ĐẠI HỌC ĐÀ NẴNG

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG VIỆT - HÀN

VIETNAM - KOREA UNIVERSITY OF INFORMATION AND COMMUNICATION TECHNOLOGY

한-베정보통신기술대학교

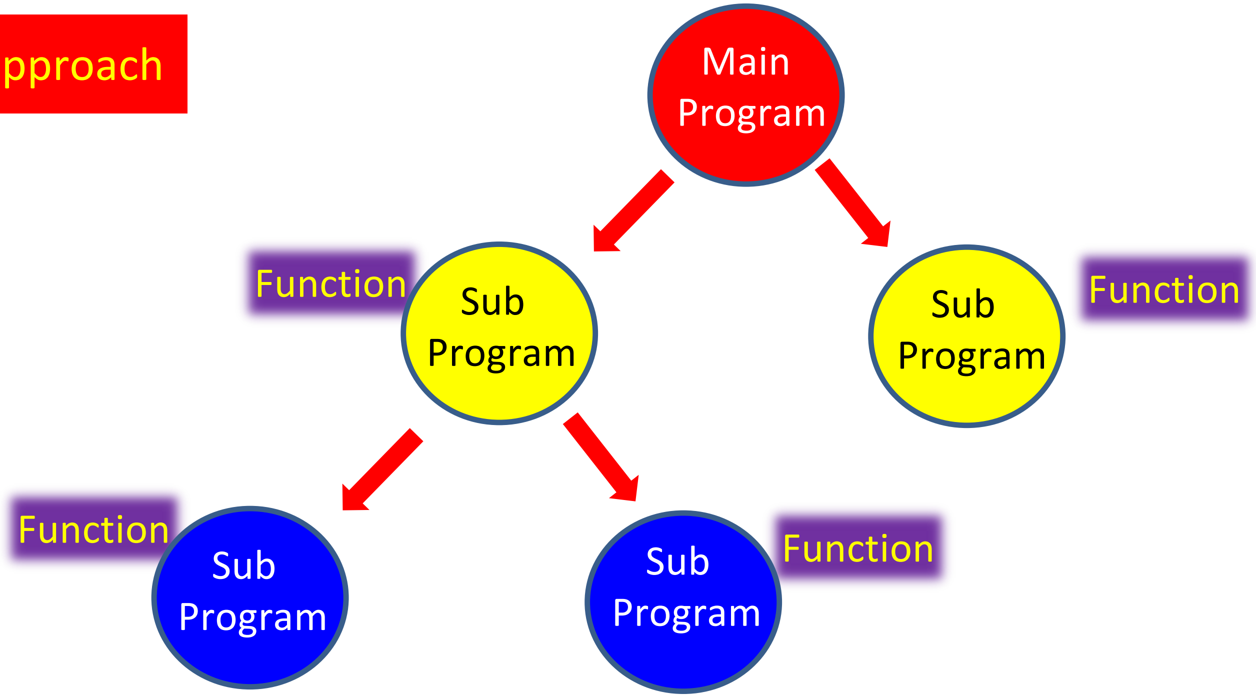
Chapter 3
Object-Oriented Programming in Python

Chapter Content

- 1. Object Oriented Programming
- 2. Class & Object
  - Attribute
  - Constructor
  - Method
- 3. Inheritance
- 4. Polymorphism
- 5. Abstraction
- 6. Encapsulation
- 7. Relationship

