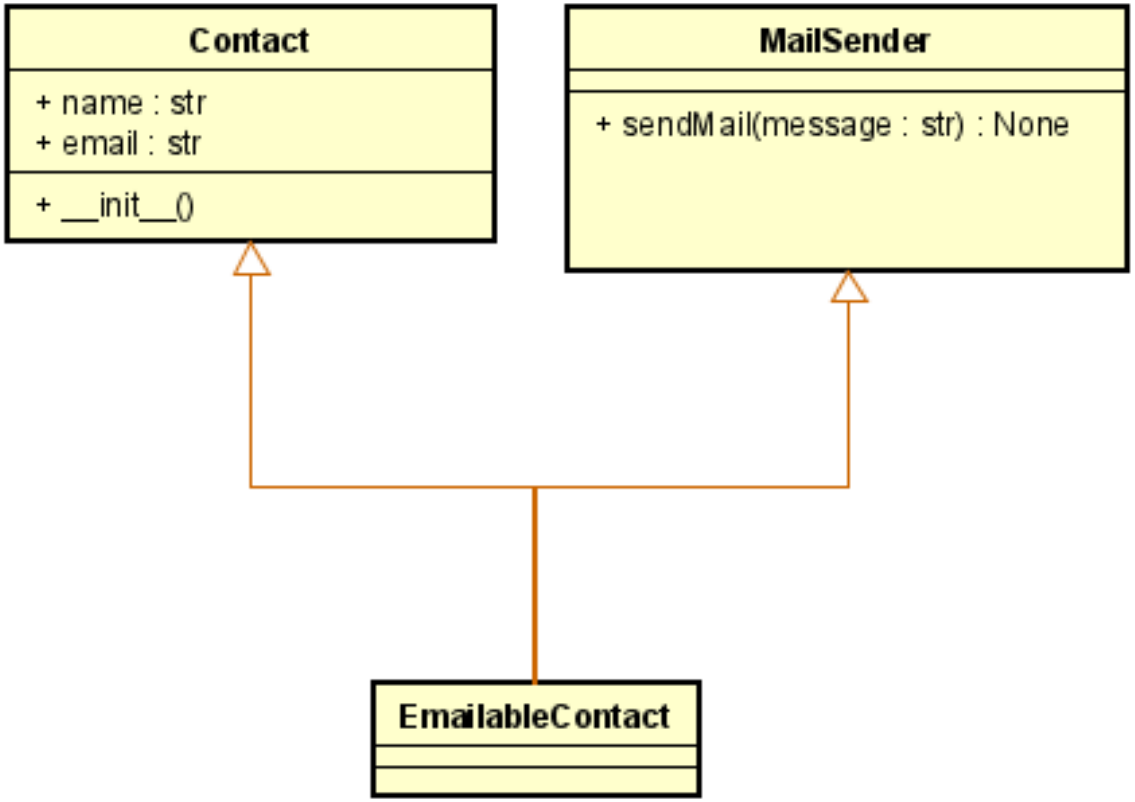
Programming Concepts and Languages

Spring 2024

Can this relation be implemented (as it is) in Java or C# ?



Learning Objectives

- By the end of this session, you should be able to:
  - ✓ explain the concepts of:
    - ✓ classes & objects
    - ✓ encapsulation
    - ✓ abstraction
    - ✓ inheritance (* multiple inheritance & mro)
    - ✓ Polymorphism
  - ✓ implement instance variables, methods, and constructors/ initializers
  - ✓ explain the behaviour of object references
  - ✓ design, implement and test your own Python classes

What is an Object?

- Everything in Python is an object.
- A software item that contains variables and methods
- Object Oriented Design focuses on
  - Encapsulation:
    - dividing the code into a public interface, and a private implementation of that interface
  - Polymorphism:
    - the ability to overload standard operators so that they have appropriate behavior based on their context
  - Inheritance:
    - the ability to create subclasses that contain specializations of their parents
  - Abstraction:
    - the ability to keep the internal mechanics of the code hidden from the user

Encapsulation – Accessibility

- In Python anything with two leading underscores is private
  - Example: __a, __my_variable
- Anything with one leading underscore is semiprivate, and you should feel guilty accessing this data directly.
  - Example: _b
  - Sometimes useful as an intermediate step to making data private

Class Definition and Object Instantiation

- A class is a kind of data type, just like a string, integer or list.
- When we create an object of that data type, we call it an *instance* of a class.

