University of
Applied Sciences

Programming 2

Basic OOP Concepts

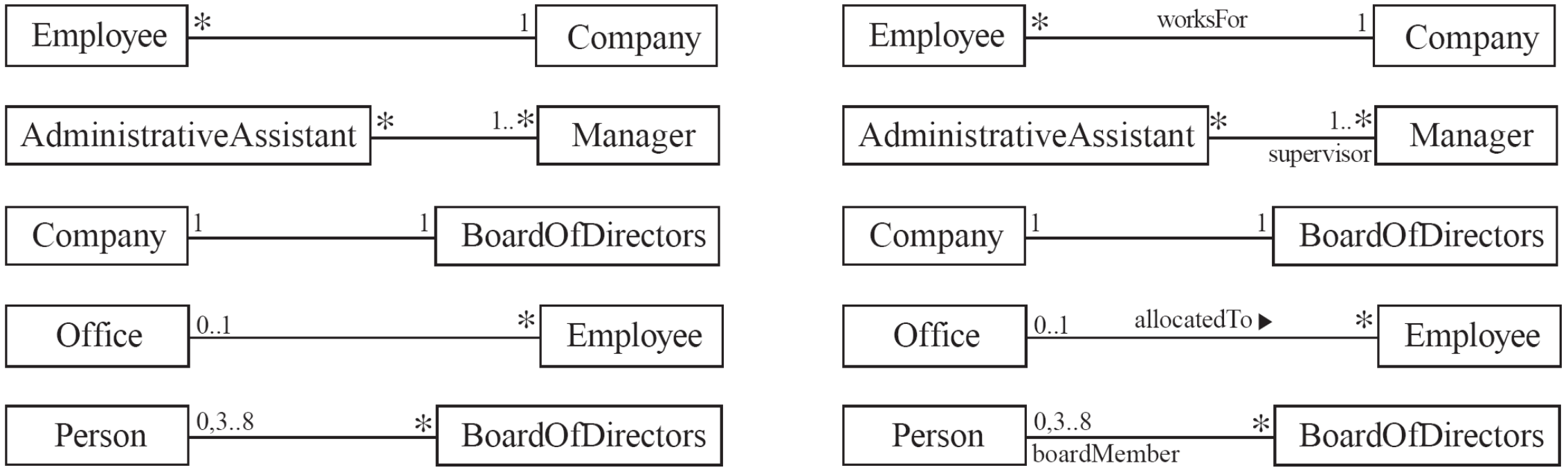
Dr. Dhungana
Dr. Klikovits
SS2024 • Krems

Associations and Multiplicity

An association is used to show how two classes are related to each other

Symbols indicating multiplicity are shown at each end of the association

Each association can be labelled, to make explicit the nature of the association

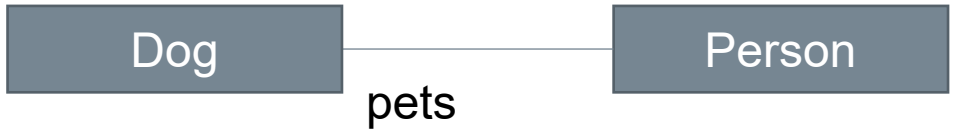


Association between Classes

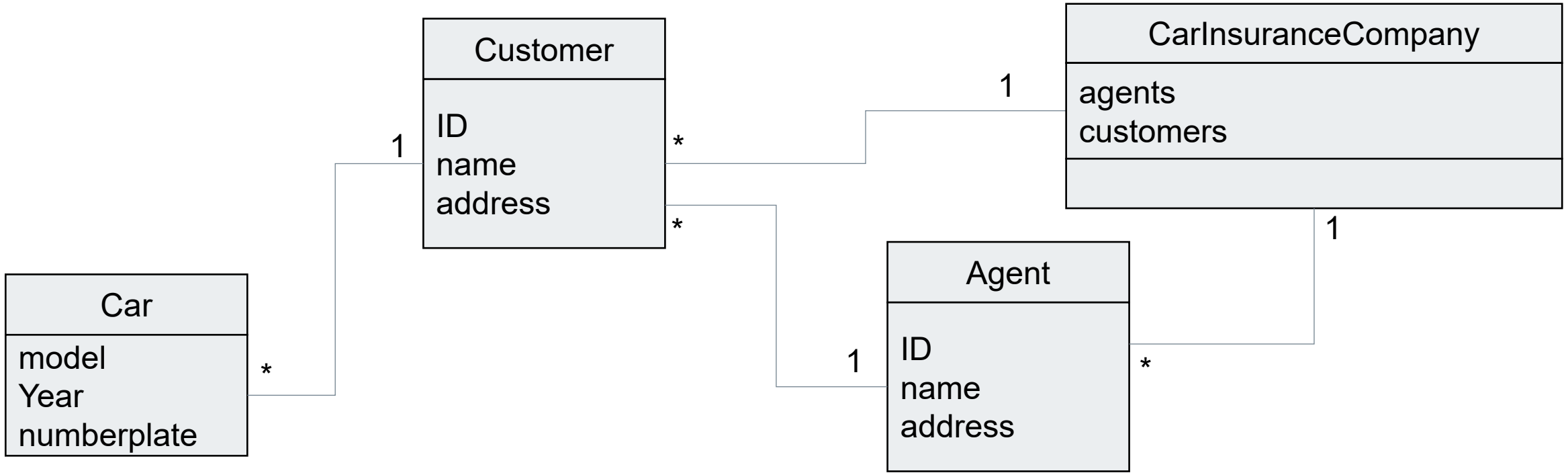
```
class Person():
    def __init__(self, name):
        self.name = name
        self.pets = []

    def addPet(self, pet):
        self.pets.append(pet)
```

```
class Dog:
    def __init__(self, name, color):
        self.name = name
        self.color = color
    def bark(self):
        print ("I am", self.color, self.name)
```