Procedural programming

Top-down approach


Main Program

Sub Program

Function

Sub Program

Function

Sub Program

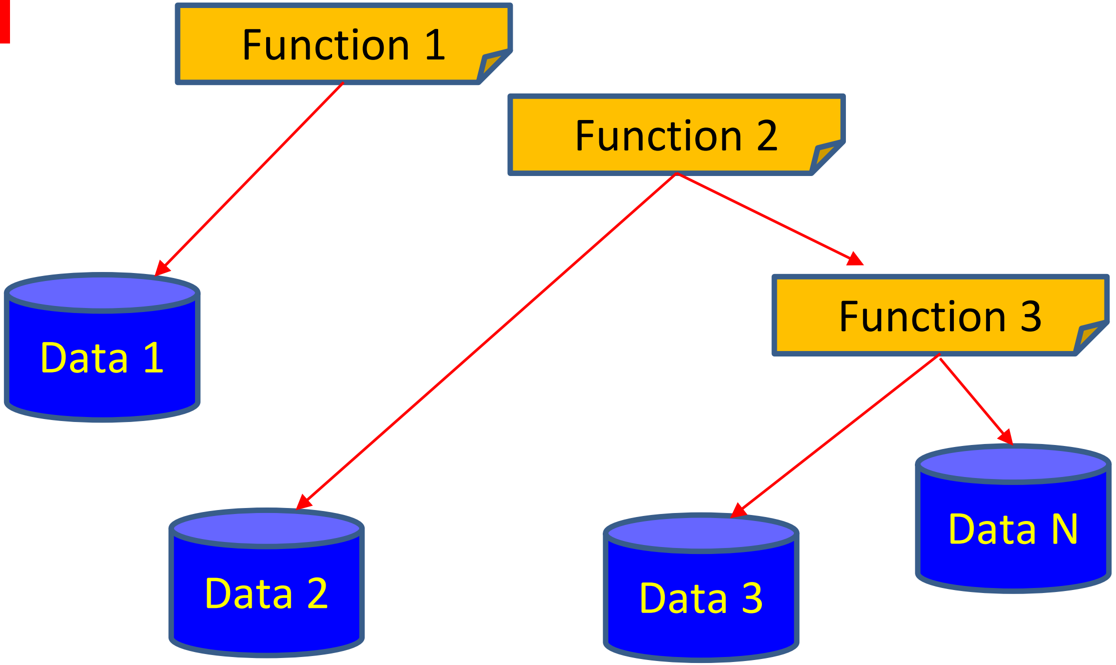
Function

Sub Program

Function

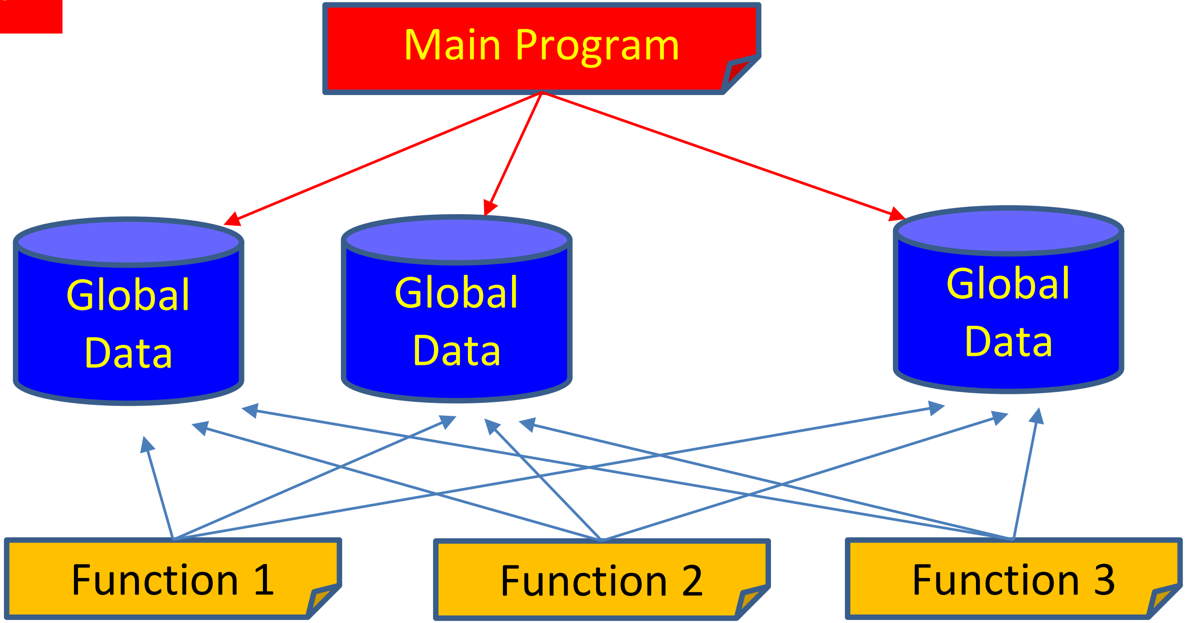
Procedural programming

Top-down approach

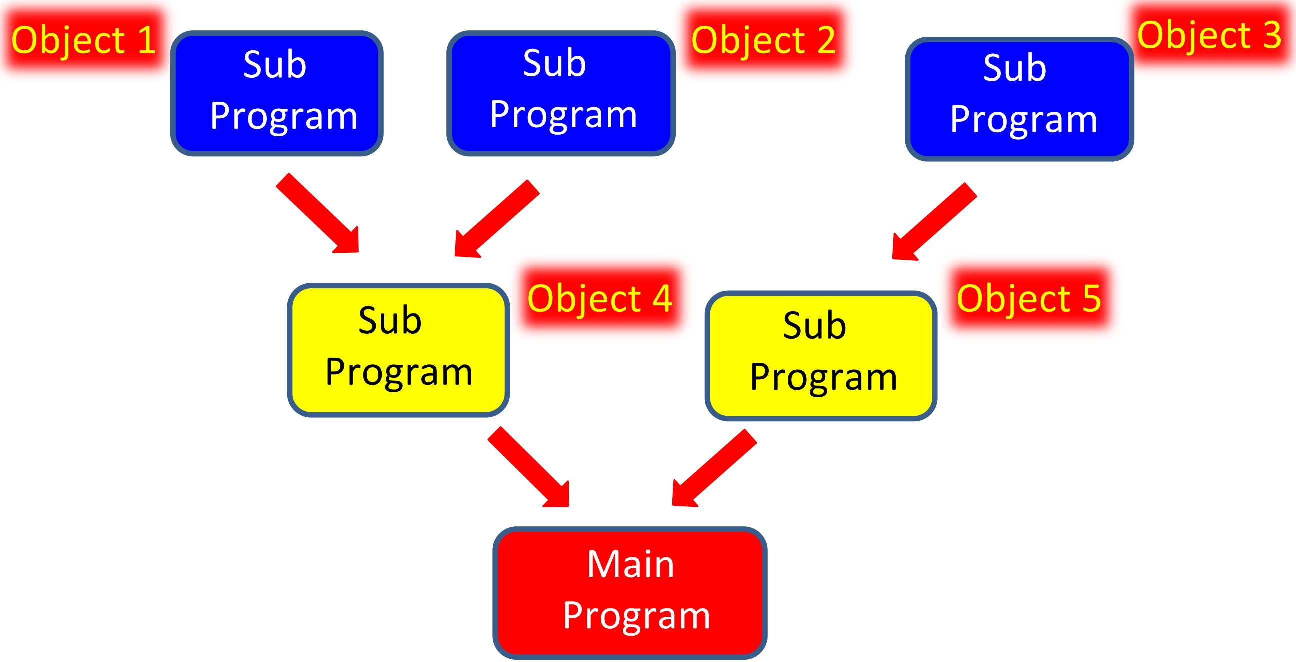


Procedural programming

Top-down approach

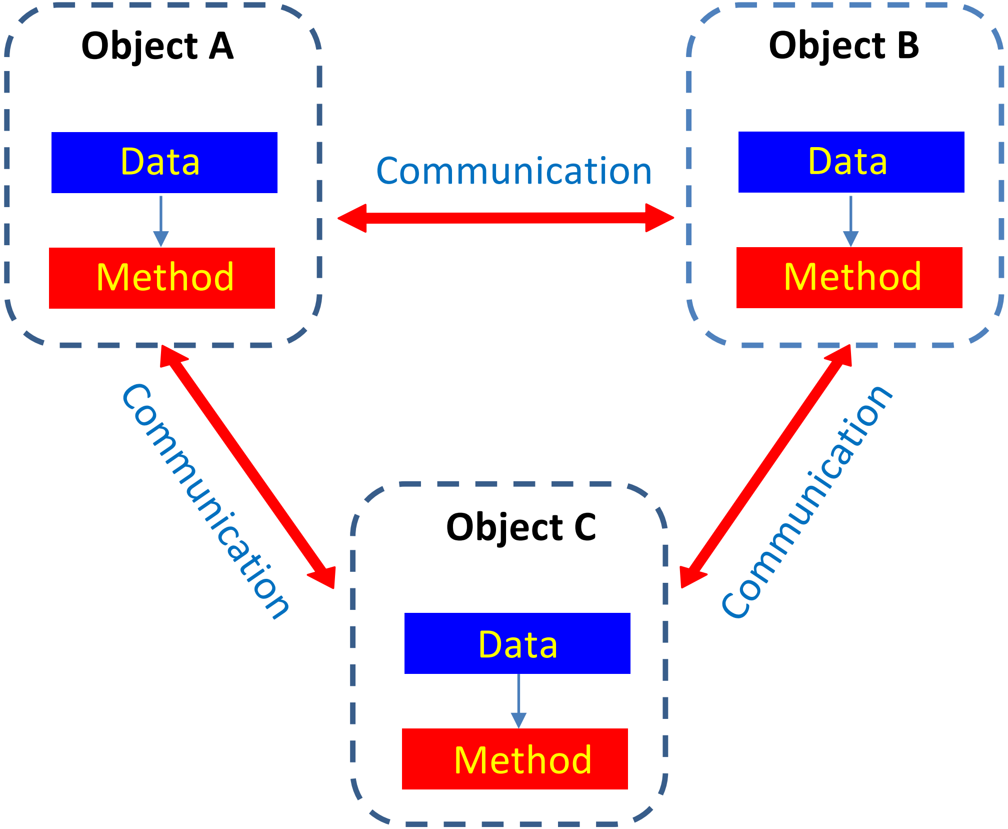


Bottom-up approach



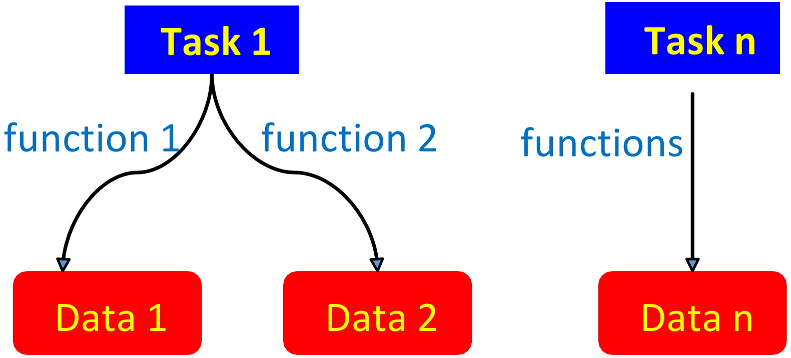
1. Object Oriented Programming

Bottom-up approach



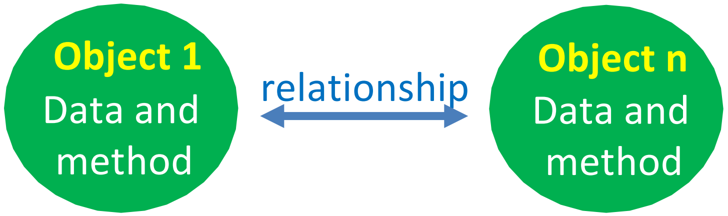
1. Object Oriented Programming

Writing functions that perform operations on the data



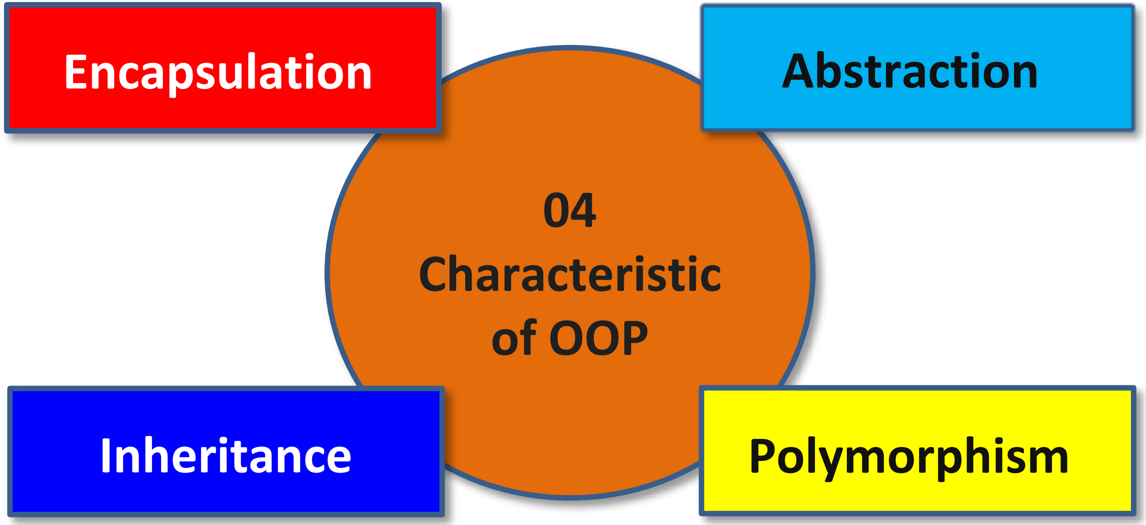
Procedural programming

Creating objects that contain both data and functions



Object-oriented programming

1. Object Oriented Programming: Characteristic



Encapsulation

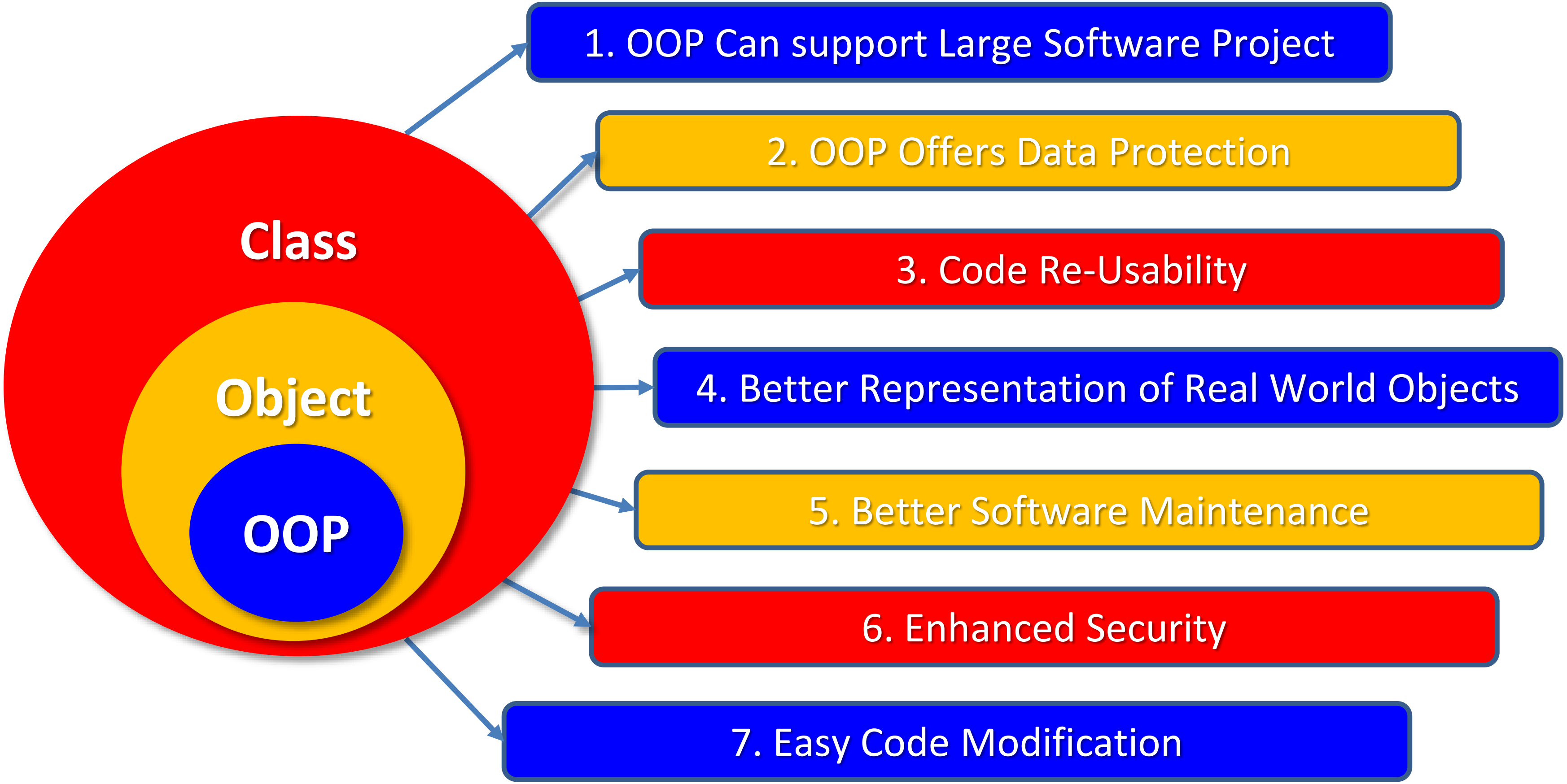
Abstraction

04 Characteristic of OOP

Inheritance

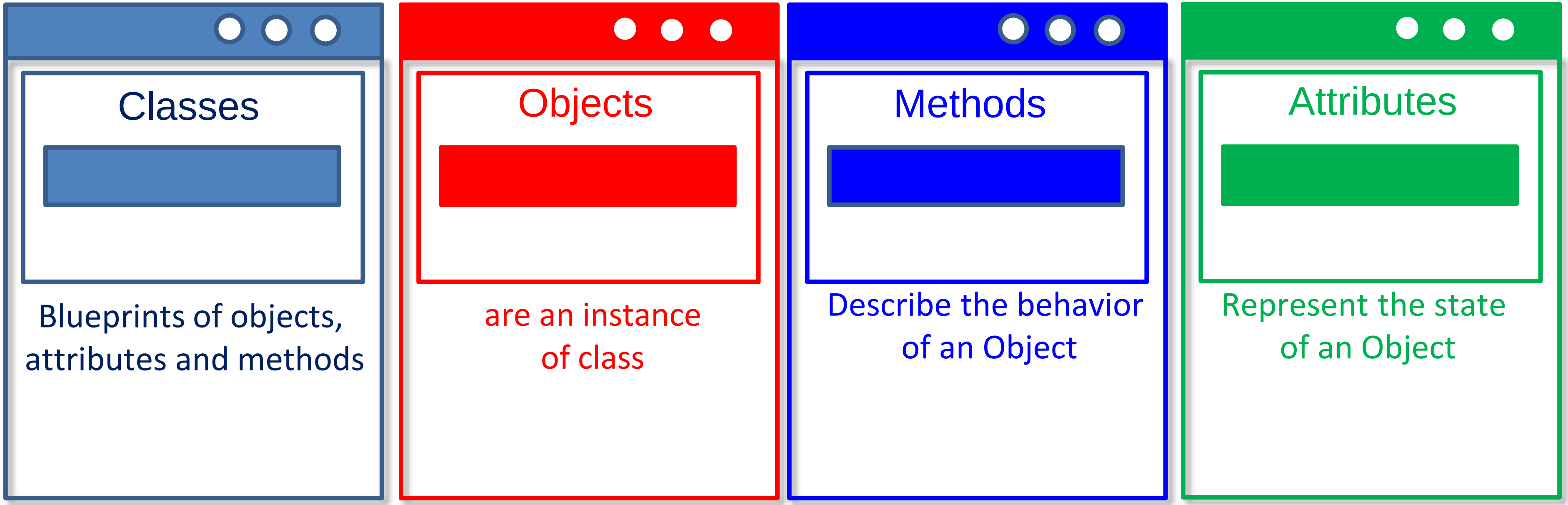
Polymorphism

1. Object Oriented Programming: Advantages



1. OOP Can support Large Software Project
2. OOP Offers Data Protection
3. Code Re-Usability
4. Better Representation of Real World Objects
5. Better Software Maintenance
6. Enhanced Security
7. Easy Code Modification

1. Object Oriented Programming: Structure



Classes

Blueprints of objects, attributes and methods

Objects

are an instance of class

Methods

Describe the behavior of an Object

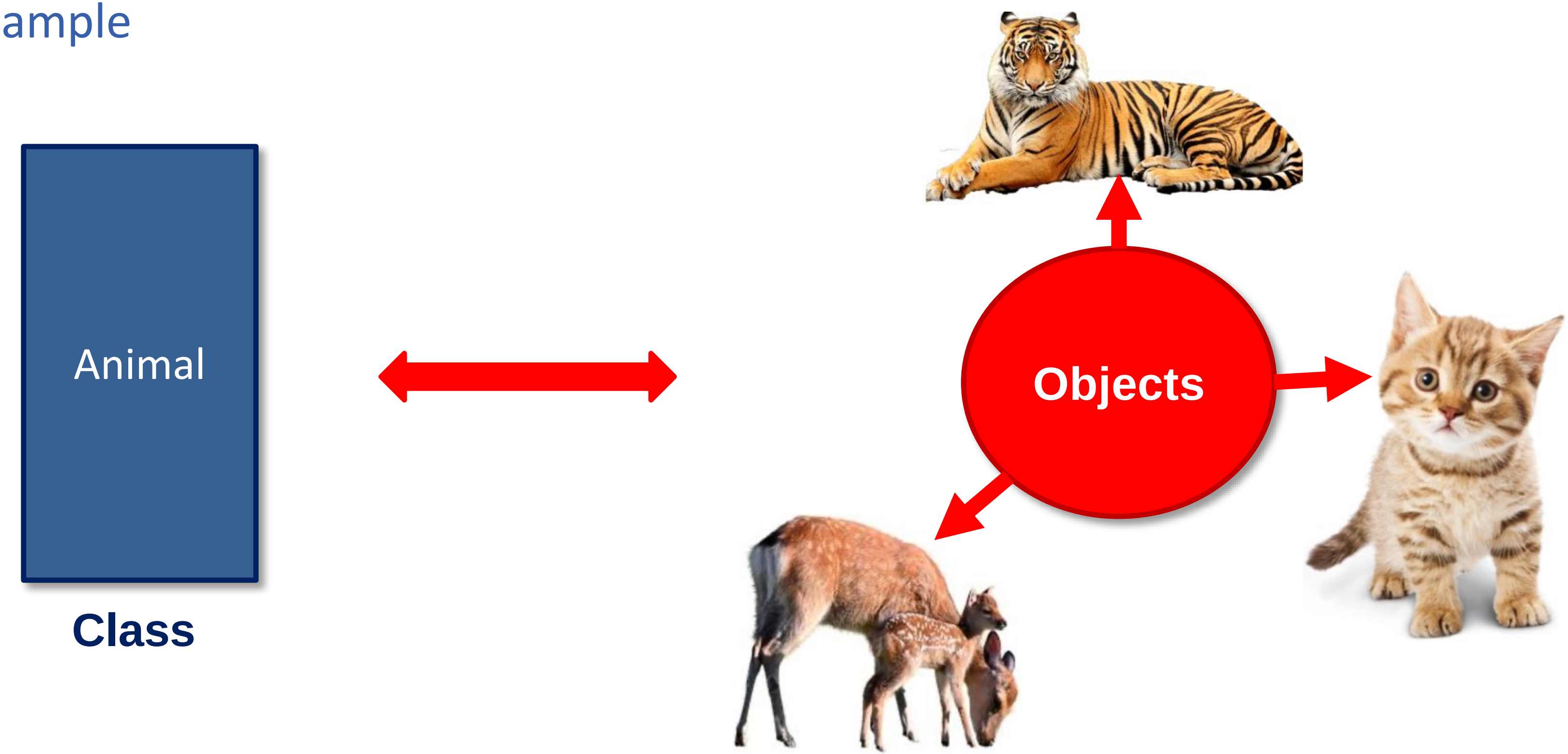
Attributes

Represent the state of an Object

Image Credit: https://www.freecodecamp.org/news/what-is-object-oriented-programming/