- Class definition syntax:
  - class Subclass[(Superclass)]:
    - [attributes and methods]

- Object instantiation syntax:
  - object = Class()
- Attributes and methods invoke:
  - object.attribute
  - object.method()

Constructing Objects

- Once a class is defined, you can create objects from the class with a *constructor*. The constructor does two things:
  - It creates an object in the memory for the class.
  - It invokes the class's `__init__` method to initialize the object.
- All methods, including the initializer, have the first parameter **self**. This parameter refers to the object that invokes the method.
- The **self** parameter in the `__init__` method is automatically set to reference the object that was just created.

- The __init__ method is run as soon as an object of a class is instantiated. Its aim is to initialize the object.

Constructor/Initializer: __init__()

- The __init__ method is run as soon as an object of a class is instantiated. Its aim is to initialize the object.

Looking at the code,
we can see that after instantiating the object, it automatically invokes __init__()

As a result, it runs self.name = 'Alfonso Ligo',
and prints the value of self.name

Class members - methods

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

- Add a *print_name()* method to the Person class above

Accessing Members of Objects

- An object's member refers to its data fields and methods.
- Data fields are also called *instance variables*, because each object (instance) has a specific value for a data field.
- Methods are also called *instance methods*, because a method is invoked by an object (instance) to perform actions on the object such as changing the values in data fields for the object.
- To access an object's data fields and invoke an object's methods, you need to assign the object to a variable by using the following syntax:

```
object_ref_var = ClassName(arguments)
```

- For example:

```
p2 = Person('Adriano Matty')
```

Form and Object for Class

- Class includes two members: form and object.
- The example in the following can reflect what is the difference between object and form for class.

- Invoke form: just invoke attribute or method in the class, where='VIA'
- Invoke object: instantialize object and then invoke attributes or methods. Here __init__(), where='VIA-Horsens'

Class decorators - @classmethod

- The @classmethod decorator is used to decorate an ordinary method
- sometimes we write classes to group related constants together with functions which act on them and we may never instantiate these classes

```
class Person:
    TITLES = ('Software engr', 'Systems analyst', 'Business analyst')
    def __init__(self, firstname, lastname):
        self.firstname = firstname
        self.lastname = lastname
    def fullname(self): # instance method
        return "%s %s" % (self.firstname, self.lastname)

    @classmethod
    def allowed_titles_starting_with(cls, startswith): # class method
        return [t for t in cls.TITLES if t.startswith(startswith)]
>>> print(pp.allowed_titles_starting_with("S"))
>>> print(Person.allowed_titles_starting_with("S"))
```

Class decorators - @staticmethod

- a static method doesn't have the calling object passed into it as the first parameter
- it doesn't have access to the rest of the class or instance
- are most commonly called from class objects, like class methods

```
class Person:
    TITLES = ('Software engineer', 'Systems analyst', 'Business analyst')
    def __init__(self, firstname, lastname):
        self.firstname = firstname
        self.lastname = lastname
    def fullname(self): # instance method
        return "%s %s" % (self.firstname, self.lastname)

    @staticmethod
    def allowed_titles_ending_with(endswith): # static method
        return [t for t in Person.TITLES if t.endswith(endswith)]
>>> print(p3.allowed_titles_ending_with("t"))
>>> print(Person.allowed_titles_ending_with("t"))
```