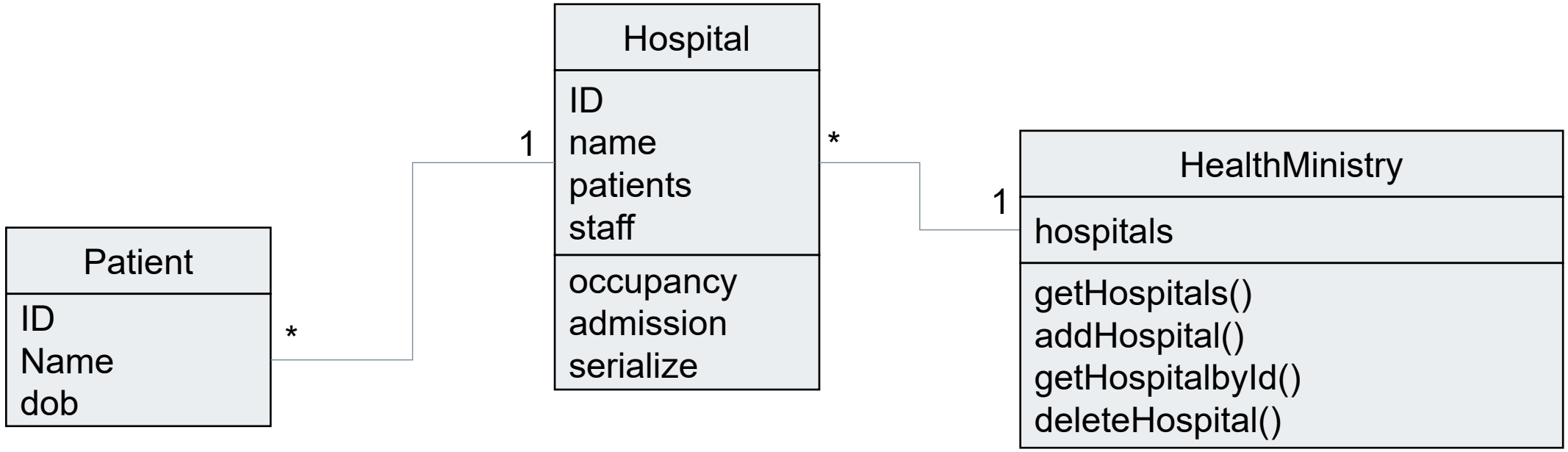
```
jessie = Person ("Jessie")
lassie = Dog("Lassie", "white")

jessie.addPet(lassie)

print (jessie.pets)
```



Example: A Car Insurance Company



Example: Hospital Management System



Special Method: str

`__str__()`: Define the string representation of the object str(obj)

```
class Exam:
   def __init__(self, name):
      self.name = name
e = Exam ("Prog2")
print (e)
```

<__main__.Exam object at 0x0000020CFCA7EDA0>

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

   def __str__(self):
      return "Exam: " + self.name

e = Exam ("Prog2")
print (e)
```

Exam: Prog2

Classroom Exercise

Given in a class Student

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

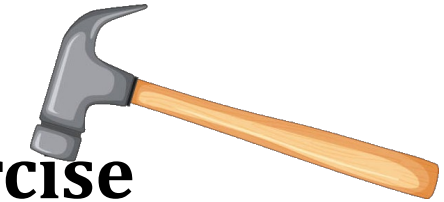
Change the class Student, so that the a proper string representation of the object is printed.