2. Class & Object

- Example



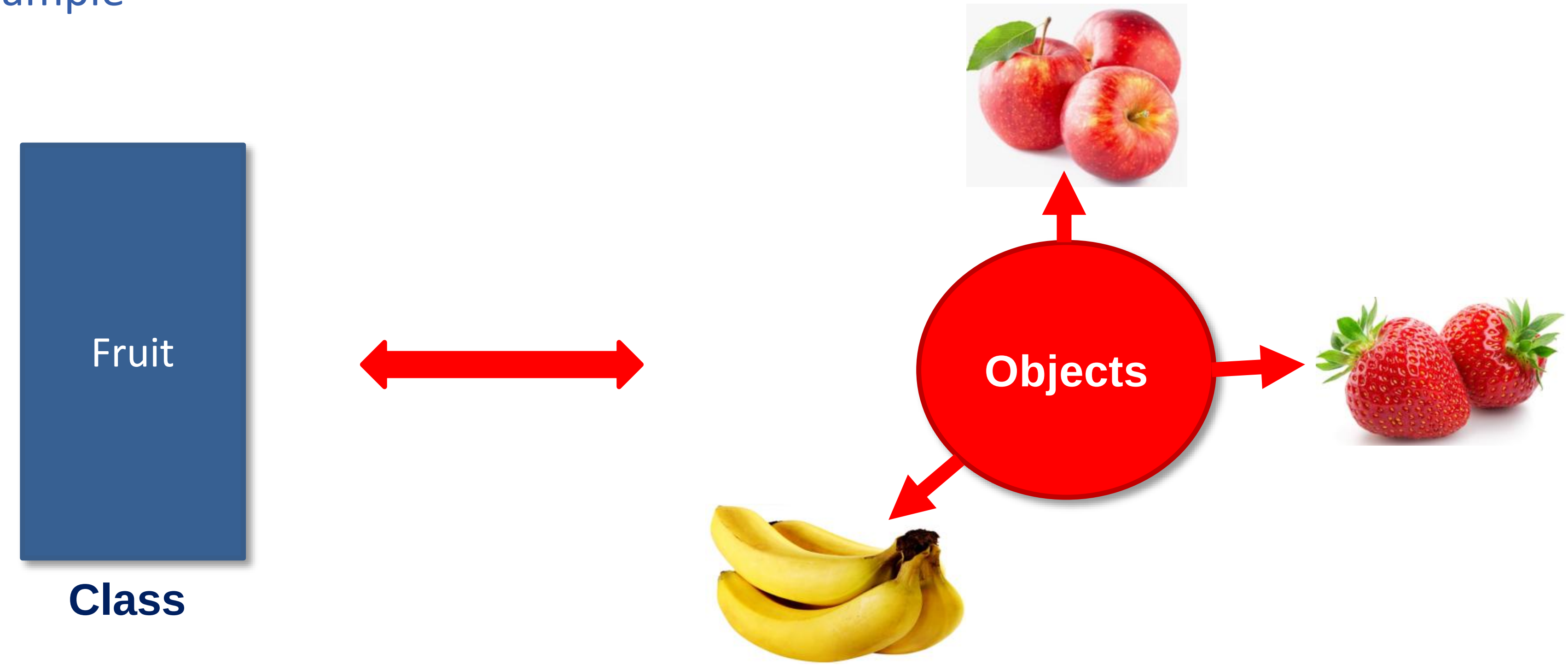
Animal

Class

Objects

2. Class & Object

- Example

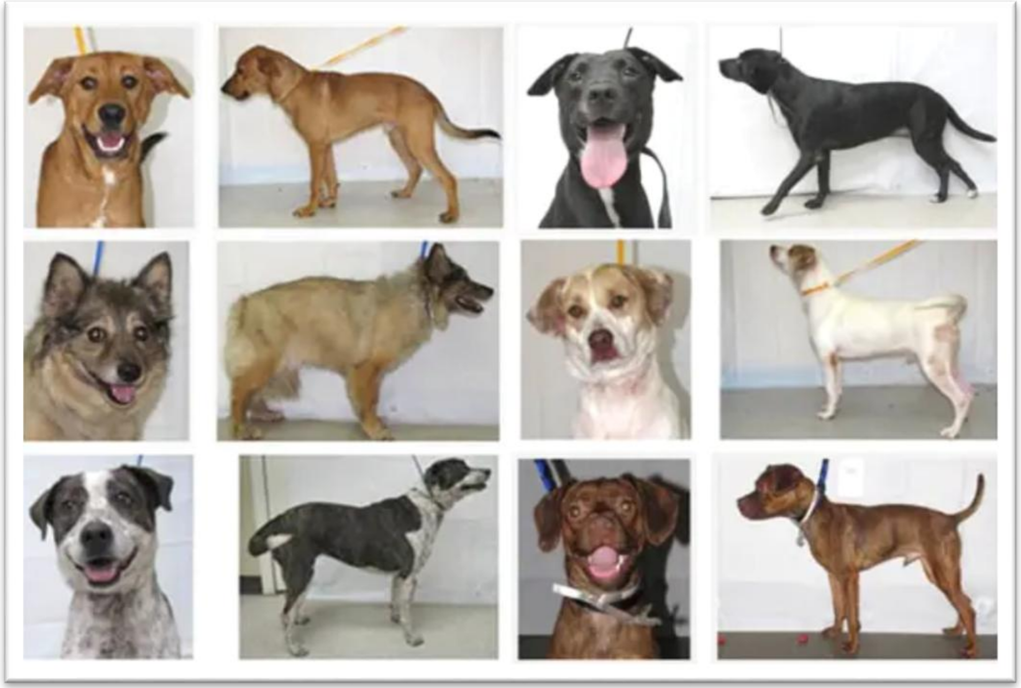


Fruit

Class

Objects

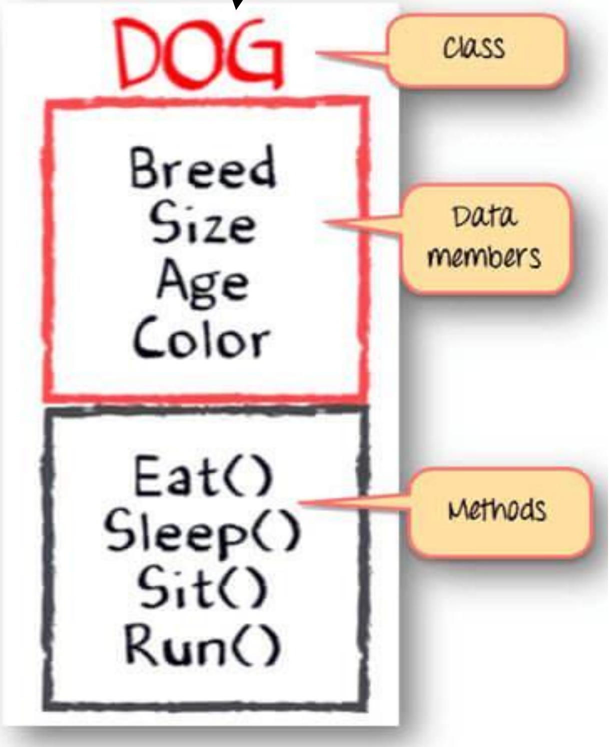
- Example: A Class of Dog
  - Common Characteristics
    - Breed
    - Size
    - Color
    - Age
  - Common Actions
    - Eat
    - Sleep
    - Sit
    - Run



2. Class & Object

- Example: A Class of Dog
  - Common Characteristics
    - Breed
    - Size
    - Color
    - Age
  - Common Actions
    - Eat
    - Sleep
    - Sit
    - Run

Class name



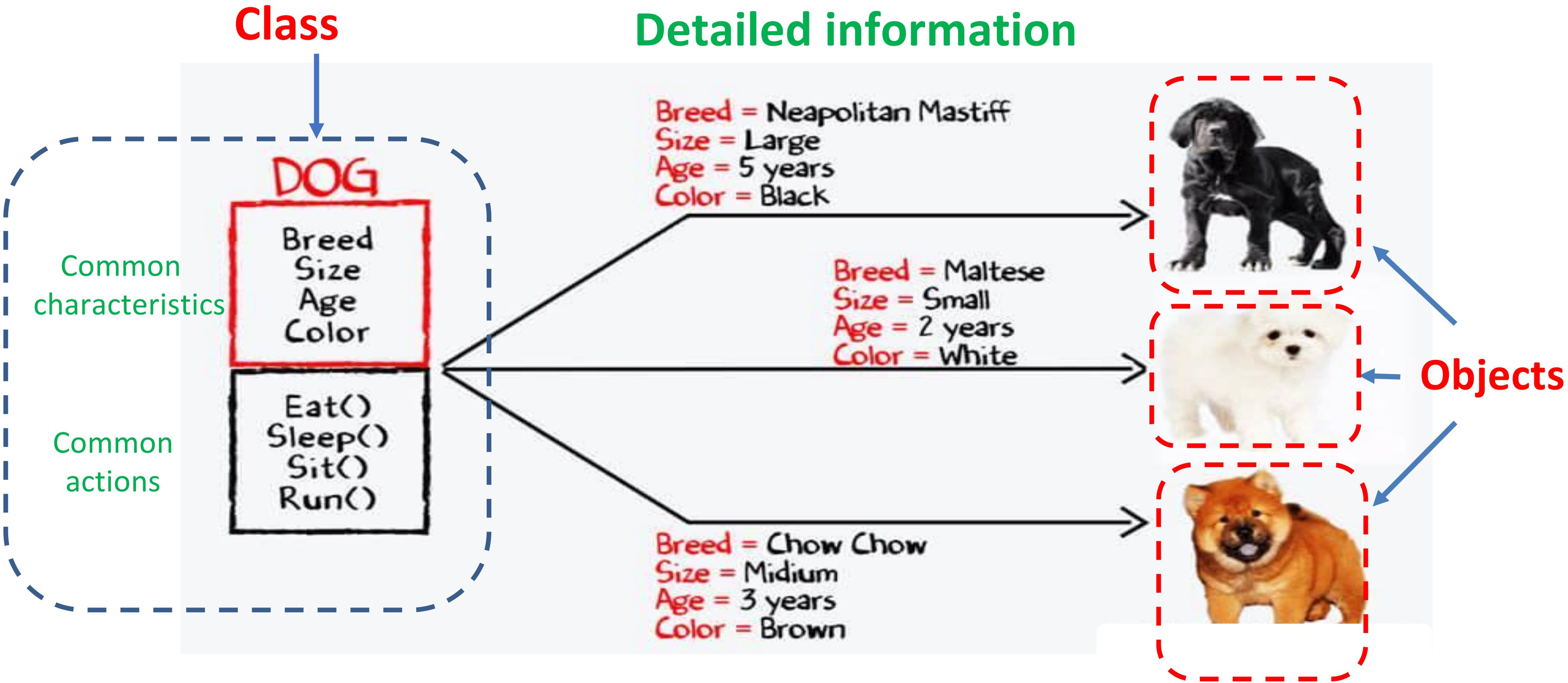


Image Credit: Guru99.com

2. Class & Object

- Class:
  - Is the template or blueprint from which objects are made
  - Can be defined as a collection of object
  - Including:
    - Attributes/ Characteristics/ field/ instance variable ←→ Data
    - Actions/Methods ←→ Functions
- Object: is an instance of class
  - → Attributes
  - → Methods
- Before we create an Object, we first need to define the Class.

2. Class & Object

- Example to define a Class:

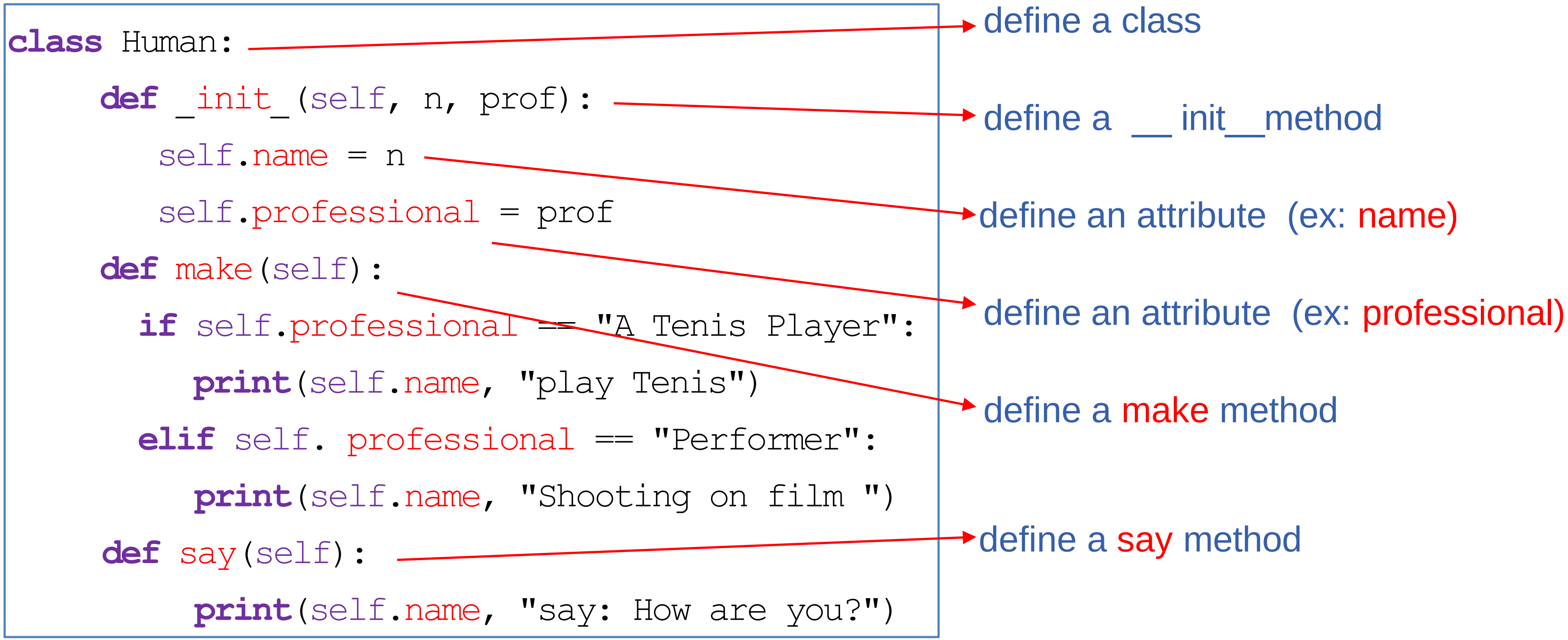
```
class Dog:
    def __init__(self, B, S, C, A):
        self.Breed = B
        self.Size  = S
        self.Age = C
        self.Color   = A
    def Eat(self):
        #...
    def Sleep(self):
        #...
    def Sit(self):
        #...
    def Run(self):
        #...
```

- Create the Objects: Dog_1, Dog_2, Dog_3

```
Dog_1 = Dog(Nepolitan Mastiff,Large,5 years, Black)
Dog_2 = Dog(Maltese,small,2 years, White)
Dog_2 = Dog(Chow Chow,midium,3 years, Brown)
```

- A Function created in a class then it's called a method
- The _init_() method is a special method, it's used to define the attributes and It's called when a Object is created
- A self parameter is the default parameter

- Example to define a class (Cont.):



```
class Human:
    def _init_(self, n, prof):
        self.name = n
        self.professional = prof
    def make(self):
        if self.professional == "A Tenis Player":
            print(self.name, "play Tenis")
        elif self. professional == "Performer":
            print(self.name, "Shooting on film ")
    def say(self):
            print(self.name, "say: How are you?")
```

define a class

define a __ init__method

define an attribute (ex: name)

define an attribute (ex: professional)

define a make method

define a say method

2. Class & Object

- Class:

- Define a Class:

```
class ClassName:
    def __int__():
        define the attributes 1
        define the attributes 2
        define the attributes 3
         …
        define the attributes N
    def MethodName_1():
     …
    def MethodName_2():
    …
    def MethodName_N():
```

- Object:

- Create a Object: ObjectName = ClassName(Attributes)
- Call a Method: ObjectName.MethodName()
- Access a Attribute: ObjectName.Attribute

- Example 1:

```
class oto:# Define a oto class
    def __init__(self, c, num):
        self.color = c
        self.numberWheel = num
    def Start_the_oto(self):
        print("Starting...")
my_oto = oto(“blue”,6)# Make a my_oto object
print("Color of Oto is: ", my_oto.color)
my_oto.Start_the_oto()
```

2. Class & Object

- Example 2:
  - Define a class: InputOutString which have 2 method:

    - getString(): get a string entered by the user from the keyboard.

