Object Oriented Programming

by

Tanmoy Sarkar Pias

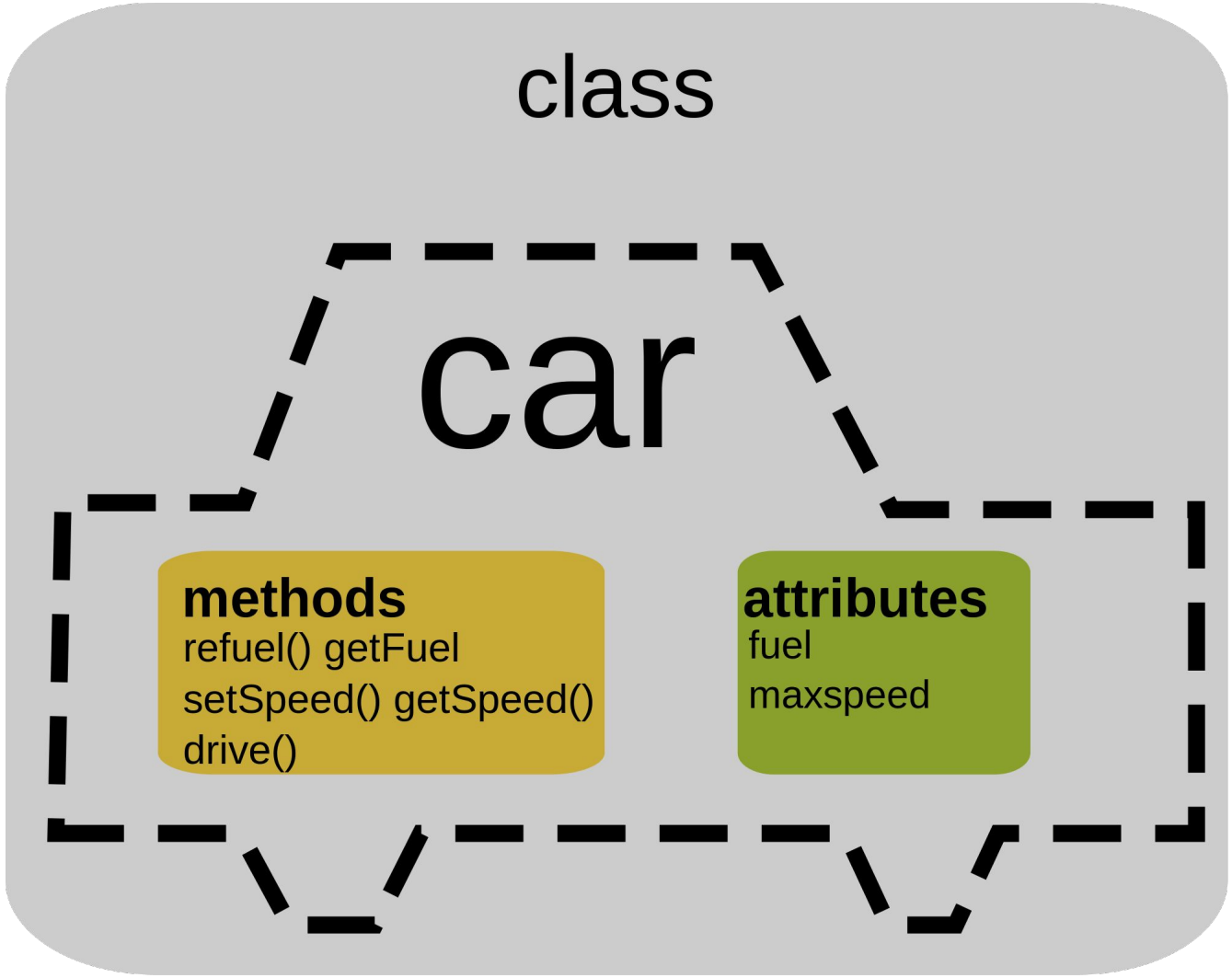
Lecturer, CSE, UAP

Definition

Object-oriented programming (**OOP**) is a programming paradigm based on the concept of "objects", which can contain data, in the form of fields (often known as *attributes),* and code, in the form of procedures (often known as *methods)*

Ref: Wikipedia

OOP


class
car
methods
refuel() getFuel
setSpeed() getSpeed()
drive()
attributes
fuel
maxspeed

Programming paradigms

Programming paradigms are a way to classify programming languages based on their features. Languages can be classified into multiple paradigms.

OOP (Java Class)

```
Class Account{
        int account_number;
        int account_balance;
public void showdata(){
        system.out.println(“Account Number”+account_number)
        system.outprintln(“Account Balance”+ account_balance)
}
}
```

Core OOPS concepts

1) **Class**
   The class is a group of similar entities.

For example, if you had a class called “Expensive Cars” it could have objects like Mercedes, BMW, Toyota, etc.

Its properties(data) can be price or speed of these cars.

While the methods may be performed with these cars are driving, reverse, braking etc.

Python Class

```
#class declaration
class Jungle:
    #constructor with default values
    def __init__(self, name="Unknown"):
        self.visitorName = name

    def welcomeMessage(self):
        print("Hello %s, Welcome to the Jungle" % self.visitorName)
```

Core OOPS concepts

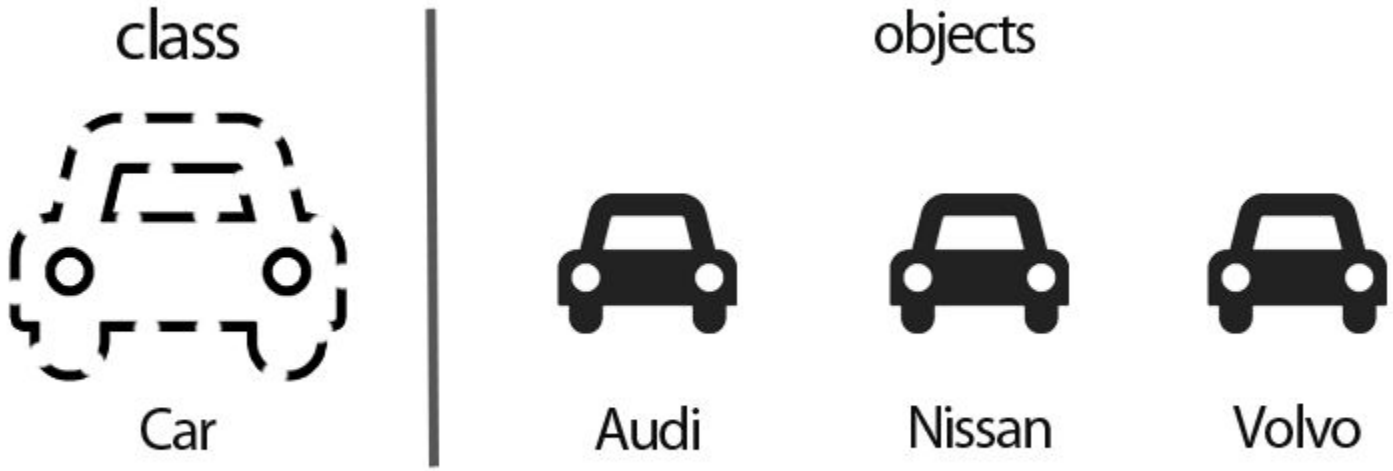
2) Object

An object can be defined as an instance of a class, and there can be multiple instances of a class in a program.

An Object contains both the data and the function, which operates on the data.

