15-110 Principles of Computing – F19

Lecture 27:

Object-Oriented Programming

Teacher:
Gianni A. Di Caro

جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

Python *objects* so far

- Built-in object types:

```
int, float, bool
```

```
a = 2
b = 2 + 1
c = 3.5 * b
```

Information data (values) only

- Built-in object types:

```
def my_function(args):
    # does something
    return something
```

Operations (actions) only

- Built-in object types:

```
str, list, tuple, dict, set, file
```

```
l = []                       r = f.readline()
l.append(44)                 a = f.readlines()
l.append(22)                 f.seek(0)
l.extend([1,0,9,12])         f.close()
l.sort()
```

Information data (values)

+

Associated methods (operations) on the data

- Importing from modules and creating variables of *module type*

```
import csv
```

```
csv_data = csv.writer (f)
csv_data.writerow([12, 100, 5.5])
```

Class objects

- Built-in object types:

  `str, list, tuple, dict, set, file`

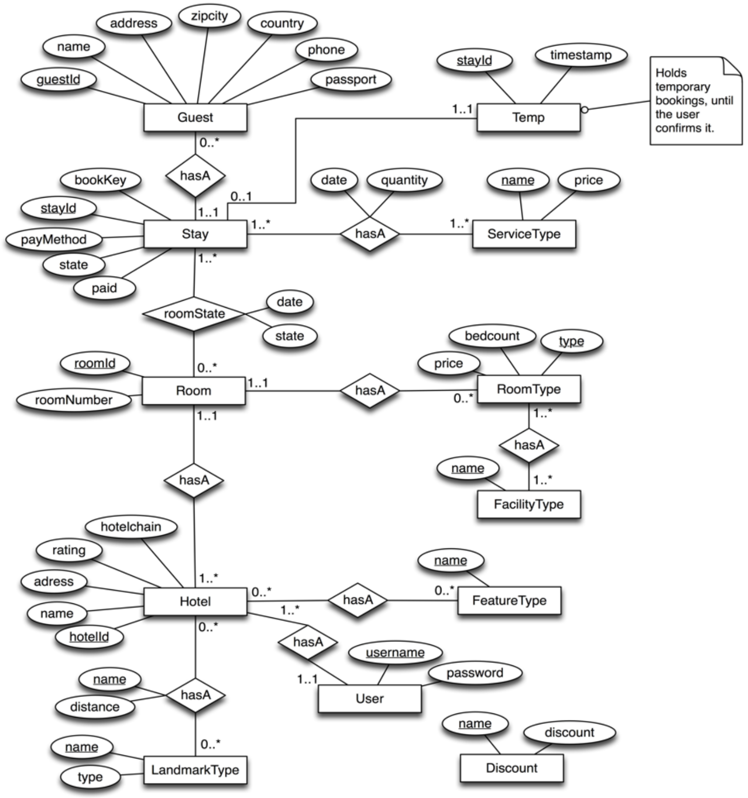
- ✓ Each data type comes with a set of specialized methods, identified through the *dot notation*, to manipulate the data type itself
  - List: `append(), insert(), extend(), remove(), pop(), count(), sort(), reverse(), copy(),` …
  - Dict: `get(), keys(), values(), items(), pop(), popitem(), clear(), update(), copy(),` …
  - …

- ✓ Each new *instance* of a data type has possibly different values, but provides the same set of methods