    - printString(): print the string to the screen as an uppercase string.
  - Create a strObj

```
class InputOutString:# define a InputOutString Class
        def __init__(self, c):
            self.str = c
        def getString(self):
            self.str = input("Enter a string:")
        def printString(self):
           print(self.str.upper())
st=''
strObj = InputOutString(st) # Create a strObj Oject
strObj.getString() # Call a getString() method
strObj.printString() # Call a printString() method
```

2. Class & Object: Organize the storage and use

- Orgranize the storage and use:
  - **Step 1**: Define classes and save in a separate file (Module file)
  - **Step 2**:
    - Create a main program file
    - Use those classes in the main program:

```
from   Class_file    import *
```

    - Create an object from the class and use attributes and methods :
      - Create a Object: ObjectName = ClassName(Attribute)
      - Call a Method: ObjectName.MethodName()
      - Access a Attribute: ObjectName.Attribute

2. Class & Object: Organize the storage and use

- Example 3
- Step 1: Make a file oto_class.py:

```
class oto:
    def __init__(self, t):
        self.ten = t
    def start(self):
        print("Starting up...")
```

- Step 2: Make a file class_test.py:

```
from oto_class import *
my_oto = oto("audi")
print("This is a: " + my_oto.ten)
print("It is: " + my_oto.start())
```

2. Class & Object: Organize the storage and use

Example 4: Make a file dientich.py which has a class DT with 2 attributes: length and width; a method Tinh_DT() to calculate a rectangular area. Then, make a file Test_dt.py to create an object from the DT class to calculate and print the area of rectangle. Requirement: the length and width is inputed from keyboard.

Step 1: Make a file **dientich.py**:

```
class DT():
    def   init_(self, l, w):
        self.length = l
        self.width = w
    def Tinh_DT(self):
        s = self.length * self.width
        print(" the area of
  rectangle is", s)
```

Step 2: Make a file **test_dt.py**:

```
from dientich import *
lenghtValue=int(input(“Enter a length:"))
widthValue = int(input(“Enter a width:"))
S = DT(lenghtValue, widthValue)
S.Tinh_DT())
```

2. Class & Object: Attributes

- The data held by an object is represented by its attributes
- Types:
  - Class variables : defined within the scope of the class, but outside of any methods
  - Instance variables: are tied to the instance (objects) than class

- Example 5:

```
class Person:
    instance_count = 0 # instance_count is class variable
    def __init__(self, name, age):
        Person.instance_count += 1
        # name, age are instance variables
        self.name = name
        self.age = age
```

2. Class & Object: Constructor

- Special type of method which is used to initialize the instance members of the class.
- Types:
  - Parameterized Constructor
  - Non-parameterized Constructor
- Syntax: `__init__(<parameter>)`
- Example:

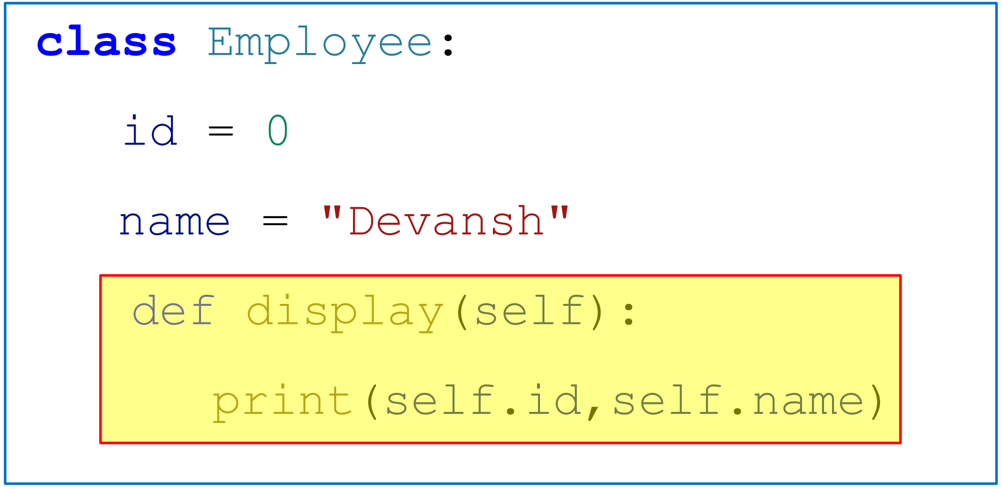
```
class Employee:
    def __init__(self, name, id):
        self.id = id
        self.name = name
    def display(self):
        print("ID:"+self.id + "\n" + self.name)
emp1 = Employee("John", 101)
emp2 = Employee("David", 102)
```

2. Class & Object: Method

- Instance method
- Class method
- Static method
- Special method
- Getter, setter method

2. Class & Object: Method

- Instance method
  - is tied to an instance of the class
  - Example:

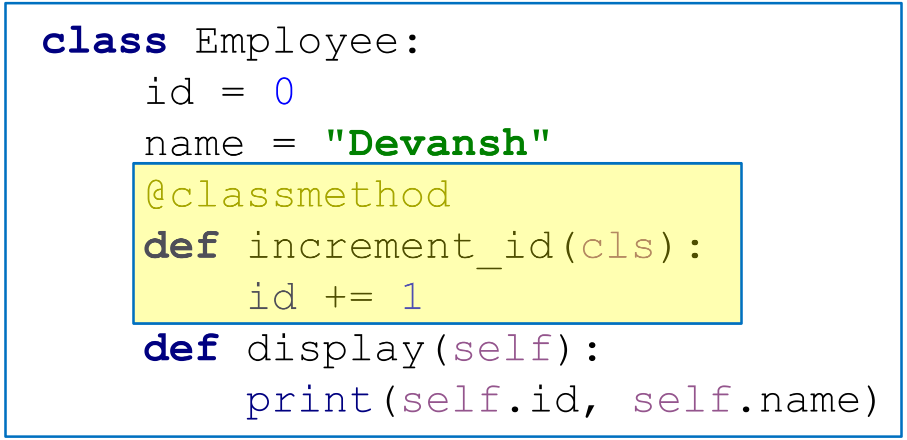


```
class Employee:
    id = 0
    name = "Devansh"
   def display(self):
       print(self.id,self.name)
```

- display(seft) is a instance method
- “self” is used as a reference variable, which refers to the current object. It is always the first argument in the function definition. However, using self is optional in the function call

2. Class & Object: Method

- Class method
  - Concept: behaviour that is linked to the class rather than an individual object
  - Example:



```
class Employee:
    id = 0
    name = "Devansh"
    @classmethod
    def increment_id(cls):
        id += 1
    def display(self):
        print(self.id, self.name)
```

- increment_id(cls) is a class method
- is decorated with“@classmethod” keyword and take a first parameter with “cls”

2. Class & Object: Method

- Static method
  - Concept: is defined within a class but are not tied to either the class nor any instance of the class
  - Example:

```
class Employee:
    id = 0
    name = "Devansh"
    @staticmethod
    def static_function():
        print(“Static method”)
```

- is decorated with the **@staticmethod** decorator
- the same as free standing functions but are defined within a class

2. Class & Object: Method

- Special method
  - Start and end with a double underbars ('__').
  - It’s called when a Object is created

__init__()	initialize the attributes of the object
__str__()	returns a string representation of the object
__len__()	returns the length of the object
__add__()	adds two objects
__call__()	call objects of the class like a normal function
__dict__()	
__doc__()	
__modulo__()	

2. Class & Object: Method

- Getter and setter methods
  - Concept: used to access the values of objects
  - Getter methods: decorated with the @property decorator
  - Setter methods: decorated with the @attribute_name.setter decorator

2. Class & Object: Method

- Class Without Getters and Setters
- Example:
  - Assume that to make a class that stores the temperature in degrees Celsius
  - implement a method to convert the temperature into degrees Fahrenheit

```
class Celsius:
    def __init__(self, temperature=0):
        self.temperature = temperature
    def to_fahrenheit(self):
        return (self.temperature*1.8)+32
# Create a new object
human = Celsius()
# Set the temperature
human.temperature = 37
# Get the temperature attribute
print(human.temperature)
# Get the to_fahrenheit method
print(human.to_fahrenheit())
```

```
37
98.60000000000001
```

2. Class & Object: Method

- Example about Getter and setter methods:

```
class Example: # Attribute
  __domain = ''
  # Getter @property
  def domain(self):
    return self.__domain
 # Setter @domain.setter
  def domain(self, domain):
    self.__domain = domain
```

3. Inheritance

- Being an object-oriented language, Python supports class inheritance

  ➔ It allows us to create a new class from an existing one.
- The newly created class is known as the subclass (child or derived class).
- The existing class from which the child class inherits is known as the superclass (parent/base class/super class)
- Inheritance Syntax:

```
class super_class:
    # attributes and method definition

# inheritance
class sub_class(super_class):
    # attributes and method of super_class
    # attributes and method of sub_class
```

3. Inheritance

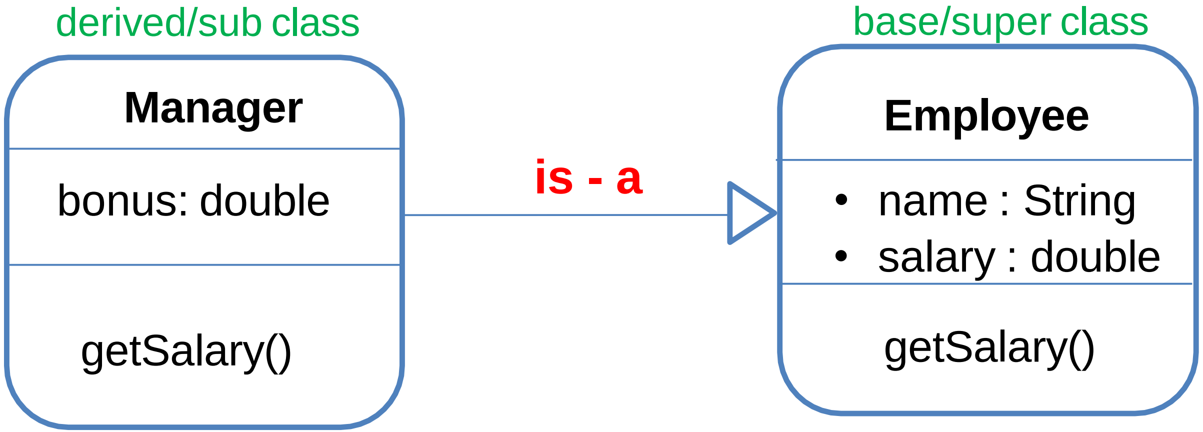
```python
class Animal:    # parent class
    # attribute and method of the parent class
    name = ""
    def eat(self):
        print("I can eat")

class Dog(Animal): # sub class is inherited from Animal
    # new method in subclass
    def display(self):
        # access name attribute of superclass using self
        print("My name is ", self.name)

labrador = Dog() # create an object of the subclass
# access superclass attribute and method
labrador.name = "Rohu"
labrador.eat()

labrador.display() # call subclass method
```

The child class acquires the properties and can access all the data members and methods defined in the parent class

```
I can eat
My name is  Rohu
```

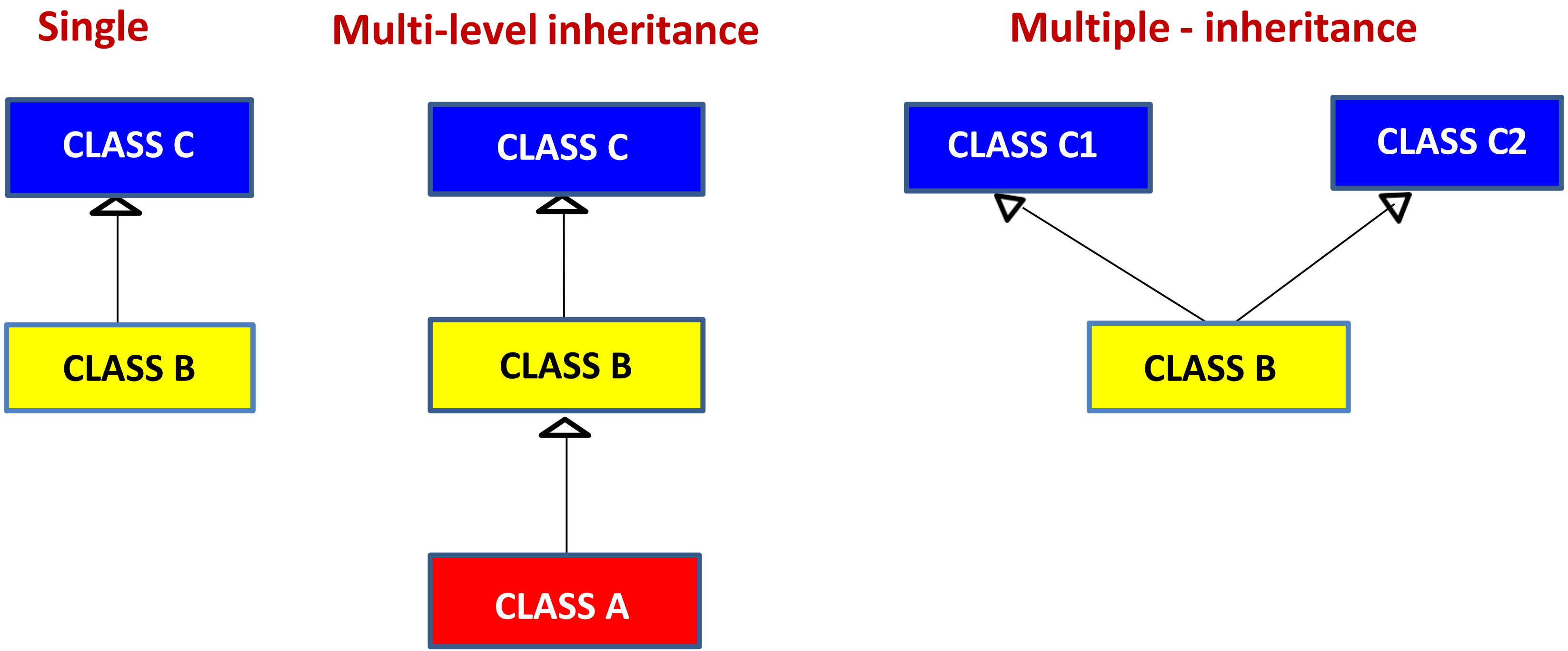
3. Inheritance

- Inheritance is an **is-a** relationship ➔ we use inheritance only if there exists an **is-a** relationship between two classes.
- Example
  - Car **is a** Vehicle
  - Apple **is a** Fruit
  - Dog **is an** Animal
  - Cat **is an** Animal
  - Manager **is a** Employee