```
s1= Student ("Sandy", "24.01.1992") # name, dob
s2= Student ("Spili", "14.10.1993") # name, dob
s3= Student ("Waile", "04.06.1994") # name, dob

for s in (s1, s2, s3):
   print (s)
```

```
<__main__.Student object at 0x0000020BCD4A8FD0>
<__main__.Student object at 0x0000020BCD4AECF8>
<__main__.Student object at 0x0000020BCD4AECC0>
```

Change to

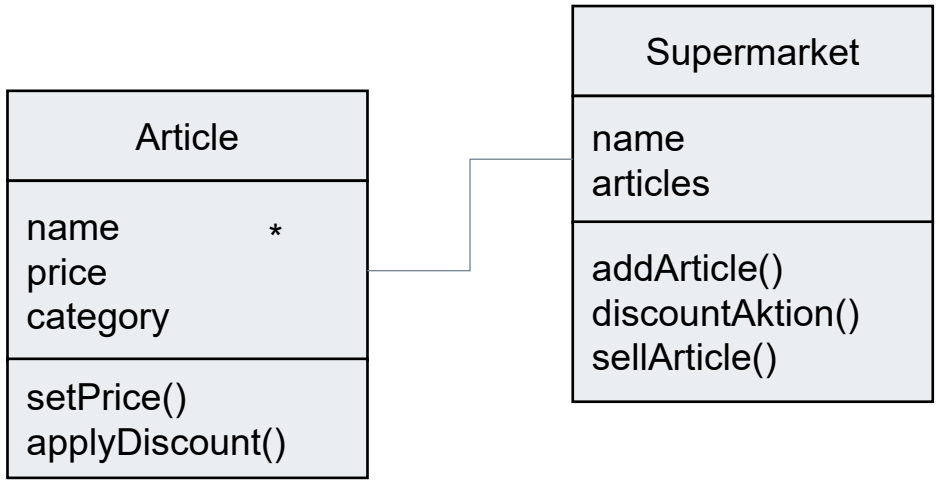
```
Sandy. DOB: 24.01.1992
Spili. DOB: 14.10.1993
Waile. DOB: 04.06.1994
```



Exercise

Task 2.1 Supermarket

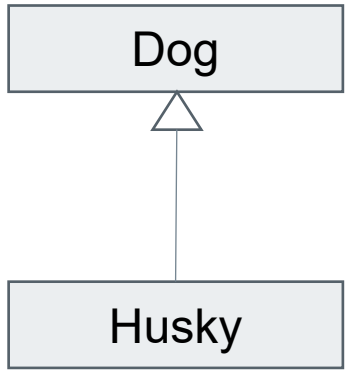
Write Python Code corresponding to this class diagram and the test code




```
a1 = Article ("Fresh Soap 200g", "hygiene")
a2 = Article ("Rose Shampoo 200 ml", "hygiene")
a3 = Article ("Coal Toothpaste 50g", "hygiene")
a4 = Article ("Mango 1 Pc", "fruits")
a5 = Article ("Orange 1 Kg", "fruits")
a6 = Article ("Apple 1 Kg", "fruits")

mk = Supermarket("Happymarket")

for a in (a1, a2, a3, a4, a5, a6):
    a.setPrice( random.randint(1,320))
    mk.addArticle(a,random.randint(1,40))

print (mk) # prints all items in supermarket

mk.discountAktion(0.25, "fruits" ) # 25% discount on fruits

print (mk) # prints all items in supermarket
```

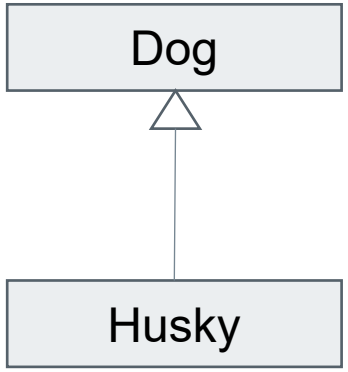
Inheritence

- When we make a new class - we can reuse an existing class and inherit all the capabilities of an existing class and then add our own little bit to make our new class
- Another form of store and reuse
- Write once - reuse many times
- The new class (child) has all the capabilities of the old class (parent) - and then some more



Subclasses are created by inheritence

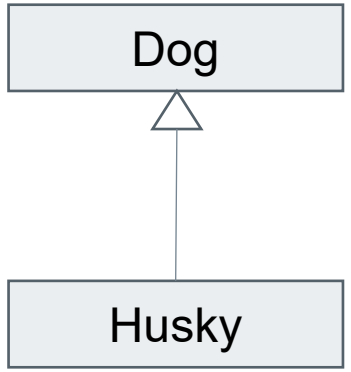
- A class can extend the definition of another class
- Allows use (or extension) of methods and attributes already defined in the previous one.
- New class: **subclass**.
- Original: parent, ancestor or **superclass**
- To define a subclass, put the name of the superclass in parentheses after the subclass's name on the first line of the definition.
- Python has no 'extends' keyword like Java.
- Multiple inheritance is supported, its use is not recommented.