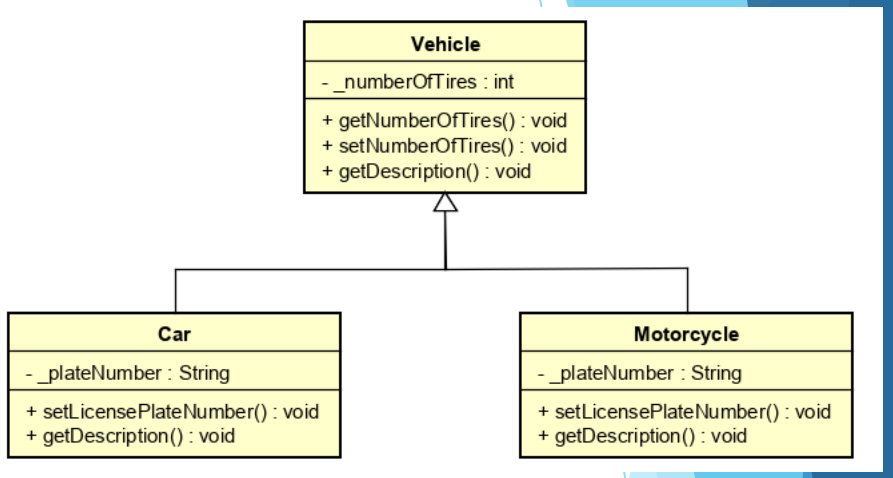
Class decorators - @property

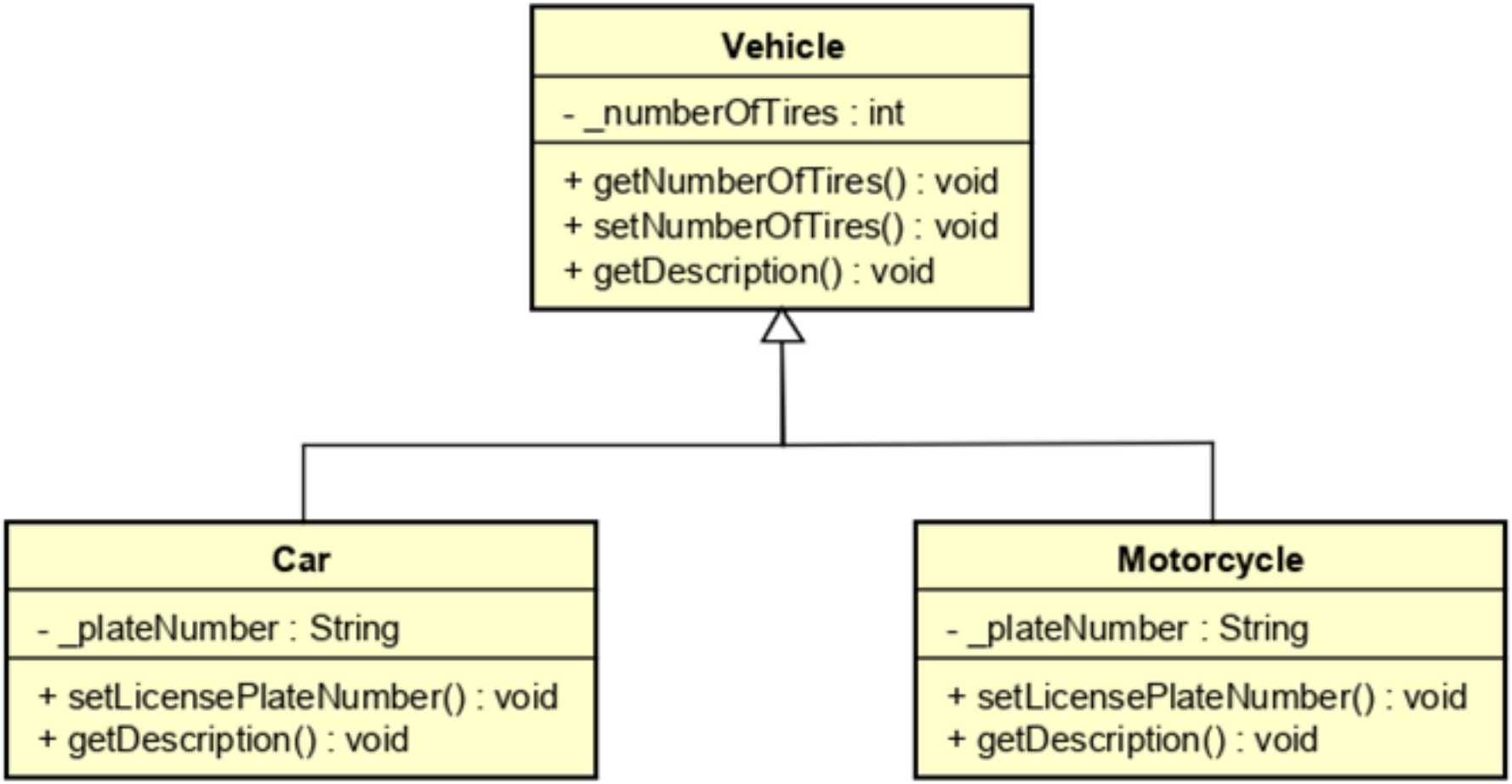
- The @property decorator lets us make a method behave like an attribute
- allows the use of a method to generate a property of an object dynamically

```
class Car(object):
    def __init__(self):
        self._speed = 90
    @property
    def speed(self):
        print("Speed is", self._speed)
        return self._speed
    @speed.setter
    def speed(self, value):
        print("Setting to", value)
        self._speed = value
```