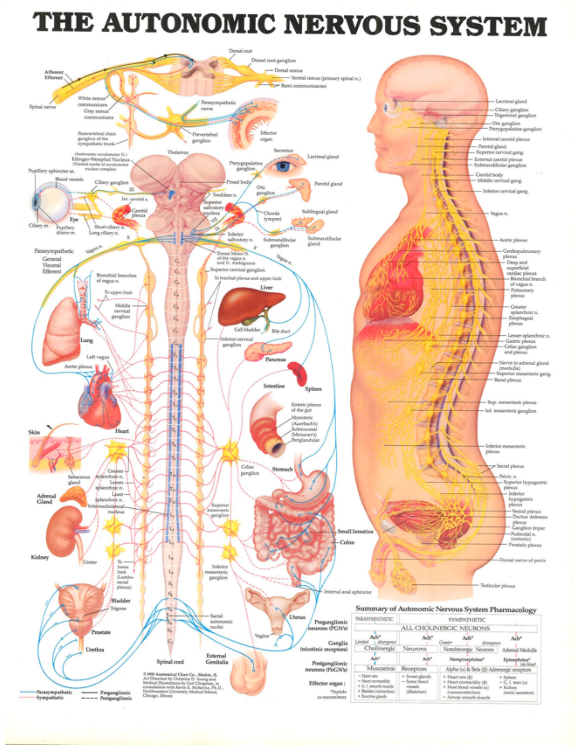
```
l1 = []
l2 = [1,2,3]
l1.append(99)
l2.append(99)
```

➢ Can we generalize and extend this approach to custom-defined types? → Yes, by using ***class* objects**

Real-world is complex, populated by complex, interacting entities



Holds temporary bookings, until the user confirms it.



How do we use software to model, study, manage, control such complex systems in a way which is effective, efficient, maintainable, reusable, extensible, scalable, adaptable, …?

- ❖ **Object-oriented** design and programming comes into help to tackle real-world complexity: it is based on the definition of ***classes***
- ✓ *Idea:* let's pack in the same object (the class) everything that we need for describing, use, and interact with a complex entity

Simple examples of the concept of a class

- A class is a template for describing and interact with specific instance objects

Class	Objects (Instances)
Fruit	Apple
	Cherry
	Mango

Class	Objects (Instances)
Car	Toyota LC
	Audi A3
	BMW 2

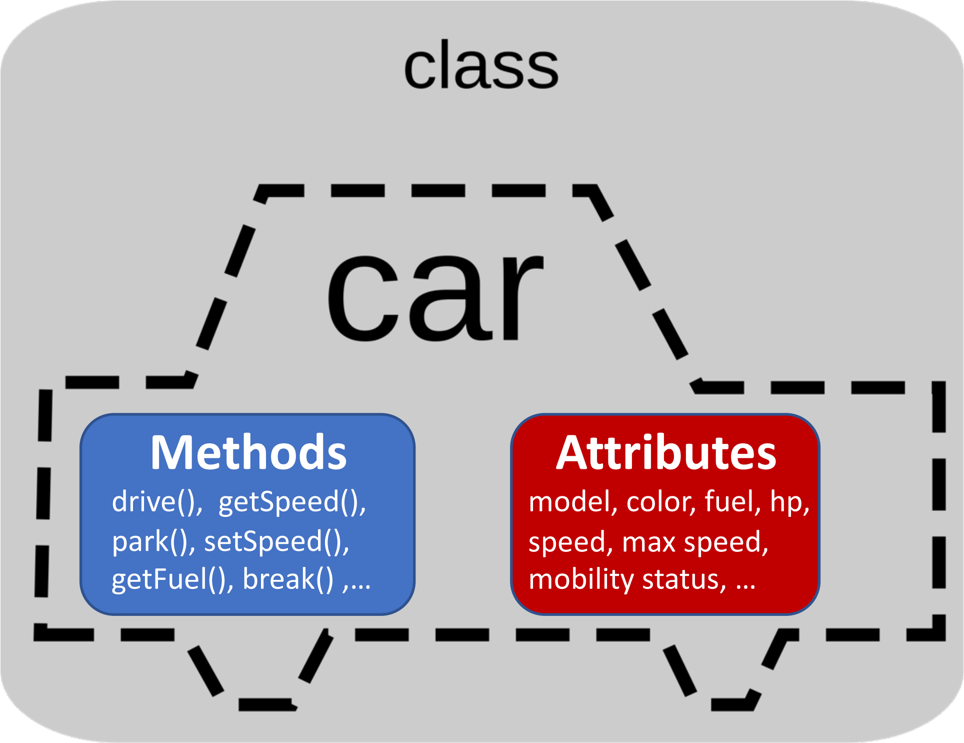
Class	Objects (Instances)
Country	Qatar
	France
	Italy
	India

