AI VIETNAM
All-in-One Course

Object-Oriented Programming

Nguyen Dinh Vinh
Ph.D in Computer Science

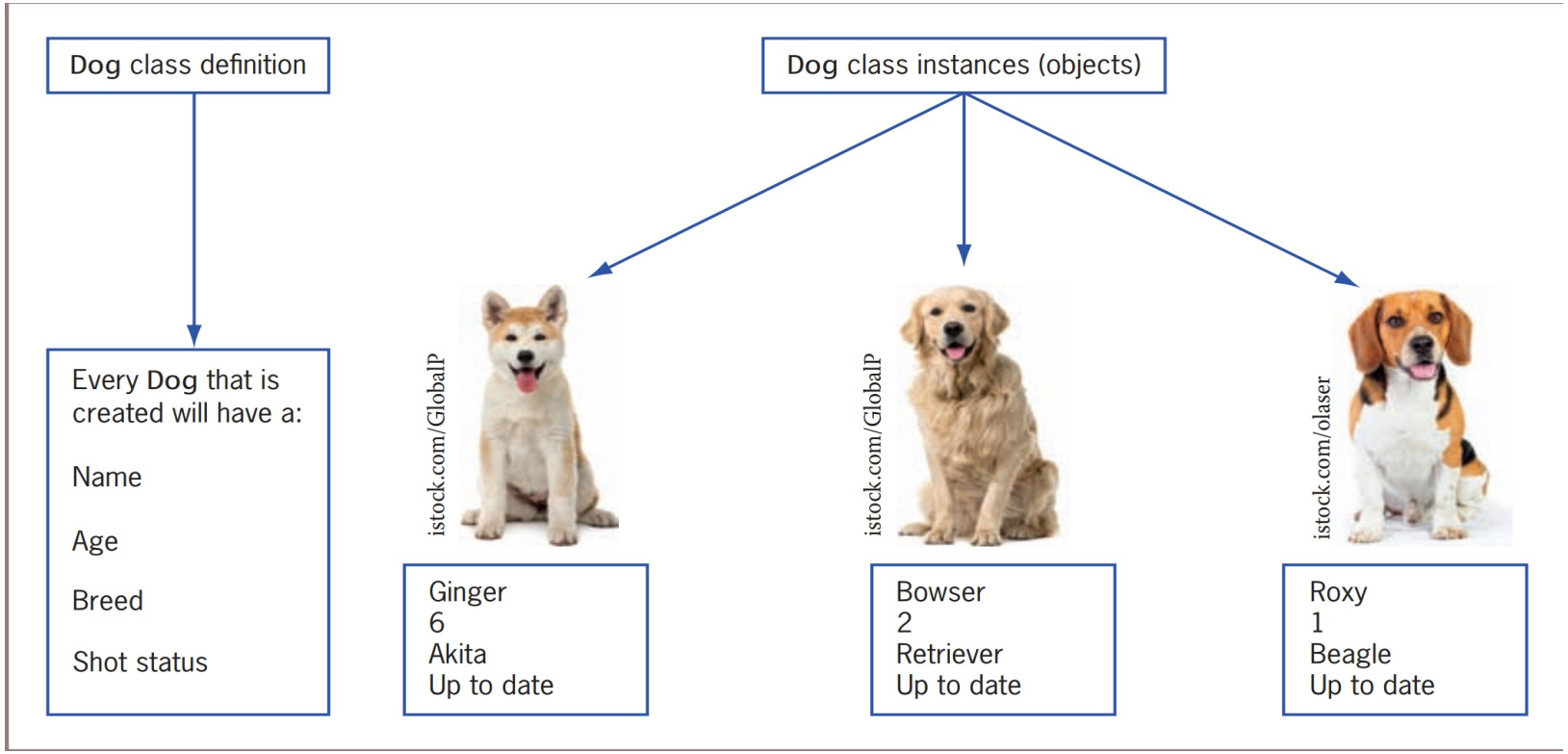
Year 2023

Outline

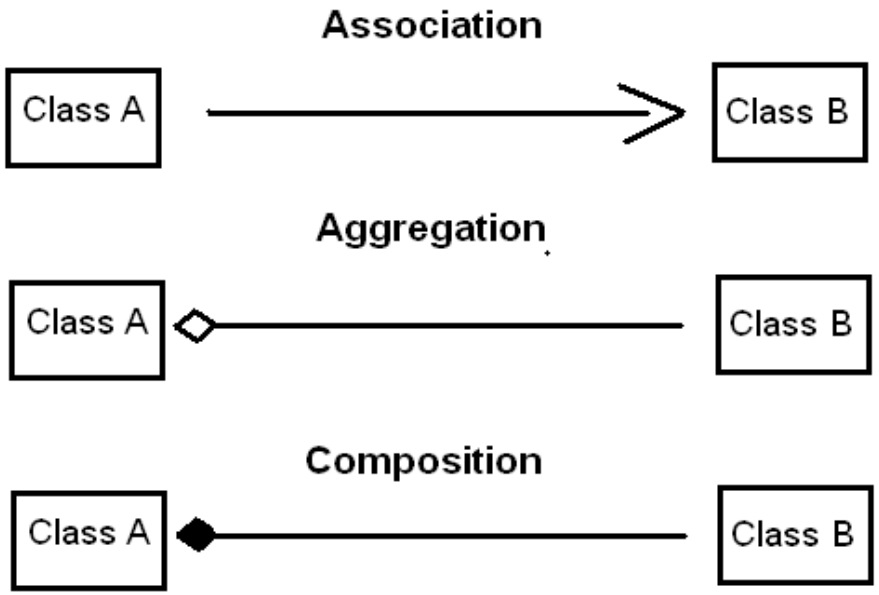
- Object Relationships (Inheritance)
- Access Modifier
- Module and Package
- Case study
- Exercises

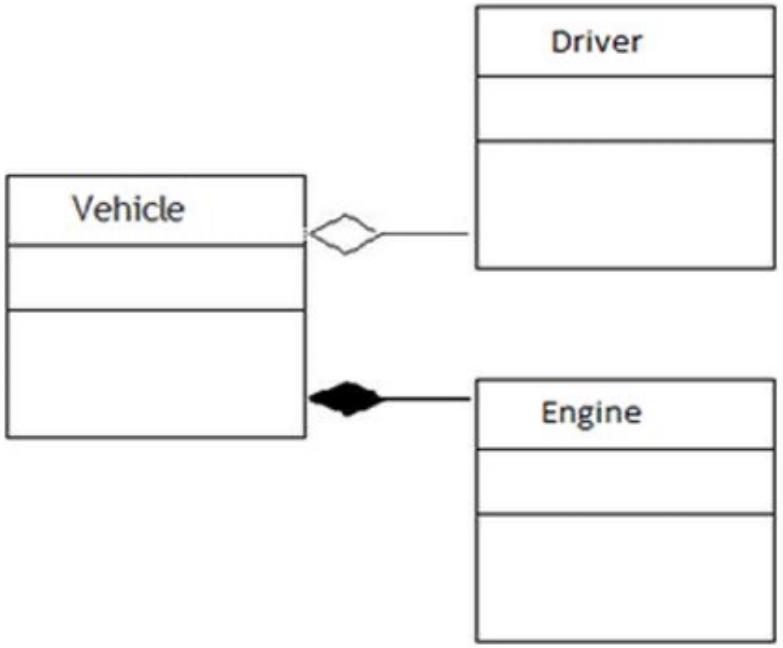
Review

➢ Classes and Objects



Object Relationships





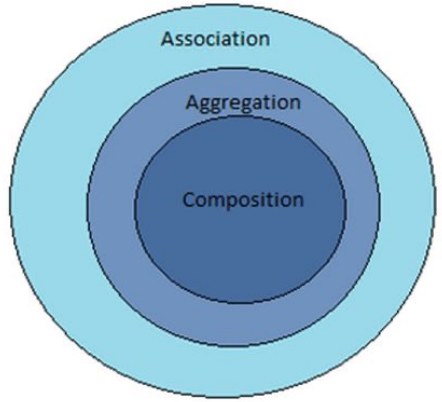


Image Credit:https://softwareengineering.stackexchange.com/

Object Relationships

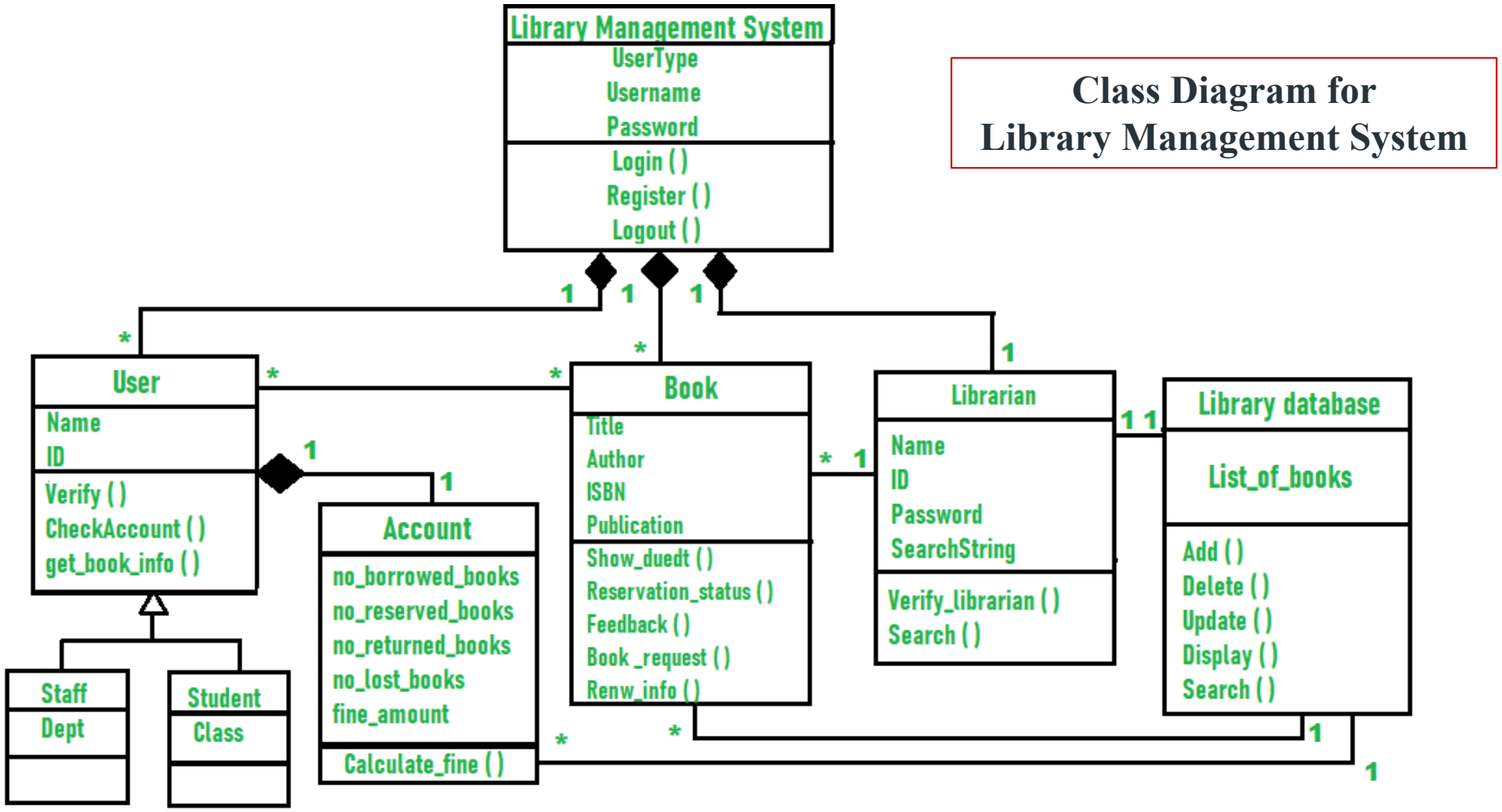


Image credits: https://www.geeksforgeeks.org/

Outline

- Object Relationships (Inheritance)
- Access Modifier
- Module and Package
- Case study
- Exercises

Why do we need Inheritance

Example

- Develop OOP-based program to store information of various types of animals.
  - Dog: name, breed, owner
  - Cat: name, breed, skin color, owner
  - Lion: name, breed, and weight

Solution 1

Animal
+ id : string + name : string + breed: string + skinColor: string + weight: float + owner: string
+ infor(): void

Limitations

```
class Animal:
  def __init__(self, type, name, breed, skinColor, weight, owner):
    self.type = type
    self.name = name
    self.breed = breed
    self.skinColor = skinColor
    self.weight = weight
    self.owner = owner

  def infor(self):

    print('{} has informaton: (name, {}), (breed: {}), (skin color: {}), (weight: {}), (owner: {})'.
          format(self.type, self.name, self.breed, self.skinColor, self.weight, self.owner))
```

```
dog = Animal("Dog", "Nini", "Phu Quoc", "N/A", "N/A",  "Nguyen Van A")
cat = Animal("Cat", "Kitty", "Canada", "Brown", "N/A", "Nguyen Van B")
lion = Animal("Lion", "Simba", "US", "N/A",  100, "N/A")

dog.infor()
cat.infor()
lion.infor()
```

```
Dog has informaton: (name, Nini), (breed: Phu Quoc), (skin color: N/A), (weight: N/A), (owner: Nguyen Van A)
Cat has informaton: (name, Kitty), (breed: Canada), (skin color: Brown), (weight: N/A), (owner: Nguyen Van B)
Lion has informaton: (name, Simba), (breed: US), (skin color: N/A), (weight: 100), (owner: N/A)
```

Solution 2

Limitations

Dog

+ name : string
+ breed: string
+ owner: string
+ infor(): void

Cat

+ name : string
+ breed: string
+ skinColor: string
+ owner: string
+ infor(): void

Lion

+ name : string
+ breed: string
+ weight: float
+ infor(): void

```
class Dog:
  def __init__(self, name, breed, owner):
    self.name = name
    self.breed = breed
    self.owner = owner

  def infor(self):
    print('Dog has informaton: (name, {}), (breed: {}), (owner: {})'.format( self.name, self.breed,  self.owner))

class Cat:
  def __init__(self, name, breed, skinColor, owner):
    self.name = name
    self.breed = breed
    self.skinColor = skinColor
    self.owner = owner

  def infor(self):
    print('Cat has informaton: (name, {}), (breed: {}), (skin color: {}) (owner: {})'.format( self.name, self.breed,  self.skinColor, self.owner))

class Lion:
  def __init__(self, name, breed, weight):
    self.name = name
    self.breed = breed
    self.weight = weight

  def infor(self):
    print('Lion has informaton: (name, {}), (breed: {}), (weight: {})'.format( self.name, self.breed,  self.weight))
```

```
dog = Dog("Nini", "Phu Quoc", "Nguyen Van A")
cat = Cat("Kitty", "Canada", "Brown", "Nguyen Van B")
lion = Lion("Simba", "US",100)

dog.infor()
cat.infor()
lion.infor()
```

```
Dog has informaton: (name, Nini), (breed: Phu Quoc), (owner: Nguyen Van A)
Cat has informaton: (name, Kitty), (breed: Canada), (skin color: Brown) (owner: Nguyen Van B)
Lion has informaton: (name, Simba), (breed: US), (weight: 100)
```

Dog	Cat	Lion	...	Monkey
- name : string - breed: string - owner: string	- name : string - breed: string - skinColor: string - owner: string	- name : string - breed: string - weight: float		- name : string - breed: string - height: float
+ infor(): void	+ infor(): void	+ infor(): void		+ infor(): void

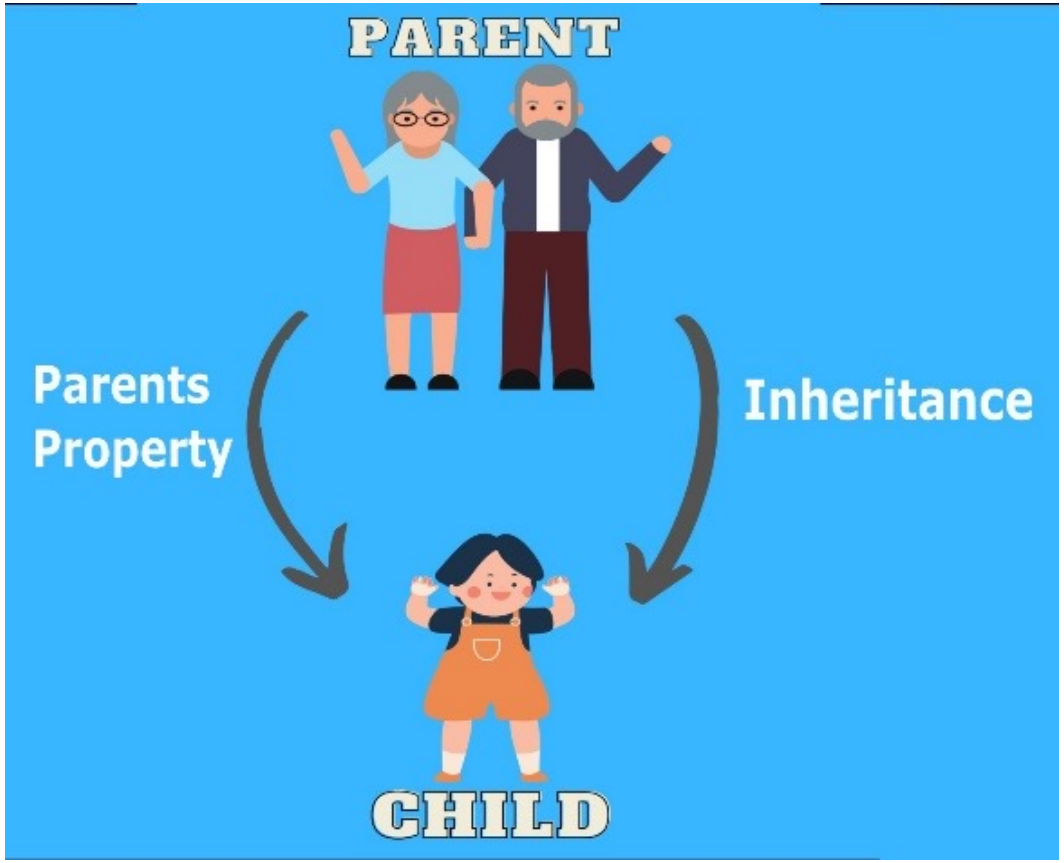
LIMITATIONS

Oh No, new terminology again?

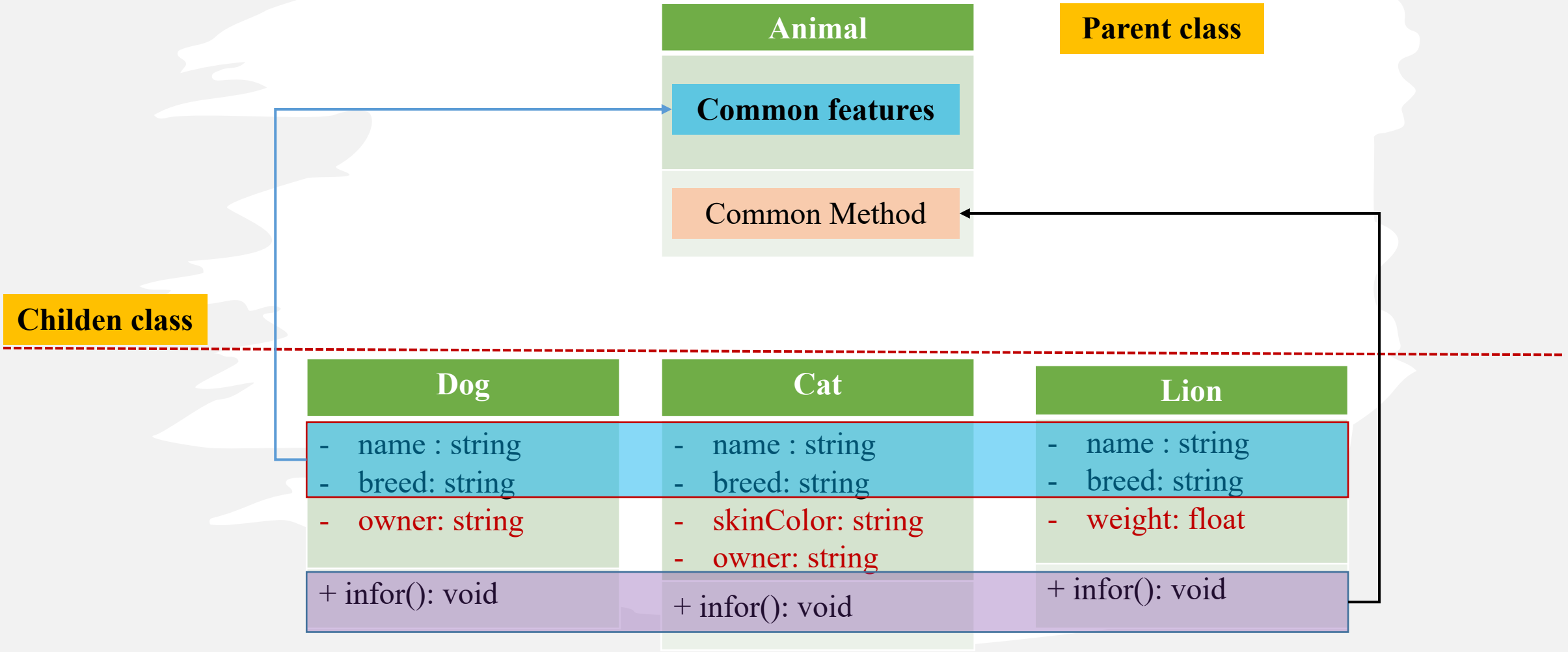
Inheritance in OOP

Reusable in OOP

Inheritance in Real life



Inheritance in OOP



Inheritance in OOP

Animal
- name : string - breed: string
+ infor(): void

Parent class

Childen class

Dog
- owner: string
+ infor(): void

Cat
- skinColor: string - owner: string
+ infor(): void

Lion
- weight: float
+ infor(): void

Inheritance in OOP

Inheritance notaion

Animal

- name : string
- breed: string

+ infor(): void

Parent class

Childen class

Dog

- owner: string

+ infor(): void

Cat

- skinColor: string
- owner: string

+ infor(): void

Lion

- weight: float

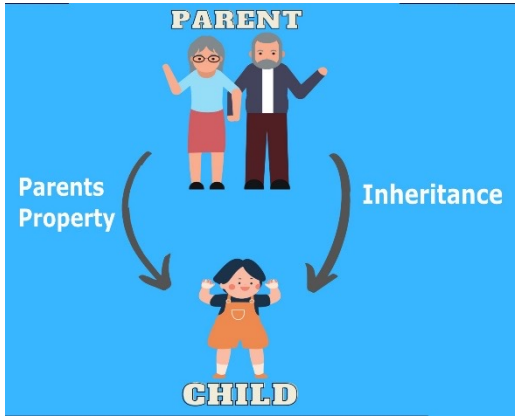
+ infor(): void

Why?