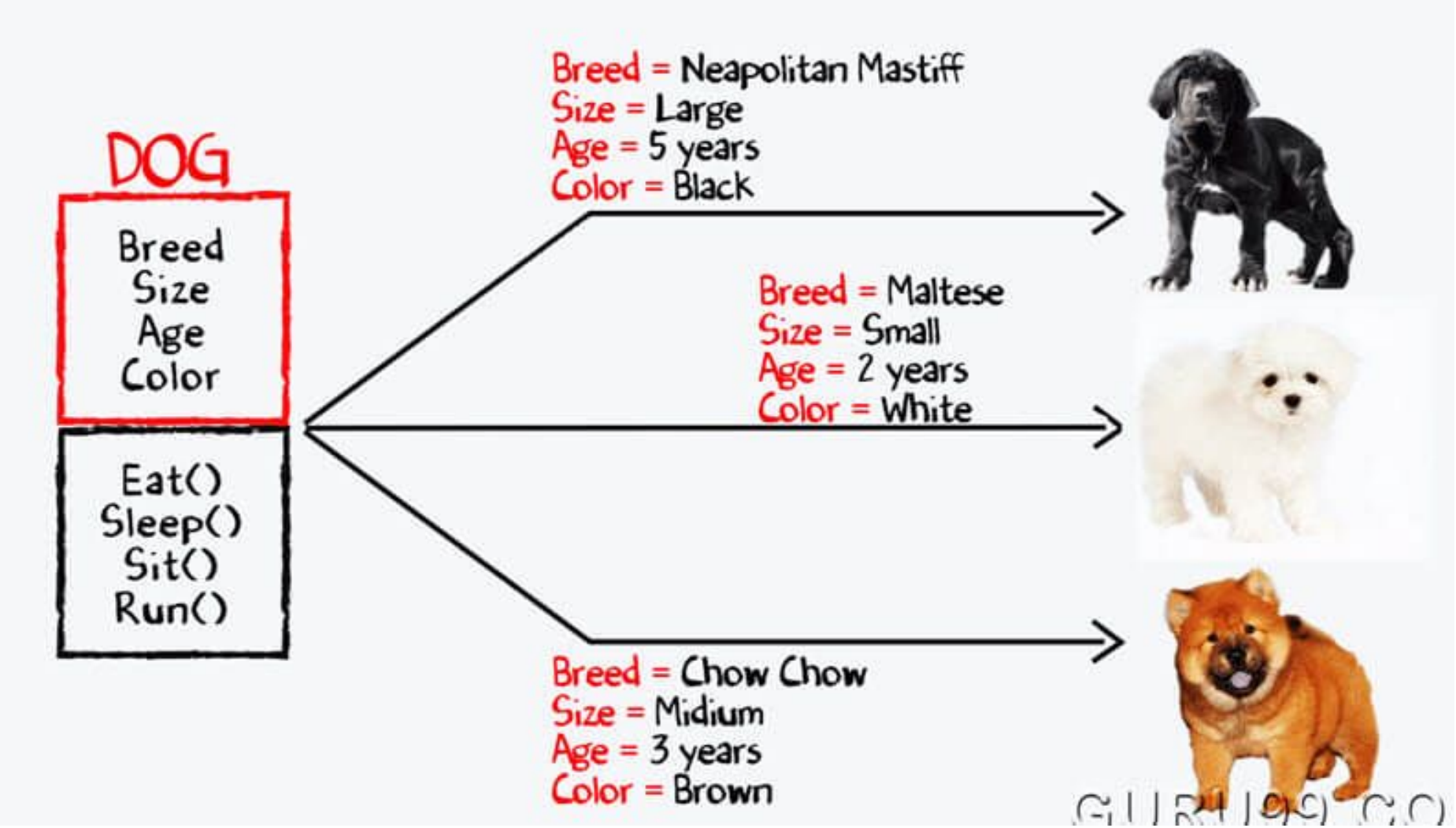
For example - chair, bike, marker, pen, table, car, etc.

Class and Object



Class and Object

Class and Object



Python Object

```
#class declaration
class Jungle:
    #constructor with default values
    def __init__(self, name="Unknown"):
        self.visitorName = name

    def welcomeMessage(self):
        print("Hello %s, Welcome to the Jungle" % self.visitorName)

# create object of class Jungle
j = Jungle("Meher")
j.welcomeMessage() # Hello Meher, Welcome to the Jungle

# if no name is passed, the default value i.e. Unknown will be used
k = Jungle()
k.welcomeMessage() # Hello Unknown, Welcome to the Jungle
```