- What are the **general characteristics** to consider for describing a fruit? What are the **actions** we want to make available to access and interact with a specific fruit object?
- Are the general characteristics and actions enough for dealing with *any specific fruit?*

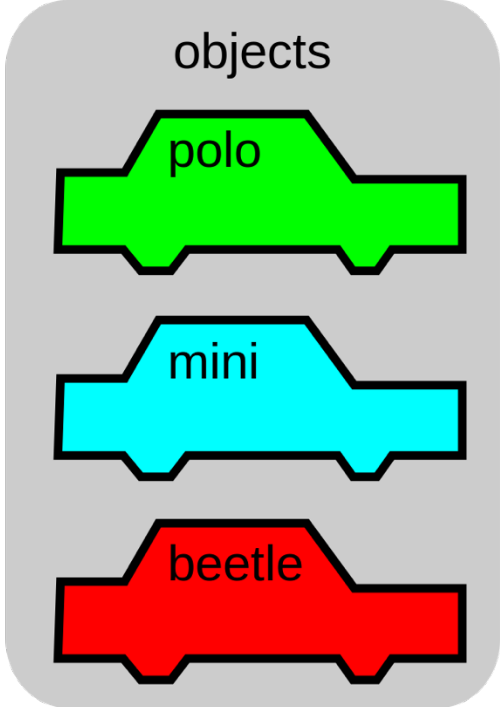
The blueprint analogy

- For instance, we want to make a software to be used in the context of road monitoring and management (whatever this could mean…)
- We first need to **represent a car**, since a road will feature many different cars, in different conditions of mobility, and in different locations

The blueprint analogy


class
car
Methods
drive(), getSpeed(), park(), setSpeed(), getFuel(), break() ,…
Attributes
model, color, fuel, hp, speed, max speed, mobility status, …

A class is a like a *blueprint*: a design guide, an operational pattern that can be used to create multiple, independent, **instances** of class objects



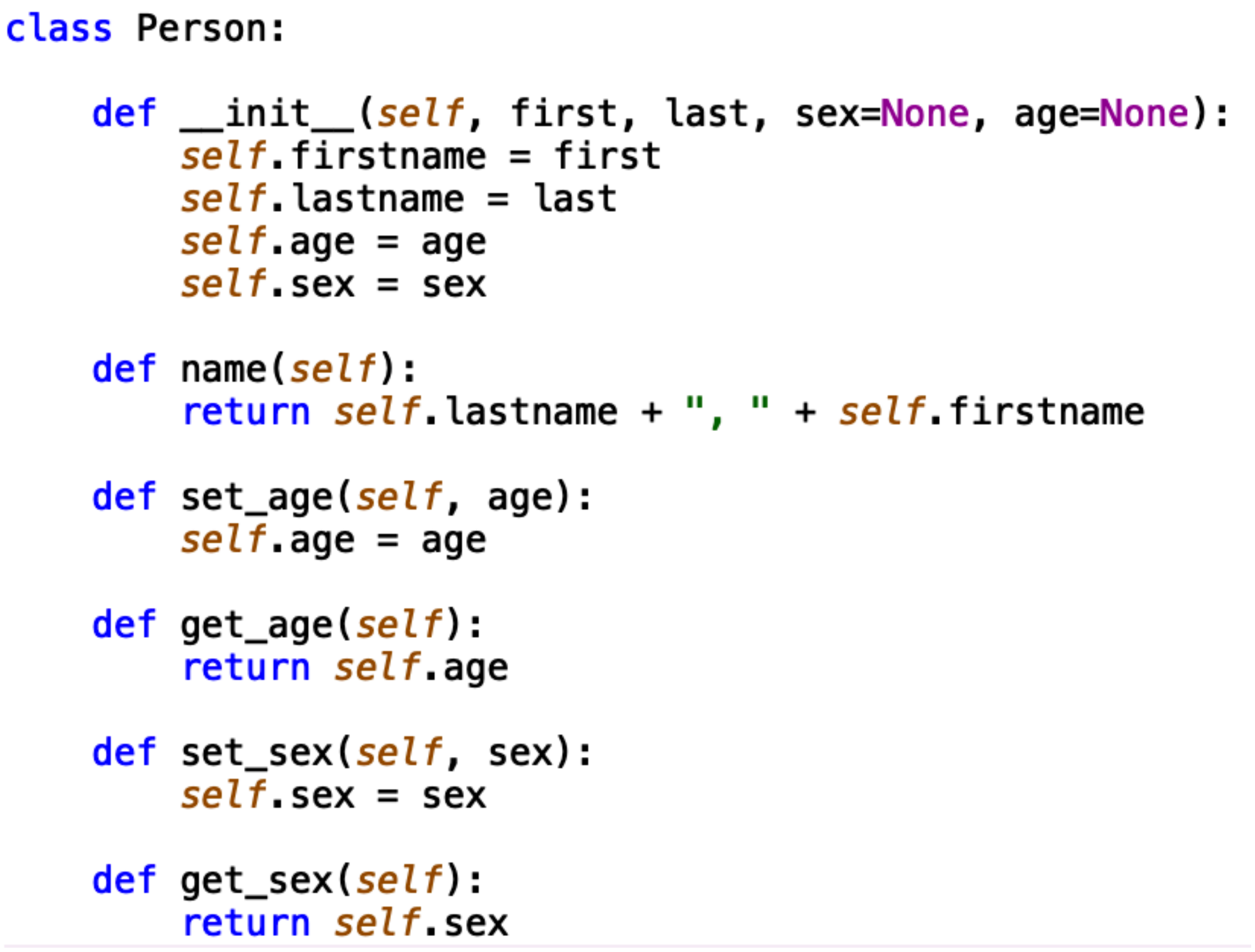
Different instances are created based on the same design: they are equipped with the same methods and attributes, but attributes might have different values, and methods might be specialized