Inheritance

- Inheritance in Python is like in Java, a subclass can invoke attributes and methods in a superclass.
- From the example, class **Car** inherits the **Vehicle** class and invoke *getNumberOfTires()*, *setNumberOfTires()* methods in the **Vehicle** class
- Syntax:
  - class subclass(superclass):



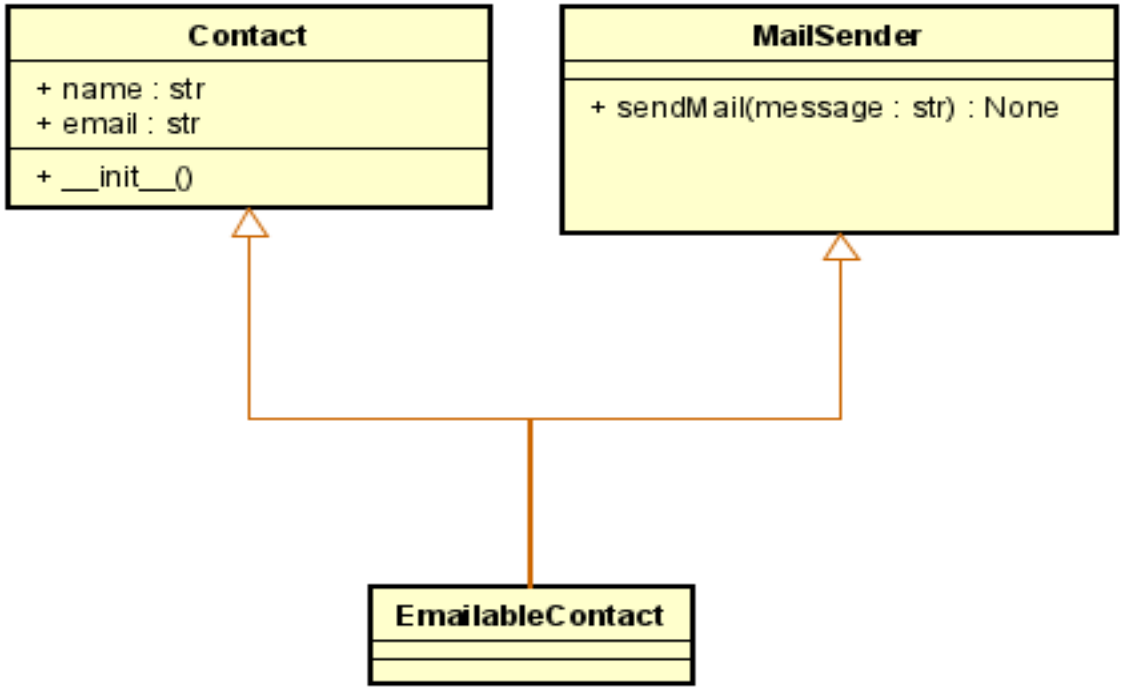

Vehicle
- _numberOfTires : int
+ getNumberOfTires() : void
+ setNumberOfTires() : void
+ getDescription() : void
Car
- _plateNumber : String
+ setLicensePlateNumber() : void
+ getDescription() : void
Motorcycle
- _plateNumber : String
+ setLicensePlateNumber() : void
+ getDescription() : void

Multiple Inheritance

- Python supports multiple inheritance.
- Different from Java/C# but closer to C++ with mixin semantics
- A mixin is a superclass that is not meant to exist on its own, but meant to be inherited by some sub-classes to provide extra functionality.
- Python supports multiple inheritance with depth-first, left-to-right resolution rule for class attributes.
- Python uses Method Resolution Order (MRO) to linearize the superclass.
- Class definition with multiple base classes syntax:

```
class DerivedClass(BaseClass1, BaseClass2, …):
    <statements1>
    <statement2>
    …
```

How do we apply multiple inheritance in practice?