```
class Dog:
    pass
```

```
class Husky(Dog):
    pass
```

Redefining (0verriding) Methods in Subclasses



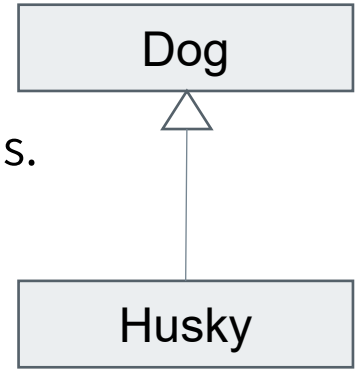
- To redefine a method of the parent class, include a new definition using the same name in the subclass.
- The old code won't get executed.
- To execute the method in the parent class in addition to new code for some method, explicitly call the parent's version of the method.
- **parentClass.methodName(self, a, b, c)**
- The only time you ever explicitly pass 'self' as an argument is when calling a method of an ancestor.

```
class Husky(Dog):
    def bark(self):
        print ("I am", self.name, "and I love snow")
```

```
h1 = Husky("Hus1", "white")
h1.bark()
```

```
class Husky(Dog):
    def bark(self):
        # calling the bark method in the super class
        Dog.bark(self)
        print ("I am", self.name, "and I love snow")
```

Overriding Constructors



- Same as for redefining any other method…
- Commonly, the ancestor's __init__ method is executed in addition to new commands.
- You'll often see something like this in the __init__ method of subclasses:
- **parentClass.__init__(self, x, y)**
  where parentClass is the name of the parent's class.

```
class Husky(Dog):
   def __init__(self, name, color, strength):
      Dog.__init__(self, name, color)
      self.strength = strength
```

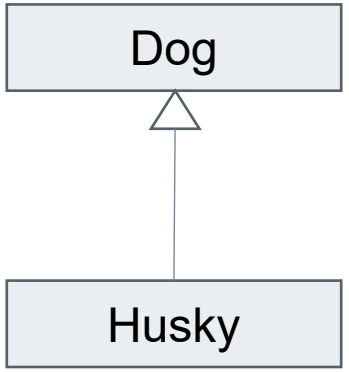
Additional Attribute

Is an object instance of a class?

- The isinstance() function returns True if the specified object is of the specified type, otherwise False

isinstance(object, type)

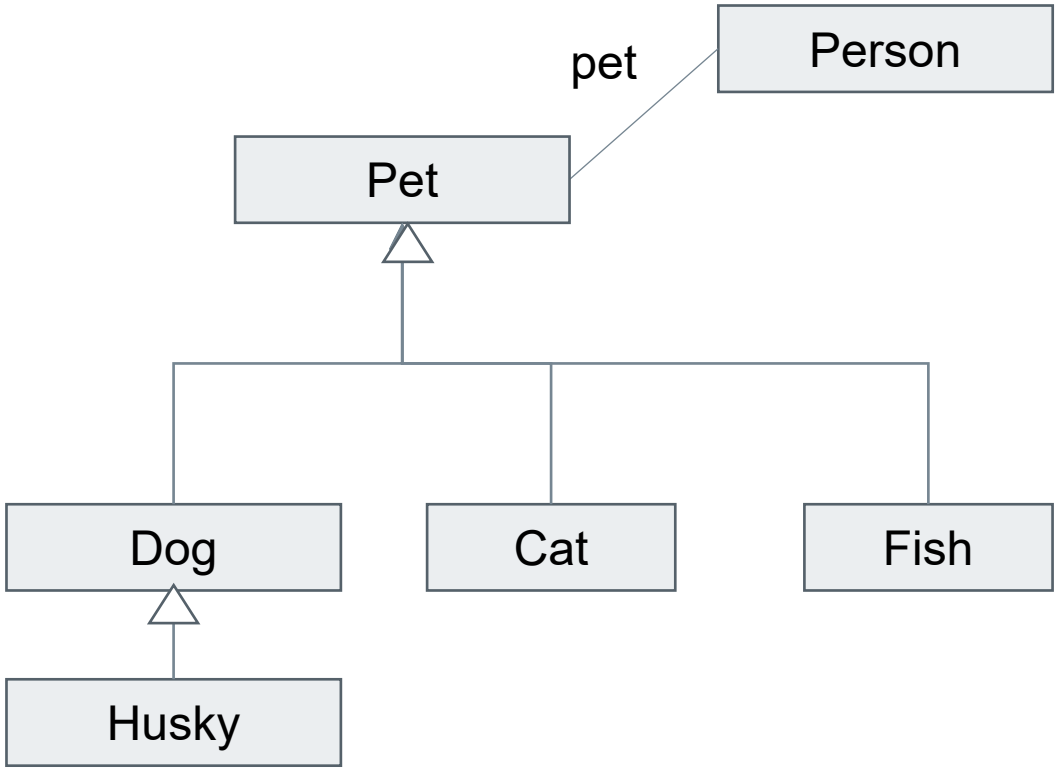
```
h1 = Husky("Hus1", "white", 45)
lassie = Dog("Lassie", "white")