Syntax for creating a class in python

```
class ClassName:

   def __init__(self, args):
      # code for object initialization

   def one_class_method(self, args):
      # code for this method

   def another_class_method(self, args):
      # code for this method
```

- `self`**: reference** (address, id) to the *specific class object*
- `__init__`: a *magic method*, it allows to **initialize** the attributes of the object, optional
- The class definition can have as many **methods** as we need
- Let’s adopt the convention ClassName to define the name of a class

A simple class to store and use information about a person



```
class Person:

    def __init__(self, first, last, sex=None, age=None):
        self.firstname = first
        self.lastname = last
        self.age = age
        self.sex = sex

    def name(self):
        return self.lastname + ", " + self.firstname

    def set_age(self, age):
        self.age = age

    def get_age(self):
        return self.age

    def set_sex(self, sex):
        self.sex = sex

    def get_sex(self):
        return self.sex
```

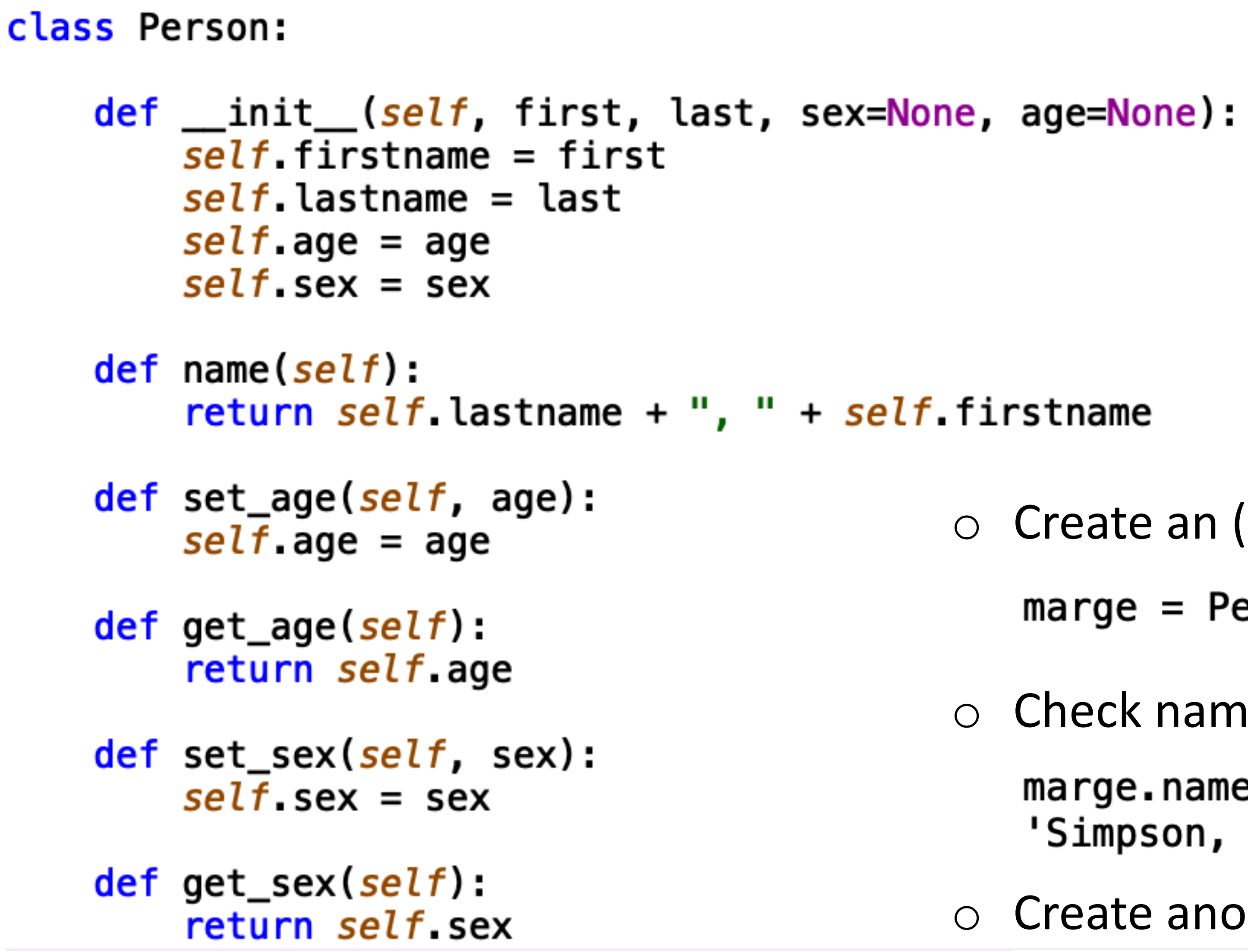