Core OOPS concepts

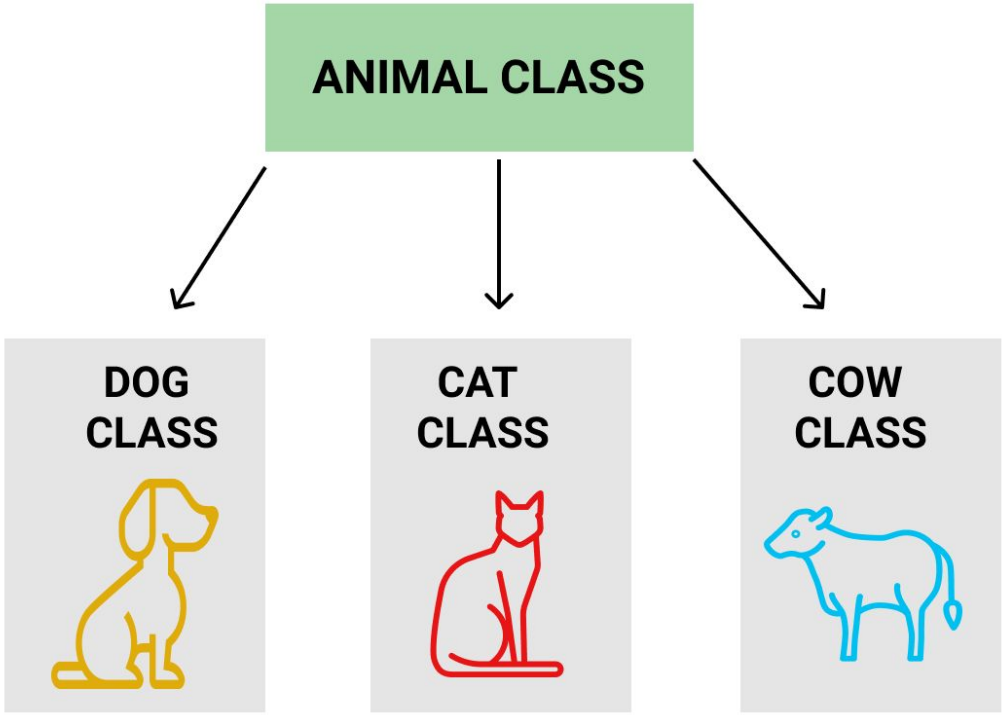
3) Inheritance

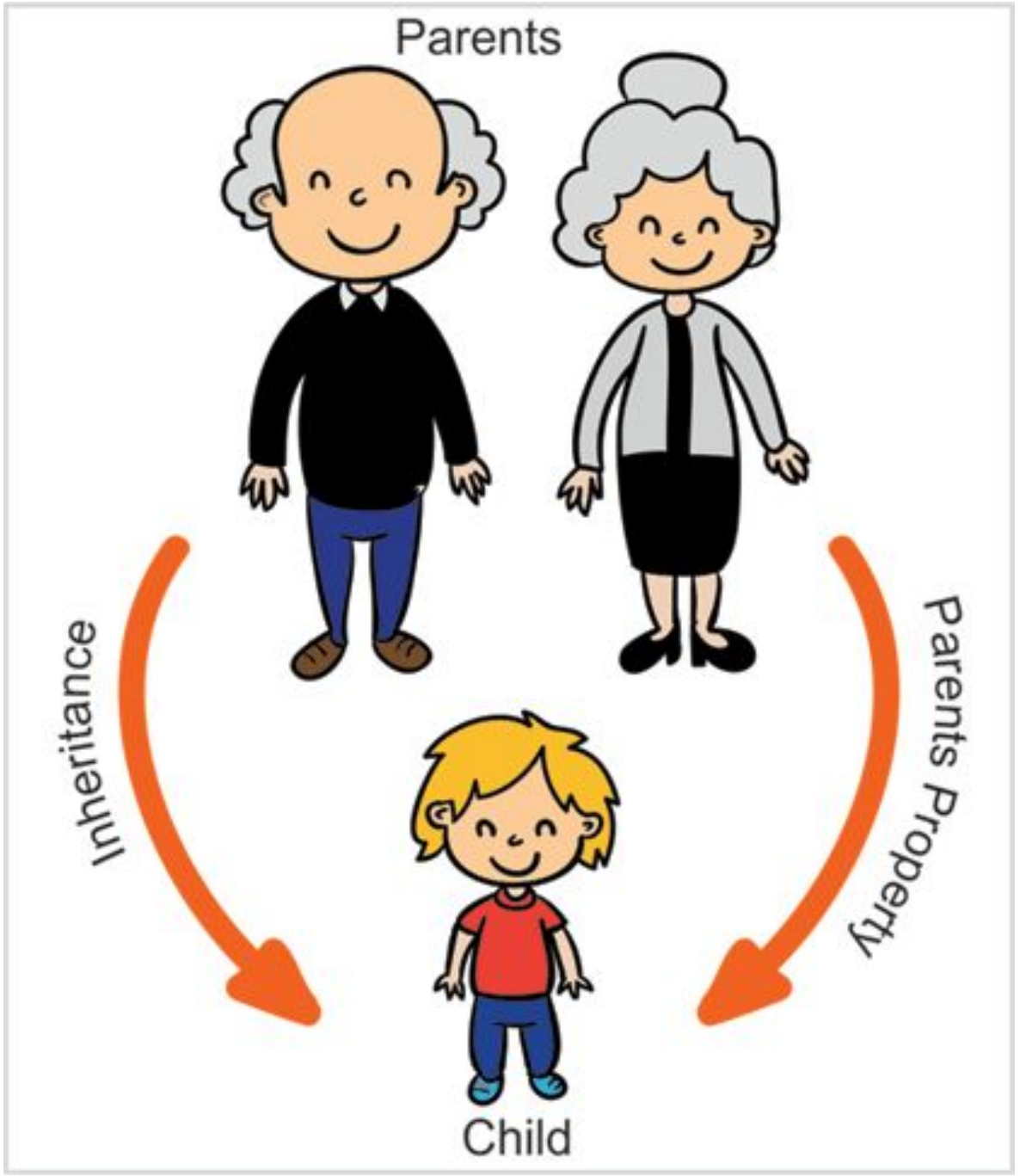
Inheritance is an OOPS concept in which one object acquires the properties and behaviors of the parent object.

It’s creating a parent-child relationship between two classes.

It offers robust and natural mechanism for organizing and structure of any software.

Inheritance





Python **Inheritance**

```
#class declaration
class Jungle:
    #constructor with default values
    def __init__(self, name="Unknown"):
        self.visitorName = name

    def welcomeMessage(self):
        print("Hello %s, Welcome to the Jungle" % self.visitorName)

class RateJungle(Jungle):
    def __init__(self, name, feedback):
        # feedback (1-10) :  1 is the best.
        self.feedback = feedback # Public Attribute

        # inheriting the constructor of the class
        super().__init__(name)

    # using parent class attribute i.e. visitorName
    def printRating(self):
        print("Thanks %s for your feedback" % self.visitorName)
```

Python **Inheritance**

```
from jungleBook import Jungle, RateJungle

def main():
    r = RateJungle("Meher", 3)

    r.printRating() # Thanks Meher for your feedback

    # calling parent class method
    r.welcomeMessage() # Hello Meher, Welcome to the Jungle
```

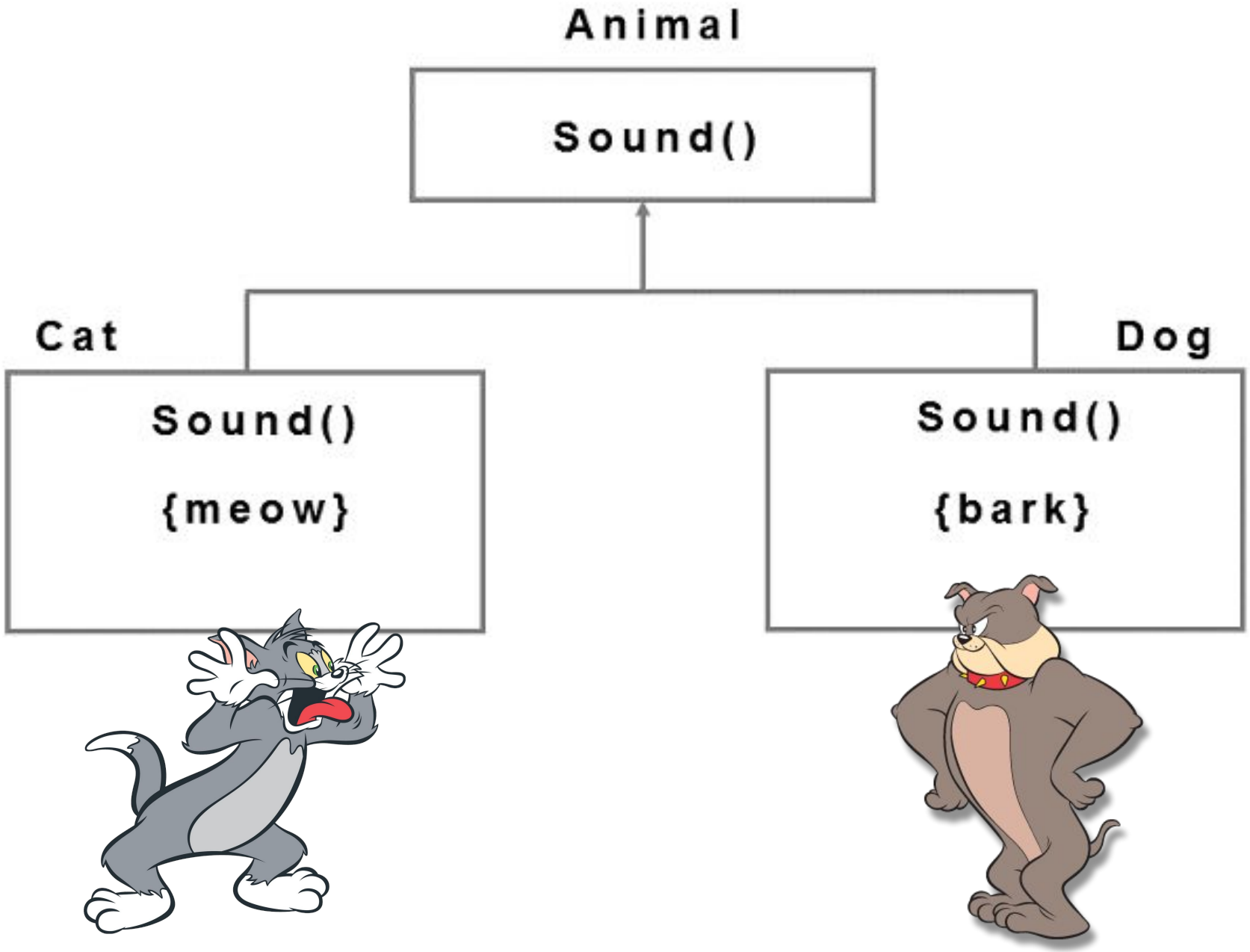
Core OOPS concepts

4) Polymorphism

Polymorphism refers to the ability of a variable, object or function to take on multiple forms.

For example, in English, the verb *run* has a different meaning if you use it with *a laptop*, *a foot race*, and *business*.

Here, we understand the meaning of *run* based on the other words used along with it. The same also applied to Polymorphism.