3. Inheritance

- Inheritance types



Single

Multi-level inheritance

Multiple - inheritance

- Method Overriding in Inheritance
  - The object of the subclass can access the method of the superclass.
  - However, what if the same method is present in both the superclass and subclass?
  - ➔ the method in the subclass overrides the method in the superclass ➔ This concept is known as method overriding ➔ run method of sub class

```
class Animal: # parent class
    # attributes and method of the
parent class
    name = ""
    def eat(self):
        print("I can eat")

# sub class is inherited from Animal
class Dog(Animal):
    # override eat() method
    def eat(self):
        print("I like to eat bones")

# create an object of the subclass
labrador = Dog()
# call the eat() on the labrador object
labrador.eat() ➔ I like to eat bones
```

3. Inheritance

- The super() method in Inheritance
  - if we need to access the superclass method from the subclass, we use the super() function.
- Example

```
class Animal: # parent class
    name = ""
    def eat(self):
        print("I can eat")

# sub class is inherited from Animal
class Dog(Animal):
    # override eat() method
    def eat(self):
        # call the eat() method of the
        # superclass using super()
        super().eat()
        print("I like to eat bones")

# create an object of the subclass
labrador = Dog()
labrador.eat()
```

```
I can eat
I like to eat bones
```

4. Polymorphism

- Polymorphism
  - Polymorphism is a very important concept in programming.
  - It refers to the use of a single type entity (method, operator or object) to represent different types in different scenarios/ways.
  - Example: “+”

```
num1 = 1
num2 = 2
print(num1+num2)
```

```
str1 = "Python"
str2 = "Programming"
print(str1+" "+str2)
```

```
print(len("Programiz"))
print(len(["Python", "Java", "C"]))
print(len({"Name": "John", "Address": "Nepal"}))
```

  - Use the concept of polymorphism while creating class methods as allows different classes to have methods with the same name ➔ Then generalize calling these methods by disregarding the object we are working with.

4. Polymorphism

```python
class Cat:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def info(self):
        print(f"I am a cat. My name is {self.name}.
              I am {self.age} years old.")
    def make_sound(self):
        print("Meow")
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def info(self):
        print(f"I am a dog. My name is {self.name}.
              I am {self.age} years old.")
    def make_sound(self):
        print("Bark")
```

```python
cat1 = Cat("Kitty", 2.5)
dog1 = Dog("Fluffy", 4)
for animal in (cat1, dog1):
    animal.make_sound()
    animal.info()
    animal.make_sound()
```

```
Meow
I am a cat. My name is Kitty. I am 2.5 years old.
Meow
Bark
I am a dog. My name is Fluffy. I am 4 years old.
Bark
```

4. Polymorphism

- Polymorphism and Inheritance
  - The child classes in Python also inherit methods and attributes from the parent class.
  - Polymorphism allows us to access these overridden methods and attributes that have the same name as the parent class.

4. Polymorphism

```
from math import pi
class Shape:
    def __init__(self, name):
        self.name = name
    def area(self):
        pass
    def fact(self):
        return “A two-dimensional shape."
    def __str__(self):
        return self.name
class Square(Shape):
    def __init__(self, length):
        super().__init__("Square")
        self.length = length
    def area(self):
        return self.length**2
    def fact(self):
        return "Squares have each angle
                equal to 90 degrees."
```

```
class Circle(Shape):
    def __init__(self, radius):
        super().__init__("Circle")
        self.radius = radius
    def area(self):
        return pi*self.radius**2

a = Square(4)
b = Circle(7)
print(b)
print(b.fact())
print(a.fact())
print(b.area())
```

```
A two-dimensional shape.
Squares have each angle equal to 90 degrees.
153.93804002589985
```

5. Abstraction

- **Abstract class** is a class in which one or more **abstract methods** are defined. When a method is declared inside the class without its implementation is known as abstract method.

- **Abstract Method**: To create abstract method and abstract classes we have to import the “ABC” and “abstractmethod” classes from abc (Abstract Base Class) library.

- Example

```
from abc import ABC, abstractmethod
class BaseClass(ABC):
    @abstractmethod
    def method_1(self):
         #empty body
         pass
```

- Abstract Class
  - Abstract Base Classes (ABCs) :
    - Can be used to define generic (potentially abstract) behaviour that can be mixed into other Python classes and act as an abstract root of a class hierarchy.
    - There are many built-in ABCs in Python including (but not limited to): IO, numbers, collection,…modules
  - Declared an Abstract Class
    - Step 1 :import ABCs, abstract method
    - Step 2: Declared an Abstract Class inheritance from ABC class in step 1
    - Step 3: Declared Abstract Methods

5. Abstraction

- Abstract Class

  - Example 1:

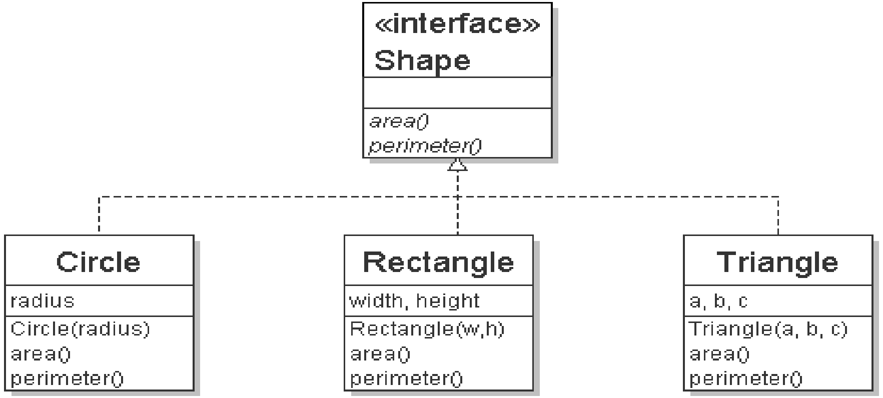
```
from abc import ABC, abstractmethod
class Vidu(ABC):
    @abstractmethod
    def methodName(self):
        pass
```

  - Example 2:

```
from collections import MutableSequence.abstractmethod
class Bag(MutableSequence):
    @abstractmethod
    def methodName(self):
        pass
```

5. Abstraction: Interface

- **Interface**: this is a contract between the implementors of an interface and the user of the implementation guaranteeing that certain facilities will be provided. Python does not explicitly have the concept of an interface contract (note here interface refers to the interface between a class and the code that utilizes that class).
- Example: Create an “interface” Shape and subclasses: Circle, Rectangle, Triangle



5. Abstraction: Interface

- Interface

- “Interface” Shape

```
from abc import ABC, abstractmethod
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass
    @abstractmethod
    def perimeter(self):
        pass
```

- Circle Class

```
class Circle(Shape):
    def __init__(self, radius) -> None:
        super().__init__()
        self.radius=radius
    def area(self):
        return 3.14*self.radius*self.radius
    def perimeter(self):
        return 2*3.14*self.radius
```

- Rectangle Class

```
class Rectangle(Shape):
    def __init__(self, width, height) -> None:
        super().__init__()
        self.width=height
        self.height=height
    def area(self):
        return self.width*self.height
    def perimeter(self):
        return (self.width+self.height)*2
```

6. Encapsulation

- It is used to restrict access to methods and variables.
- In encapsulation, code and data are wrapped together within a single unit from being modified by accident.
- Note:
  - Private Attributes
  - Using getter and setter methods to access data

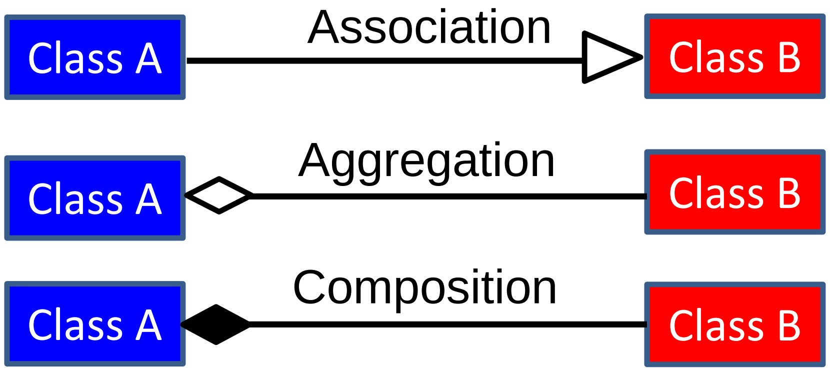
```
class Student:
    def __init__(self, univer):
        self.__univer=univer
    # getter
    @property
    def univer(self):
        return self.__univer
    # Missing setter method
    # only read data
```

```
a= Student(“VKU")
a.__univer=“VKU University"
print(a.univer)
# result is not changed: “VKU”
```

```
class Student:
    def __init__(self, univer):
        self.__univer=univer
    # setter
    @univer.setter
    def univer(self,univer):
        self.__univer=univer
    # Missing getter method
    # Only write data
```

```
a= Student(“VKU")
a.univer=“VKU University"
print(a.__univer)
# Cannot print
```

7. Relationships



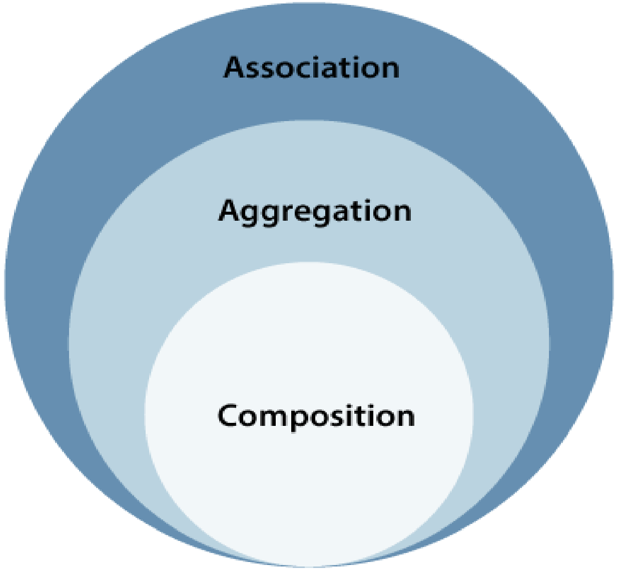
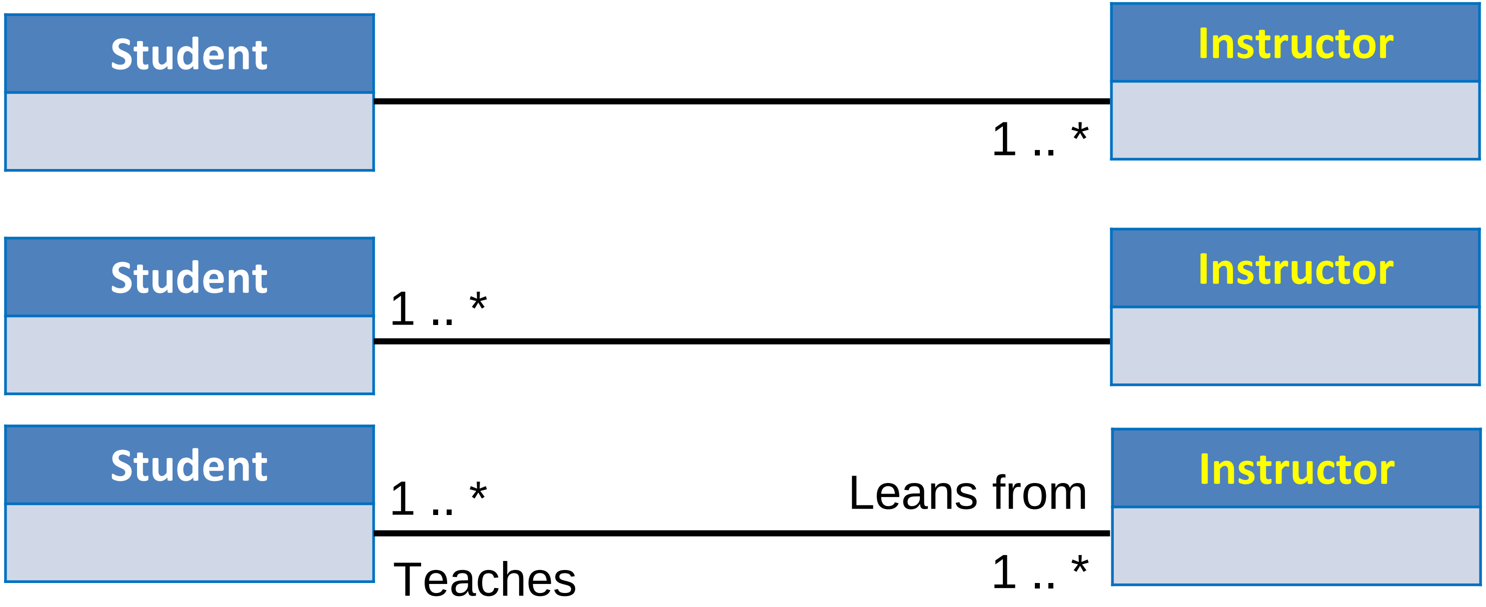
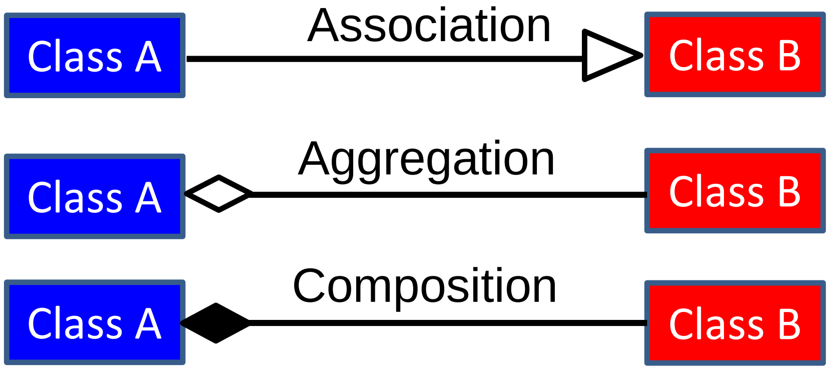


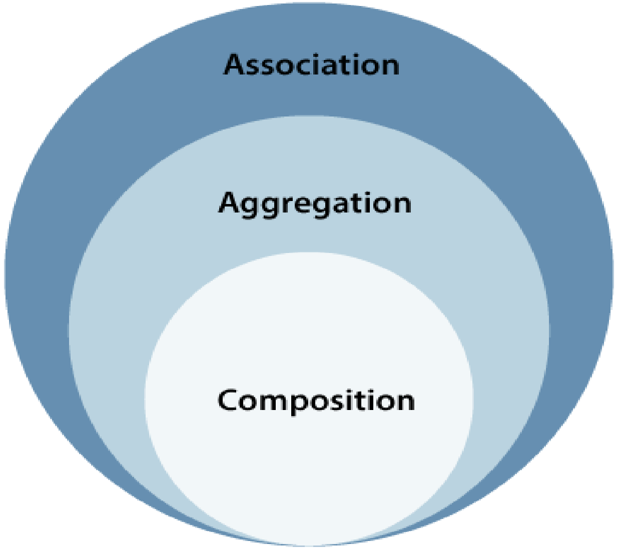
Image Credit:https://softwareengineering.stackexchange.com/

- Association
  - If two classes in a model need to communicate with each other, there must be a link between them, and that can be represented by an association (connector).