Inheritance

❖ Introduction

Mechanism by which one class is allowed to inherit the features (attributes and methods) of another class.



Super Class: The class whose features are inherited is known as superclass (a base class or a parent class).

Subclass: The class that inherits the other class is known as subclass (a derived class, extended class, or child class).

The subclass can add its own attributes and methods in addition to the superclass attributes and methods.

Inheritance

❖ Introduction

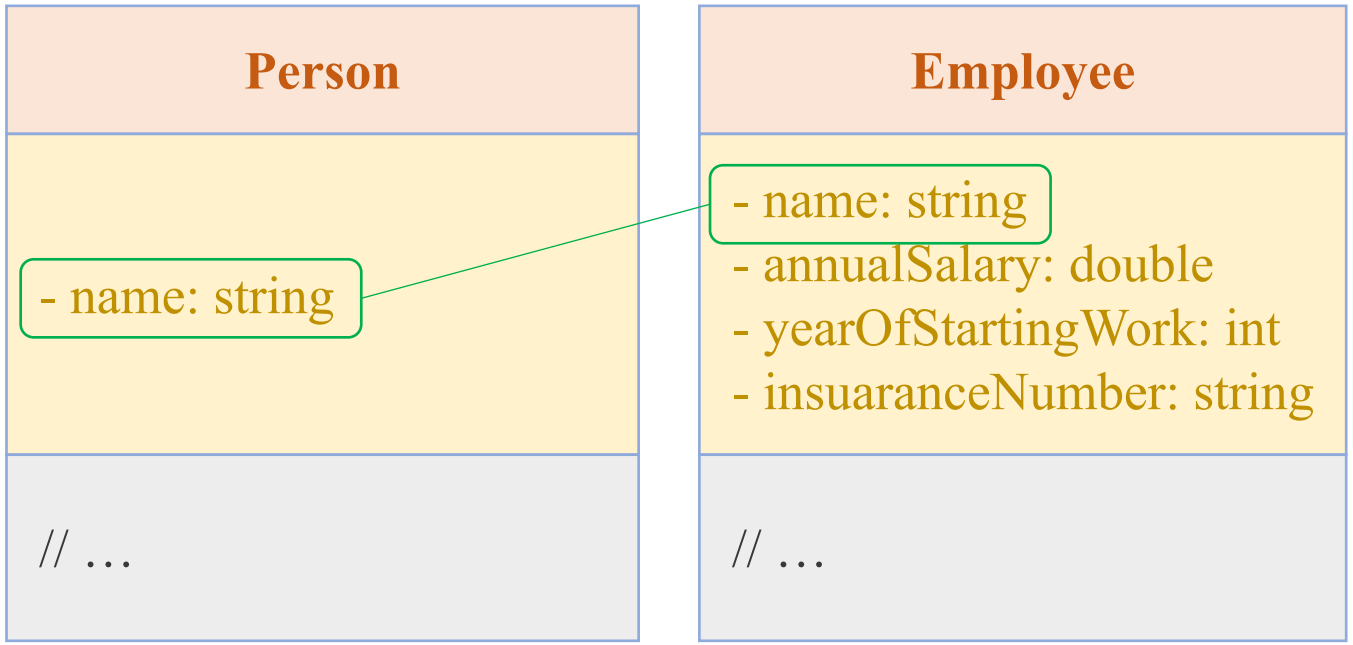
Create a class called Employee whose objects are records for an employee. This class will be a derived class of the class Person.

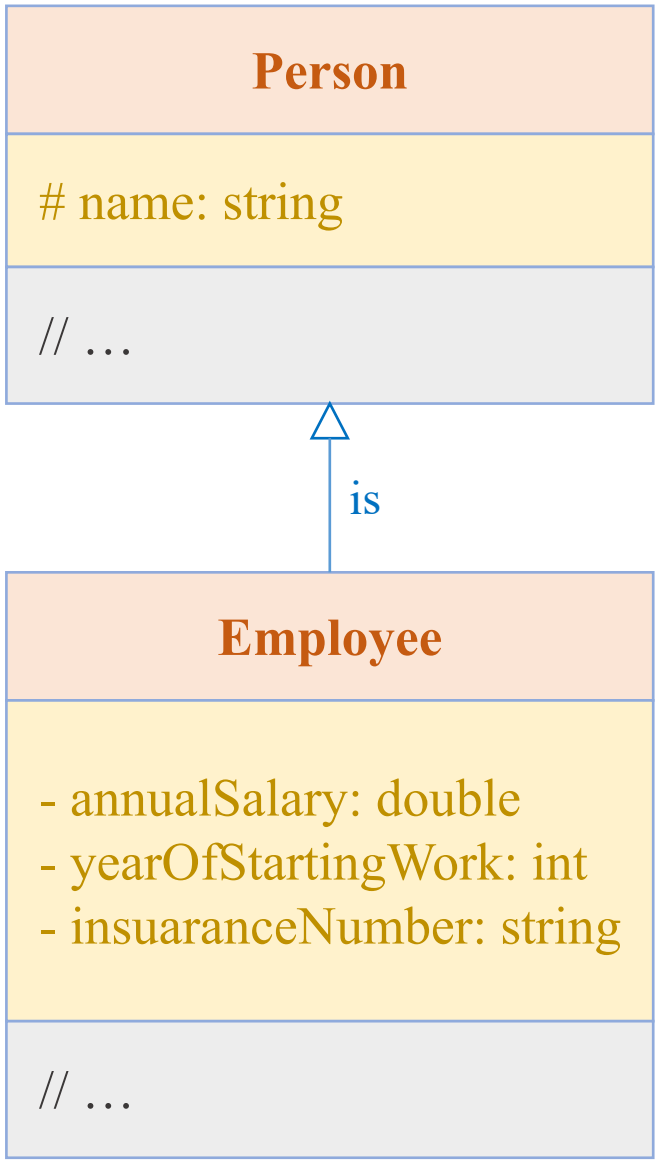
An employee record has an employee's name (inherited from the class Person), an annual salary represented as a single value of type double, a year the employee started work as a single value of type int and a national insurance number, which is a value of type String.

Person
- name: string
// …

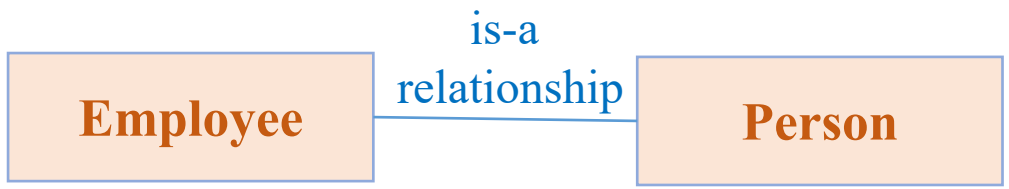
Employee
- name: string - annualSalary: double - yearOfStartingWork: int - insuaranceNumber: string
// …

Inheritance





Access modifiers
- private
+ public
# protected



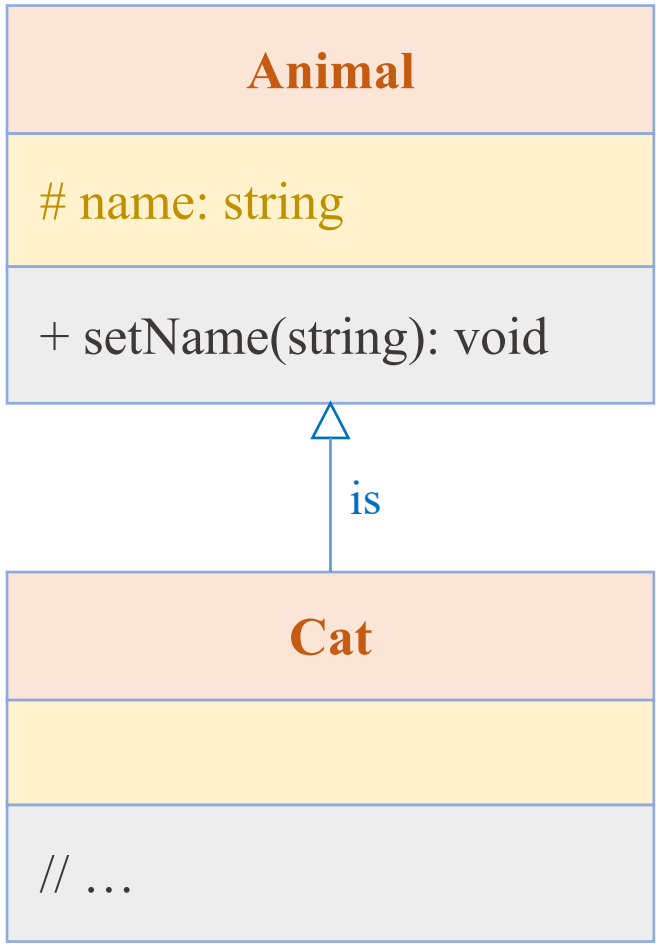
An employee is a person.

AI VIETNAM
All-in-One Course

Inheritance

Inherit attributes and methods from one class to another

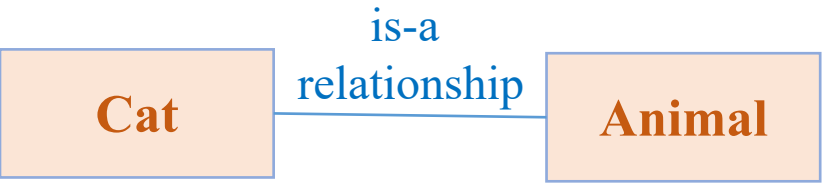
Benefit: Code reusability



Derived class (child) - the class that inherits from another class

Base class (parent) - the class being inherited from

DerivedClass(BaseClass)



A cat is an animal.

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

    def setName(self, name):
        self._name = name

    def describe(self):
        print(f'Name: {self._name}')

class Cat(Animal):
    def __init__(self, name):
        Animal.__init__(self, name)

cat = Cat('Calico')
cat.describe()
```

```
Name: Calico
```

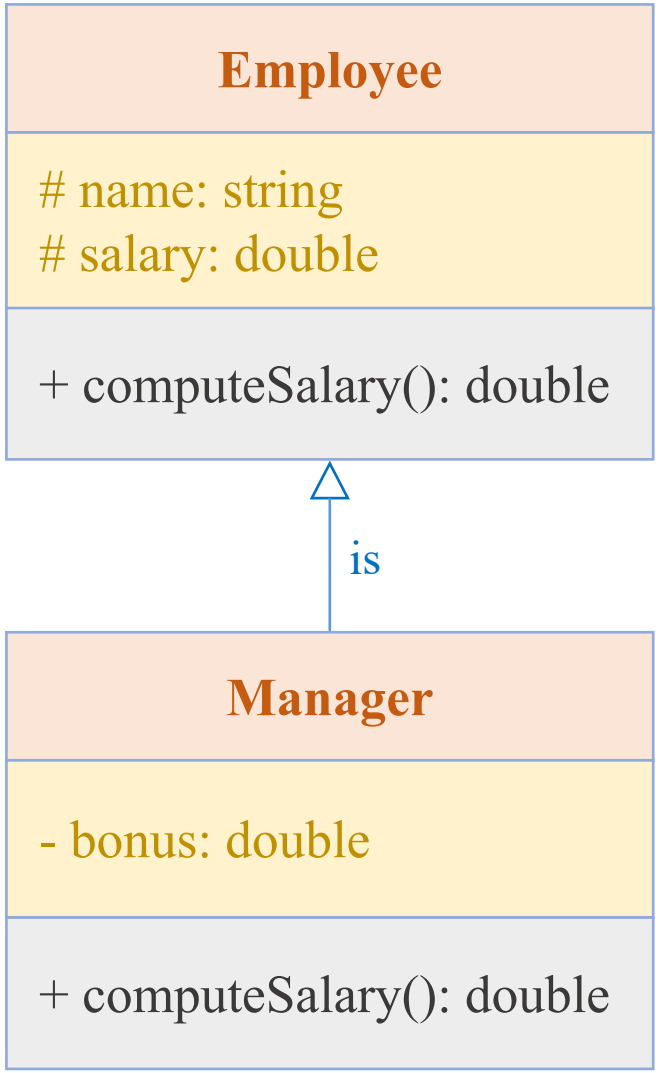
Inheritance

❖ Introduction

To extend an existing class

UML Annotation
'-' stands for 'private'
'#' stands for 'protected'
'+' stands for 'public'

Super Class



Subclass

What features does a manager inherit?

Inheritance

❖ As a template

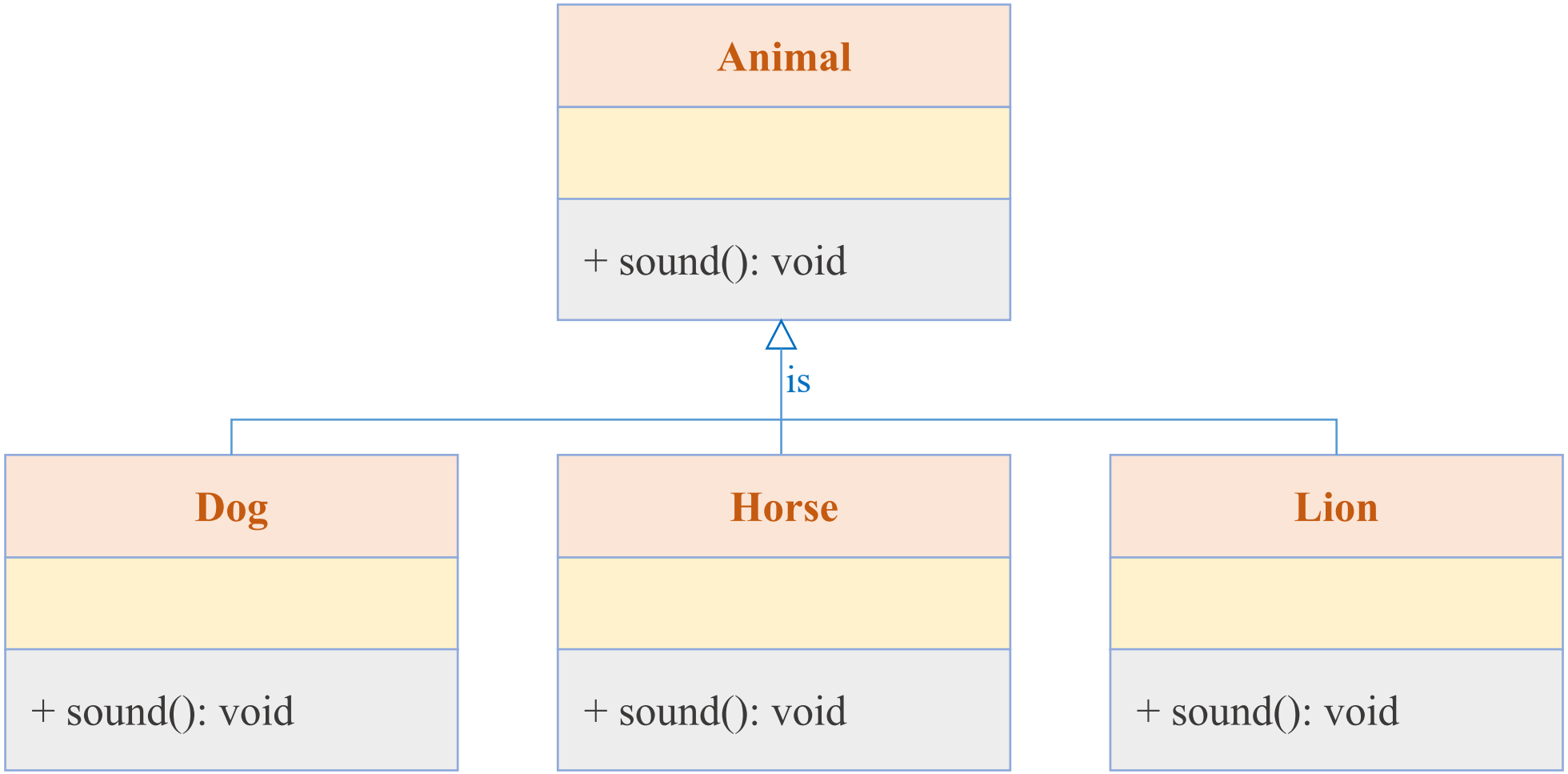
A class Animal that has a method sound() and the subclasses of it like Dog, Lion, Horse, Cat, etc.

Since the animal sound differs from one animal to another, there is no point to implement this method in parent class.

This is because every child class must override this method to give its own implementation details, like Lion class will say “Roar” in this method and Dog class will say “Woof”.

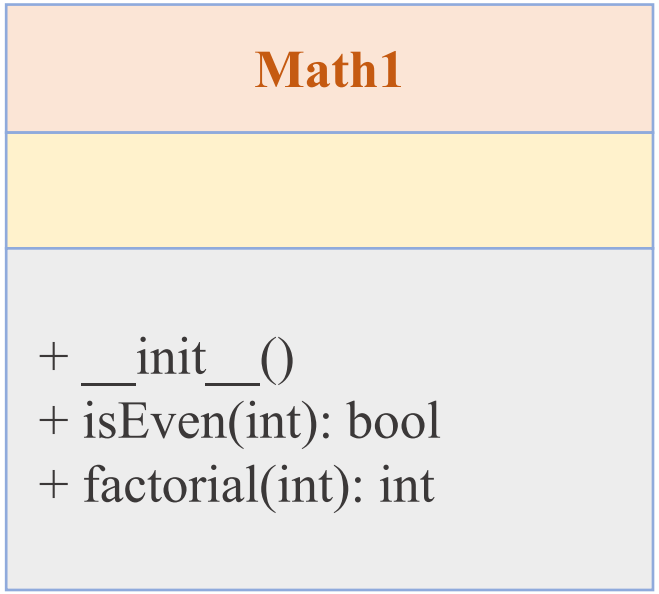
Inheritance

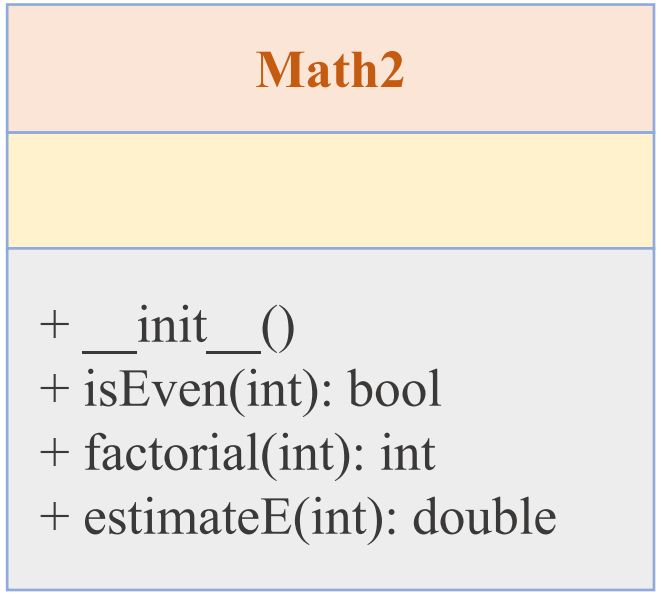
❖ As a template



Example

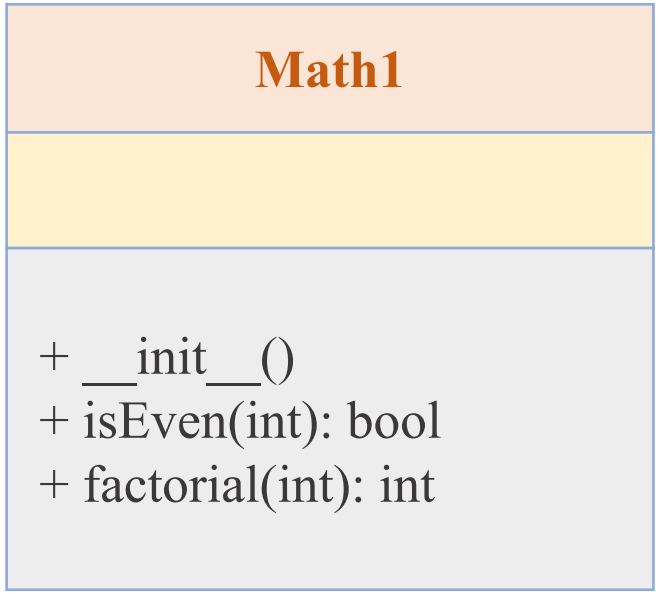
❖ **Implement the two classes below**





Example

Implement the two classes below