Polymorphism

Python **Polymorphism**

```
#scarySound.py

class Animal:
    def scarySound(self):
        print("Animals are running away due to scary sound.")

class Bird:
    def scarySound(self):
        print("Birds are flying away due to scary sound.")

# scaryScound is not defined for Insect
class Insect:
    pass
```

Python **Polymorphism**

```
#main.py

## import class 'Animal, Bird' from scarySound.py
from scarySound import Animal, Bird

def main():
    # create objects of Animal and Bird class
    a = Animal()
    b = Bird()

    # polymorphism
    a.scarySound() # Animals are running away due to scary sound.
    b.scarySound() # Birds are flying away due to scary sound.
```

Core OOPS concepts

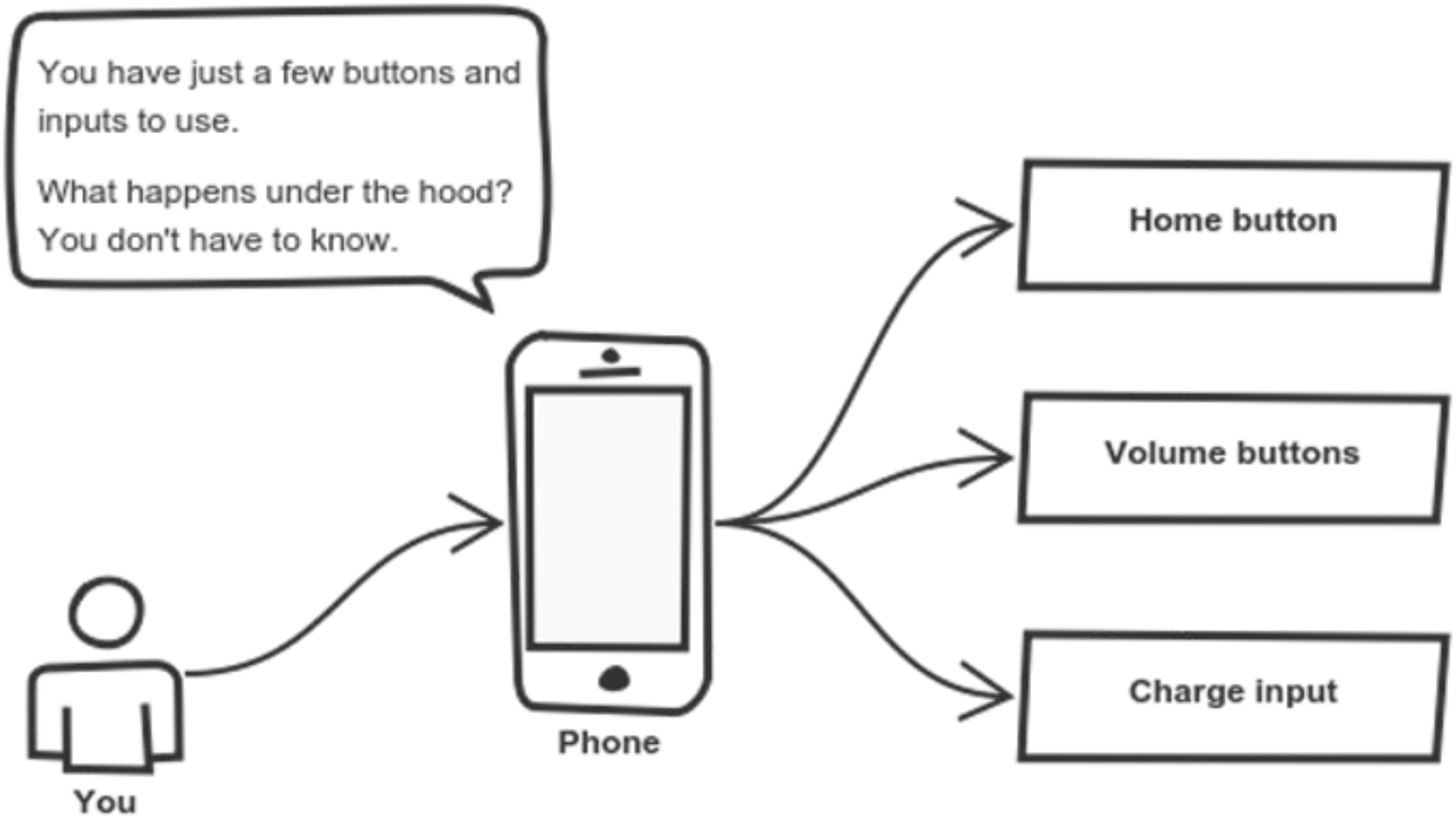
5) Abstraction

An abstraction is an act of representing essential features without including background details.

It is a technique of creating a new data type that is suited for a specific application.

For example, while driving a car, you do not have to be concerned with its internal working. Here you just need to concern about parts like steering wheel, Gears, accelerator, etc.

Abstraction



Abstraction

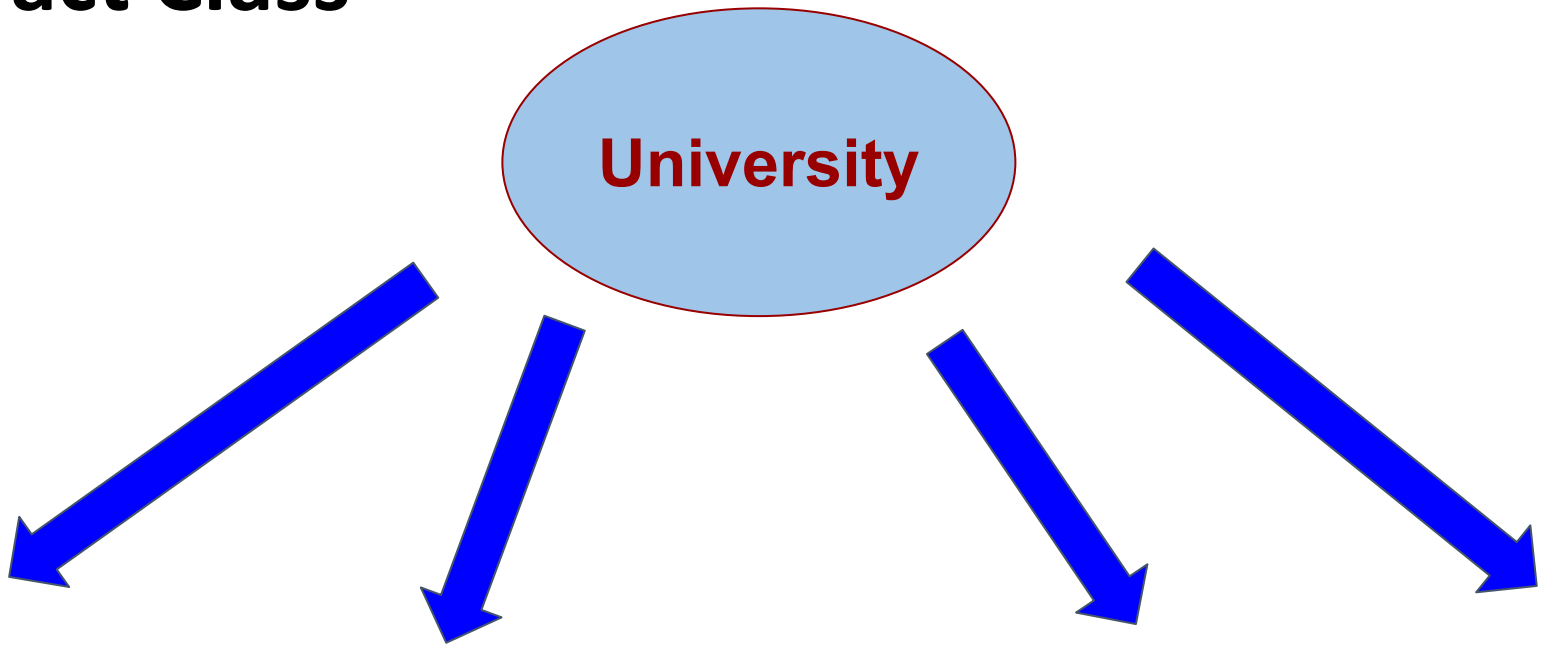
```
def main:
    car = Tesla(“Version 2.0”)
    car.start()
    car.accelerate(3.0)
    car.run(300)
    car.travel(100)
    car.stop()
```

Abstract Class

- An abstract class can be considered as a blueprint for other classes. It allows you to create a set of methods that must be created within any child classes built from the abstract class.
- A class which contains one or more abstract methods is called an abstract class.
- An abstract method is a method that has a declaration but does not have an implementation.
- While we are designing large functional units we use an abstract class. When we want to provide a common interface for different implementations of a component, we use an abstract class.