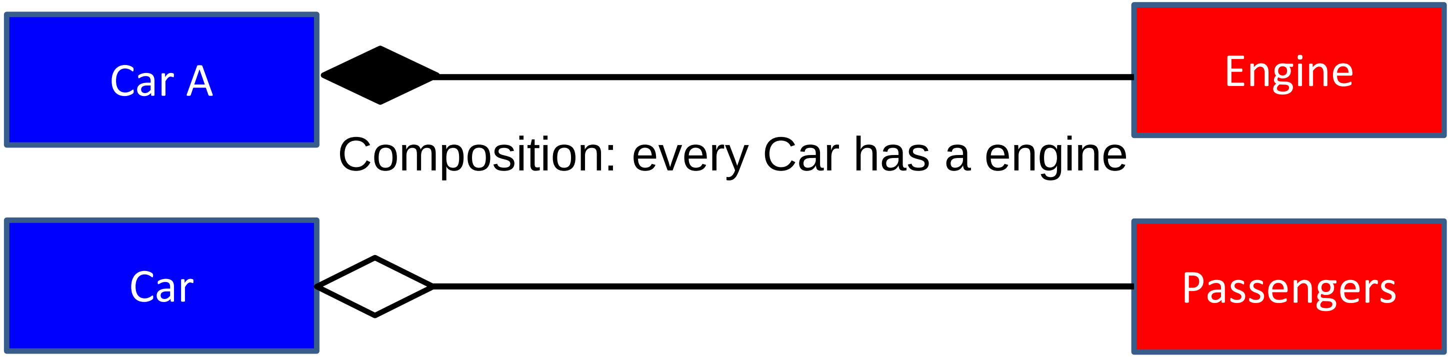
- Aggregation and Composition





- **Aggregation** and **Composition** are subsets of association meaning they are specific cases of association.
- In both **Aggregation** and **Composition** object of one class "owns" object of another class. But there is a difference meaning

- Aggregation

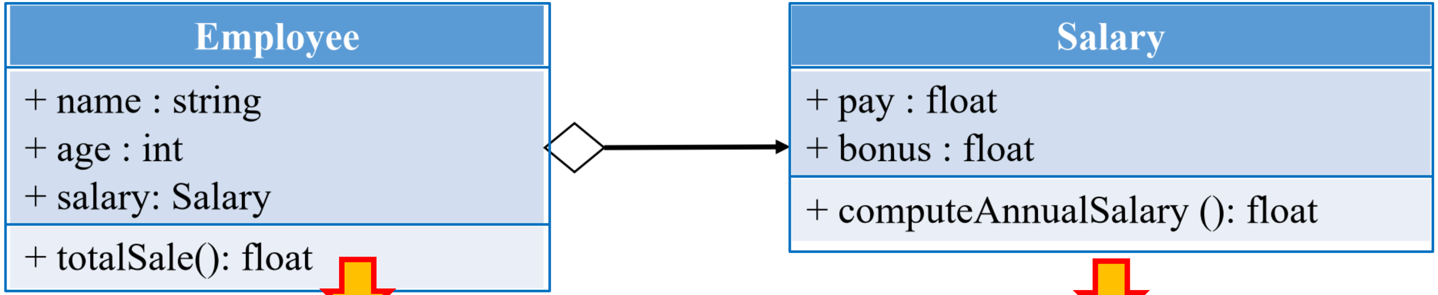


Composition: every Car has a engine

Aggregation: Cars may have Passengers, they come and go

Aggregation is one type of association between two objects describing the "**have/has a**" relationship, while Composition is a specific type of Aggregation which implies ownership.

- Aggregation: Example



Employee

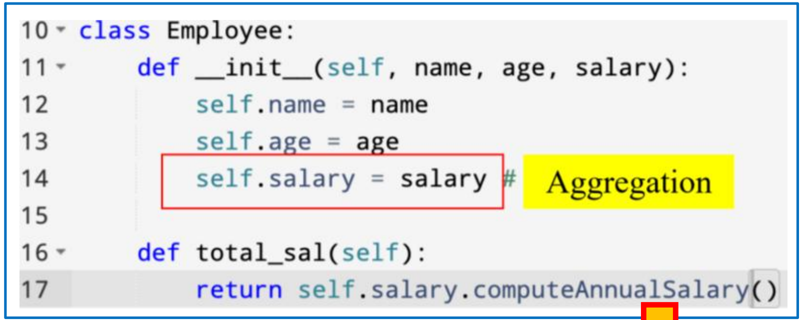
+ name : string
+ age : int
+ salary: Salary

+ totalSale(): float

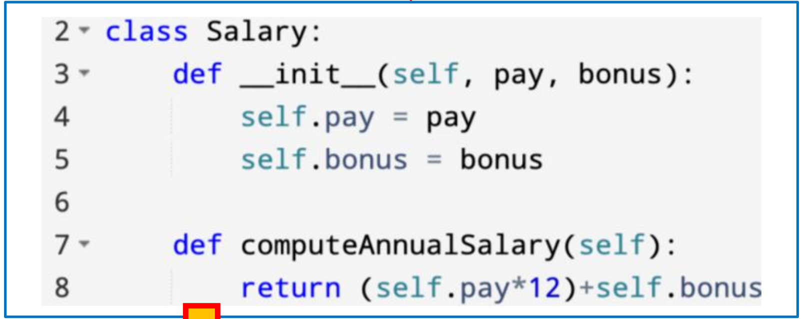
Salary

+ pay : float
+ bonus : float

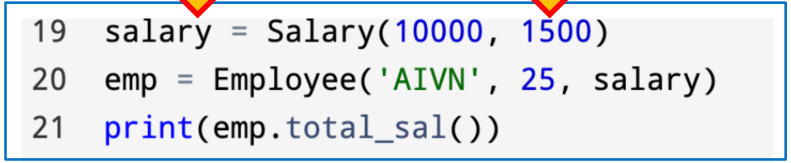
+ computeAnnualSalary (): float



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

    def total_sal(self):
        return self.salary.computeAnnualSalary()
```



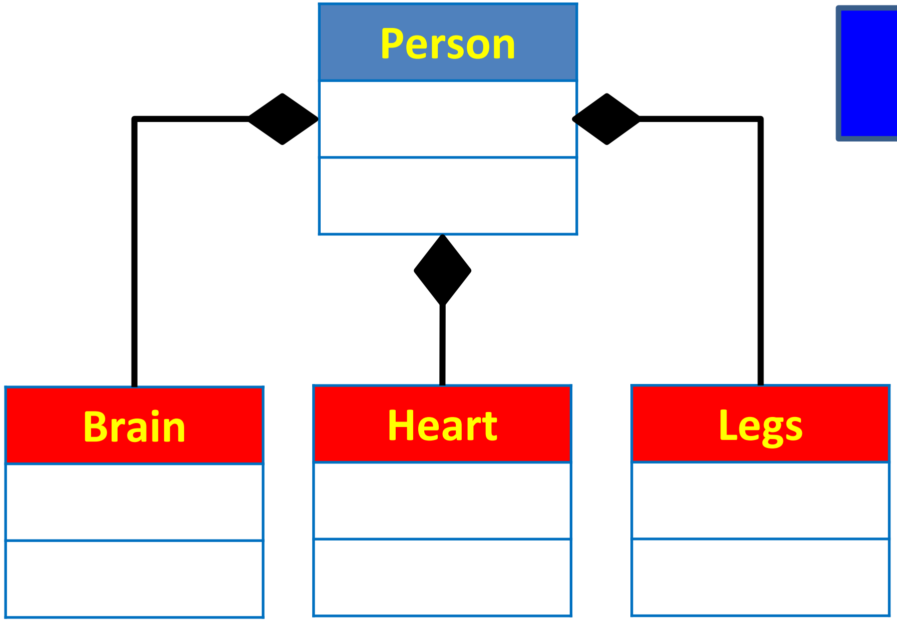
```
class Salary:
    def __init__(self, pay, bonus):
        self.pay = pay
        self.bonus = bonus

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

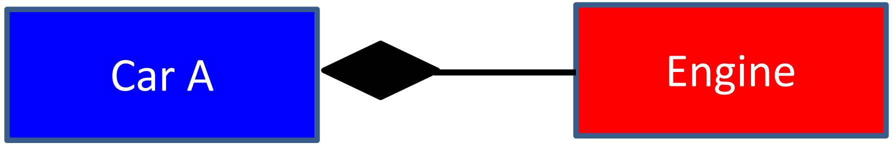


```
salary = Salary(10000, 1500)
emp = Employee('AIVN', 25, salary)
print(emp.total_sal())
```

121500

- Composition



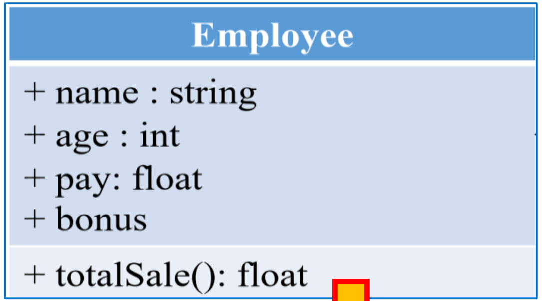


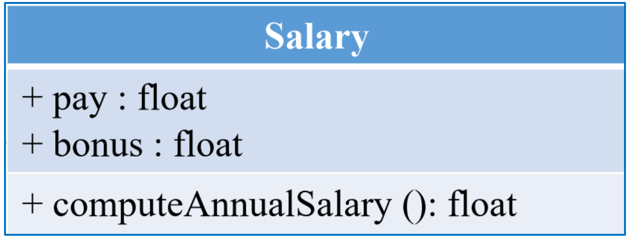
Composition: every Car has a engine

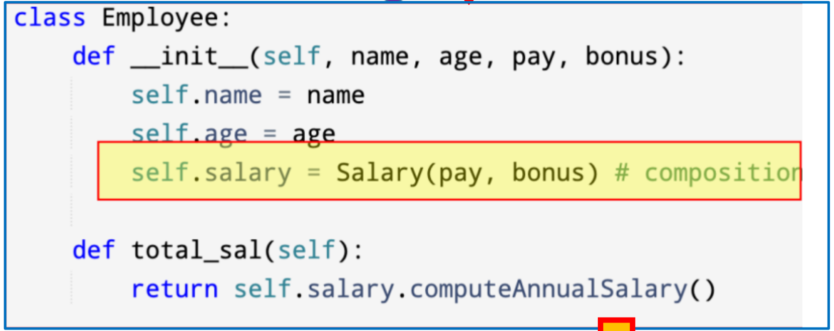
Composition is defined by the PART-OF relationship which means that one object IS PART-OF ANOTHER OBJECT

7. Relationships

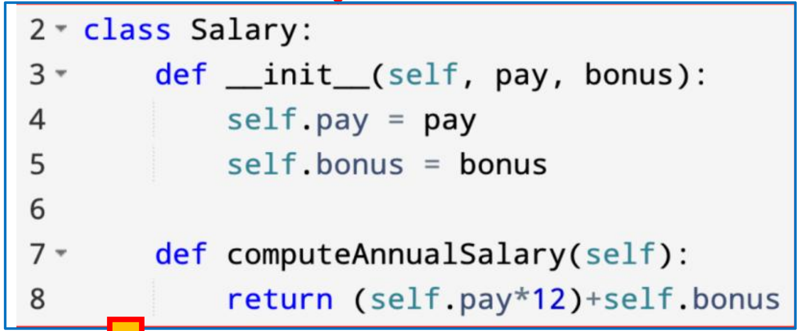
- Composition: Example



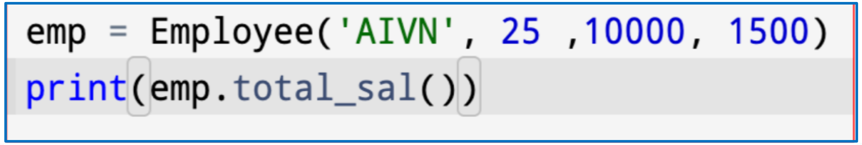




```python
class Employee:
    def __init__(self, name, age, pay, bonus):
        self.name = name
        self.age = age
        self.salary = Salary(pay, bonus) # composition

    def total_sal(self):
        return self.salary.computeAnnualSalary()
```