```python
class Math1:
    def isEven(self, number):
        if number%2:
            return False
        else:
            return True

    def factorial(self, number):
        result = 1

        for i in range(1, number+1):
            result = result*i

        return result
```

```python
# test Math1
math1 = Math1()

# isEven() sample: number=5 -> False
# isEven() sample: number=6 -> True
print(math1.isEven(5))
print(math1.isEven(6))

# factorial() sample: number=4 -> 24
# factorial() sample: number=5 -> 120
print(math1.factorial(4))
print(math1.factorial(5))
```

```
False
True
24
120
```

AI VIETNAM
All-in-One Course

Example

Implement the two classes below

$$e = 2.71828$$

$$e \approx 1 + \frac{1}{1!} + \frac{1}{2!} + \ldots + \frac{1}{n!}$$

Math2
+ __init__() + isEven(int): bool + factorial(int): int + estimateEuler(int): double

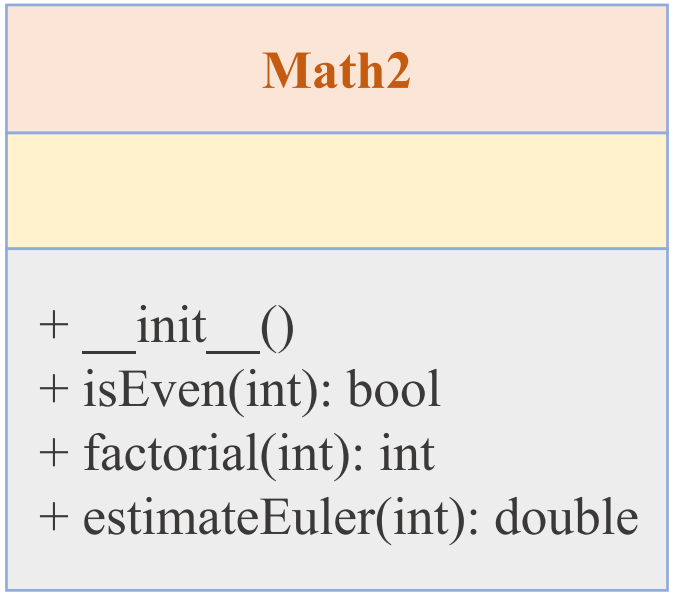
```python
class Math2:
    def isEven(self, number):
        if number%2:
            return False
        else:
            return True

    def factorial(self, number):
        result = 1

        for i in range(1, number+1):
            result = result*i

        return result

    def estimateEuler(self, number):
        result = 1

        for i in range(1, number+1):
            result = result + 1/self.factorial(i)

        return result
```

Example

Implement the two classes below

$$e = 2.71828$$

$$e \approx 1 + \frac{1}{1!} + \frac{1}{2!} + \ldots + \frac{1}{n!}$$




```python
# test Math2
math2 = Math2()

# isEven() sample: number=5 -> False
# isEven() sample: number=6 -> True
print(math2.isEven(5))
print(math2.isEven(6))

# factorial() sample: number=4 -> 24
# factorial() sample: number=5 -> 120
print(math2.factorial(4))
print(math2.factorial(5))

# estimateEuler() sample: number=2 -> 2.5
# estimateEuler() sample: number=8 -> 2.71
print(math2.estimateEuler(2))
print(math2.estimateEuler(8))
```

```
False
True
24
120
2.5
2.71827876984127
```

AI VIETNAM
All-in-One Course

Example

How to reuse an existing class?

Math1

+ __init__()
+ isEven(int): bool
+ factorial(int): int

Math2

+ __init__()
+ isEven(int): bool
+ factorial(int): int
+ estimateEuler(int): double

```python
class Math1:
    def isEven(self, number):
        if number%2:
            return False
        else:
            return True

    def factorial(self, number):
        result = 1

        for i in range(1, number+1):
            result = result*i

        return result
```

```python
class Math2:
    def isEven(self, number):
        if number%2:
            return False
        else:
            return True

    def factorial(self, number):
        result = 1

        for i in range(1, number+1):
            result = result*i

        return result

    def estimateEuler(self, number):
        result = 1

        for i in range(1, number+1):
            result = result + 1/self.factorial(i)

        return result
```

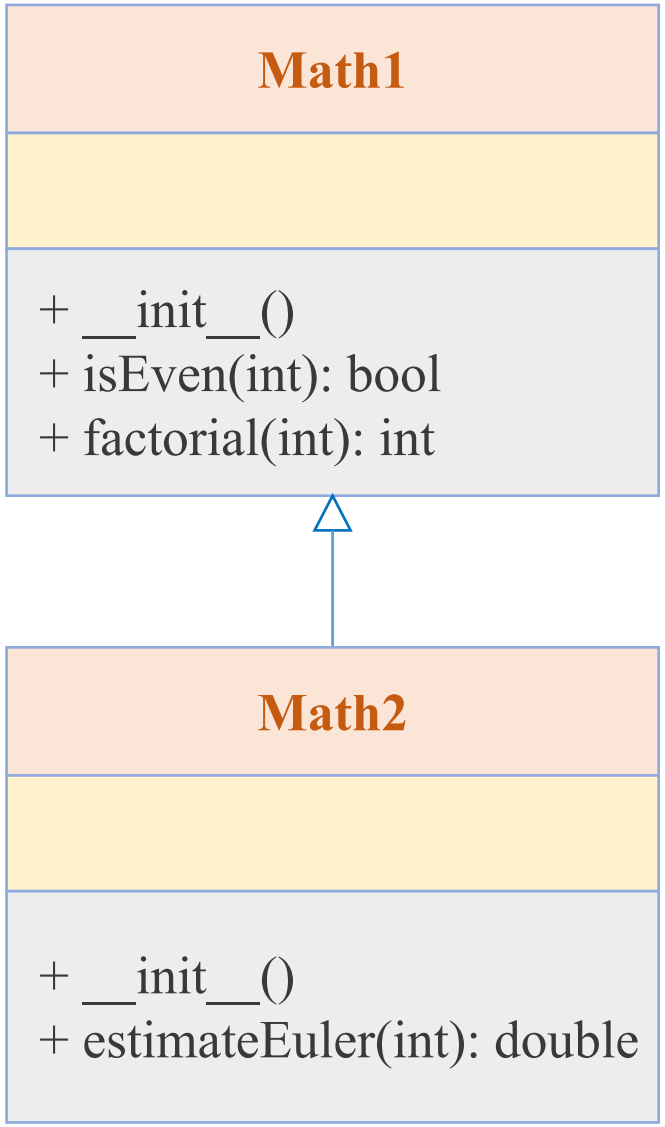
Example

❖ Inheritance

Math1: super class or parent class

Math2: child class or derived class

Child classes can use the public and protected attributes and methods of the super classes.



```python
class Math1:
    def isEven(self, number):
        if number%2:
            return False
        else:
            return True

    def factorial(self, number):
        result = 1

        for i in range(1, number+1):
            result = result*i

        return result
```

```python
class Math2(Math1):
    def estimateEuler(self, number):
        result = 1

        for i in range(1, number+1):
            result = result + 1/self.factorial(i)

        return result
```

```python
# test Math2
math2 = Math2()

# isEven() sample: number=5 -> False
# isEven() sample: number=6 -> True
print(math2.isEven(5))
print(math2.isEven(6))

# factorial() sample: number=4 -> 24
# factorial() sample: number=5 -> 120
print(math2.factorial(4))
print(math2.factorial(5))

# estimateEuler() sample: number=2 -> 2.5
# estimateEuler() sample: number=8 -> 2.71
print(math2.estimateEuler(2))
print(math2.estimateEuler(8))
```

```
False
True
24
120
2.5
2.71827876984127
```

```python
class Math1:
    def isEven(self, number):
        if number%2:
            return False
        else:
            return True

    def factorial(self, number):
        result = 1

        for i in range(1, number+1):
            result = result*i

        return result
```

```python
class Math2(Math1):
    def estimateEuler(self, number):
        result = 1

        for i in range(1, number+1):
            result = result + 1/super().factorial(i)

        return result
```

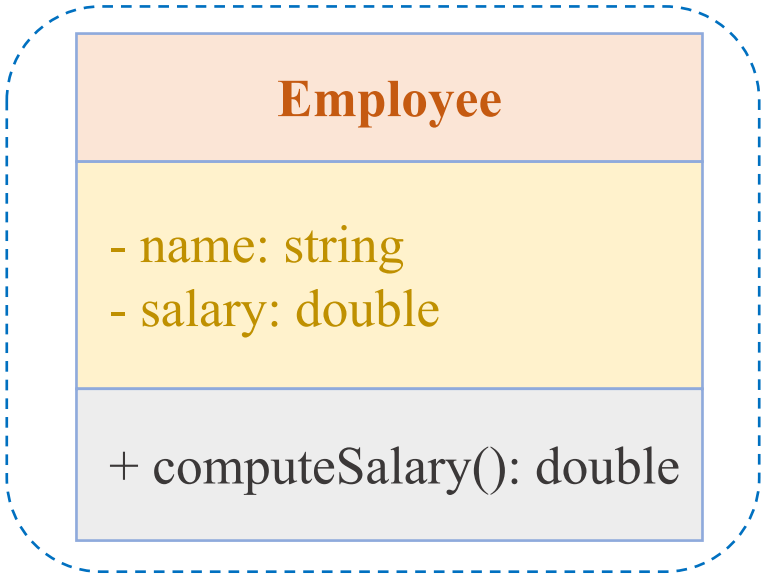
```python
# test Math2
math2 = Math2()

# isEven() sample: number=5 -> False
# isEven() sample: number=6 -> True
print(math2.isEven(5))
print(math2.isEven(6))

# factorial() sample: number=4 -> 24
# factorial() sample: number=5 -> 120
print(math2.factorial(4))
print(math2.factorial(5))

# estimateEuler() sample: number=2 -> 2.5
# estimateEuler() sample: number=8 -> 2.71
print(math2.estimateEuler(2))
print(math2.estimateEuler(8))
```

```
False
True
24
120
2.5
2.71827876984127
```

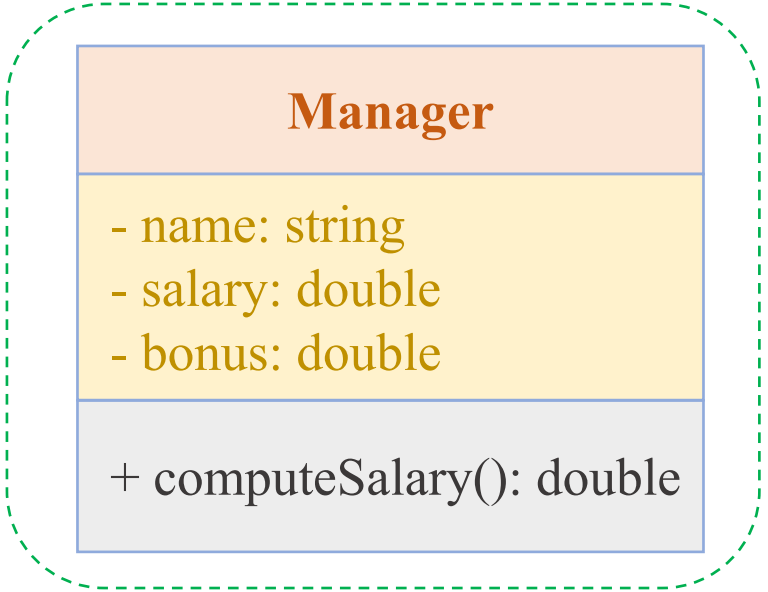
Another Example

Employee-Manager Example: Simple requirement

A standard employee of company X includes his/her name and base salary. For example, Peter is working for X, and his base salary is 60000$ a year. Implement the Employee class and the computeSalary() method to compute the final salary for an employee. The salary for an employee is his/her base salary.



A manager includes his/her name, base salary, and bonus. The final salary for the manager comprises the base salary and a bonus. For example, Mary is a manager in the company. Her base salary and bonus are 60000$ and 20000$ a year, respectively. Yearly, she gets paid 80000$ a year. Implement the Manager class and the computeSalary() method to compute the final salary.



Another Example

Employee-Manager Example (Using inheritance)

A standard employee of company X includes his/her name and base salary. For example, Peter is working for X, and his base salary is 60000$ a year.

Implement the Employee class and the computeSalary() method to compute the final salary for an employee. The salary for an employee is his/her base salary.

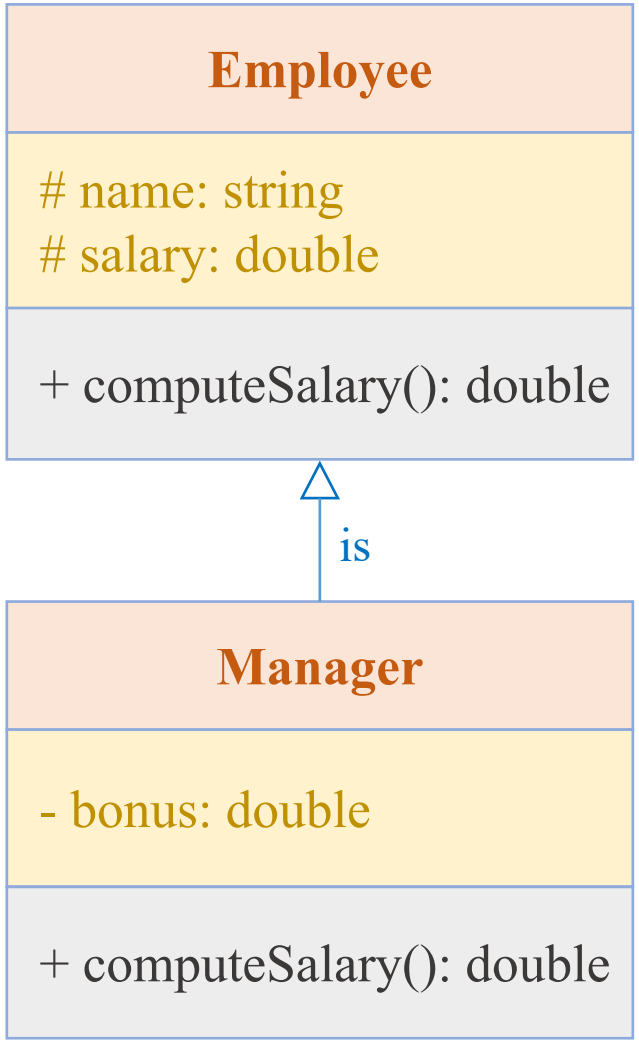
A manager is an employee who has the name and base salary attributes. However, the final salary for the manager comprises the base salary and a bonus.

For example, Mary is a manager in the company. Her base salary and bonus are 60000$ and 20000$ a year, respectively. Yearly, she gets paid 80000$ a year.

Implement the Manager class and the computeSalary() method to compute the final salary.

Another Example

Employee-Manager




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

    def computeSalary(self):
        return self._salary

class Manager(Employee):
    def __init__(self, name, salary, bonus):
        self._name = name
        self._salary = salary
        self.__bonus = bonus

    def computeSalary(self):
        return super().computeSalary() + self.__bonus
```

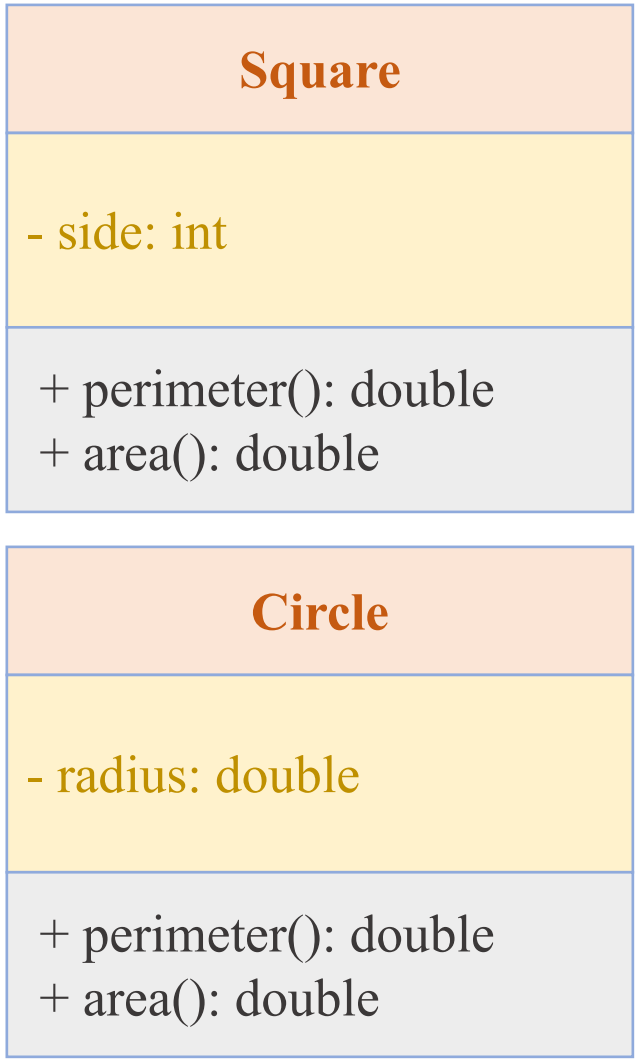
```python
peter = Manager('Peter', 100, 20)
salary = peter.computeSalary()
print(f'Peter Salary: {salary}')
```

```
Peter Salary: 120
```

Example

❖ Inheritance recognition

Squares and circles are both examples of shapes. There are certain questions one can reasonably ask of both a circle and a square (such as, ‘what is the area?’ or ‘what is the perimeter?’) but some questions can be asked only of one or the other but not both (such as, ‘what is the length of a side?’ or ‘what is the radius?’)



