# Object Oriented Programming

## Polymorphism and abstract classes

Jolan Philippe

March 8th, 2024

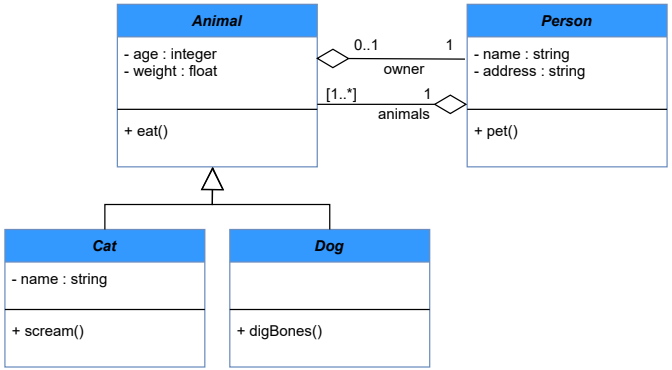
IMT Atlantique

# Last times

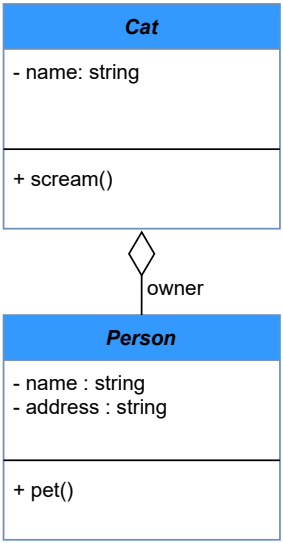
# Representing objects with UML

## The Unified Modeling Language (UML)

- Standard way to visualize a system
- Two concepts:
  - Inheritance: to specialize a class into sub-classes
  - Aggregation: to compose classes



# UML to Python code




```python
class Person:

    def __init__(self, name, address):
        self.name = name
        self.address

class Cat(Animal):

    def __init__(self, name, age, weight,
        person):
        Animal.__init__(self, age, weight)
        self.name = name
        self.owner = person
```

# More behavioral functions

```
1  class Animal:
2
3      def __init__(self, name, owner):
4          self.__name = name
5          self.__owner = owner
6
7      def getName(self):
8          return self.__name
9
10     def getOwner(self):
11         return self.__owner
12
13     def __str__(self):
14         return f"{self.name} (owned by {self.owner})"
```

# More behavioral functions

```
1  class Animal:
2      ...
3
4      def __copy__(self):
5          return Animal(self.__name, self.__owner)
6
7      def __deepcopy__(self):
8          copied_name = copy.deepcopy(self.__name)
9          copied_owner = copy.deepcopy(self.__owner)
10         return Animal(copied_name, copied_owner)
11
12     def __eq__(self, other):
13         if isinstance(other, Animal):
14             return (self.getName() == other.getName()
15                 and self.getOwner() == other.getOwner())
16         return False
```

# Encapsulation and visibility

**Visibility**

A visibility can be defined for each attribute and each function of a class.

- **public**: Accessible from everywhere
- **private**: Accessible from anywhere else than in the class definition
- **protected**: Accessible in the same module (class and subclasses)

# In Python

```
1  class MyClass:
2
3      def __init__(self, value):
4          self.__value = value
5          self.__elements = []
6
7      def getValue(self):
8          return self.__value
9
10     def setValue(self, new_value):
11         self.__value = new_value
12
13     def addElements(self, element):
14         self.__elements.append(element)
```

# Final notions

Two last points for this class

- Abstract classes
- Polymorphism

# Abstract class

Cannot be instantiated !

## Pros

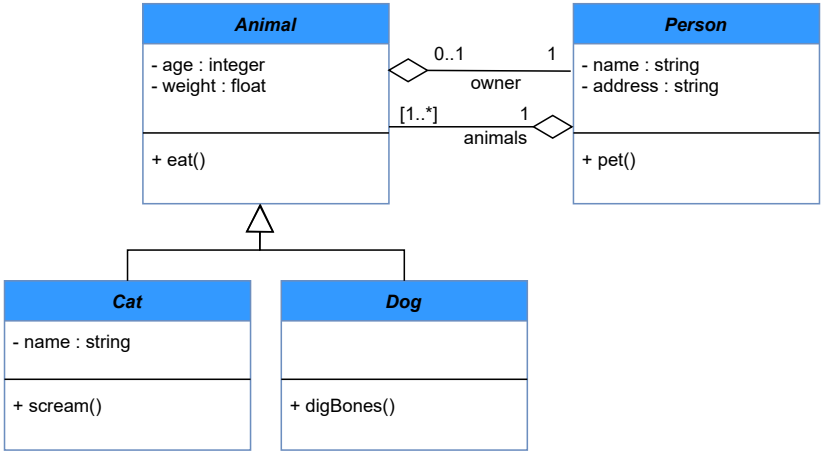
- Give a general definition, guidelines, for subclasses
- Give a list of needed functions without their implementations
- Give a default implementation for functions

## In Python

- An abstract class extends `abc.ABC`
- If a class extends an abstract class, it must implements non-defined functions

# Example

Animal can be abstracted



# Python

```
1  import abc
2
3  Class Animal (abc.ABC):
4
5      def __init__(self, age, weight):
6          ...
7
8      def eat():
9          print("Eat")
10
11 Class Cat(Animal):
12
13     def __init__(self, age, weight, name):
14         ...
15
16     def eat(self):
17         print(f"The cat {self.name} eats")
18
19     def scream(self):
20         print("Grr")
```

# Polymorphism

**General definition**

Two functions, with the same name, but a different behavior

**In Python**

- Two functions with the same name, but in different class
- Two functions with the same name, same parameters, but in sub-classes
- Two functions with the same name, in the same class, but with different parameters

# Python: Two functions with the same name, but in different class

```
1  class Person:
2
3      def __init__(self):
4          ...
5
6      def eat(self):
7          print("Person is eating")
8
9  class Animal:
10
11     def __init__(self):
12         ...
13
14     def eat(self):
15         print("Animal is eating")
```

# Python: Two functions with the same name, same parameters, but in sub-classes

```
class Person:

    def __init__(self):
        ...

    def eat(self):
        print("Person is eating")

class MVP(Person):

    def __init__(self):
        ...

    def eat(self):
        print("The MVP is eating")
```

# Python: Two functions with the same name, in the same class, but with different parameters

```
1  class Person:
2
3      def __init__(self):
4          ...
5
6      def eat(self):
7          print("Person is eating")
8
9      def eat(self, food):
10         print(f"Person is eating {food}")
```

# Exercices

# Small project

By group, you will need to create a small project to illustrate all the points we have covered in this class: inheritance, aggregation, class/object attributes, visibility, ...

- 2 groups: 3-4 students
- 2 subjects:
  - Represent your class, with students, teachers, courses, etc. with functions to manage it (add, remove, etc.)
  - Represent a bank, with clients, several kind of account, cards (debit or credit) etc. with functions to manage it (add, remove, etc.)
- Additional subject ? You are open

1. Make a UML diagram (45min)
2. Write Python code for it (45min)
3. Presentation (10min each group)