Source: https://www.geeksforgeeks.org/abstract-classes-in-python/

Abstract Class



Abstract Class

```
#jungleBook.py

from abc import ABCMeta, abstractmethod

#Abstract class and abstract method declaration
class Jungle(metaclass=ABCMeta):
    #constructor with default values
    def __init__(self, name="Unknown"):
        self.visitorName = name

    def welcomeMessage(self):
        print("Hello %s, Welcome to the Jungle" % self.visitorName)

    # abstract method is compulsory to defined in child-class
    @abstractmethod
    def scarySound(self):
        pass
```

Abstract Class

```
#scarySound.py

from jungleBook import Jungle

class Animal(Jungle):
    def scarySound(self):
        print("Animals are running away due to scary sound.")

class Bird(Jungle):
    def scarySound(self):
        print("Birds are flying away due to scary sound.")

# since Jungle is defined as metaclass
# therefore all the abstract methods are compulsory be defined in child class
class Insect(Jungle):
    def scarySound(self):
        print("Insects do not care about scary sound.")
```

Abstract Class

```python
#main.py

## import class 'Animal, Bird' from scarySound.py
from scarySound import Animal, Bird, Insect

def main():
    # create objects of Animal and Bird class
    a = Animal()
    b = Bird()
    i = Insect()

    # polymorphism
    a.scarySound() # Animals are running away due to scary sound.
    b.scarySound() # Birds are flying away due to scary sound.
    i.scarySound() # Insects do not care about scary sound.
```

Interface

In object-oriented languages like Python, the interface is a collection of method signatures that should be provided by the implementing class. Implementing an interface is a way of writing an organized code and achieve abstraction.

Interface

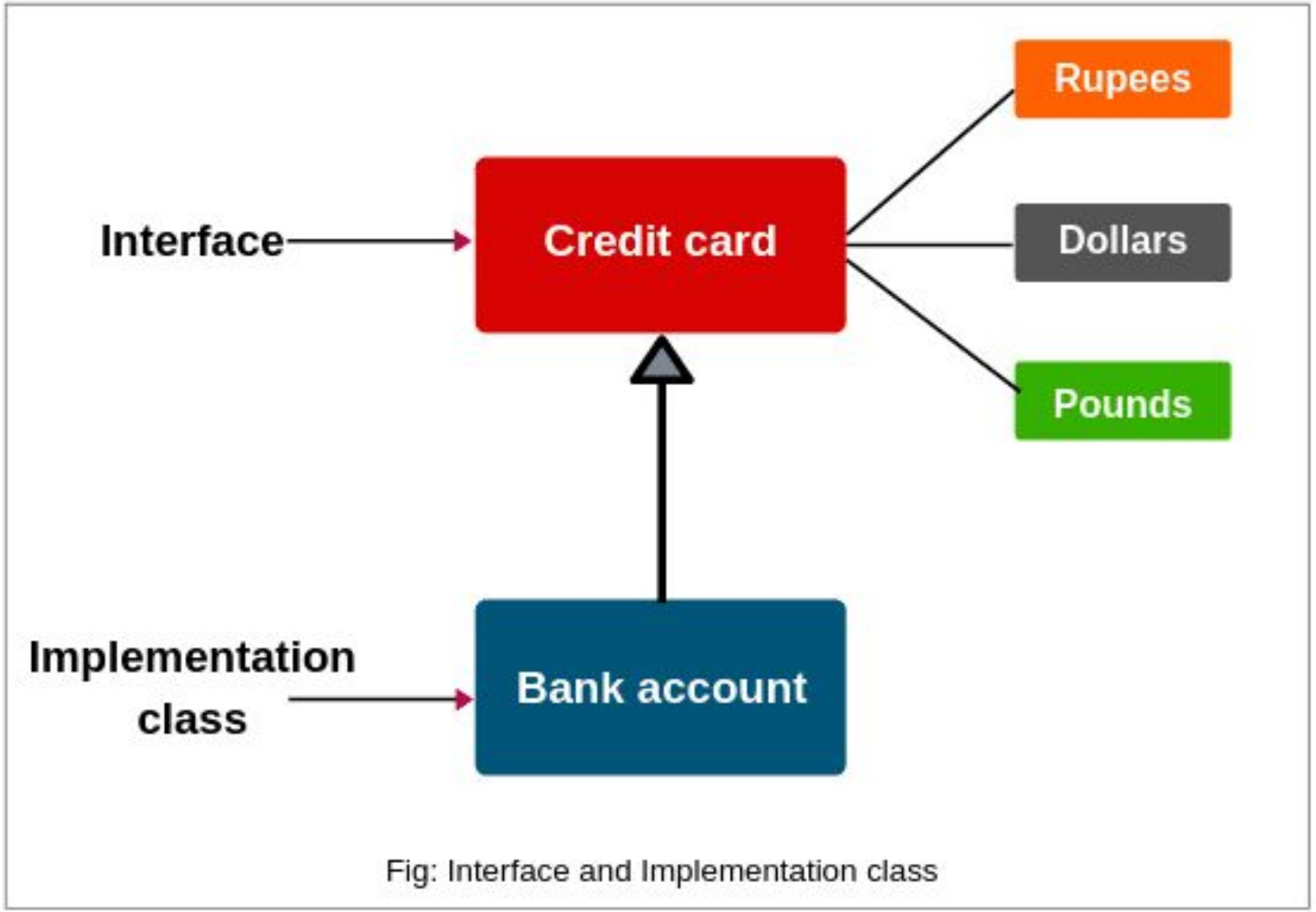


Fig: Interface and Implementation class

Interface

```
from interface import implements, Interface

class MyInterface(Interface):

    def method1(self, x):
        pass

    def method2(self, x, y):
        pass


class MyClass(implements(MyInterface)):

    def method1(self, x):
        return x * 2

    def method2(self, x, y):
        return x + y
```

interface vs. abc

```
>>> from abc import ABCMeta, abstractmethod
>>> class Base(metaclass=ABCMeta):
...
...     @abstractmethod
...     def method(self, a, b):
...         pass
...
>>> class Implementation(MyABC):
...
...     def method(self):  # Missing a and b parameters.
...         return "This shouldn't work."
...
>>> impl = Implementation()
>>>
```

interface vs. abc

```
>>> from interface import implements, Interface
>>> class I(Interface):
...     def method(self, a, b):
...         pass
...
>>> class C(implements(I)):
...     def method(self):
...         return "This shouldn't work"
...
TypeError:
class C failed to implement interface I:

The following methods were implemented but had invalid signatures:
  - method(self) != method(self, a, b)
```

Core OOPS concepts

6) Encapsulation

Encapsulation is an OOP technique of wrapping the data and code.

In this OOPS concept, the variables of a class are always hidden from other classes. It can only be accessed using the methods of their current class.