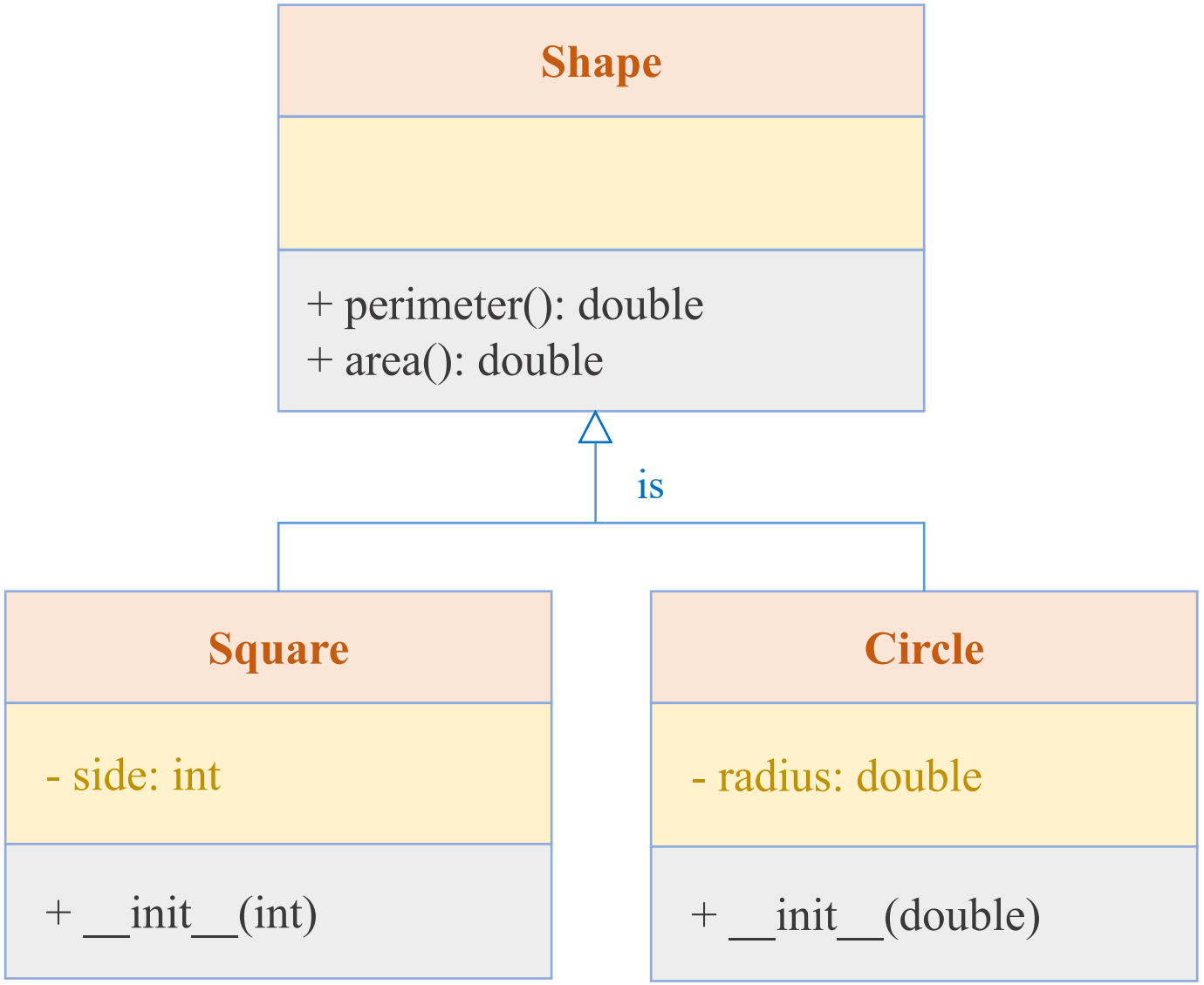
Example

❖ Inheritance recognition

Shape does not know how to compute its perimeter and area

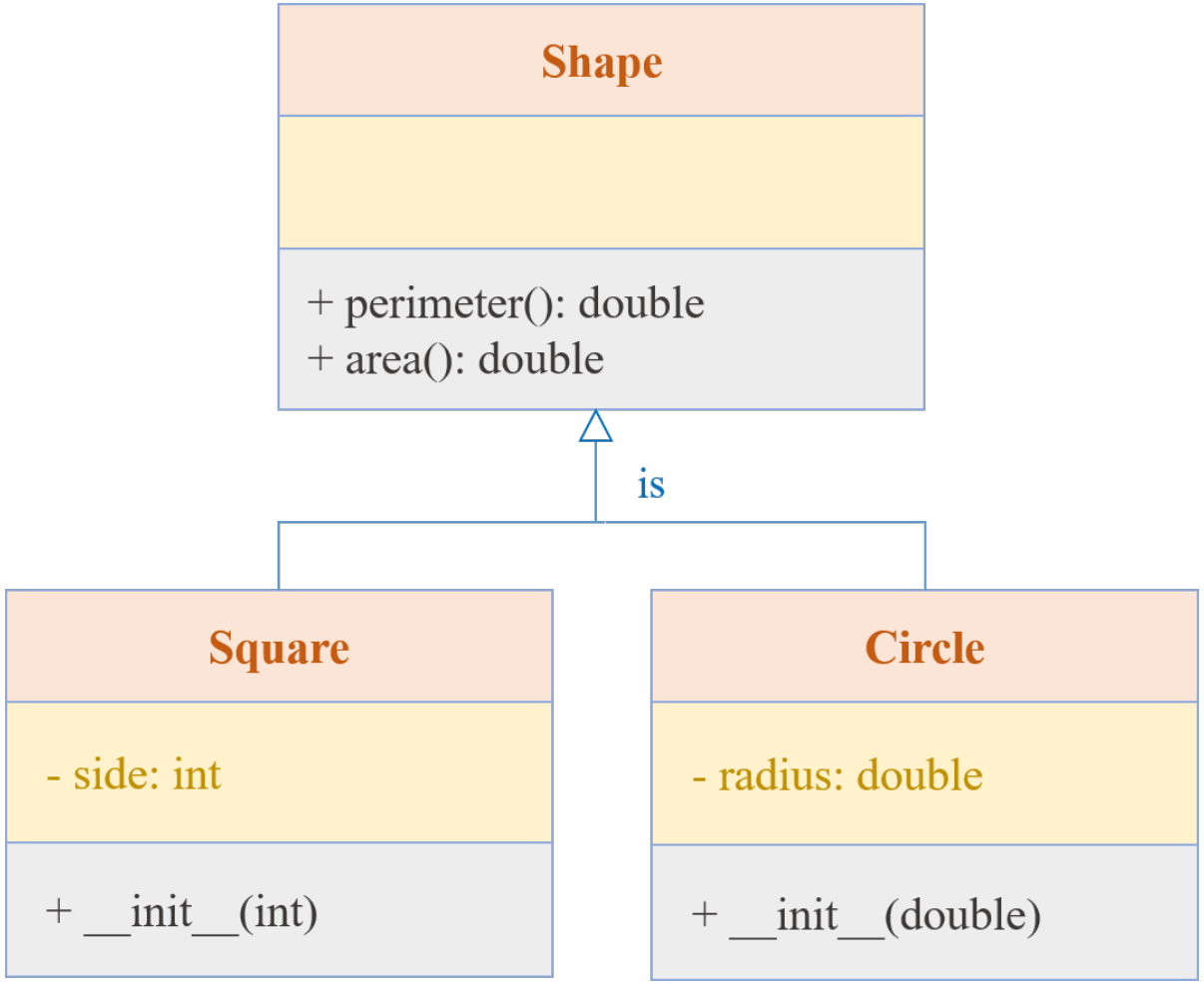
Use @abstractmethod to ask its child to implement them

Using pass in the abstract method



Example

❖ Inheritance recognition



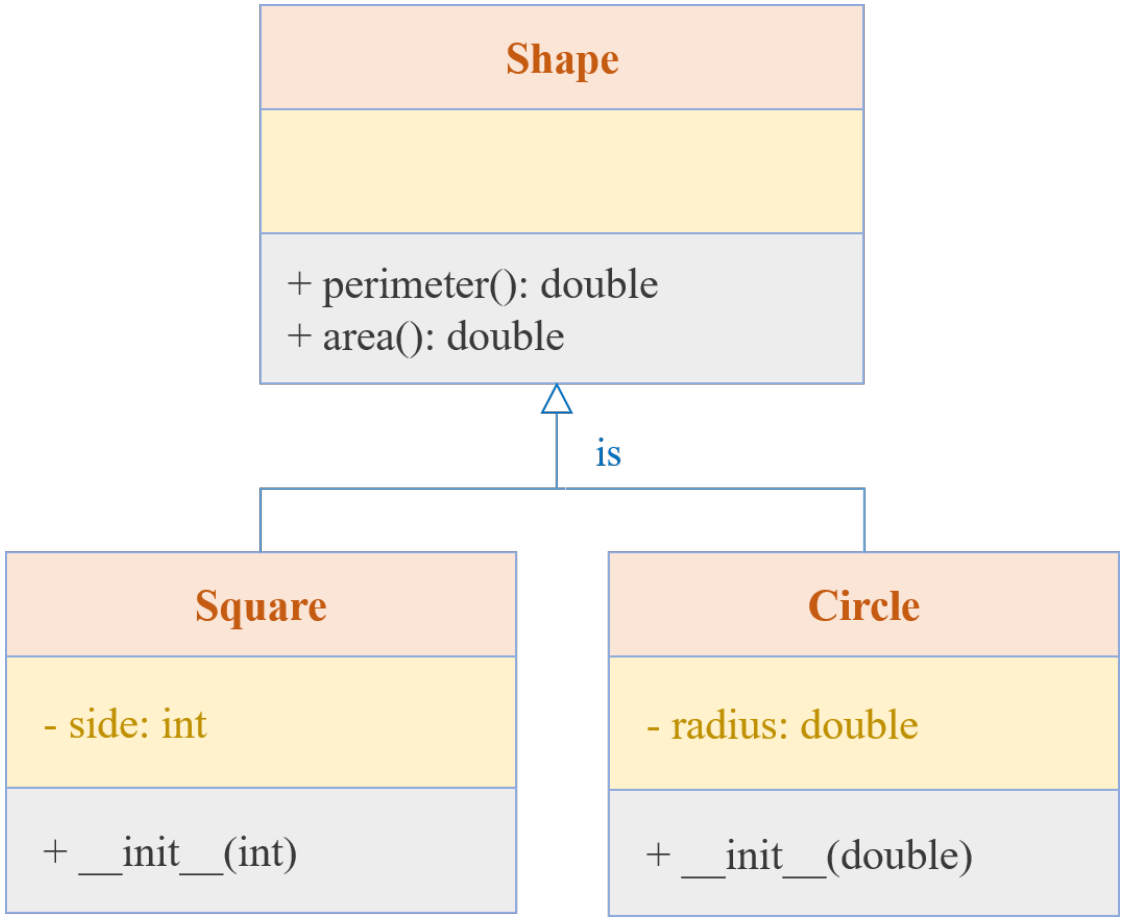

```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def computeArea(self):
        pass
```

```python
class Square(Shape):
    def __init__(self, side):
        self.__side = side

    def computeArea(self):
        return self.__side*self.__side
```

```python
square = Square(5)
print(square.computeArea())
```

Example

❖ Inheritance recognition




```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def computeArea(self):
        pass
```

```python
class Cicle(Shape):
    def __init__(self, radius):
        self.__radius = radius
```

```python
circle = Cicle(5)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [8], in <cell line: 1>()
----> 1 circle = Cicle(5)

TypeError: Can't instantiate abstract class Cicle with abstract method computeArea
```

Example

❖ Inheritance recognition

```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def computeArea(self):
        pass
```

```python
import math

class Cicle(Shape):
    def __init__(self, radius):
        self.__radius = radius

    def computeArea(self):
        return self.__radius*self.__radius*math.pi
```

```python
class Square(Shape):
    def __init__(self, side):
        self.__side = side

    def computeArea(self):
        return self.__side*self.__side
```

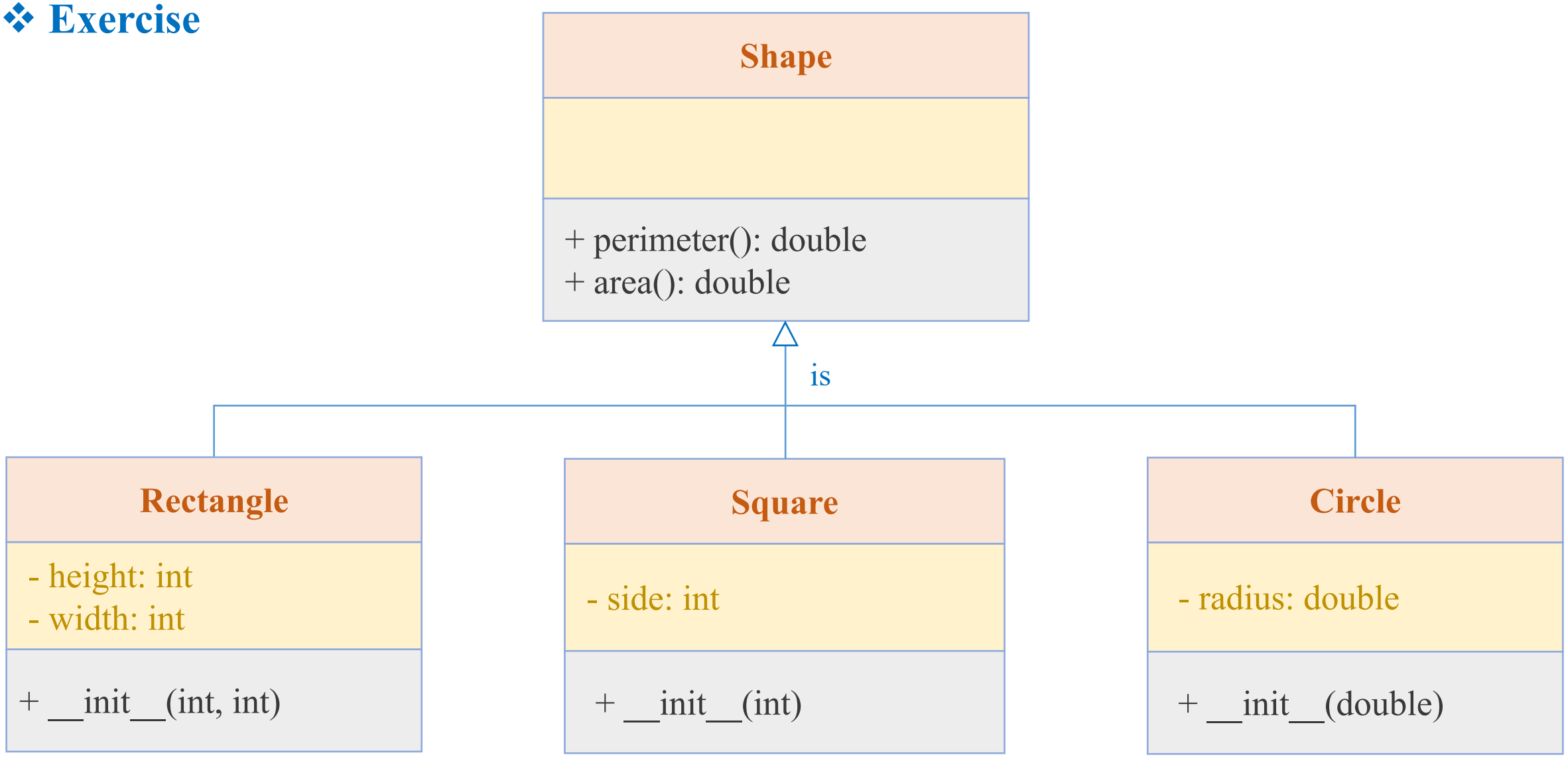
```python
circle = Cicle(5)
print(circle.computeArea())
```

```
78.53981633974483
```

```python
square = Square(5)
print(square.computeArea())
```

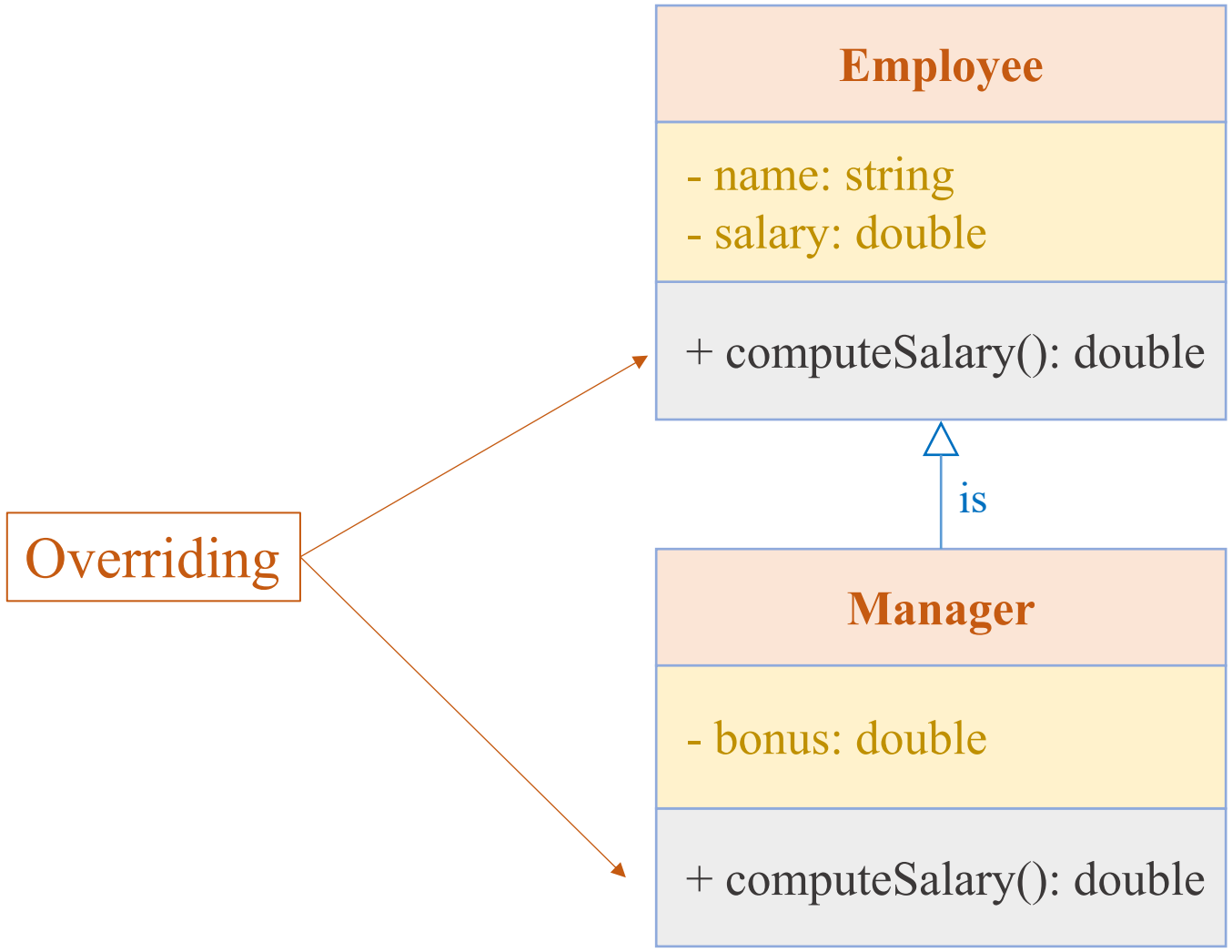
Example

❖ Exercise



Overriding

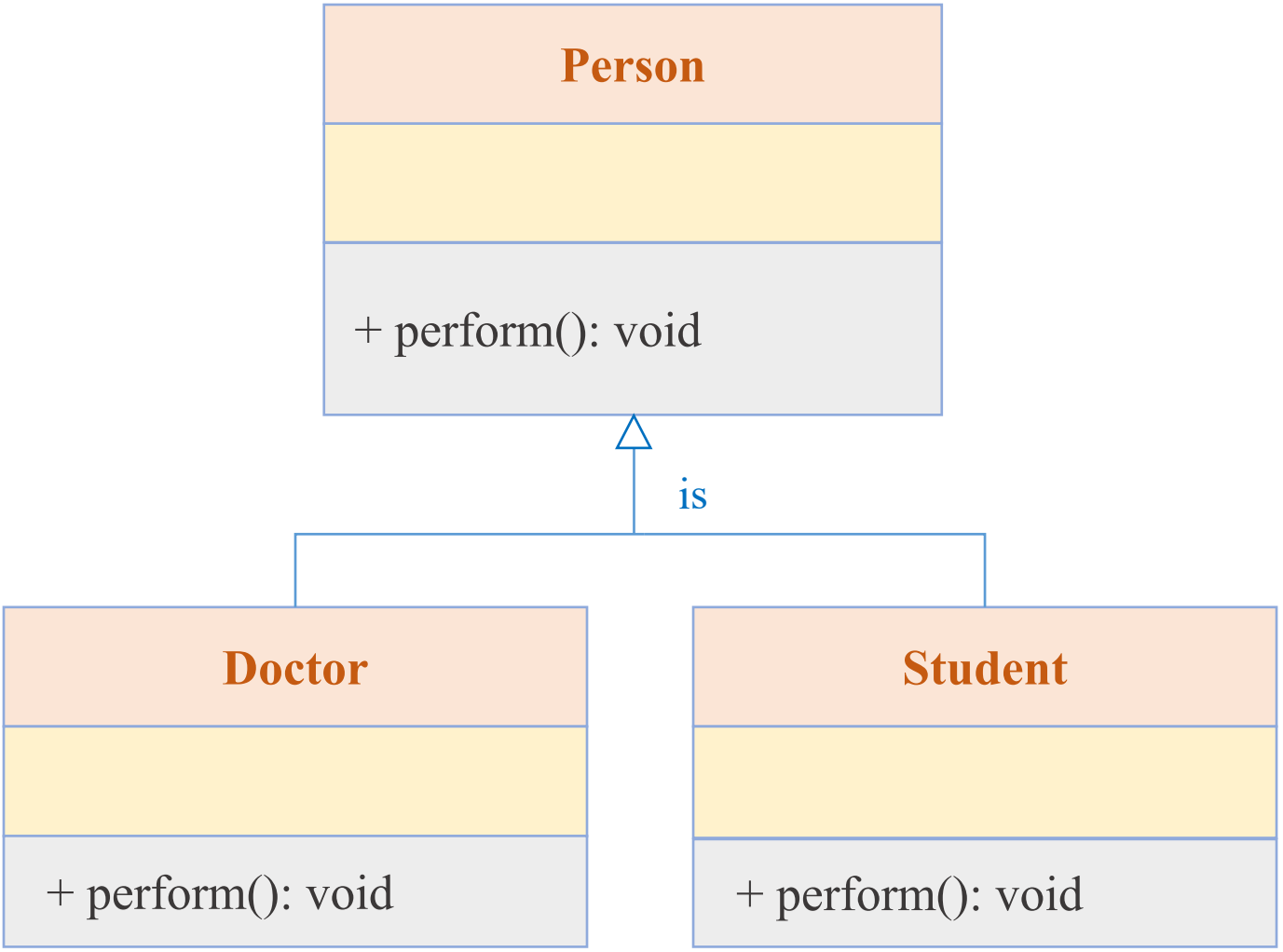
Overriding is a feature that allows a child class to provide a specific implementation of a method that is already provided by its super-class.