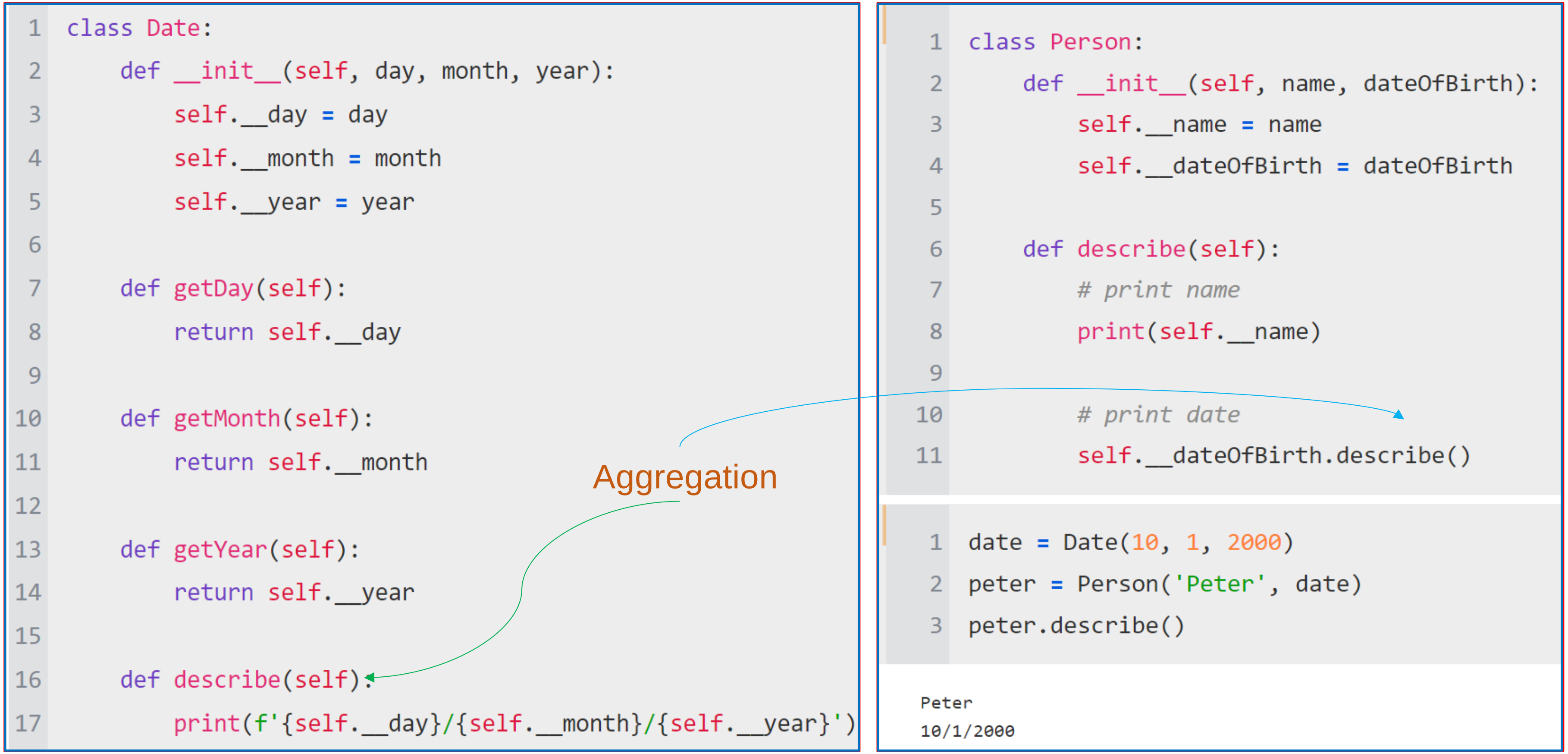
```python
class Salary:
    def __init__(self, pay, bonus):
        self.pay = pay
        self.bonus = bonus

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



```python
emp = Employee('AIVN', 25 ,10000, 1500)
print(emp.total_sal())
```

→ 121500

7. Relationships



```python
class Date:
    def __init__(self, day, month, year):
        self.__day = day
        self.__month = month
        self.__year = year

    def getDay(self):
        return self.__day

    def getMonth(self):
        return self.__month

    def getYear(self):
        return self.__year

    def describe(self):
        print(f'{self.__day}/{self.__month}/{self.__year}')
```

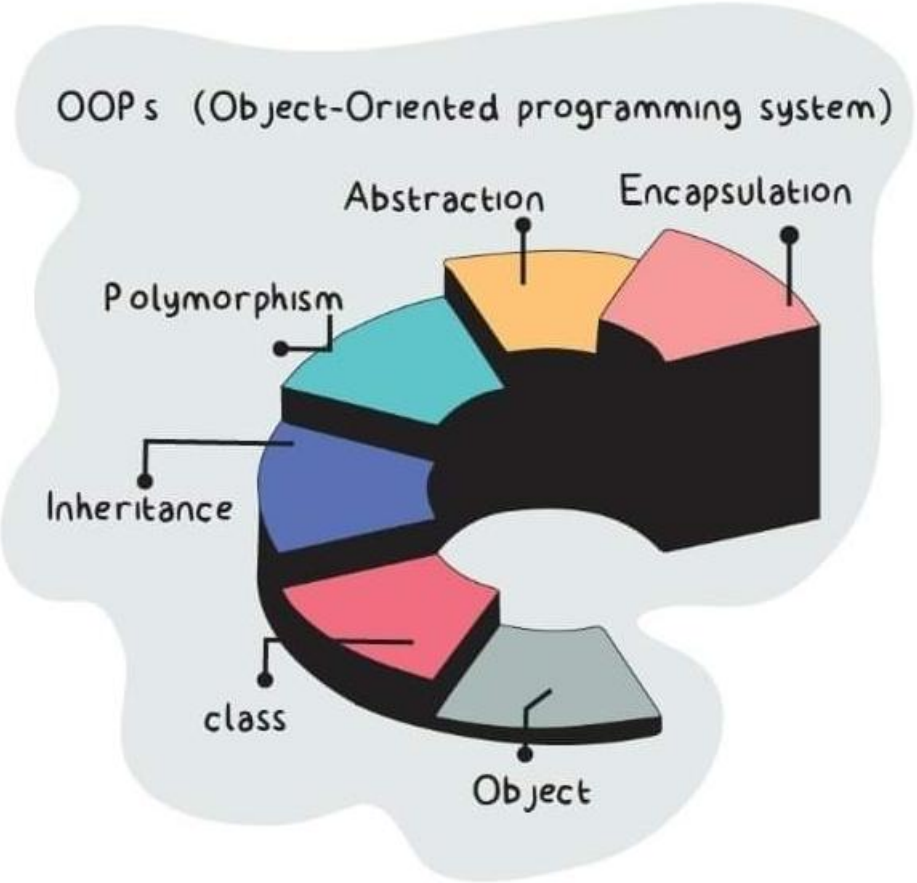
Aggregation

```python
class Person:
    def __init__(self, name, dateOfBirth):
        self.__name = name
        self.__dateOfBirth = dateOfBirth

    def describe(self):
        # print name
        print(self.__name)

        # print date
        self.__dateOfBirth.describe()
```

```python
date = Date(10, 1, 2000)
peter = Person('Peter', date)
peter.describe()
```

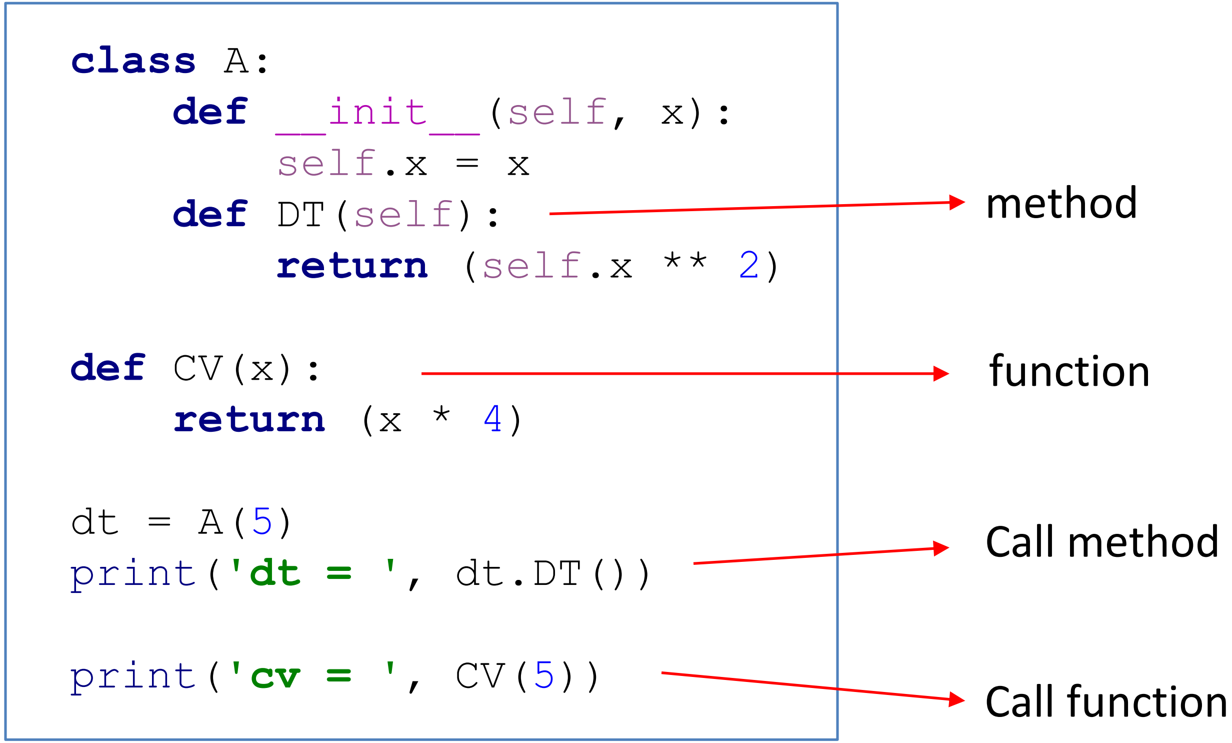
```
Peter
10/1/2000
```

Summarize


OOPs (Object-Oriented programming system)
Abstraction
Encapsulation
Polymorphism
Inheritance
class
Object

Summarize

- Differences between Method and Function



```
class A:
    def __init__(self, x):
        self.x = x
    def DT(self):
        return (self.x ** 2)

def CV(x):
    return (x * 4)

dt = A(5)
print('dt = ', dt.DT())

print('cv = ', CV(5))
```

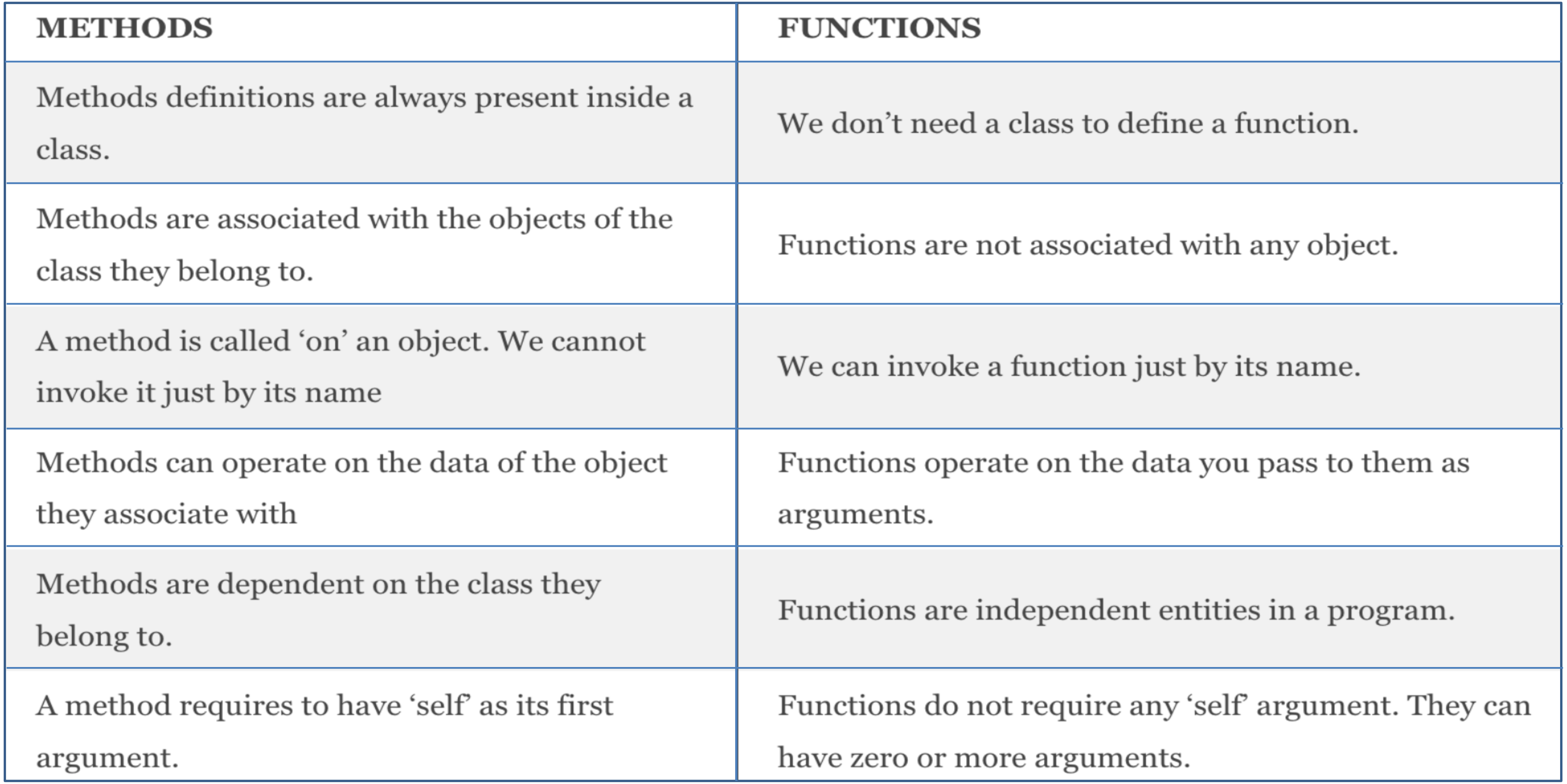
method

function

Call method

Call function

Summarize



METHODS	FUNCTIONS
Methods definitions are always present inside a class.	We don’t need a class to define a function.
Methods are associated with the objects of the class they belong to.	Functions are not associated with any object.
A method is called ‘on’ an object. We cannot invoke it just by its name	We can invoke a function just by its name.
Methods can operate on the data of the object they associate with	Functions operate on the data you pass to them as arguments.
Methods are dependent on the class they belong to.	Functions are independent entities in a program.
A method requires to have ‘self’ as its first argument.	Functions do not require any ‘self’ argument. They can have zero or more arguments.

Summarize

- Class and Object

Dog Class

Every Dog Class that is created will have a:

Name
Age
Breed
Shot status

Object is instance of Class

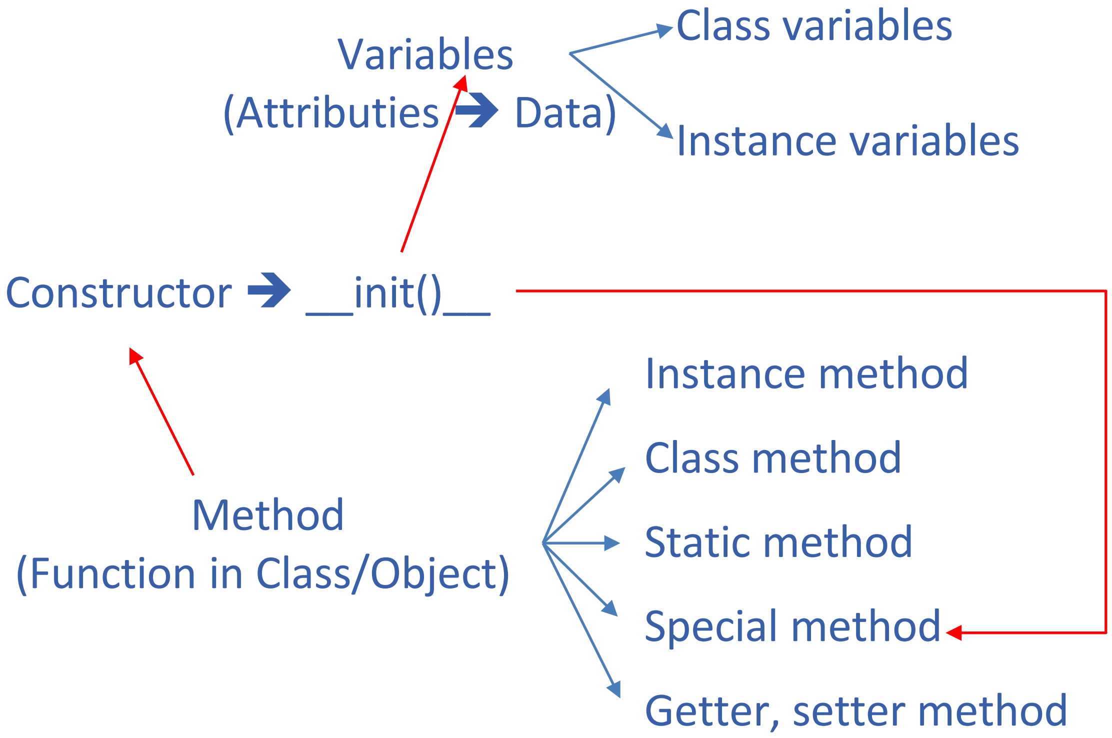
Name: Ginner
Age: 6
Breed: Akita
Shot status: Up to date

Name: Bowser
Age: 4
Breed: Retriever
Shot status: Up to date

Name: Roxy
Age: 5
Breed: Beagle
Shot status: Up to date

Summarize

- Attributies and Method



Variables

(Attributies ➔ Data)

Class variables

Instance variables

Constructor ➔ __init()__

Method

(Function in Class/Object)

Instance method

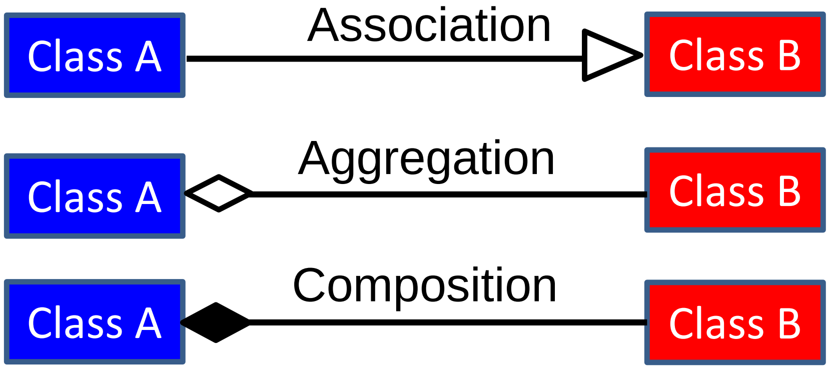
Class method

Static method

Special method

Getter, setter method

- Relationships



Summarize

- Orgranize the storage and use:
  - **Step 1**: Define classes and save in a separate file (Module file)
  - **Step 2**:
    - Create a main program file
    - Use those classes in the main program:

```
from   Class_file    import *
```

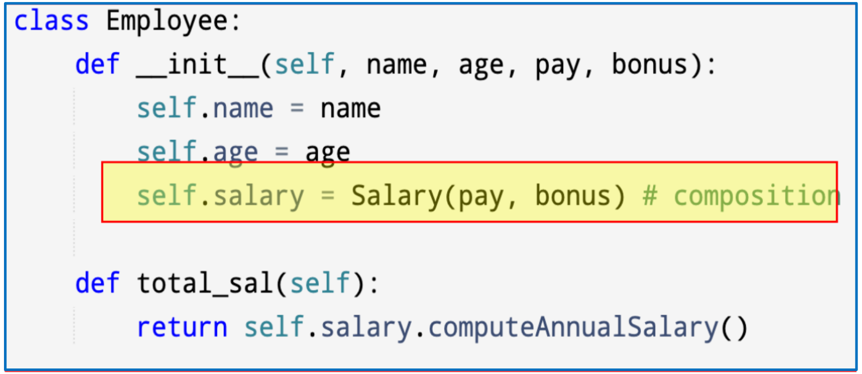
    - Create an object from the class and use attributes and methods :
      - Create a Object: ObjectName = ClassName(Attribute)
      - Call a Method: ObjectName.MethodName()
      - Access a Attribute: ObjectName.Attribute

Practice and exercises

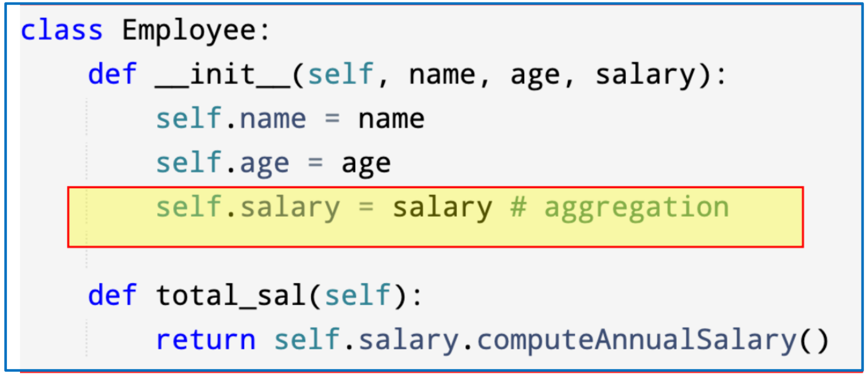
- **1. Practice**: Practice all the examples of Chapter 3
- **2. Exercise**: Using object-oriented programming techniques to complete from 17 to 25 of Chapter 2
- 3. Complete the exercises in the next slide below

Exercises Chapter 3

- Question 1: What is main different between Aggregation and Composition



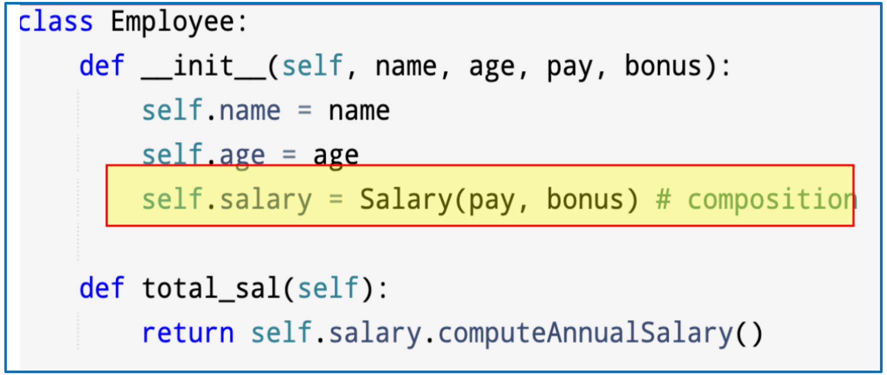
```python
class Employee:
    def __init__(self, name, age, pay, bonus):
        self.name = name
        self.age = age
        self.salary = Salary(pay, bonus) # composition

    def total_sal(self):
        return self.salary.computeAnnualSalary()
```



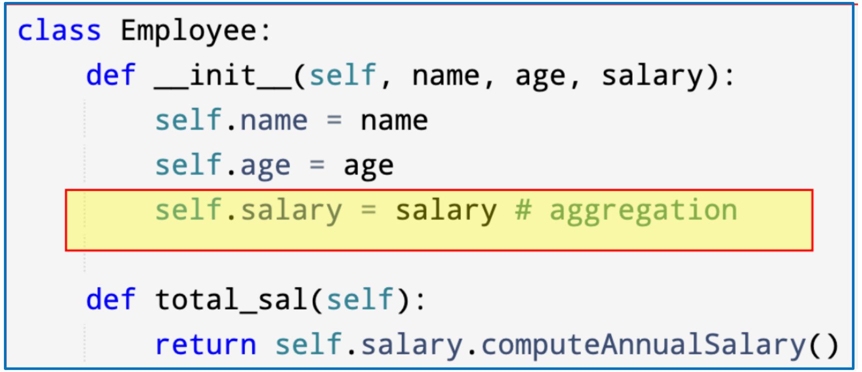
```python
class Employee:
    def __init__(self, name, age, salary):
        self.name = name
        self.age = age
        self.salary = salary # aggregation

    def total_sal(self):
        return self.salary.computeAnnualSalary()
```

- Answer

- Question 2: When should we use Aggregation and Composition in OOP?



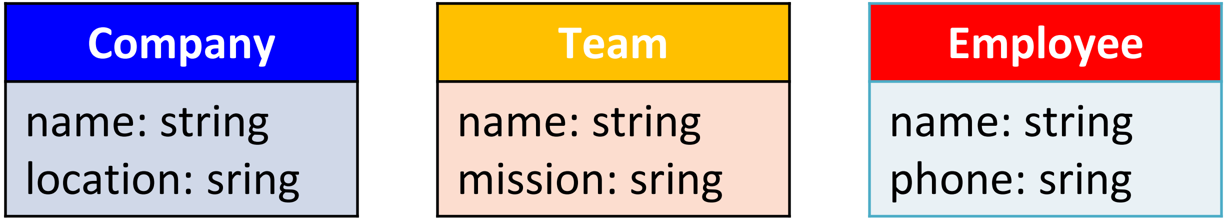
```python
class Employee:
    def __init__(self, name, age, pay, bonus):
        self.name = name
        self.age = age
        self.salary = Salary(pay, bonus) # composition

    def total_sal(self):
        return self.salary.computeAnnualSalary()
```



```python
class Employee:
    def __init__(self, name, age, salary):
        self.name = name
        self.age = age
        self.salary = salary # aggregation

    def total_sal(self):
        return self.salary.computeAnnualSalary()
```

- Answer

- Question 3: Determine and implement the relationship between the following classes in Python



Company

name: string
location: sring

Team

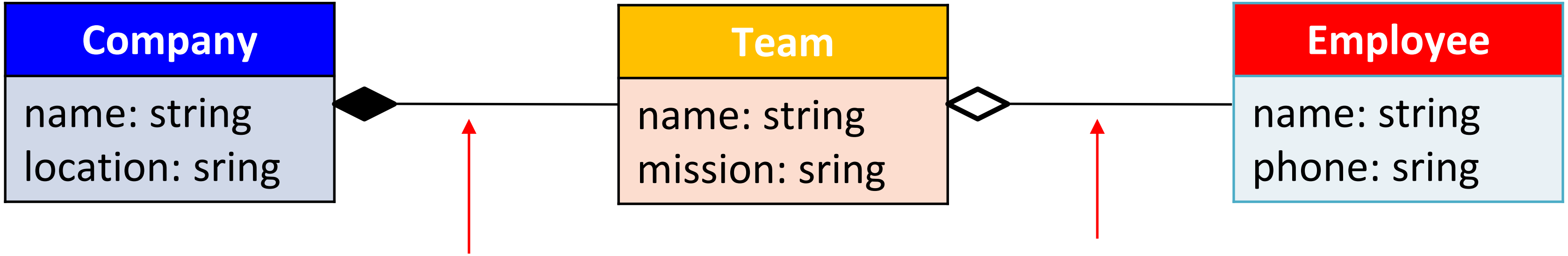
name: string
mission: sring

Employee

name: string
phone: sring

- Suggest:
  - If the company object is deleted, the teams will have no reason to exist anymore.
  - When a team is deleted, the employees that were in the team, still exist. Employees might also belong to multiple teams. A team object does not "own" an employee object.
- Answer:

- Answer



Company

name: string
location: sring

Team

name: string
mission: sring

Employee

name: string
phone: sring

If the company object is deleted, the teams will have no reason to exist anymore

When a team is deleted, the employees that were in the team, still exist. Employees might also belong to multiple teams. A team object does not "own" an employee object.

- Question 4: Determine and implement the relationship between the following classes in Python

Mechanic
name: string
phone: sring

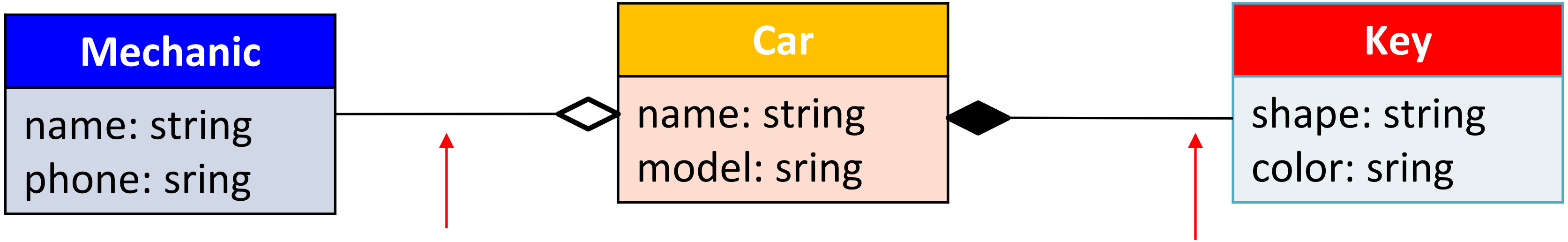
Car
name: string
model: sring

Key
shape: string
color: sring

- Suggest:
  - A car needs a mechanic to work on it. A mechanic can also work on other cars at the same time. If a car object is deleted, the mechanic keeps working at the factory
  - If the car object is deleted from the system, the key is useless and mustbe deleted also.
- Answer:

Exercises Chapter 3

- Question 4: Determine and implement the relationship between the following classes in Python
- Answer



Mechanic

name: string
phone: sring

Car

name: string
model: sring

Key

shape: string
color: sring

A mechanic can also work on other cars at the same time. If a car object is deleted, the mechanic keeps working at the factory

If the car object is deleted from the system, the key is useless and mustbe deleted also.

- Question 5: Using object-oriented programming technique to finish the questions from 43 to 55 in exercises of Chapter 2

Exercises Chapter 3

- https://v1study.com/python-bai-tap-bai-tap-phan-class.html
- https://v1study.com/python-bai-tap-bai-tap-phan-thua-ke.html

The end of Chapter