Encapsulation

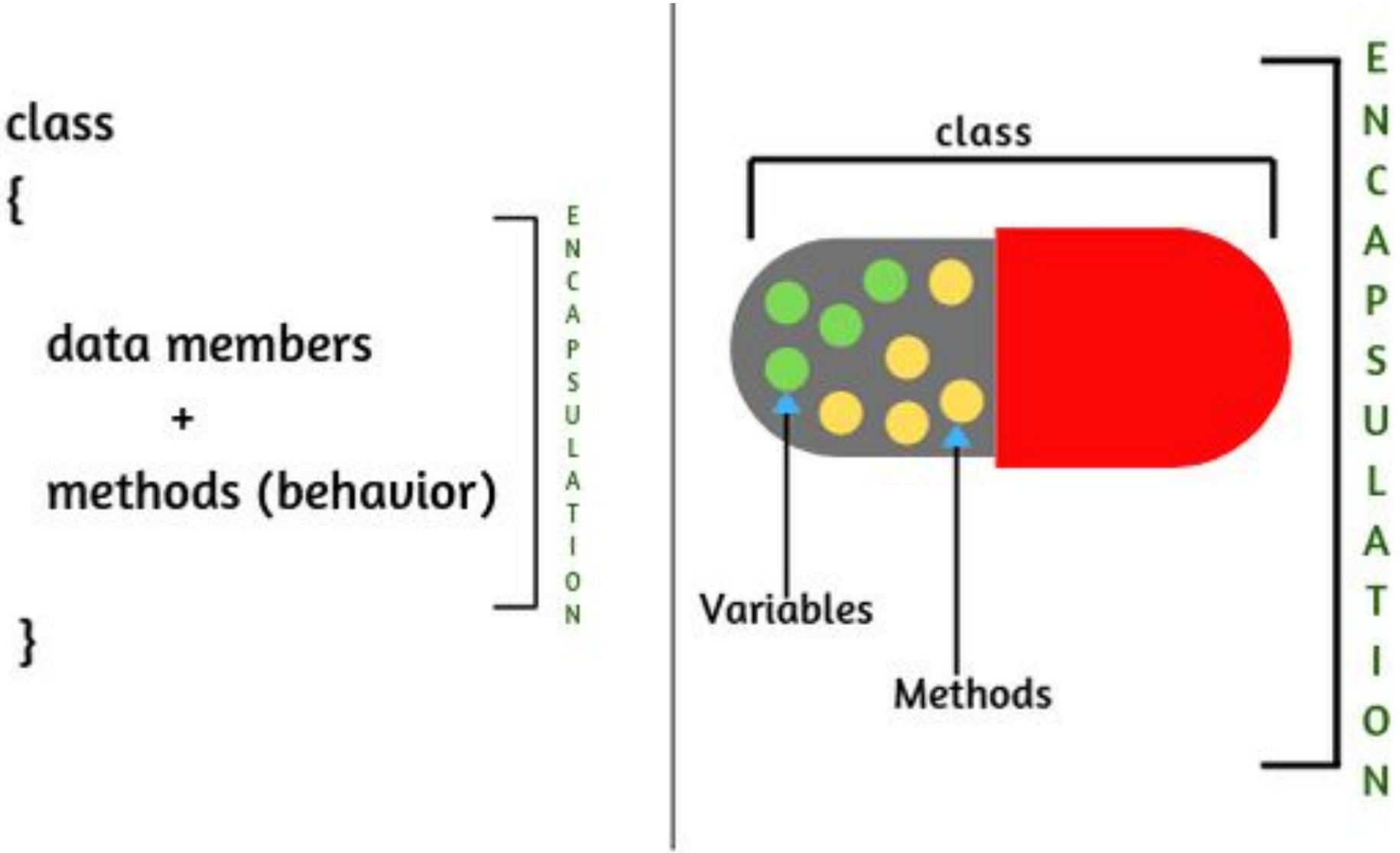


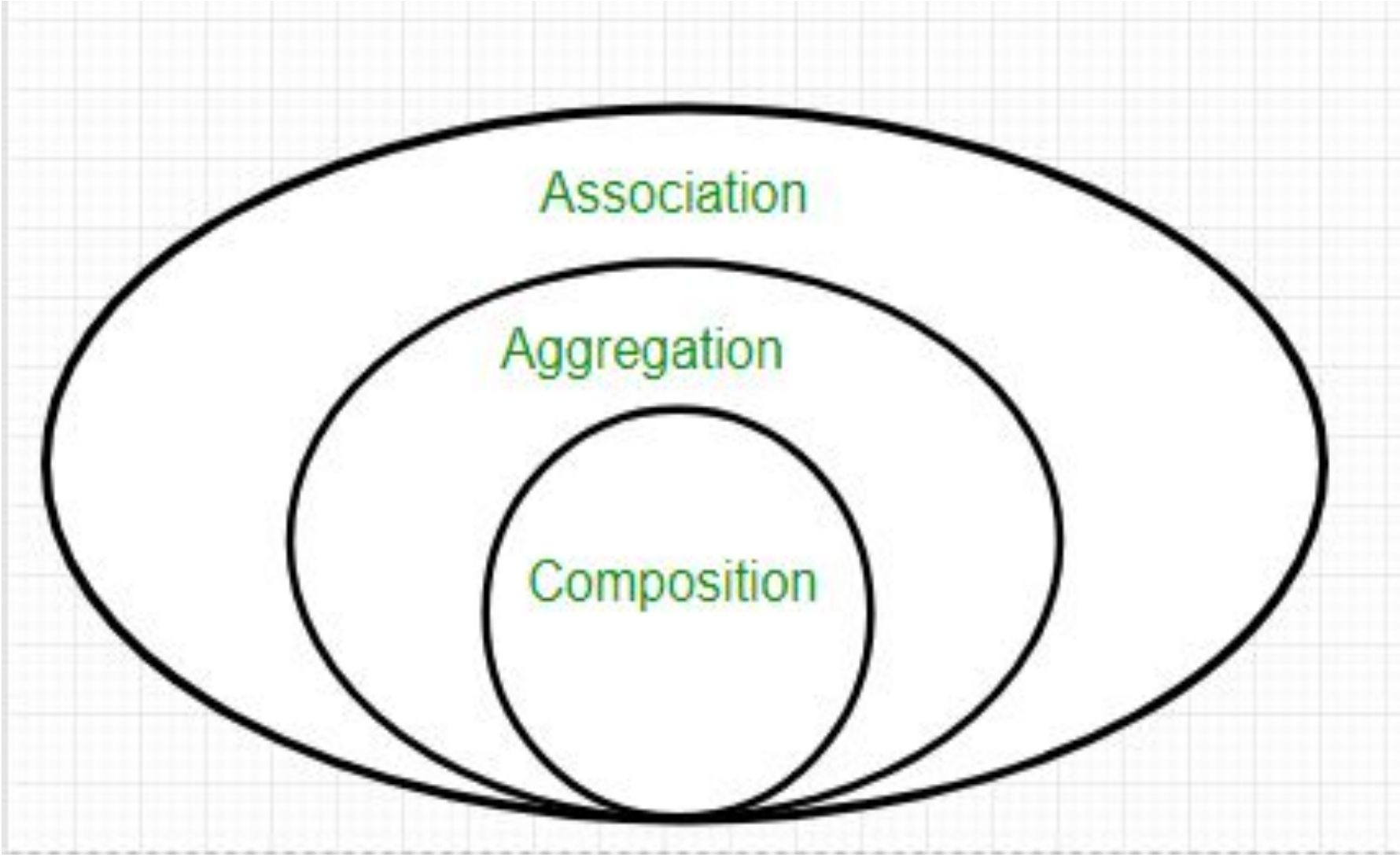
Fig: Encapsulation

Encapsulation

```
Class Employee:
    totalEmployee = 0

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

Core OOPS concepts



Core OOPS concepts

7) Association

Association is a relationship between two objects. It defines the diversity between objects.

In this OOP concept, all object have their separate lifecycle, and there is no owner.

For example, many students can associate with one teacher while one student can also associate with multiple teachers.

Association

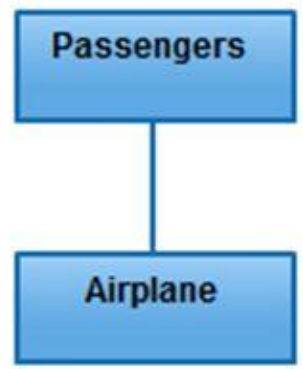
- where objects of the two classes can know about each other, but they do not affect each others lifetime.
- The objects can exist independently and which class A object knows about which class B objects can vary over time.

Association

Relationship where **<u>all object have their own</u>** lifecycle and there is **<u>no owner</u>**.

Ex: Airplane and passengers.

No ownership between the objects and both have their own lifecycle. Both can create and delete independently



Core OOPS concepts

8) Aggregation

In this technique, all objects have their separate lifecycle. However, there is ownership such that child object can’t belong to another parent object.