AI VIETNAM
All-in-One Course

Overriding

Example




```python
class Person:
    def perform(self):
        print('Person activities')

class Doctor(Person):
    def perform(self):
        print('Doctor activities')

class Student(Person):
    def perform(self):
        print('Student activities')
```

```python
person = Person()
person.perform()
```

```
Person activities
```

```python
doctor = Doctor()
doctor.perform()
```

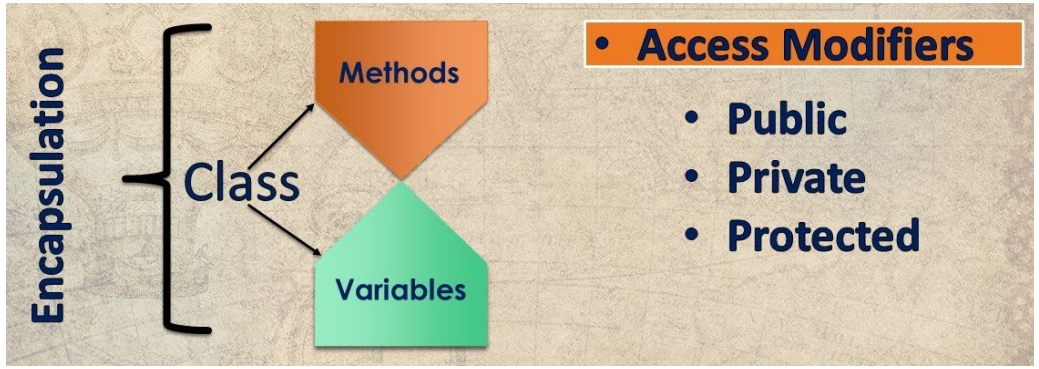
```
Doctor activities
```

Outline

- Object Relationships (Inheritance)
- Access Modifier
- Module and Package
- Case study
- Exercises

Access Modifiers in Python : Public, Private and Protected

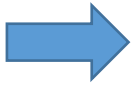
- A Class in Python has three types of access modifiers:
  - ✓Public Access Modifier
  - ✓Protected Access Modifier
  - ✓Private Access Modifier



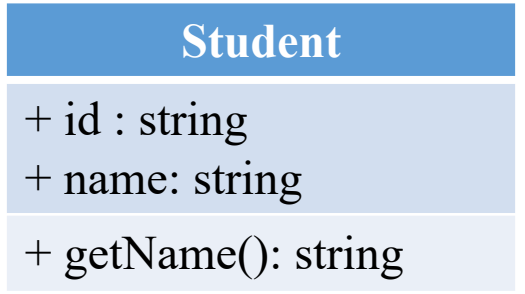
Marker	Visibility	Description
+	Public	All classes can view the information.
-	Private	Information is hidden from all classes outside the partition.
#	Protected	Child class can access the parent class inherited information.

Example
Public access modifier

```
#Declare class
class Student:

    # __init__ is known as the constructor
    def __init__(self, name):
        self.id = id
        self.name = name

    def getName(self):
      return self.name

# Object instantiation
vinh = Student("Nguyen Dinh Vinh")
print("Access public variable: ", vinh.name)
print("Access public method: ", vinh.getName())
```

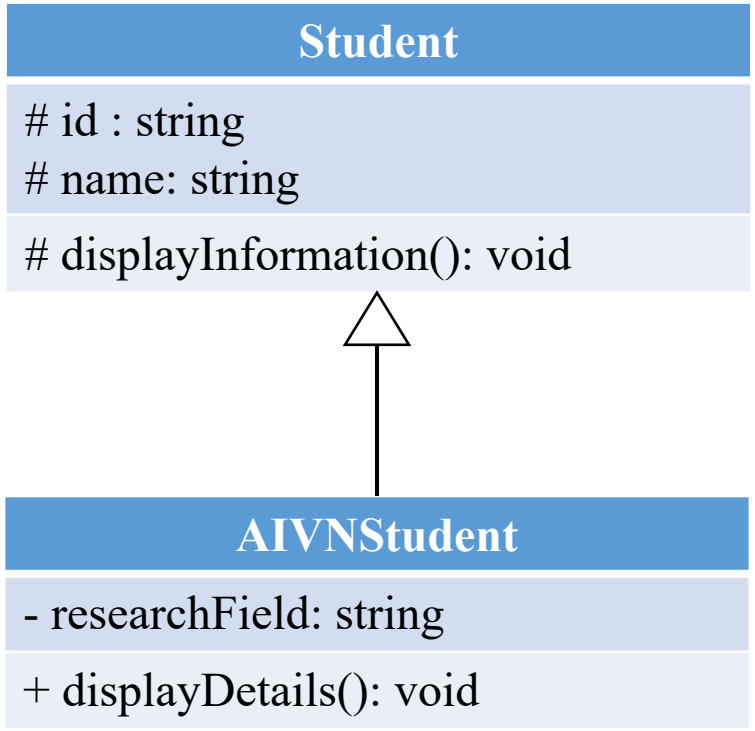


```
Access public variable:  Nguyen Dinh Vinh
Access public method:  Nguyen Dinh Vinh
```



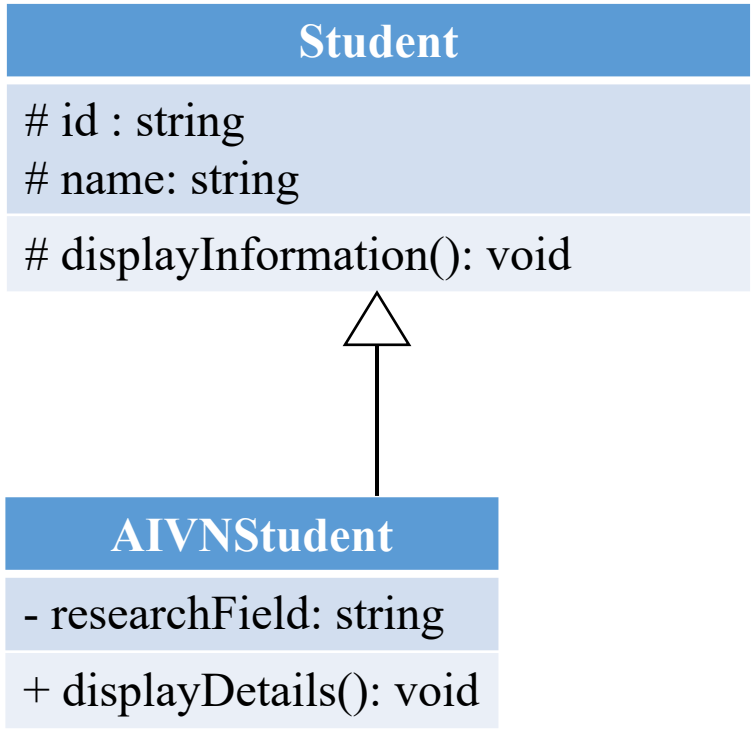
The members of a class that are declared public are easily accessible from any part of the program. All data members and member method of a class are public by default.

Example
Protected Access Modifier



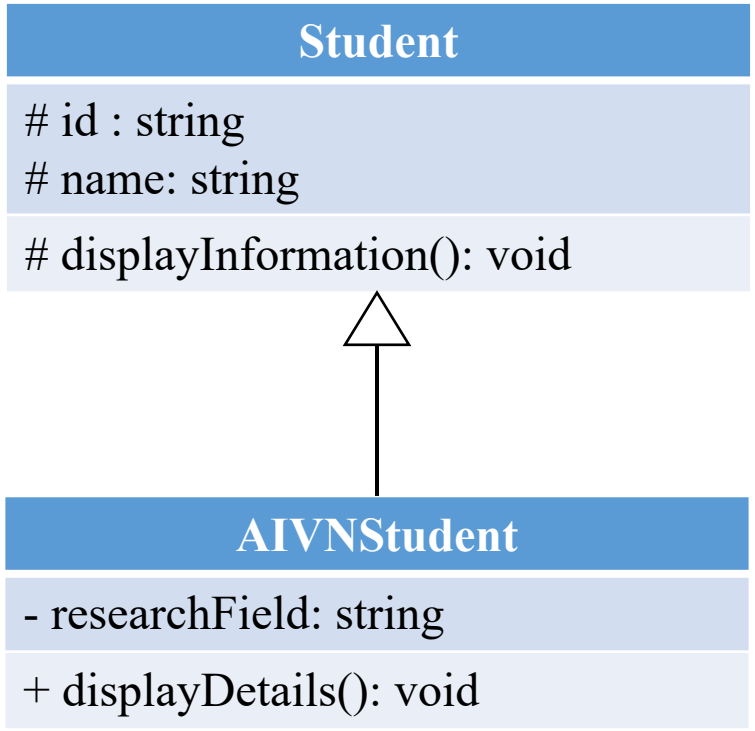

```python
# super class
class Student:
    # protected data members
    _id = None
    _name = None

    # constructor
    def __init__(self, id, name):
        self._id = id
        self._name = name

    # protected member function
    def _displayInformation(self):
        # accessing protected data members
        print("ID: ", self._id)
        print("Name: ", self._name)
```

Example
Protected Access Modifier




```python
# derived class
class AIVNStudent(Student):
    # constructor
    def __init__(self, id, name, researchField):
        self.__researchField = researchField
        Student.__init__(self, id, name)

    # public member function
    def displayDetails(self):
        # accessing protected member functions of super class
        self._displayInformation()

        # accessing private data members of child class
        print("Research Field: ", self.__researchField)
```

Example
Protected Access Modifier



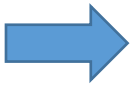

```
35  # creating objects of the derived class
36  obj = AIVNStudent(23032023, "Nguyen Dinh Vinh", "Computer
        Vision")
37
38  # calling public member functions of the class
39  obj.displayDetails()
```

```
1  ID:  23032023
2  Name:  Nguyen Dinh Vinh
3  Research Field:  Computer Vision
```

The members of a class that are declared protected are only accessible to a class derived from it. Data members of a class are declared protected by adding a single underscore '_' symbol before the data member of that class.

Example
Private access modifier

```
1  #Declare class
2  class Student:
3
4      # __init__ is known as the constructor
5      def __init__(self, name):
6          self.__id = id  #private variable
7          self.__name = name #private variable
8
9      #public function
10     def getName(self):
11       return self.__name
12
13 # Object instantiation
14 vinh = Student("Nguyen Dinh Vinh")
15 print("Access public function: ", vinh.getName())
16 print("Access private variable: ", vinh.name)
```



```
Access public function:  Nguyen Dinh Vinh
Traceback (most recent call last):
  File "<string>", line 16, in <module>
ERROR!
AttributeError: 'Student' object has no attribute 'name'
```

Student
- id : string - name: string
+ getName(): string

The members of a class that are declared private are accessible within the class only, private access modifier is the most secure access modifier. Data members of a class are declared private by adding a double underscore '__' symbol before the data member of that class.

Outline

- Object Relationships (Inheritance)
- Access Modifier
- Module and Package
- Case study
- Exercises

Motivation

Program.py

```
1  def getDate():
2      return "19.05.2023"
3
4  def getDate():
5      return "May 19, 2023"
6
7  print(getDate())
```

???

May 19, 2023

Program1.py

```
1
2  def getDate():
3      return "19.06.2023"
```

Program2.py

```
1
2  def getDate():
3      return "Jun 19, 2023"
```

Program.py

```
1  from Program1 import getDate as p1
2  from Program2 import getDate as p2
3
4  print(p1())
5  print(p2())
```

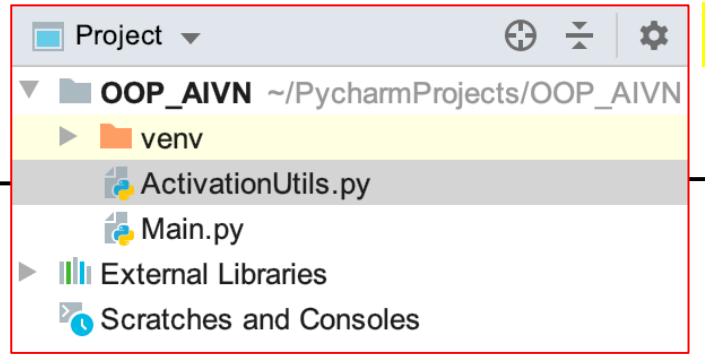
```
19.06.2023
Jun 19, 2023
```

Program1.py, Program2.py, and Program.py are located at the same folder/package

AI VIETNAM
All-in-One Course

Python Module & Package

In essence, a module is a file that contains Python statements and definitions.



Pycharm project

```python
import numpy as np

#compute sigmoid value of x
def sigmoid(x):
    return 1.0/ (1.0 + np.exp(-x))

#compute tanh value of x
def tanh(x):
    return (exp(x)-exp(-x))/(exp(x)+exp(-x))

#compute relu value of x
def relu(x):
    return max(0.0, x)
```

Module: ActivationUtils.py

```
Python
from <module_name> import <name(s)>
```

How to import **ActivationUtils module** to use in main.py

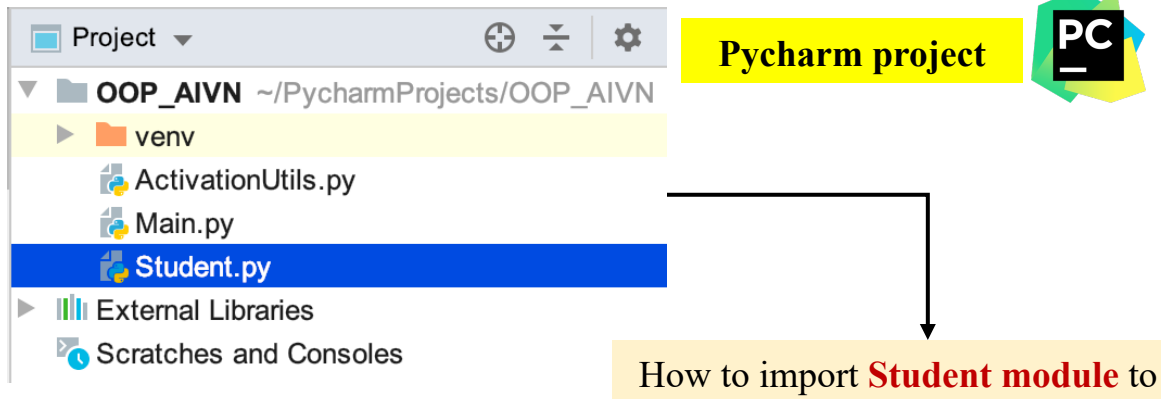
main.py

```python
from ActivationUtils import sigmoid

x = 10.0
print(sigmoid(x))
```

AI VIETNAM
All-in-One Course

Python Module & Package

In essence, a module is a file that contains Python statements and definitions.



Pycharm project

```python
# Declare class
class Student:

    # __init__ is known as the constructor
    def __init__(self, name):
        self.id = id
        self.name = name

    def getName(self):
        return self.name
```

Module: Student.py

How to import **Student module** to use in main.py

```
Python
from <module_name> import *
```

main.py

```python
from Student import *

# Object instantiation
vinh = Student("Nguyen Dinh Vinh")
print("Access public variable: ", vinh.name)
print("Access public function: ", vinh.getName())
```

AI VIETNAM
All-in-One Course

Python Module & Package

Suppose you have developed a very large application that includes many modules. As the number of modules grows, it becomes difficult to keep track of them all if they are dumped into one location. This is particularly so if they have similar names or functionality. You might wish for a means of grouping and organizing them.

University

- Student.py
- Grading.py
- StudentAttendance.py
- Lecturer.py
- Salary.py
- LectureAttendance.py
- Staff.py
- Service.py
- StaffAttendance.py

Python Module & Package

A python package creates a hierarchical directory structure with numerous modules and sub-packages to give an application development environment. They are simply a collection of modules and sub-packages.

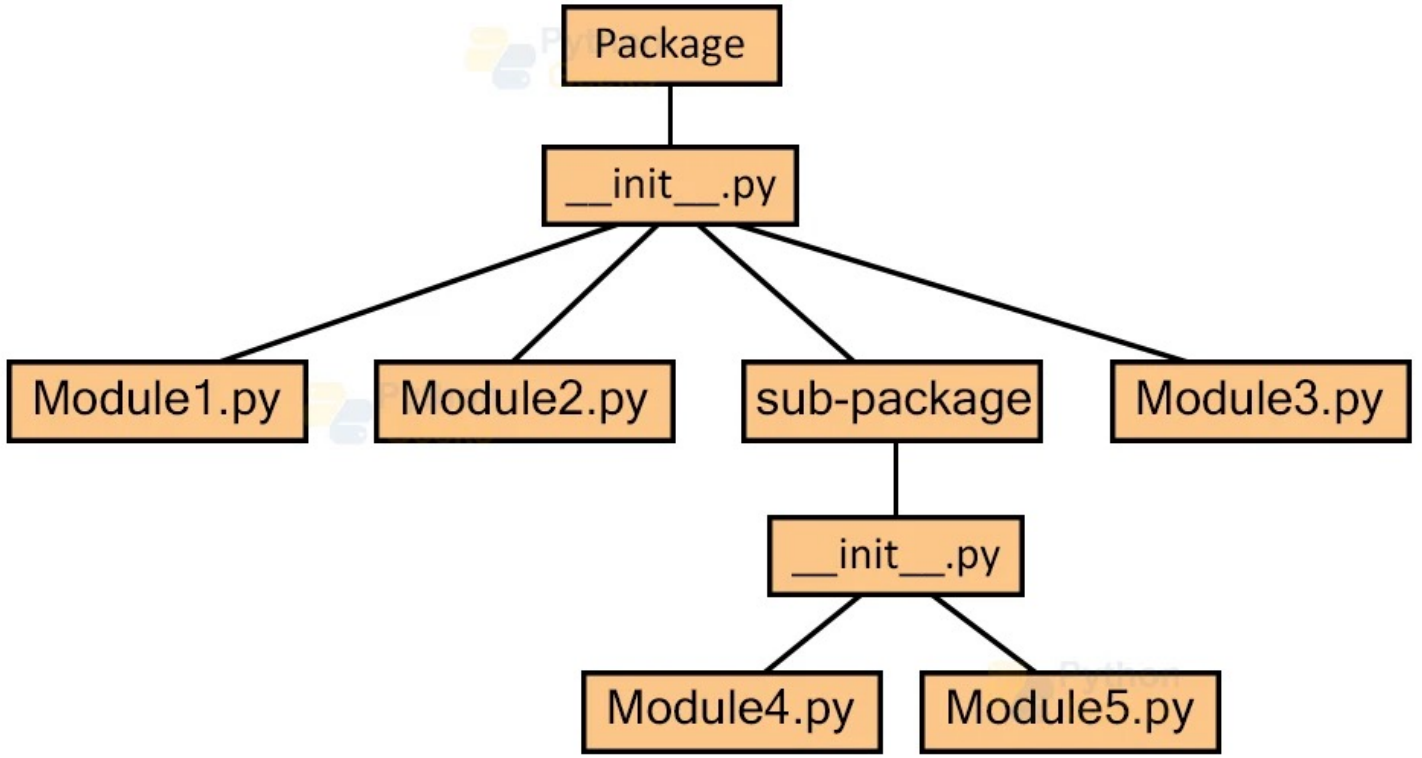
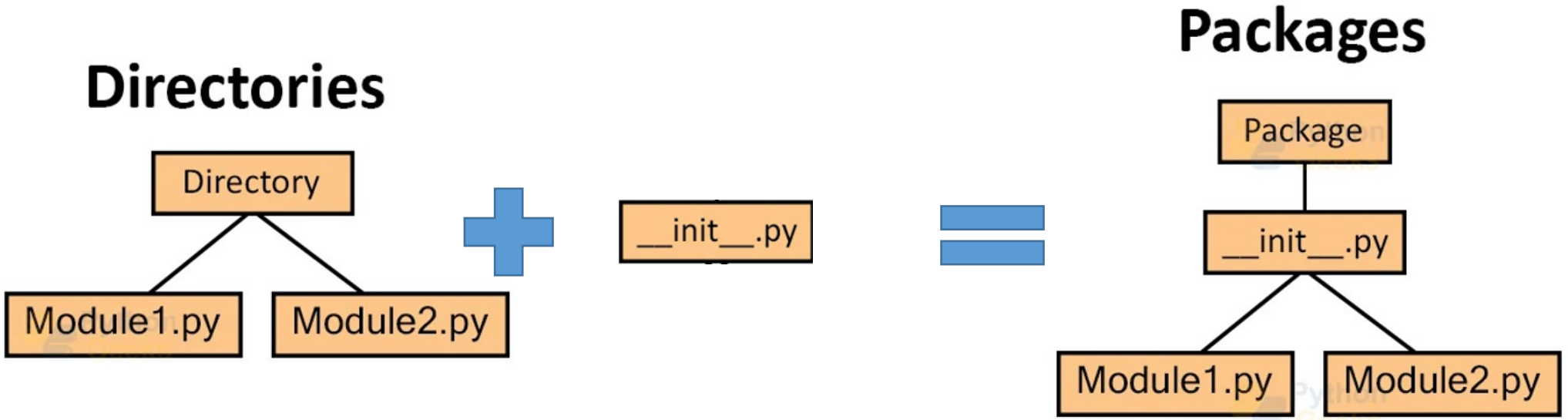