isinstance(h1,Dog)
isinstance(lassie,Dog)
isinstance(h1,Husky)
isinstance(lassie,Husky)
```

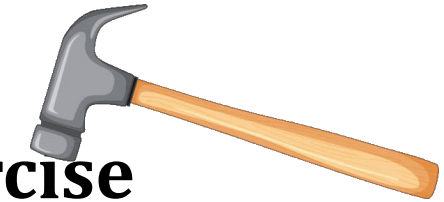


Isinstance vs. issubclass

- The issubclass() function checks if the object argument (first argument) is a subclass of classinfo class (second argument).

```
issubclass(Husky, Dog)
issubclass(Dog, Husky)
issubclass(Dog, Dog)
issubclass(Dog, Pet)
issubclass(Husky, Pet)
issubclass(Person, Husky)
issubclass(Person, Pet)
```

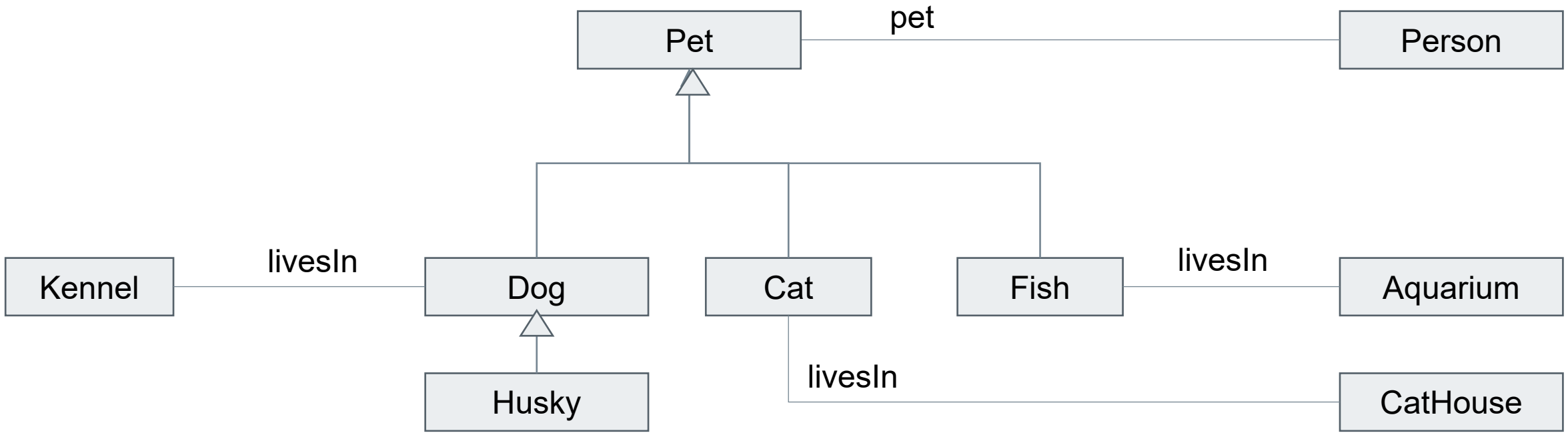




Exercise

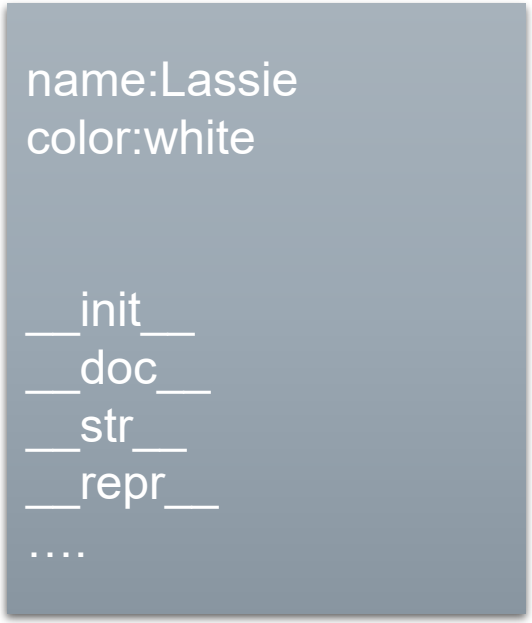
Task 2.2 Animals everywhere

Implement the given class diagram in Python. Write minimal lines of code to reflect the information contained in the class diagram **only**.



Built-in members of a class

- Classes contain many methods and attributes that are included by Python even if you don’t define them explicitly.
- Most of these methods define automatic functionality triggered by special operators or usage of that class.
- The built-in attributes define information that must be stored for all classes.
- All built-in members have double underscores around their names: __init__ __doc__

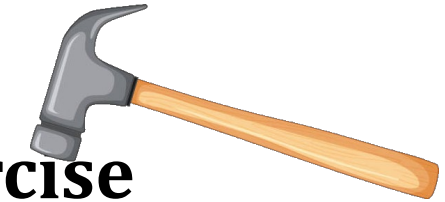


Special methods (aka Dunder or magic methods)

`__init__()`	: The constructor for the class
`__eq__()`	: Define how == works for class (c.f. `__ne__`, `__lt__`, `__gt__`,`__le__`, `__ge__`)
`__len__()`	: Define how **len(obj)** works
`__copy__()`	: Define how to copy a class
`__repr__()`	: Define the string representation of the object
`__str__()`	: Define the string representation of the object **str(obj)**
`__add__()`	: Define the meaning of the operator **+** on objects of this type
`__mul__()`	: Define the meaning of the operator * on objects of this type
`__pow__()`	: Define the meaning of the operator ** on objects of this type
`__getitem__()`	: Define how **obj[index]** works
`__contains__()`	: define the meaning of **in** operator

…

Exercise



Task 2.3 Dunder Methods

Given in a class Student.
Change the class Student, so that when objects of this type are added together, a new object of type Student is created, with the names concatenated

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

