# Object Oriented Programming

## Encapsulation

Jolan Philippe

March 4th, 2024

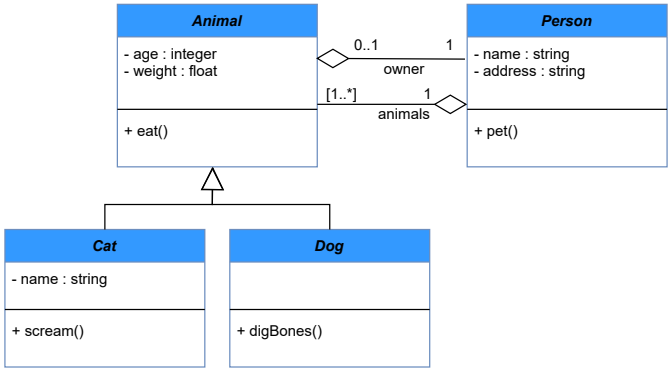
IMT Atlantique

# Last time

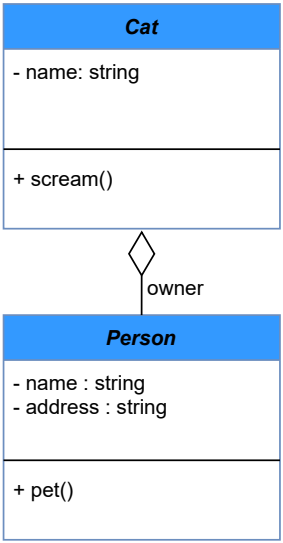
# Representing objects with UML

## 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



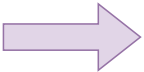
# UML to Python code

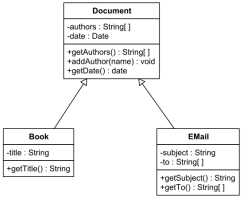



```python
class Person:

    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
```

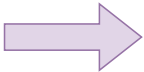
# Mechanized process

A specification describing data






A visual model understandable by everybody




A code representation of the model (to be used for programming)

# Practicing with additional examples: Library

**Specification for a Library**

A **Library** has shelves. On each **Shelf**, there are **Book**s. A book is characterized by a title, an author, and a release date. We want to be able to add a book to library, by specifying the shelf number.

1. Create a UML model for this specification
2. Create the corresponding classes in Python
3. Make instances and test your code

# Practicing with additional examples: Family

**Specification for a Family**

A **Family** is identified with a name, and is composed by family members, who are **Person**s. Each **Person** has a firstname, an age, an height and a favorite color. Then, a **Person** has a **Gender** (for instance **Male**, **Female**, **Other**). A family can also has **Animal**s, which could be a **Cat**, a **Dog** or a **Rabbit**. An animal has a name, an age, and a specie. All animals can eat foot. Then animals make a different kind of scream.

1. Create a UML model for this specification
2. Create the corresponding classes in Python
3. Make instances representing your family

# One step closer to encapsulation

Choose one of the previous examples. Select one class with at least 2 attributes.

- For each **attribute** of the selected class, write one function named **get_attribute** returning the value of this attribute.
- For each **attribute** of the selected class, write one function named **set_attribute**, taking as argument a new value, and set the value of this attribute with this argument's value.

# Encapsulation

# Class attributes vs. Instance (ie Objects) attributes

## Object attributes

Previously, we accessed attributes from an instance of a class (i.e., an object)

```
1  class MyClass:
2
3      def __init__(self, value):
4          self.attribute = value
5
6      def getAttribute(self):
7          return self.attribute
```

# Class attributes vs. Instance (ie Objects) attributes

## Class attributes

Let's see how we can access attributes from a class itself

```
1 class Animal:
2     currentId=0
3
4     def __init__(self):
5         self.id = Animal.currentId
6         Animal.currentId = Animal.currentId + 1
```

It is dangerous to not hide currentId. Any external code can modify it

# Protecting internal content of classes

## Visibility

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

- **public**
  - For an attribute: everybody can use and modify the value of the attribute, even from outside the class definition.
  - For a function: everybody can call the function, even from outside the class definition.
- **private**
  - For an attribute: can only be used and modified within the class definition
  - For a function: can only be called within the class definition
- **protected**
  - Usage in the same module only.