Image Credit: https://pythongeeks.org/python-packages/

Python Module & Package

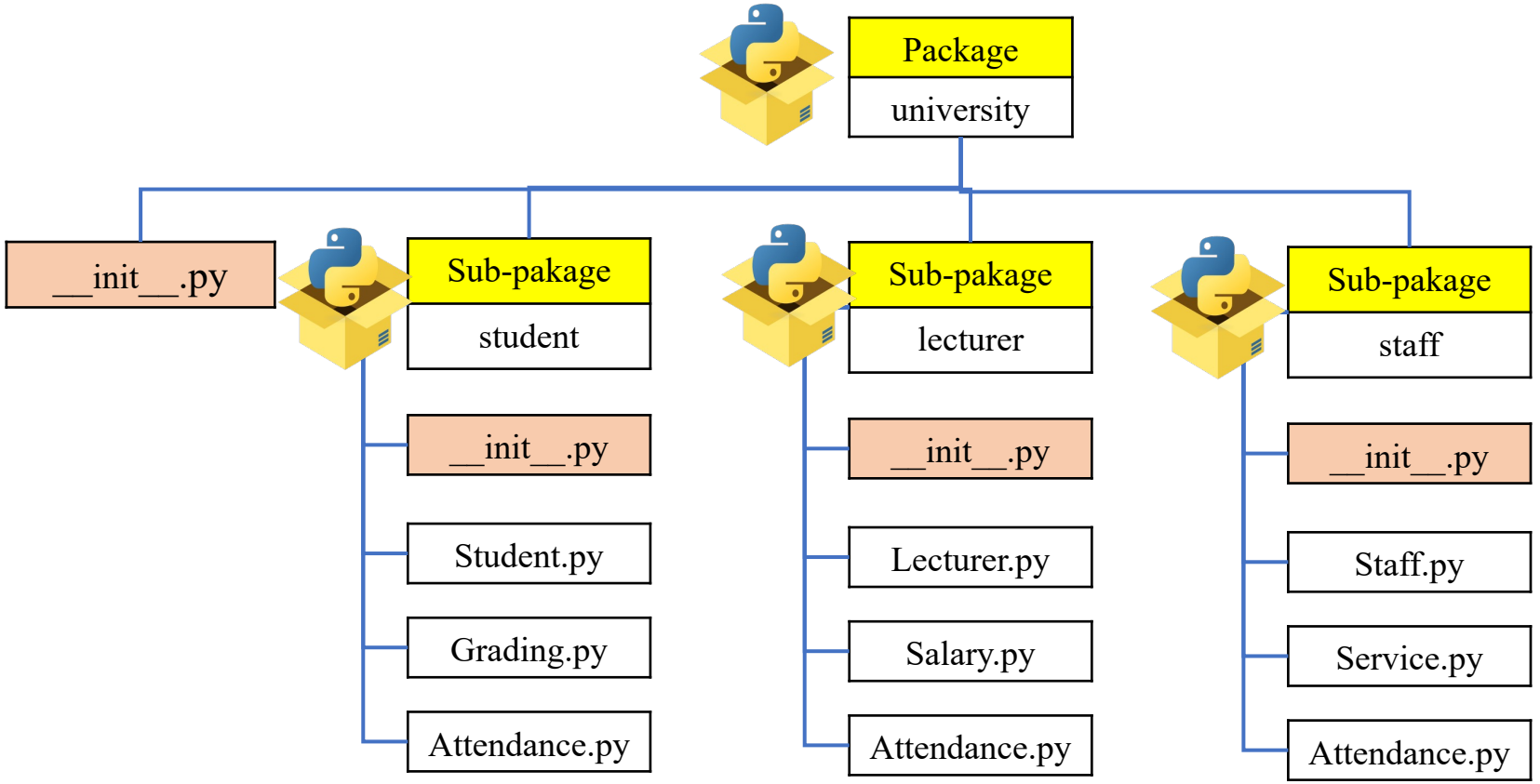
Python Packages vs Directories



The __init__.py file makes an ordinary directory into a Python package

Python Module & Package

Suppose you have developed a very large application that includes many modules. As the number of modules grows, it becomes difficult to keep track of them all if they are dumped into one location. This is particularly so if they have similar names or functionality. You might wish for a means of grouping and organizing them.



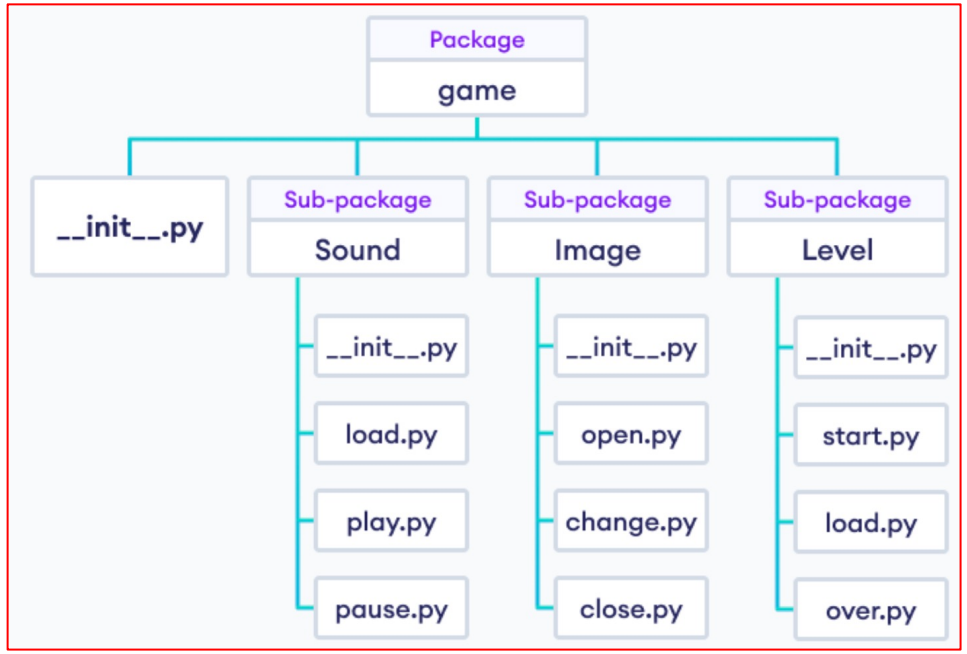
Python Module & Package

Example

Suppose you want to design a collection of modules (a “package”) for the uniform handling of sound files and sound data. There are many different sound file formats (usually recognized by their extension, for example: .wav, .aiff, .au), so you may need to create and maintain a growing collection of modules for the conversion between the various file formats. There are also many different operations you might want to perform on sound data (such as mixing, adding echo, applying an equalizer function, creating an artificial stereo effect).

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

Importing module from a package



For example, if we want to import the **start** module in the example:

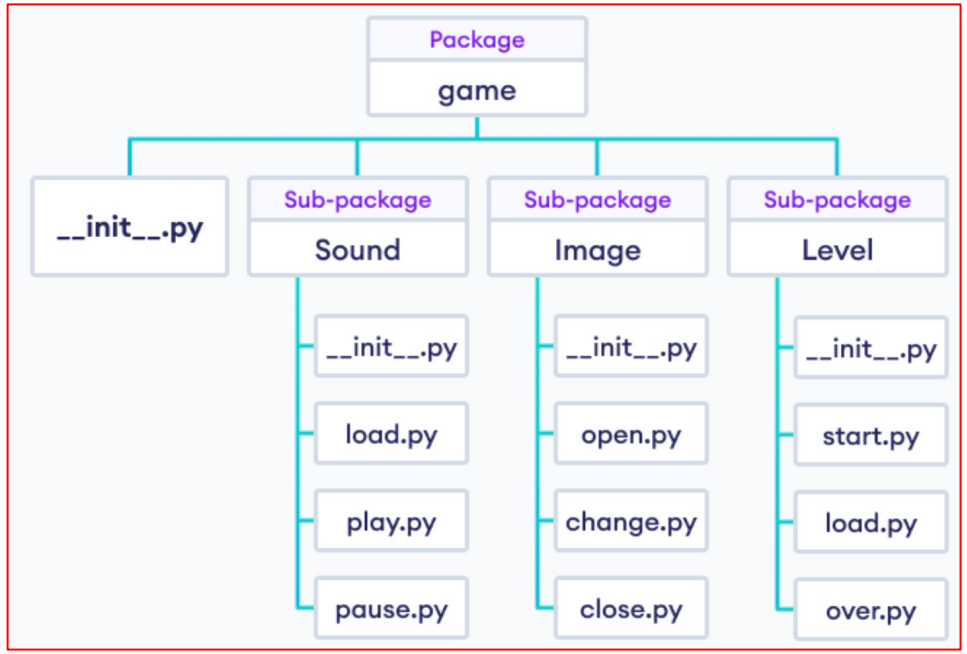
```
import Game.Level.start
```

if **start** module contains a function named **select_difficulty**(). How to use it?

```
Game.Level.start.select_difficulty(2)
```

Image credits: https://www.programiz.com/python-programming/package

Importing module from a package



For example, if we want to import the **start** module in the example:

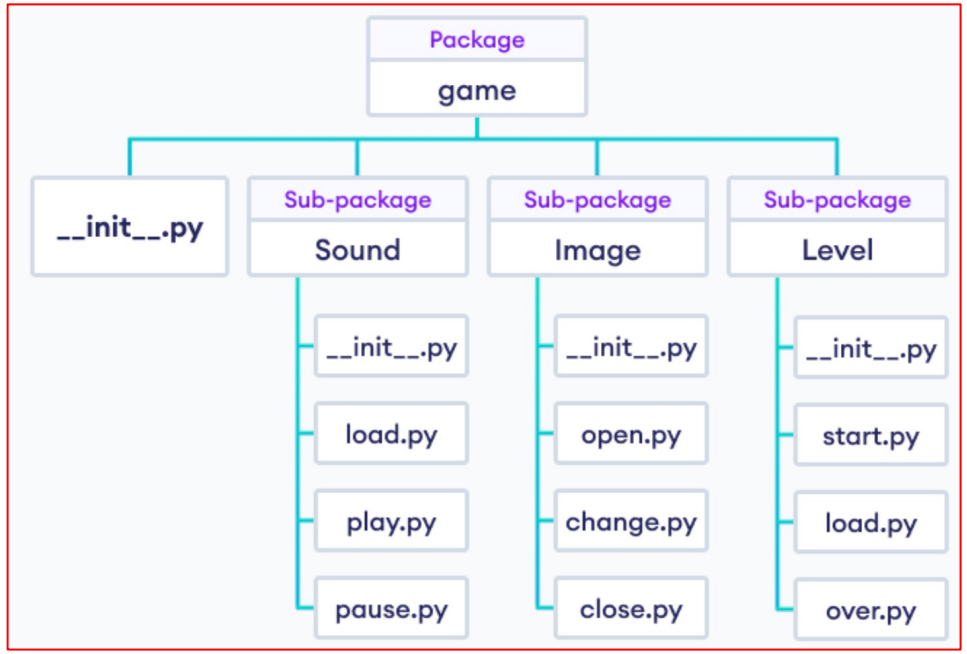
```
from Game.Level import start
```

if **start** module contains a function named **select_difficulty**(). How to use it?

```
start.select_difficulty(2)
```

Image credits: https://www.programiz.com/python-programming/package

Importing module from a package



For example, if we want to import the **start** module in the example:

```
from Game.Level.start import select_difficulty
```

if **start** module contains a function named **select_difficulty**(). How to use it?

```
select_difficulty(2)
```

Image credits: https://www.programiz.com/python-programming/package

Outline

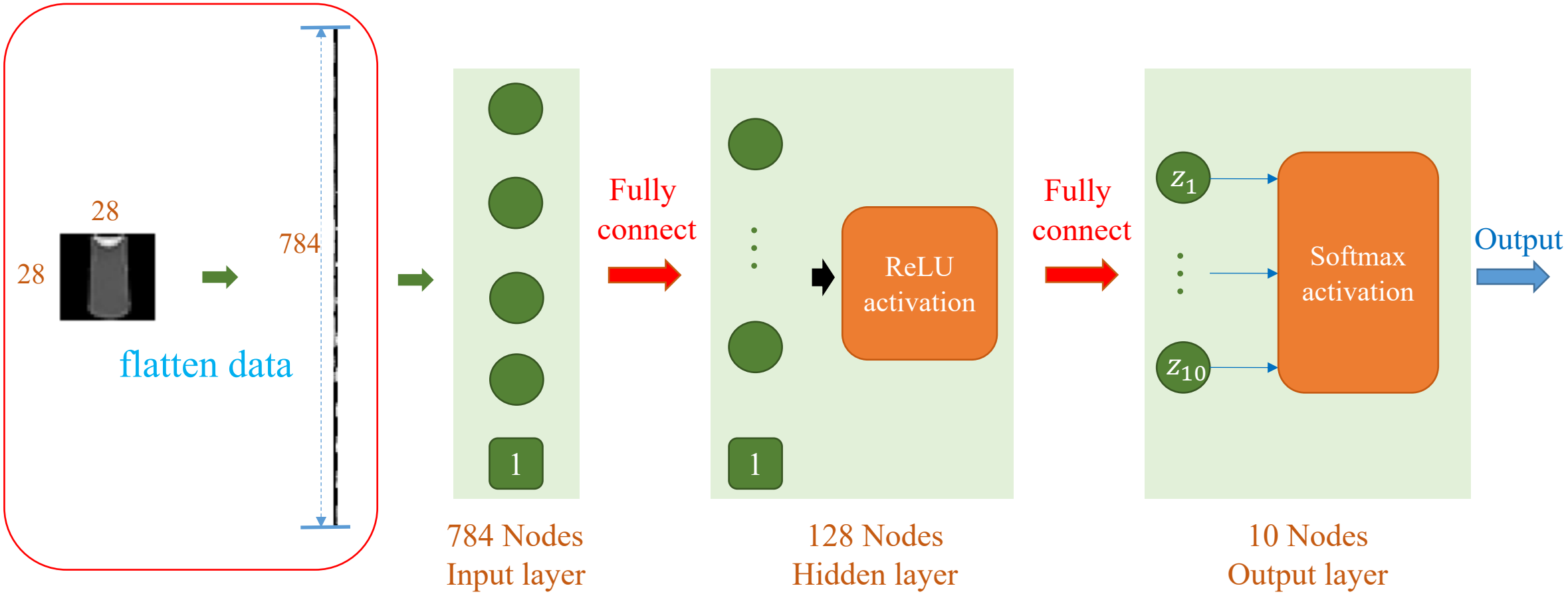
- Object Relationships (Inheritance)
- Access Modifier
- Module and Package
- Case study
- Exercises

Case study: Custom MLP for Fashion-MNIST

How to create Class in Keras Deep Learning Framework

Case study: Custom MLP for Fashion-MNIST
How to create Class in Keras Deep Learning Framework

❖ ReLU and SGD



Custom MLP for Fashion-MNIST

❖ Custom ReLU

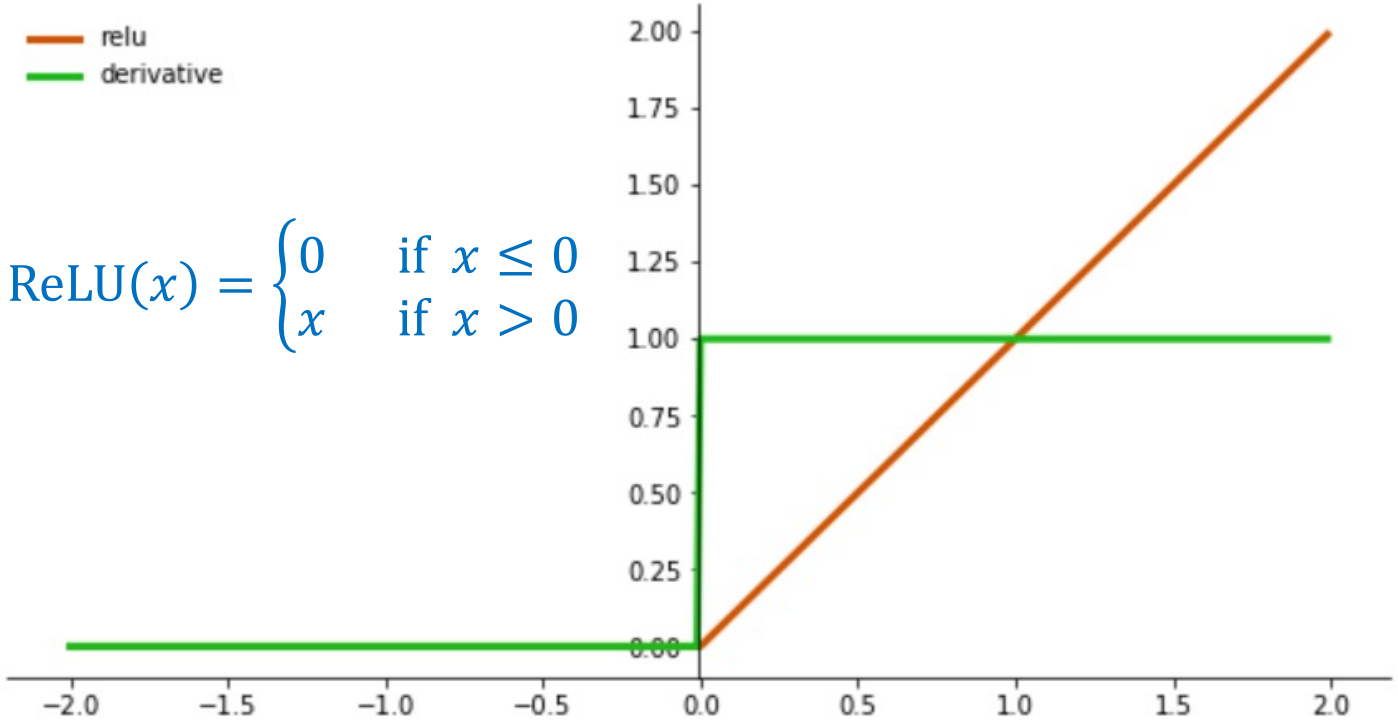
init method

Initialize values/variables necessary for a class

call method

Forward computation max(0, x)

$$\text{ReLU}(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$$




```python
class MyReluActivation(tf.keras.layers.Layer):
    def __init__(self):
        super(MyReluActivation, self).__init__()

    def call(self, inputs):
        return tf.maximum(inputs, 0)
```

Custom MLP for Fashion-MNIST

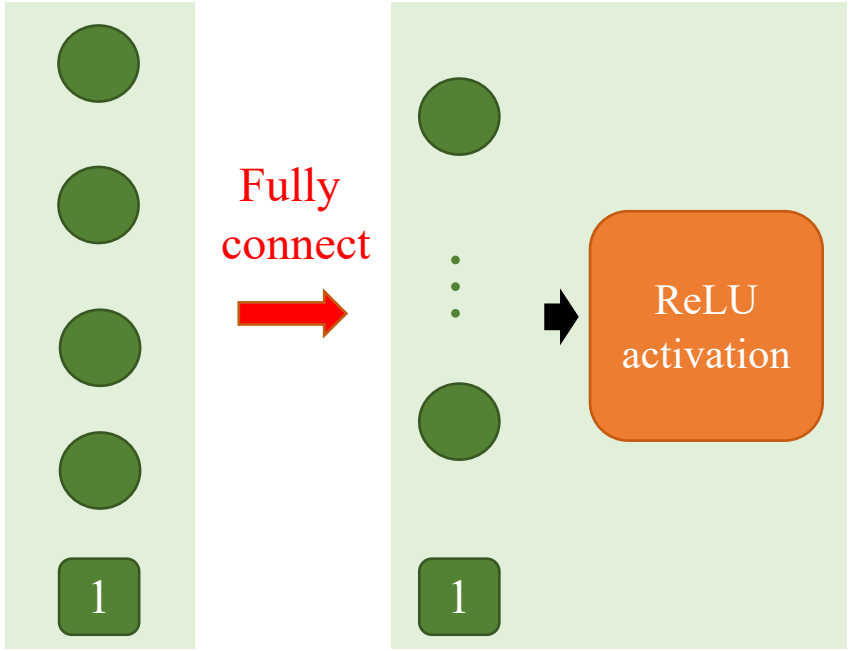
Custom Dense

init method
Initialize values/variables necessary for a class
build method
Do something using the shape of the input tensors
call method
Forward computation $z = x\theta$

$\theta = [\theta_1 \ \ \theta_2 \ \ \dots \ \ \theta_{128}]$

num_outputs = 128

input_shape = (None, 784)