```
s1= Student ("Sandy") # name
s2= Student ("Spili") # name
s3= Student ("Waile") # name

print (s1+s2+s3)
```

Should print: Sandy+Spili+Waile

Special Data Items

These attributes exist for all classes.

__doc__ : Documentation string for class

__class__: Reference to the class from any instance of it

__module__ : Reference to the module in which the particular class is defined

__dict__:The dictionary that is actually the namespace for a class

…

Private and Public Attributes

- Any attribute/method with 2 leading under-scores in its name (but none at the end) is private and can't be accessed outside of class (__secret, __data)
- **Note**: Names with two underscores at the beginning and the end are for built-in methods or attributes for the class (__init__, __doc__)
- **Note**: There is no '**protected**' status in Python; so, subclasses would be unable to access these private data either.
- Actually, the private accessibility method is just a rule, not the limitation of compiler.
- The trick is to change name of private name like __variable or __function() to _ClassName__variable or _ClassName__function().
- So we can't access them because of wrong names.

```
class Exam:
   def __init__(self, name):
      self.name = name
      self.__participants = []


   def addParticipant(self, std):
      self.__participants.append(std)

e1 = Exam("")
```

```
In [18]: dir (e1)
Out[18]:
['_Exam__participants',
 '__class__',
 '__delattr__',
 '__dict__',
```

Object and Class Attributes/Methods

Object attributes/methods

- Variables/functions owned by a particular instance of a class
- Each instance has its own value for it

Class attributes

- Owned by the class as a whole
- All class instances share the same value for it
- Also called "**static**" variables/methods
- Good for (1) class-wide constants and (2) building counter of how many instances of the class have been made

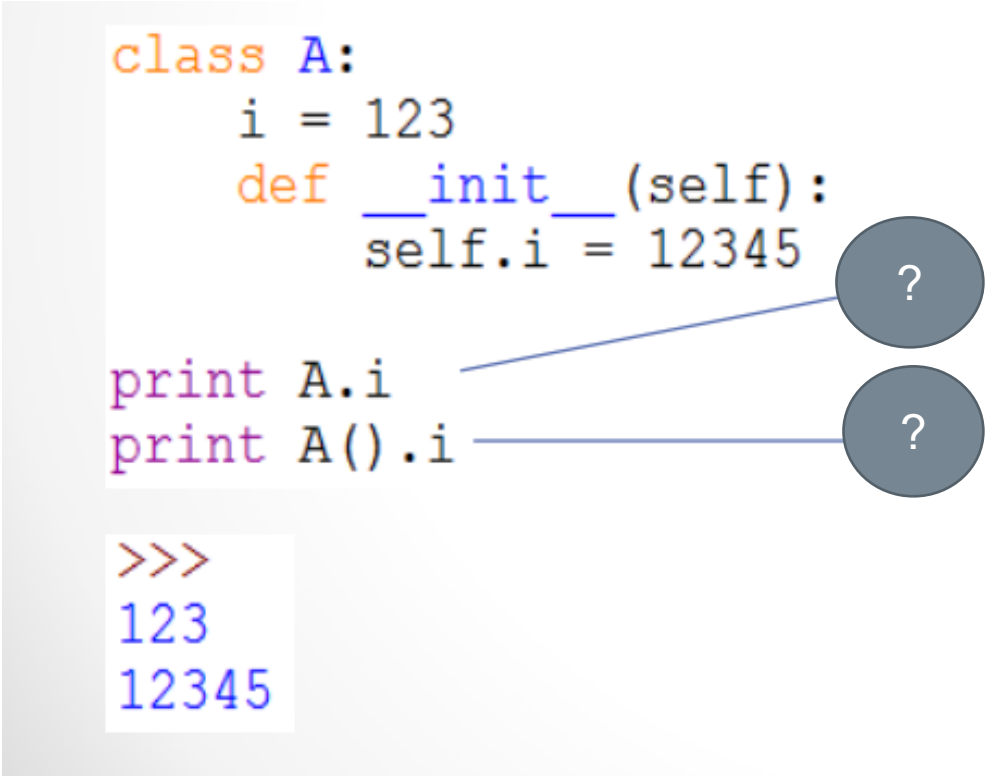
Class attribute: A single attribute that's associated with a **class** itself (not an instance!)

Static method: A method that's associated with a class itself
Class attribute could be used for counting the total number of objects instantiated, for example
Static methods often work with class attributes

Creating a class attribute

- Assignment statement in class but outside method creates class attribute
- Assignment statement executed only once, when Python first sees class definition
- Class attribute exists even before single object created
- Can use class attribute without any objects of class in existence