# Python code

```
class Animal:

    def __init__(self, name, age, weight):
        self.public_attribute = name
        self.__private_attribute = age # __ before the name
        self._protected_attribute = weight # _ before the name

    def public_function(self):
        ... # Can be used from any program

    def __private_function(self):
        ... # This function can only be used in the class definition

    def _protected_function(self):
        ... # Can be used in the code of the same module
```

# Protecting internal content of classes

**Class as black box, to control usage**

- Avoid accidental usage
- Protect internal data
- Simplifies interface for users
- Facilitates inheritance

# Private attributes ⇒ Accessors

Let is consider all our attributes private... How can we access some of them externally ?

**Getter**

One function for attribute you want to expose. The function returns the value of the attribute

**Setter**

One function for attribute you want to be modifiable. The function takes as argument the new value. You can control the modification (for example: check validity of new value)

# Python code

```
1  class Animal:
2
3      def __init__(self, age):
4          if age < 0:
5              raise Exception ("Age cannot be negative")
6          self.__age = age
7
8      def getAge():
9          return self.__age
10
11     def setAge(new_age):
12         if new_age < 0:
13             raise Exception ("Age cannot be negative")
14         self.__age = new_age
```

# Copying and Deep Copying Objects in Python

- In Python, copying objects is a common operation, but it's important to understand the difference between shallow copy and deep copy.

## Shallow Copy

- Shallow copy creates a new object and inserts references to the objects found in the original.
- Changes made to the original object's elements affect the copied object and vice versa.
- Python provides a built-in 'copy()' method and 'copy' module to perform shallow copy.

## Deep Copy

- Deep copy creates a new object and recursively inserts copies of the objects found in the original.
- Changes made to the original object's elements do not affect the copied object.
- Python provides a built-in 'deepcopy()' method from the 'copy' module to perform deep copy.

# Example

```python
import copy

list = [1, 2, [3, 4]]
copy_list = copy.copy(list)

copy_list[2][0] = 'x'

print(list)  # Output: [1, 2, ['x', 4]]
print(copy_list)  # Output: [1, 2, ['x', 4]]

# ------------------------------------------

list = [1, 2, [3, 4]]
copy_list = copy.deepcopy(list)

copy_list[2][0] = 'x'

print(list)  # Output: [1, 2, [3, 4]]
print(copy_list)  # Output: [1, 2, ['x', 4]]
```

# Copying objects

```
1  class Animal:
2
3      def __init__(self, name, owner):
4          self.name
5          self.owner
6
7      def __str__(self):
8          return f"{self.name}"
9
10     def __copy__(self):
11         return Animal(self.name, self.owner)
12
13     def __deepcopy__(self):
14         copied_name = copy.deepcopy(self.name)
15         copied_owner = copy.deepcopy(self.owner)
16         return Animal(copied_name, copied_owner)
```

# Exercise

Let's start from previous code:
`https://jolanphilippe.github.io/course/docs/24-oop/country.py`

1. Change the visibility to private for each attribute, and make accessors accordingly: getters and setters
2. Add an id to the cities, such that all the cities have a different one when created
3. Create a class function **areTheSameCities**(A,B) that return True if the cities are the same, False otherwise. Test your method with the previously created cities (Paris, Nantes, Orleans).
4. Create a class function **getTheBestCity**(votes) returning the best city from votes. The entry "votes" is a table of "City". Test your function with the following votes: [Orleans,Nantes,Orleans,Nantes,Nantes,Nantes,Paris,Paris]
5. Change the capital city of France with the winner of the previous vote.