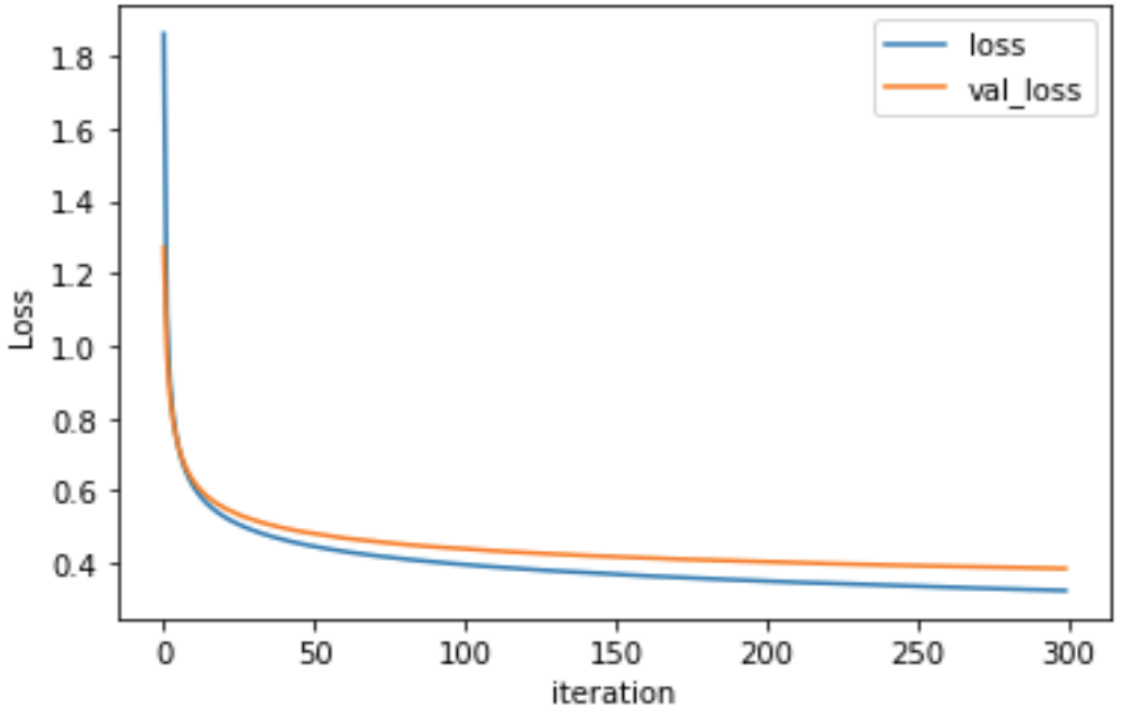


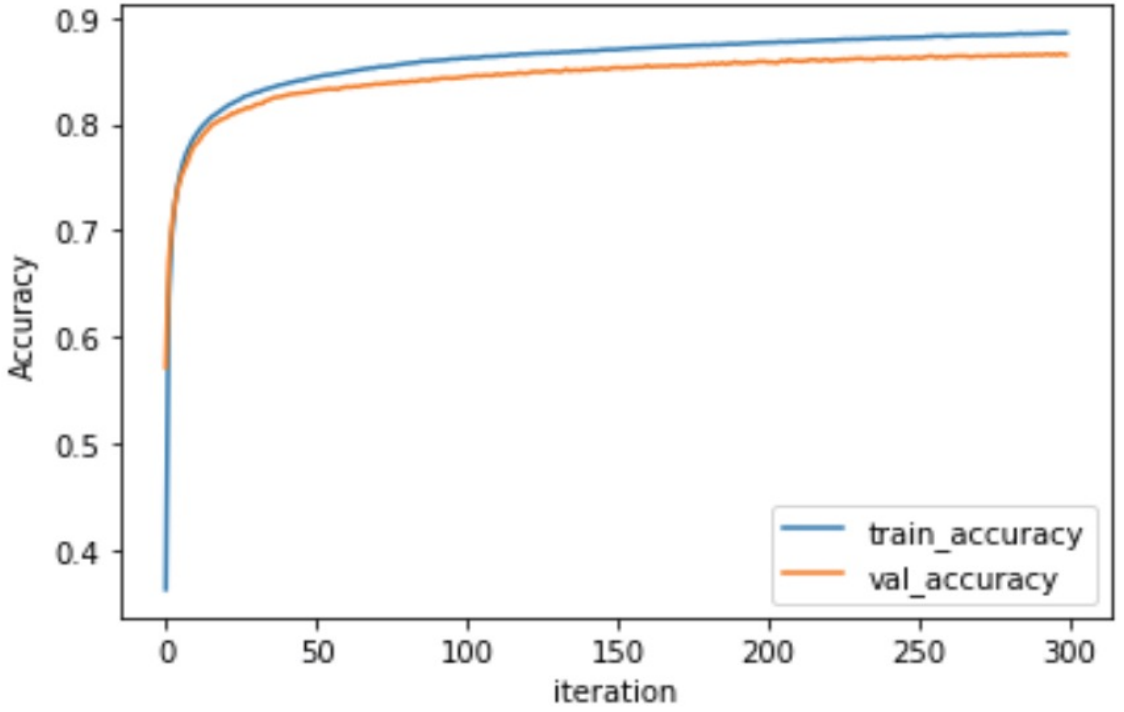

```python
class MyDenseLayer(tf.keras.layers.Layer):
    def __init__(self, num_outputs):
        super(MyDenseLayer, self).__init__()
        self.num_outputs = num_outputs

    def build(self, input_shape):
        kernel_shape = [int(input_shape[-1]), self.num_outputs]
        self.kernel = self.add_weight("kernel", shape=kernel_shape)

    def call(self, inputs):
        return tf.matmul(inputs, self.kernel)
```

Custom MLP for Fashion-MNIST

Result

```python
import tensorflow as tf
import tensorflow.keras as keras

# create model
model = keras.Sequential()
model.add(keras.Input(shape=(784,)))
model.add(MyDenseLayer(128))
model.add(MyReluActivation())
model.add(MyDenseLayer(10))
model.add(tf.keras.layers.Softmax())

# optimizer and loss
opt = tf.keras.optimizers.SGD(0.001)
model.compile(optimizer=opt,
              loss='sparse_categorical_crossentropy',
              metrics=['sparse_categorical_accuracy'])

# training
batch_size = 256
history = model.fit(X_train, y_train, batch_size,
                    validation_data=(X_test, y_test),
                    epochs=300, verbose=2)
```



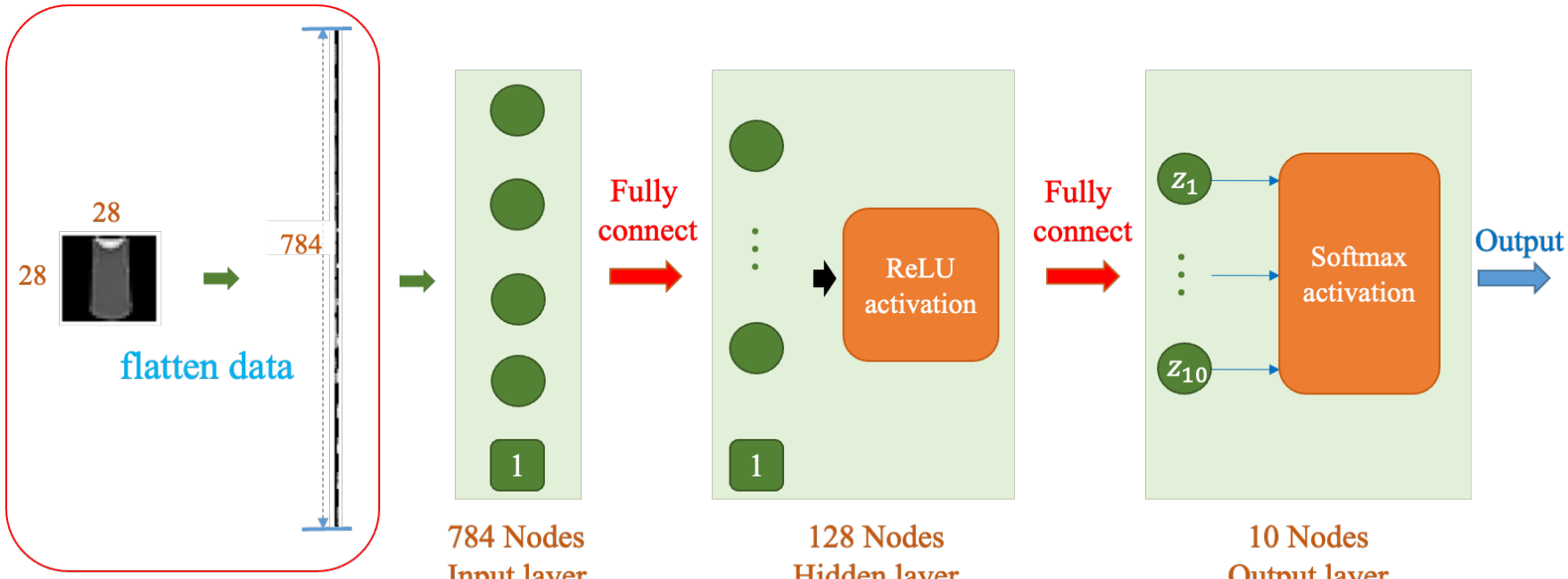


AI VIETNAM
All-in-One Course

Case study: Custom MLP for Fashion-MNIST
SubClass in Pytorch Framework

- ❖Neural networks comprise of layers/modules that perform operations on data
- ❖Every module in PyTorch subclasses the nn.Module

Develop a neural network to classify images in the FashionMNIST dataset.
Your neural network acts as a subclassing from torch.nn.Module



https://pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html

Case study: Custom MLP for Fashion-MNIST
Class in Pytorch Framework

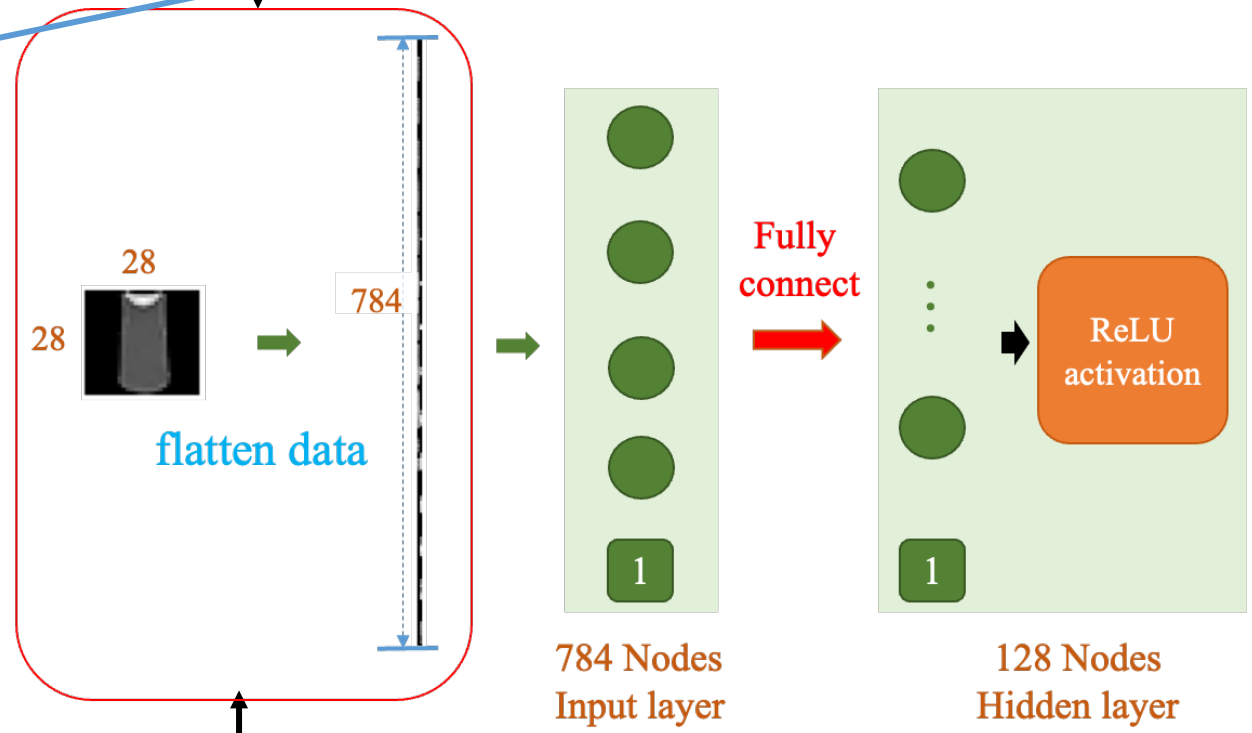
We define our neural network by subclassing nn.Module

Define a subclass

Inheritance

```
1  class NeuralNetwork(nn.Module):
2      def __init__(self):
3          super().__init__()
4          self.flatten = nn.Flatten()
5          self.linear_relu_stack = nn.Sequential(
6              nn.Linear(28*28, 784),
7              nn.Linear(784, 128),
8              nn.ReLU(),
9              nn.Linear(128, 10),
10         )
11
12     def forward(self, x):
13         x = self.flatten(x)
14         logits = self.linear_relu_stack(x)
15         return logits
```

Define Forward path

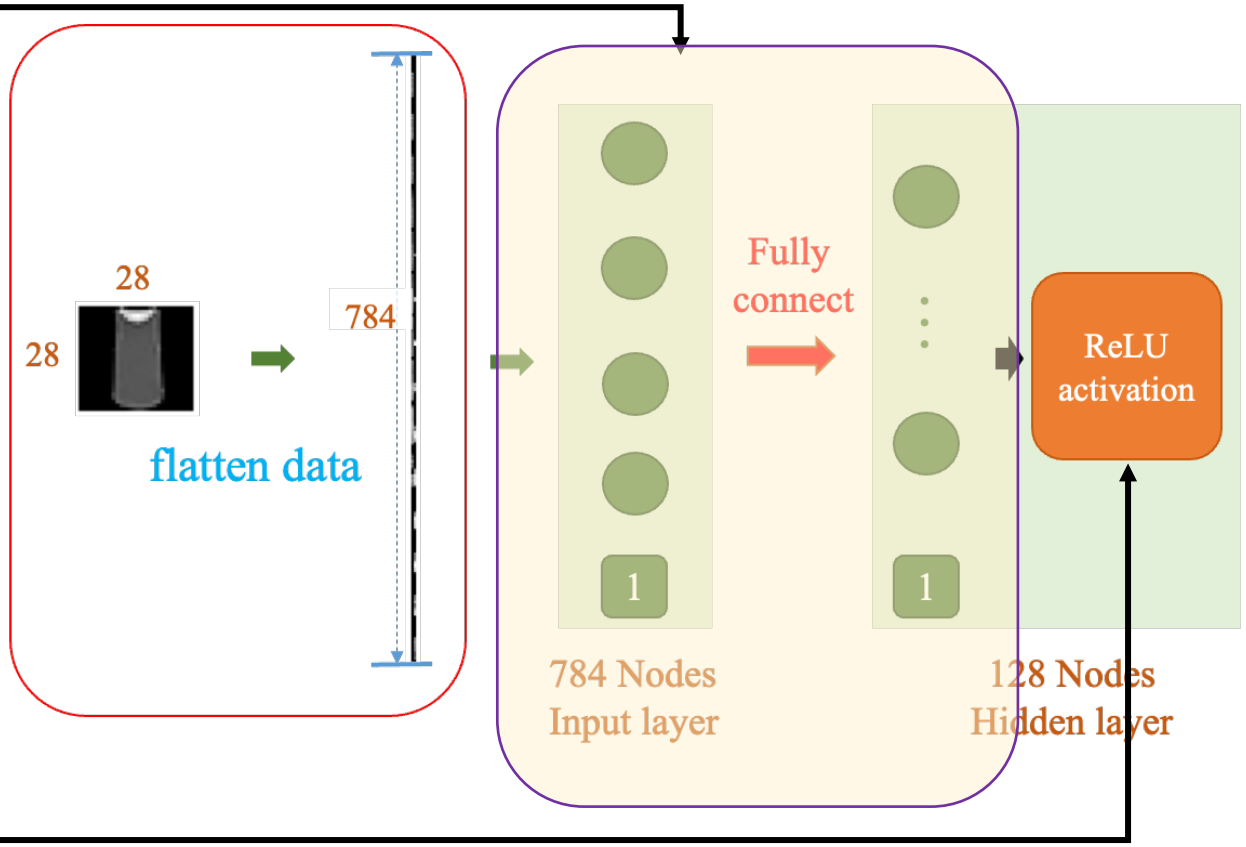


Case study: Custom MLP for Fashion-MNIST Class in Pytorch Framework

Define the Class

We define our neural network by subclassing nn.Module

```
1  class NeuralNetwork(nn.Module):
2      def __init__(self):
3          super().__init__()
4          self.flatten = nn.Flatten()
5          self.linear_relu_stack = nn.Sequential(
6              nn.Linear(28*28, 784),
7              nn.Linear(784, 128),
8              nn.ReLU(),
9              nn.Linear(128, 10),
10         )
11
12     def forward(self, x):
13         x = self.flatten(x)
14         logits = self.linear_relu_stack(x)
15         return logits
```

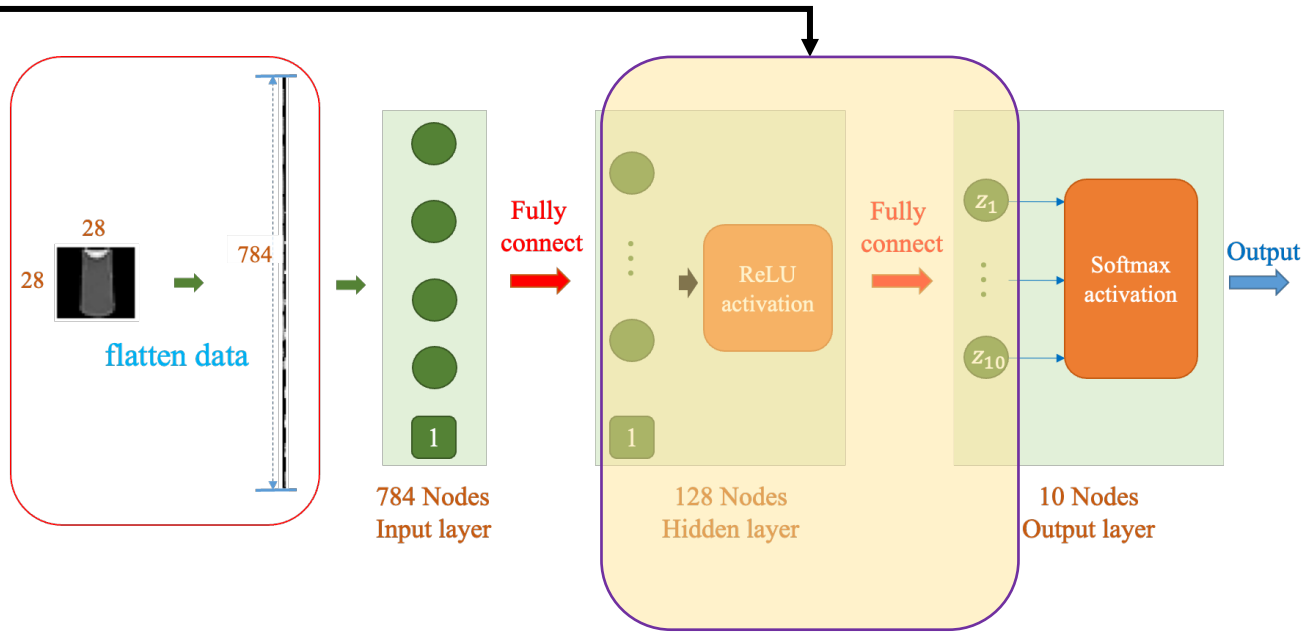


Case study: Custom MLP for Fashion-MNIST Class in Pytorch Framework

Define the Class

We define our neural network by subclassing nn.Module

```
1  class NeuralNetwork(nn.Module):
2      def __init__(self):
3          super().__init__()
4          self.flatten = nn.Flatten()
5          self.linear_relu_stack = nn.Sequential(
6              nn.Linear(28*28, 784),
7              nn.Linear(784, 128),
8              nn.ReLU(),
9              nn.Linear(128, 10),
10         )
11
12     def forward(self, x):
13         x = self.flatten(x)
14         logits = self.linear_relu_stack(x)
15         return logits
```



Case study: Custom MLP for Fashion-MNIST Class in Pytorch Framework

Define the Class

We define our neural network by subclassing nn.Module

```python
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 784),
            nn.Linear(784, 128),
            nn.ReLU(),
            nn.Linear(128, 10),
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

```python
model = NeuralNetwork().to(device)
print(model)
```

```
NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=10, bias=True)
  )
)
```

Case study: Custom MLP for Fashion-MNIST
Class in Pytorch Framework

Define the Class

We define our neural network by subclassing nn.Module

```python
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 784),
            nn.Linear(784, 128),
            nn.ReLU(),
            nn.Linear(128, 10),
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

```python
model = NeuralNetwork().to(device)
print(model)
```

```python
X = torch.rand(1, 28, 28, device=device)
logits = model(X)
pred_probab = nn.Softmax(dim=1)(logits)
y_pred = pred_probab.argmax(1)
print(pred_probab)
print(f"Predicted class: {y_pred}")
```