Class Attributes:

- `firstname`
- `lastname`
- `age`
- `sex`

Class Methods:

- `__init__()`
- `name()`
- `set_age()`
- `get_age()`
- `set_sex()`
- `get_sex()`

A simple class to store and use information about a person



```
class Person:

    def __init__(self, first, last, sex=None, age=None):
        self.firstname = first
        self.lastname = last
        self.age = age
        self.sex = sex

    def name(self):
        return self.lastname + ", " + self.firstname

    def set_age(self, age):
        self.age = age

    def get_age(self):
        return self.age

    def set_sex(self, sex):
        self.sex = sex

    def get_sex(self):
        return self.sex
```

○ Create an (instance) object of class `Person`

```
marge = Person('Marge', 'Simpson', 'F')
```

○ Check name attribute of object `marge`

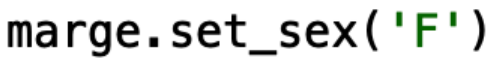
```
marge.name()
'Simpson, Marge'
```

○ Create another (instance) object of class `Person`

```
homer = Person('Homer', 'Simpson', 'M', 39)
```

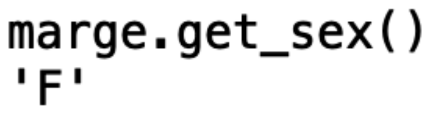
A simple class to store and use information about a person