```
class Dog():
   total = 0 #creates class attribute total set to 0
```



Creating and calling class methods

decorator classmethod!

```
class Dog():
 total = 0

 @classmethod
 def status(cls):
   print ("Total dogs created:", Dog.total )


status = Dog.status()
print (Dog.total)
```

- A class method is a method that is bound to a class rather than its object.
- The class method can be called both by the class and its object.
- But no matter what, the class method is always attached to a class with the first argument as the class itself **cls**.

Accessing unknown members

Problem: Occasionally the name of an attribute or method of a class is only given at run time…

Solution:

getattr(object_instance, string)

`string` is a string which contains the name of an attribute or method of a class

getattr(object_instance, string) returns a reference to that attribute or method

```
f = student("Bob Smith", 23)
getattr(f, "full_name")
getattr(f, "get_age")
getattr(f, "get_age")() # call it
getattr(f, "get_birthday")# Raises AttributeError – No method!
```

```
f = student("Bob Smith", 23)
hasattr(f, "full_name")
hasattr(f, "get_age")
hasattr(f, "get_birthday")
```

Use hasattr to avoid runtime errors, when accessing unknown members

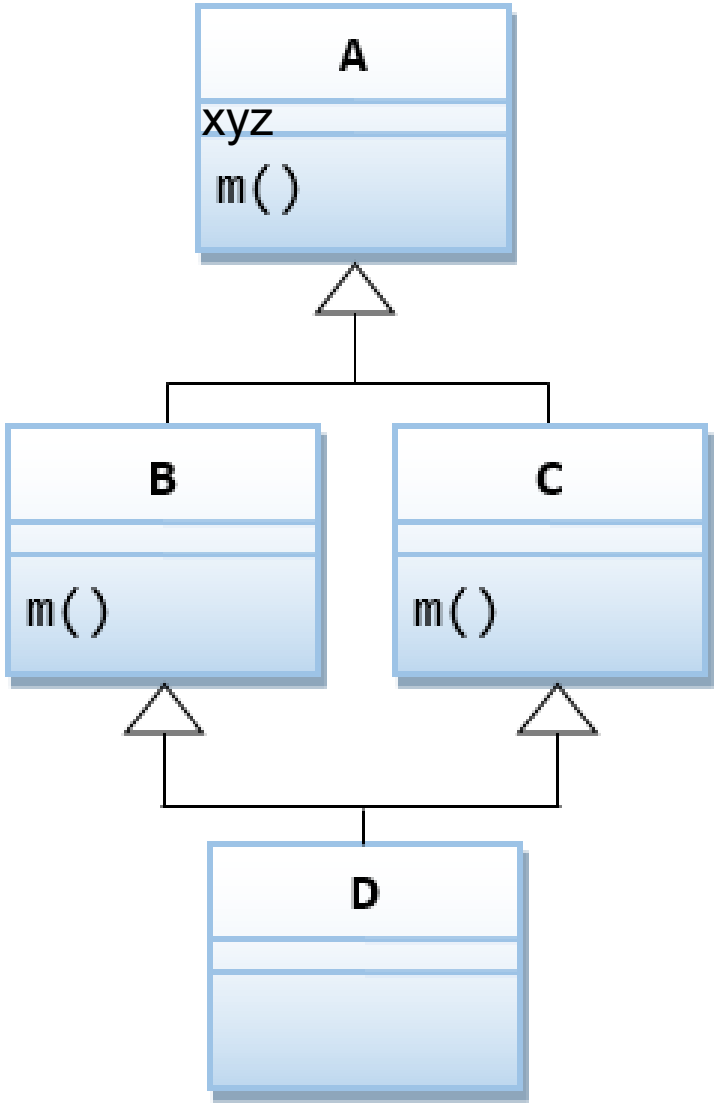
Multiple Inheritence (possible but not recommended)

If an attribute is not found in **D**, it is searched in **B**, then recursively in the classes of **C**, and only if it is not found there, it is searched in **A**, and so on.

Method Resolution Order (MRO)

https://medium.com/@__hungrywolf/mro-in-python-3-e2bcd2bd6851