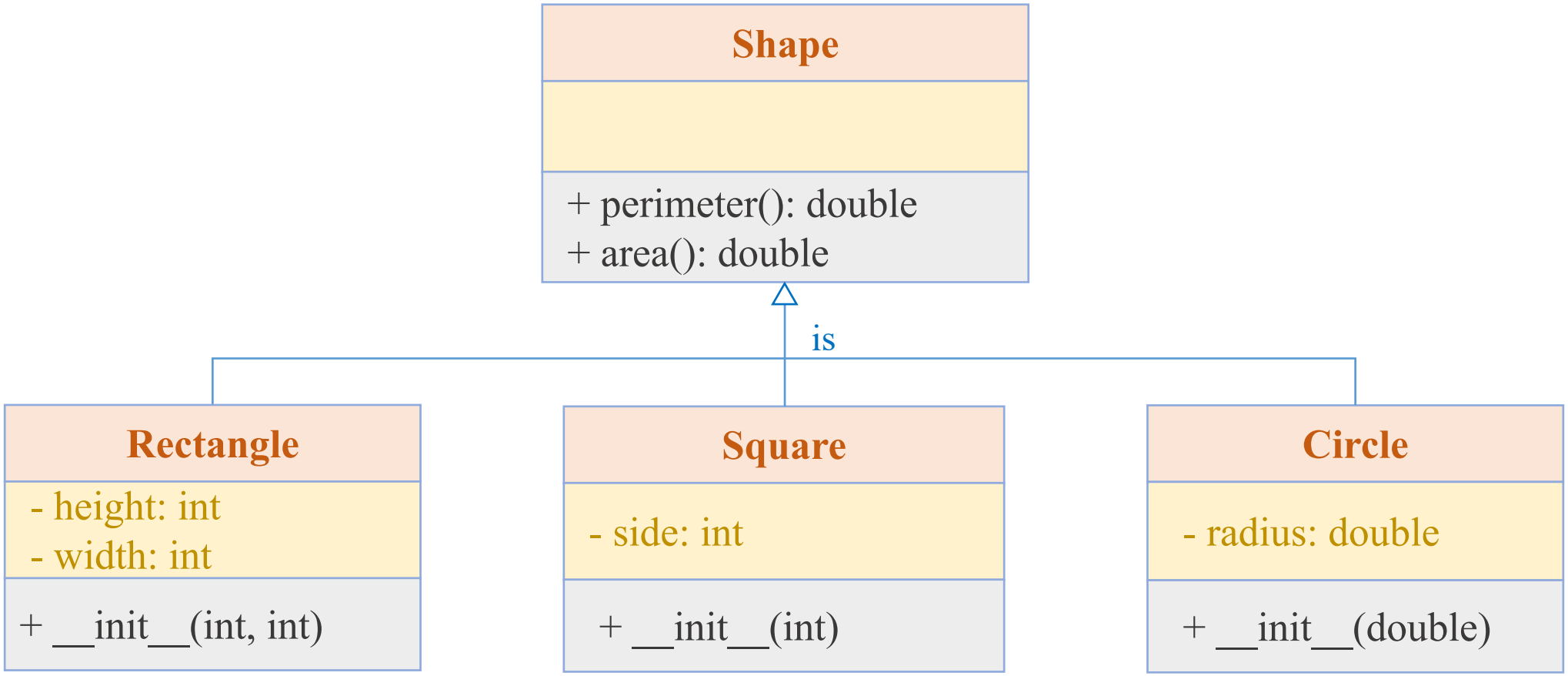
```
tensor([[0.0926, 0.1064, 0.1026, 0.0988, 0.0989, 0.1032, 0
    .1043, 0.0940, 0.1043,
         0.0948]], grad_fn=<SoftmaxBackward0>)

Predicted class: tensor([1])
```

Outline

- Object Relationships (Inheritance)
- Access Modifier
- Module and Package
- Case study
- Summarize

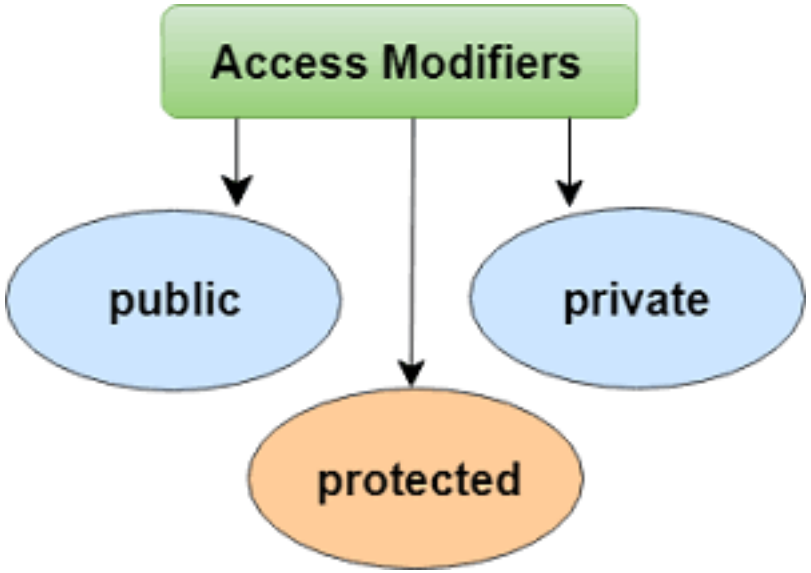
Summarize

➢ Object Relationships (Inheritance)



Summarize

➢ Access Modifier



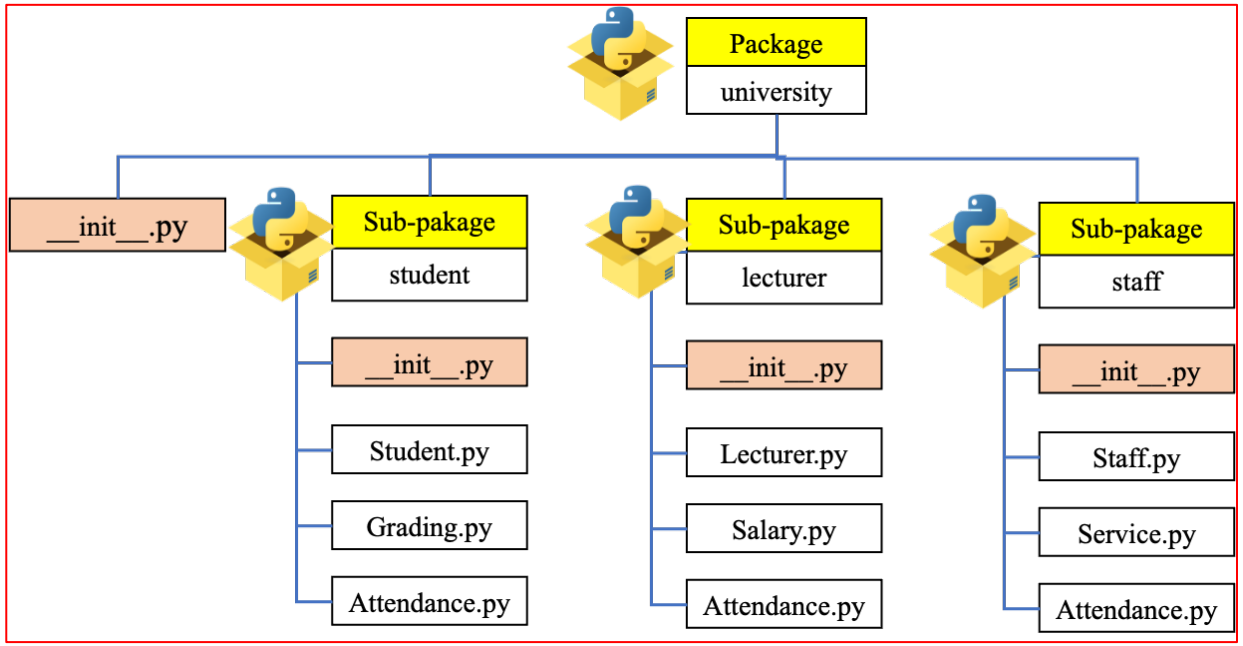
Access Modifiers	Same Class	Same Package	Sub Class	Other Packages
Public	*Y*	*Y*	*Y*	*Y*
Protected	*Y*	*Y*	*Y*	*N*
Private	*Y*	*N*	*N*	*N*

Summarize

➢ Module and Package

University

- Student.py
- Grading.py
- StudentAttendance.py
- Lecturer.py
- Salary.py
- LectureAttendance.py
- Staff.py
- Service.py
- StaffAttendance.py



Outline

- Object Relationships (Inheritance)
- Access Modifier
- Module and Package
- Case study
- Further reading

Polymorphism in Python

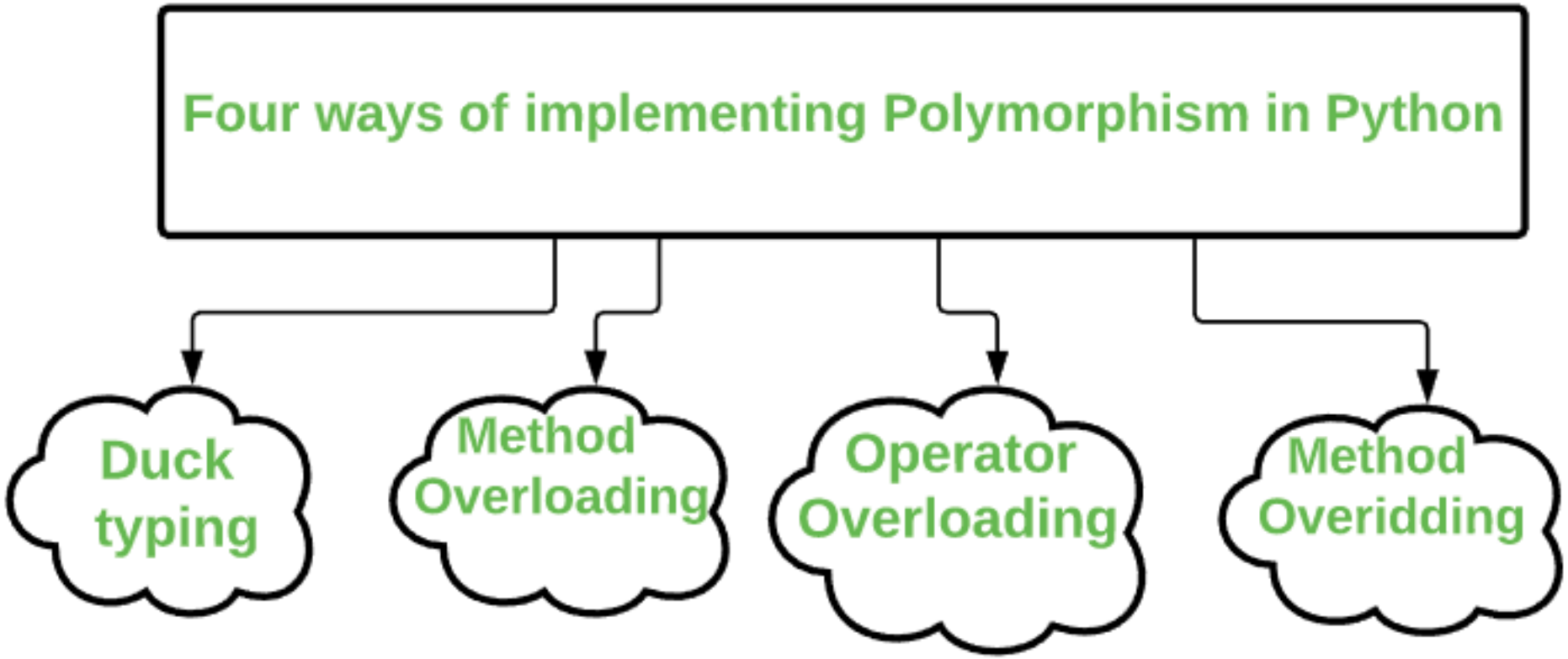


Image credit: https://www.geeksforgeeks.org/ways-of-implementing-polymorphism-in-python/

Duck Typing

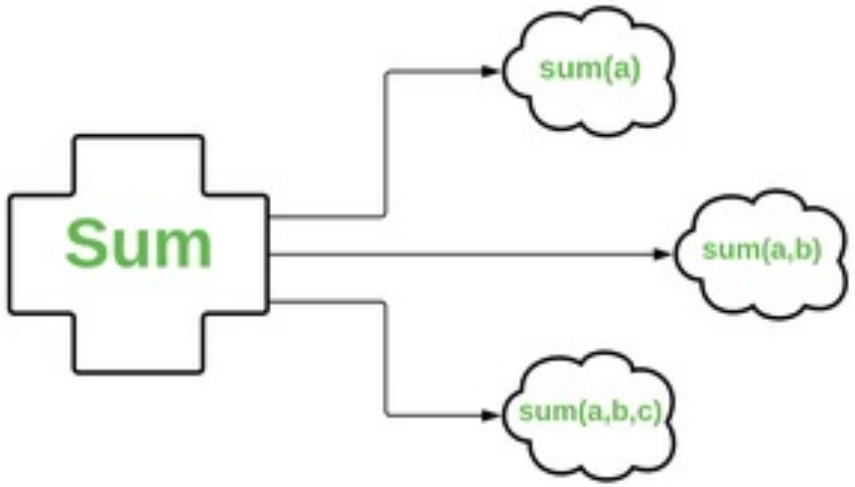
```python
class AIVN1:
    def execute (self, ide):
        ide.execute()

class AIVN2:
    def execute (self):
        print("Welcome to AI Vietnam!")

# create object of AIVN2
aivn2 = AIVN2()

# create object of class AIVN1
aivn1 = AIVN1()

# calling the function by giving ide as the argument.
aivn1.execute(aivn2)
```

Duck Typing

```
class Bird:
    def fly(self):
        print("fly with wings")

class Airplane:
    def fly(self):
        print("fly with fuel")

class Fish:
    def swim(self):
        print("fish swim in sea")

# Attributes having same name are
# considered as duck typing
for obj in Bird(), Airplane(), Fish():
    obj.fly()
```

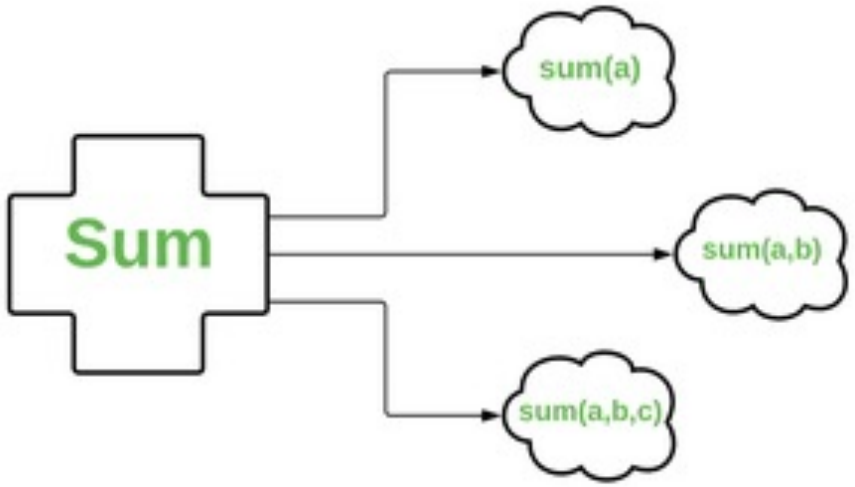
```
# Python program to demonstrate
# duck typing

class Specialstring:
    def __len__(self):
        return 21

# Driver's code
if __name__ == "__main__":

    string = Specialstring()
    print(len(string))
```

Over Loading




```python
class MathUtils:
    def sum(self, a = None, b = None, c = None):
        s = 0
        if a != None and b != None and c != None:
            s = a + b + c
        elif a != None and b != None:
            s = a + b
        else:
            s = a
        return s

mu = MathUtils()

# sum of 2 integers
print(mu.sum(3, 5))

# sum of 3 integers
print(mu.sum(1, 2, 3))
```

Operator Overloading



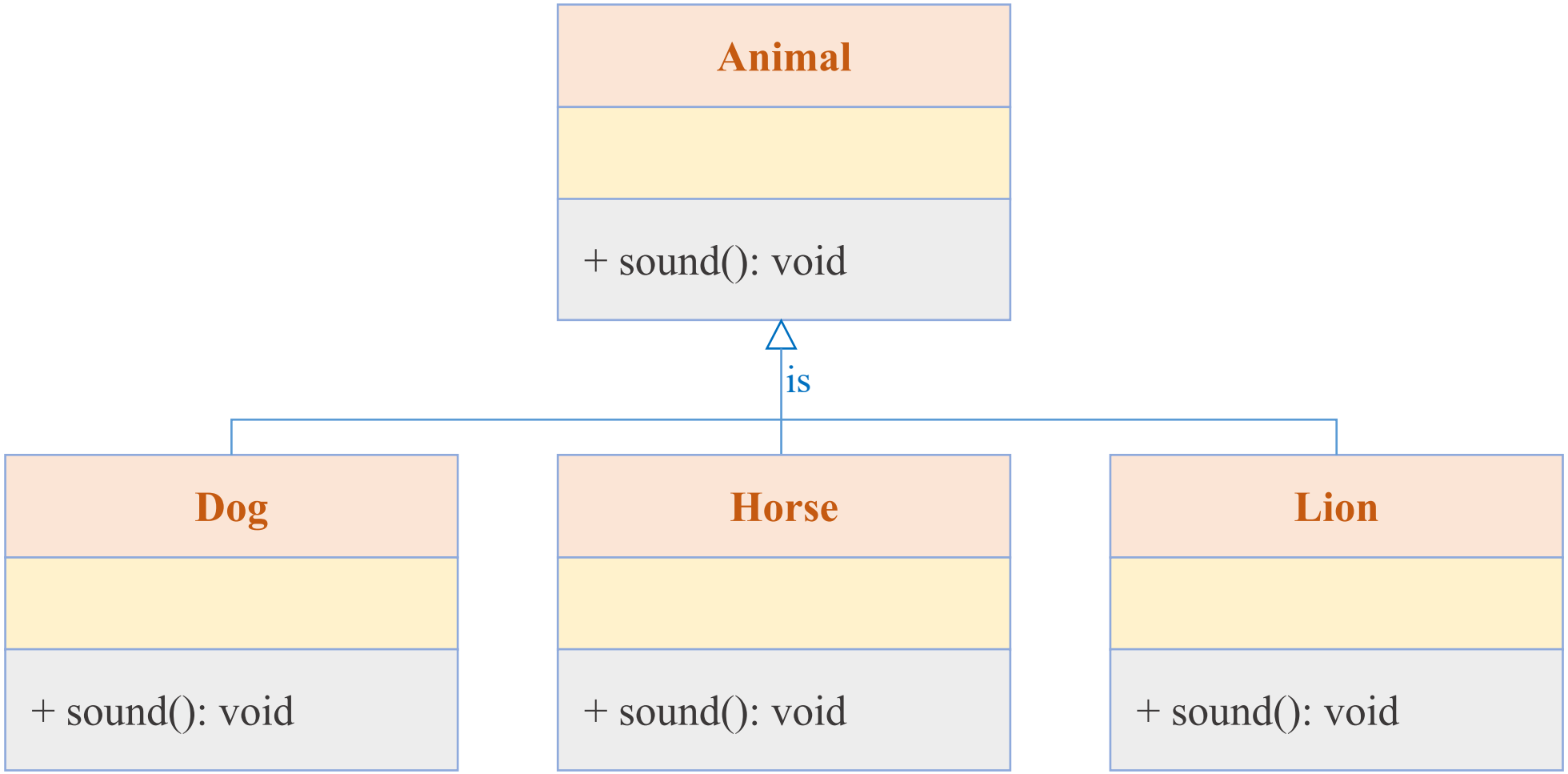

```python
class Student:
    def __init__(self, name, hometown):
        self.name = name
        self.hometown = hometown

vinh = Student ("Vinh", "Can Tho")
hau = Student ("Hau", "An Giang")

# this will generate an error
sum = vinh + hau
```

???

Operator Overloading

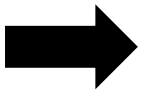
```python
class Student:
    def __init__(self, name, hometown):
        self.name = name
        self.hometown = hometown

    # overloading the + operator
    def __add__(self, other):
        name = self.name + other.name
        homtown = self.hometown + other.hometown
        result = Student(name, homtown)
        return result

vinh = Student ("Vinh", "Can Tho")
hau = Student ("Hau", "An Giang")

# this will generate an error
sum = vinh + hau
print("Name: ", sum.name, ",Hometown: ", sum.hometown)
```

Method Overriding

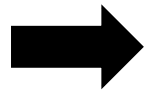


Method Overriding

```python
class Person:
    def perform(self):
        print('Person activities')

class Doctor(Person):
    def perform(self):
        print('Doctor activities')

class Student(Person):
    def perform(self):
        print('Student activities')
```



```python
def showInformation(object):
  object.perform()

showInformation(Person())
showInformation(Doctor())
showInformation(Student())
```



???

```
Person activities
Doctor activities
Student activities
```

Reference Books