For example consider class/objects department and teacher. Here, a single teacher can’t belong to multiple departments, but even if we delete the department, the teacher object will never be destroyed.

Aggregation

It is a special form of Association where:

- It represents **Has-A** relationship.
- It is a **unidirectional association** i.e. a one way relationship. For example, department can have students but vice versa is not possible and thus unidirectional in nature.
- In Aggregation, **both the entries can survive individually** which means ending one entity will not effect the other entity

Aggregation

There is an Institute which has no. of departments like CSE, EE.

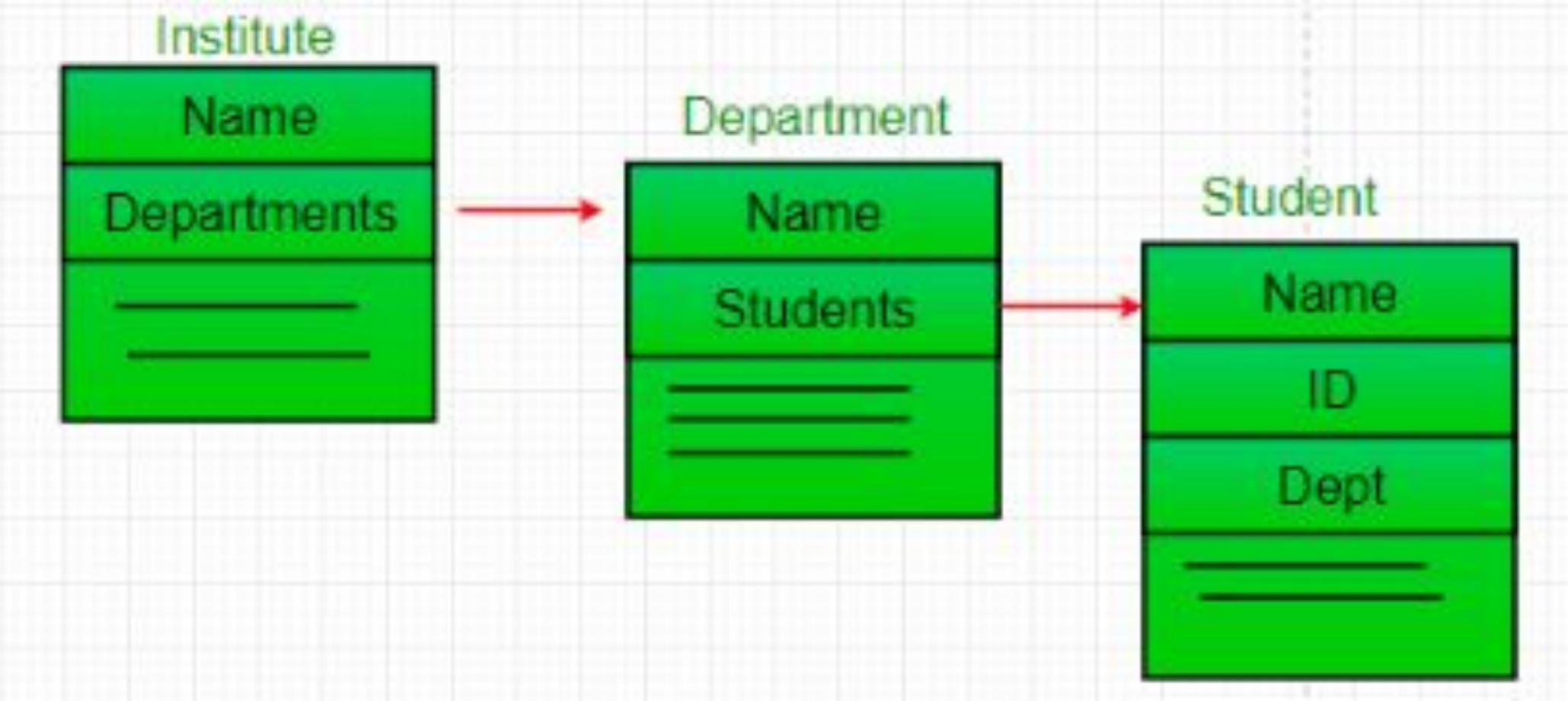
Every department has no. of students.

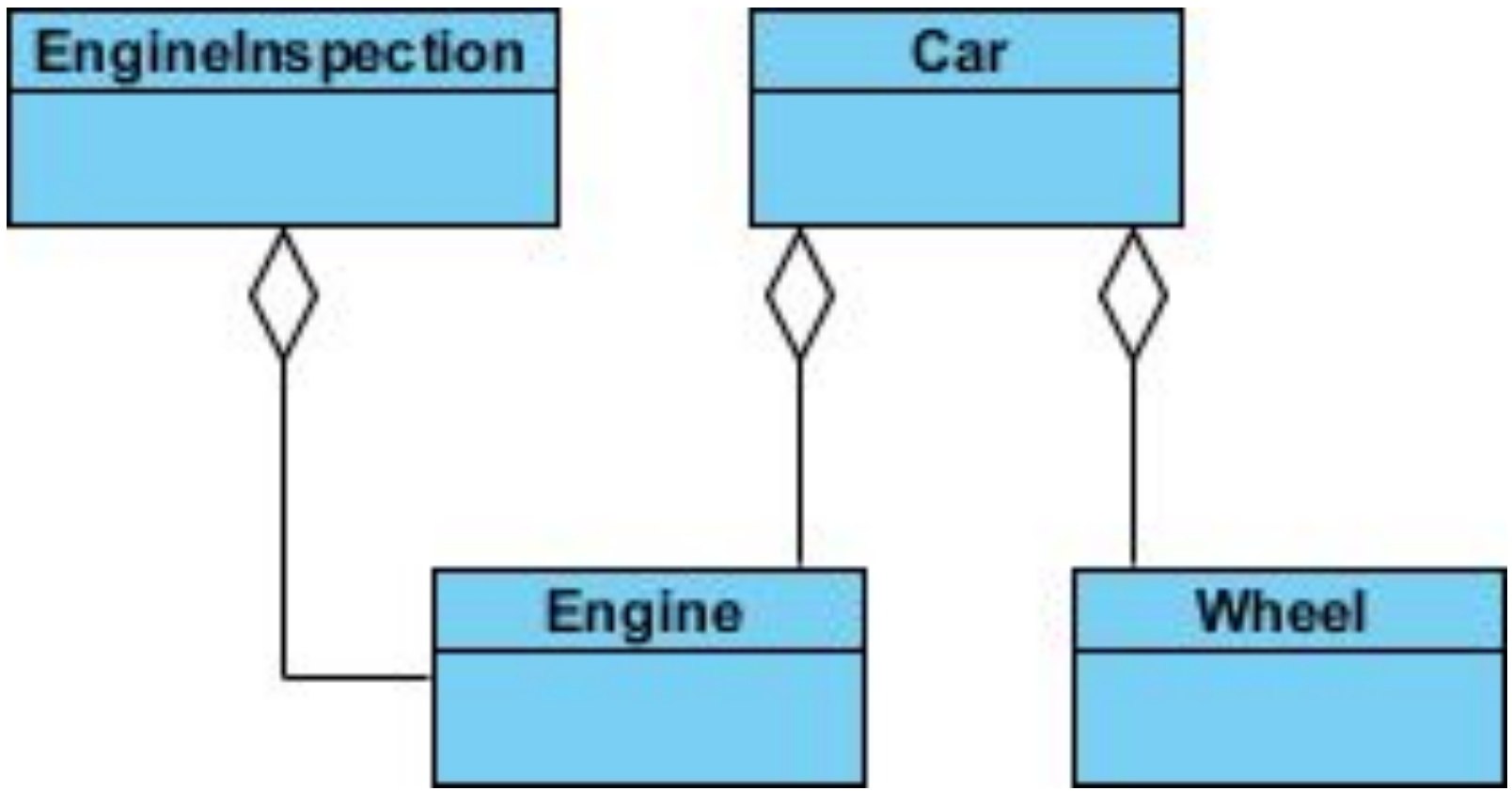
So, we make a **Institute** class which has a reference to Object or no. of Objects (i.e. List of Objects) of the **Department** class. That means Institute class is associated with Department class through its Object(s).

Department class has also a reference to Object or Objects (i.e. List of Objects) of **Student** class means it is associated with **Student** class through its Object(s).

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

Aggregation




EngineInspection
Car
Engine
Wheel

Aggregation