```
class A :
   def __init__(self):
      self.xyz = 123

class B(A):
   pass

class C (A):
   pass

class D (B, C):
   pass

d = D()
print (d.xyz )
```

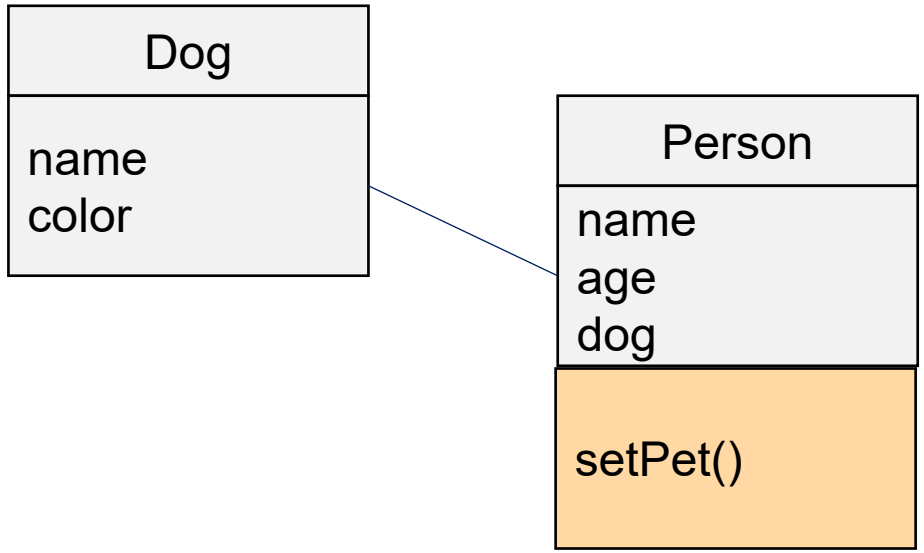


Methods

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

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

    def setPet(self, d):
        self.dog = d
```

```
p = Person ("Mr. Smith")
d = Dog("Lassie")
p.setPet(d)

print (d)
print(p.dog)
```

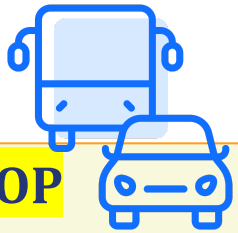



```
<__main__.Dog object at 0x000002AC4030E978>
<__main__.Dog object at 0x000002AC4030E978>
```

Summary

- Object-oriented Programming (OOP) is a methodology of programming where new types of objects are defined
- An object is a single software unit that combines attributes and methods
- An attribute is a “characteristic” of an object; it’s a variable associated with an object (“instance variable”)
- A method is a “behavior” of an object; it’s a function associated with an object
- A class defines the attributes and methods of a kind of object
- Each instance method must have a special first parameter, called self by convention, which provides a way for a method to refer to object itself
- A constructor is a special method that is automatically invoked right after a new object is created
- A class attribute is a single attribute that’s associated with a class itself
- A static method is a method that’s associated with a class itself

Try at Home



Task 2.4 Shapes

Write a Python program to create a class that represents a shape. Include methods to calculate its area and perimeter. Implement subclasses for different shapes like circle, triangle, and square.

Task 2.5 Shopping cart

Write a Python program to create a class representing a shopping cart. Include methods for adding and removing items, and calculating the total price.

Task 2.6 More OOP

- Create a **Vehicle** class with a method fare().
- Create a child class **Bus** and a child class **Car** that will inherit all of the variables and methods of the Vehicle class. Give the capacity argument of **Bus.seating_capacity() a default value of 50. Car.seating_capacity() has a default value of 5.**
- Define a **class** attribute"**color**" with a default value **white**. i.e., Every Vehicle should be white.
- The default fare charge of any vehicle is **seating capacity * 100**. If Vehicle is **Bus** instance, we need to add an extra 10% on full fare as a maintenance charge. So total fare for bus instance will become the **final amount = total fare + 10% of the total fare.** You need to override the fare() method of a Vehicle class in Bus class.