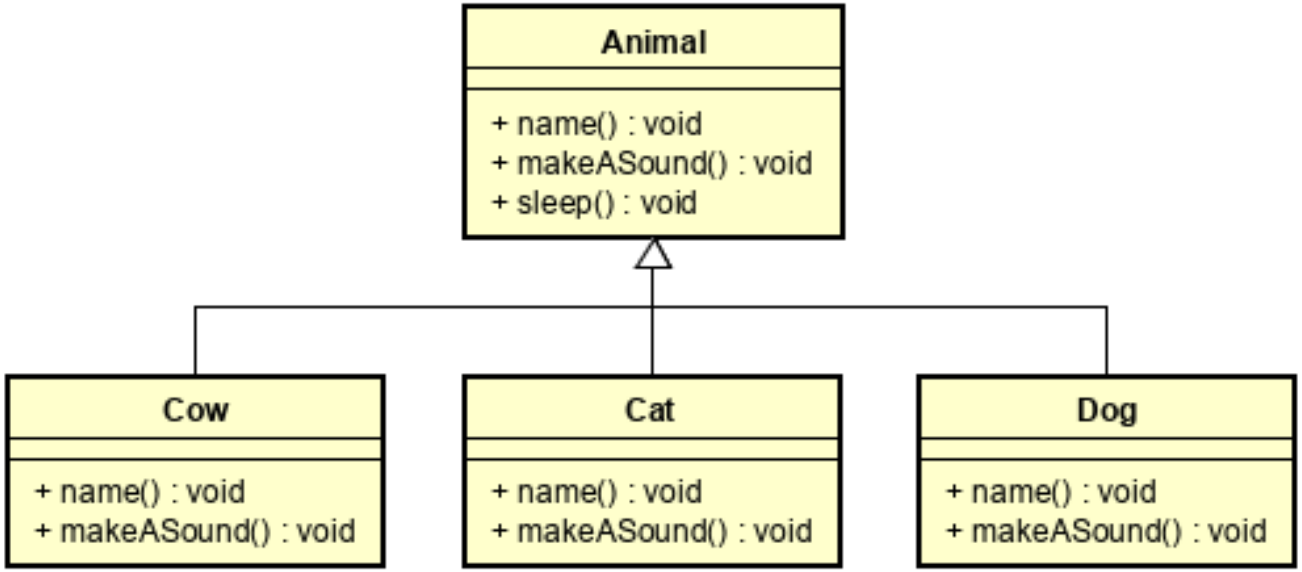
Polymorphism

- Polymorphism means “many forms”. Polymorphic operations have many implementations.
- ie. objects of different classes have operations with the same signature but different implementations.

- In Python, we have the “traditional polymorphism” as described above

- Also, there is polymorphism everywhere in Python
  - Python is a dynamic programming language, however, the interpreter keeps track of all variables types.
    - Thus reflecting the polymorphism character in Python
  - Many operators have the property of polymorphism

Polymorphism in practice I



Polymorphism in practice II

- Many operators in Python have polymorphism property
  - here variables can support any objects which support '+' operation. Not only integer but also string, list, and tuple

```
>>> 1+2
3
>>> 'cross' + 'word'
'crossword'
>>> [2, 4, 6] + [8, 10]
[2, 4, 6, 8, 10]
>>> (1,3,5) + (7, 9)
(1, 3, 5, 7, 9)
```

- Some Python built-in functions have polymorphism property
- Example: repr
  - return the canonical string representation of the object.
  - In the example beside, converts integer to string and also added the two string to '12 Excellent'

```
>>> g = 12
>>> gs = repr(g)
>>> gs
'12'
>>> s = ' Excellent'
>>> ggs = gs + s
>>> ggs
'12 Excellent'
```

Abstraction: Abstract Base Class

- Python provides an easy way of creating an abstract base class through a built-in module abc (Abstract Base Class).

```
from abc import ABC, abstractmethod

class Car(ABC):
    @abstractmethod
    def car_model(self):
        pass
```