```
class Car:
    def __init__ (self, engine, body):
        self.engine = engine
        self.body = body

def main:
    engine300 = Engine(300)
    stealBody = Body(“Steal”)

    teslaV2 = Car(engine, stealBody)
```

Core OOPS concepts

9) Composition

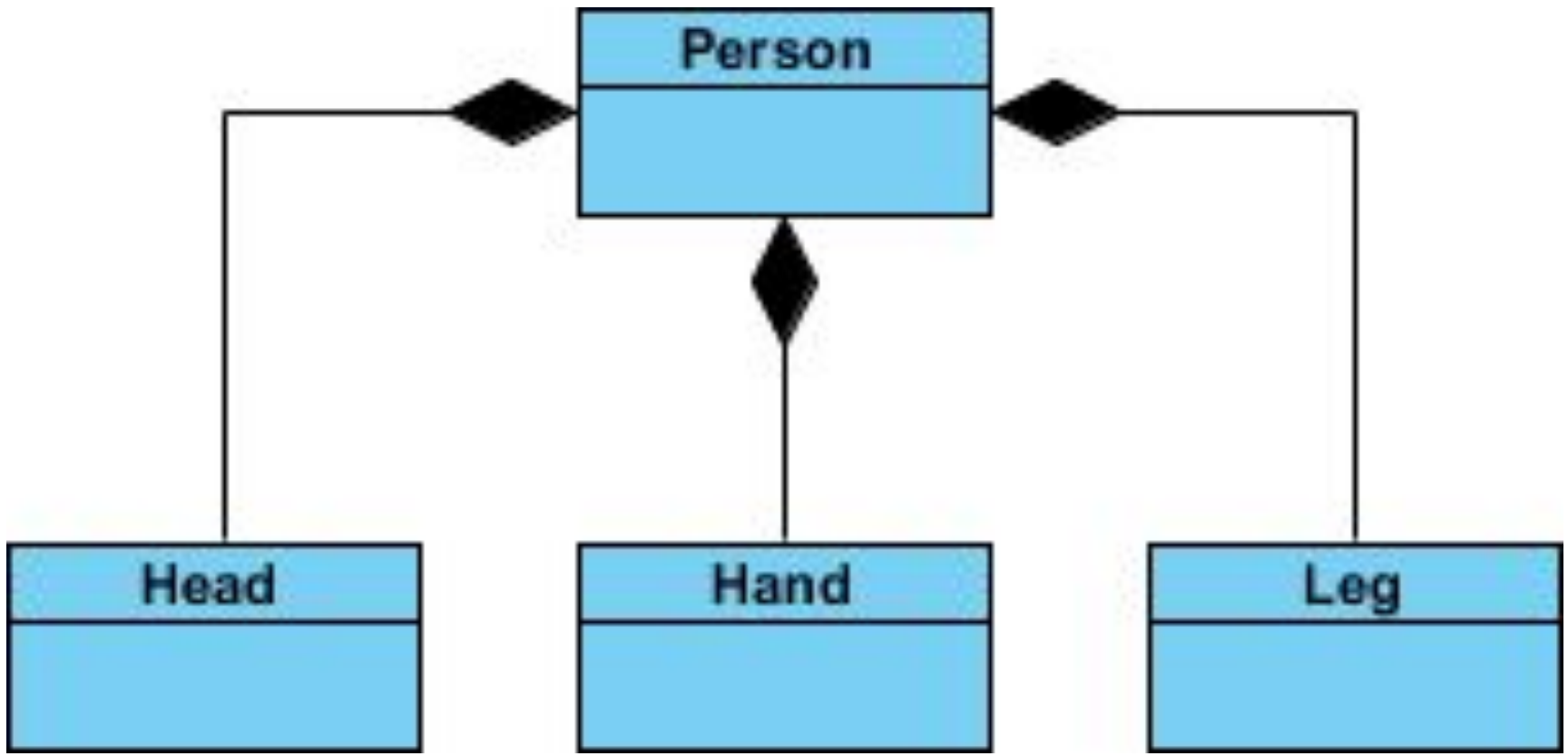
A composition is a specialized form of Aggregation. It is also called "death" relationship. Child objects do not have their lifecycle so when parent object is deleted, then all child object will also be deleted automatically.

For that, let’s take an example of House and rooms. Any house can have several rooms. One room can’t become part of two different houses. So, if you delete the house, room will also be deleted.

Composition

Composition is a restricted form of Aggregation in which two entities are highly dependent on each other.

- It represents **part-of** relationship.
- In composition, both the entities are dependent on each other.
- When there is a composition between two entities, the composed object **cannot exist** without the other entity.


Person
Head
Hand
Leg

Composition

```
class Human:
    def __init__ (self):
        brain = Brain()
        heart = heart()

def main:
    asif = Human()
```

Aggregation vs. Composition

- **Dependency:** Aggregation implies a relationship where the child **can exist independently** of the parent. For example, Bank and Employee and Human and heart.
- **Type of Relationship:** Aggregation relation is **“has-a”** and composition is **“part-of”** relation.
- **Type of association:** Composition is a **strong** Association whereas Aggregation is a **weak** Association.

Association vs. Aggregation vs. Composition

	Association	Aggregation	Composition
Owner	No Owner	Single owner	Single Owner
Life time	Have their own life time.	Have their own life time.	Owners life time
Child object	No Child objects all are independent	Child objects belong to single parent.	Child objects belong to single parent.

Advantages of OOPS:

- OOP offers easy to understand and a clear modular structure for programs.
- Objects created for Object-Oriented Programs can be reused in other programs. Thus it saves significant development cost.
- Large programs are difficult to write, but if the development and designing team follow OOPS concept then they can better design with minimum flaws.
- It also enhances program modularity because every object exists independently.

Thanks to the References:

1. Wikipedia
2. Advance Python Tutorials by Meher Krishna Patel
3. Python Official tutorial
4. Geeksforgeeks
5. Stackoverflow
6. Stackexchange
7. Tutorials Point
8. https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-aggregation-vs-composition/
9. https://www.geeksforgeeks.org/association-composition-aggregation-java/


thank you!
grammarly