class B, C, and D.



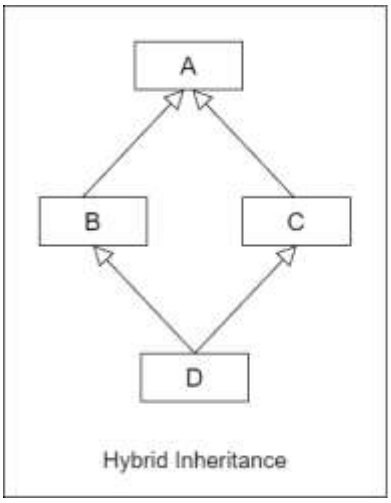
3.4 Multiple Inheritance

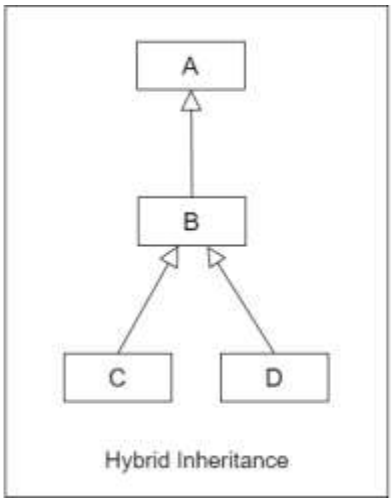
In Multiple inheritance, one class can have more than one parent classes and inherit features from all parent class classes. In a diagram below, class C inherits from the class A and class B.

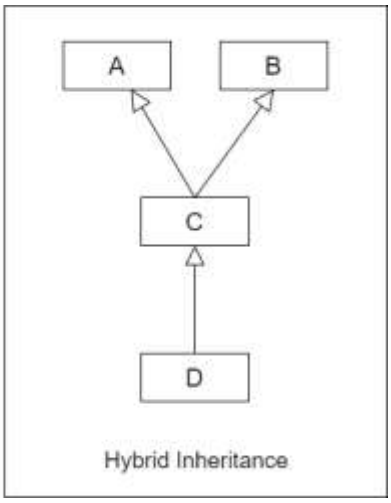


3.5 Hybrid Inheritance

When inheritance consists of multiple different types of inheritance is called **hybrid inheritance**. Below are some examples:







Exercises

II. Draw class diagrams and write codes for each of the following programs:

1. Create a `Room` class with attributes **`room_number`** and **`number_of_seats`**. Create another class called `Building` that contains **`name`**, **`number_of_floors`**, and a method to display the building's details, including room specifications.
2. Create a `Book` class with attributes like **`id`**, **`title`** and **`author`**. Create another class called `Library` that contains a list of book objects as its attribute. Implement methods in the `Library` class to:
   - Add a new book to the library.
   - Remove a book by id.
   - Display all books in the library.
3. Create a class `Teacher` with attributes **`id`** and **`name`**. Create a `Student` class with attributes **`id`**, and name. Create a `Subject` class to represent the association between `Teacher` and `Student`. The `Subject` class should store a teacher, a student, **`subject_name`**, **`department_name`**, **`year`**, and **`semester`**.
4. Create a class `Product` with attributes **`id`**, **`name`** and **`price`**. Create a class called `ShoppingCart` with a method that accepts a list of product objects and returns the total price.
5. Create an `Order` class with attributes **`items`** (list of strings) and **`table_number`**. Create a class called `Restaurant` with a method `show_order`() that accepts an order object and prints which table to serve and the items ordered.
6. Create a `Customer` class with attributes **`id`**, **`name`** and **`balance`**. Create a class called `Bank` with a method transfer that accepts two customer objects and an amount.
   - The `transfer`() method should deduct the amount from one customer and add it to the other.
   - The `Bank` class does not need to store the customer objects—it interacts with them only for the transfer operation.
7. Create a `Person` class with attributes like **`name`**, **`age`**, and **`gender`** and a method called `display`() that will display the information of the person. Create `Employee` and `Customer` classes that inherit from `Person`. The `Employee` class should have attributes like **`employee_id`**, **`job_title`**, and `salary`, while the `Customer` class should have attributes like **`customer_id`**, `address`, and **`phone_number`**. Override the `display`() method in each child class to display the their own information. Write a test program that creates an employee and a customer, and call their `display`() methods.