- Change the value of an attribute:



```
marge.set_sex('F')
```

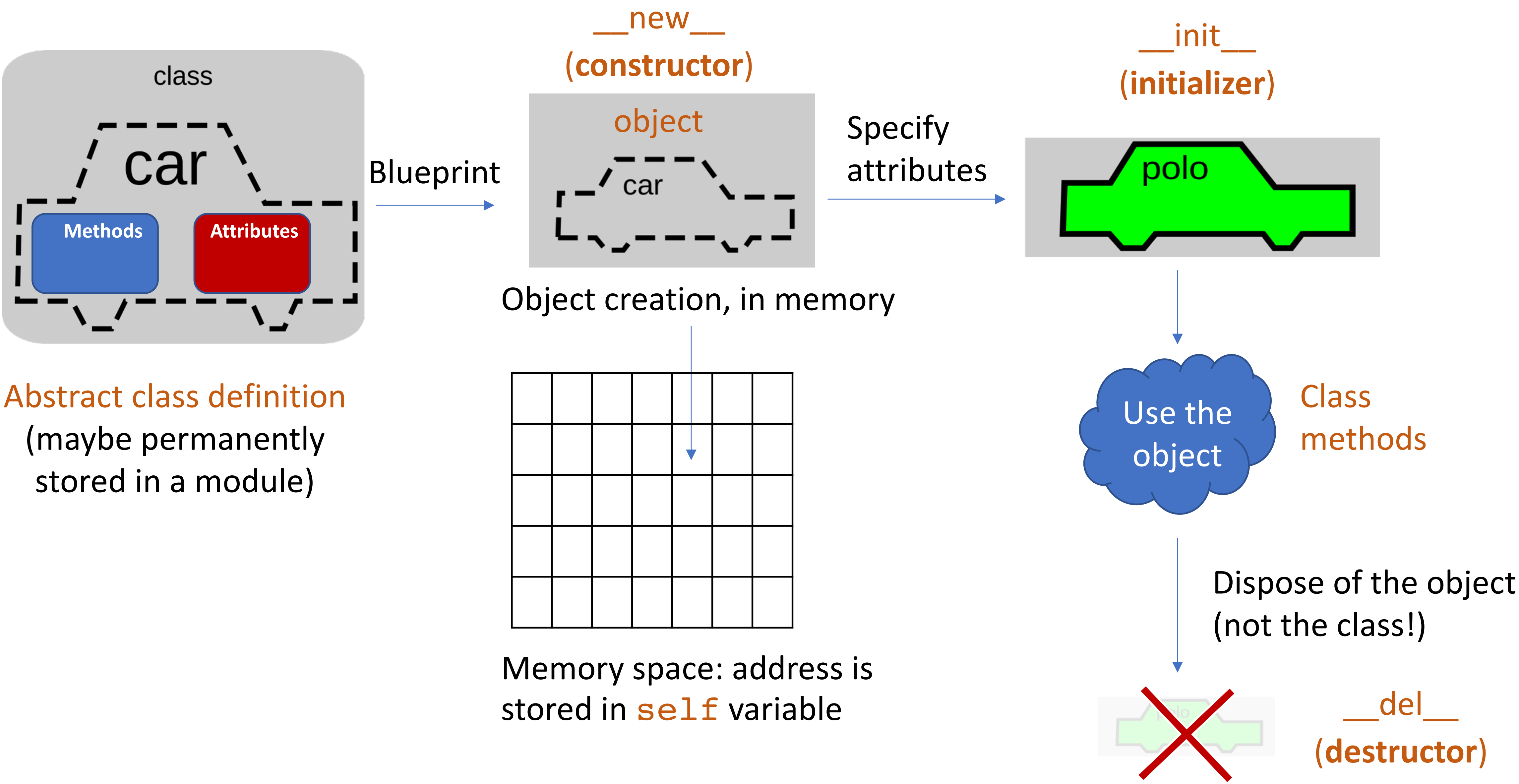
- Check the value of the attribute



```
marge.get_sex()
'F'
```

➢ Let’s **add** some useful attributes and methods!

The life cycle of a class object



Abstract class definition (maybe permanently stored in a module)

Blueprint

__new__ (**constructor**)

Object creation, in memory

Memory space: address is stored in `self` variable

Specify attributes

__init__ (**initializer**)

Class methods

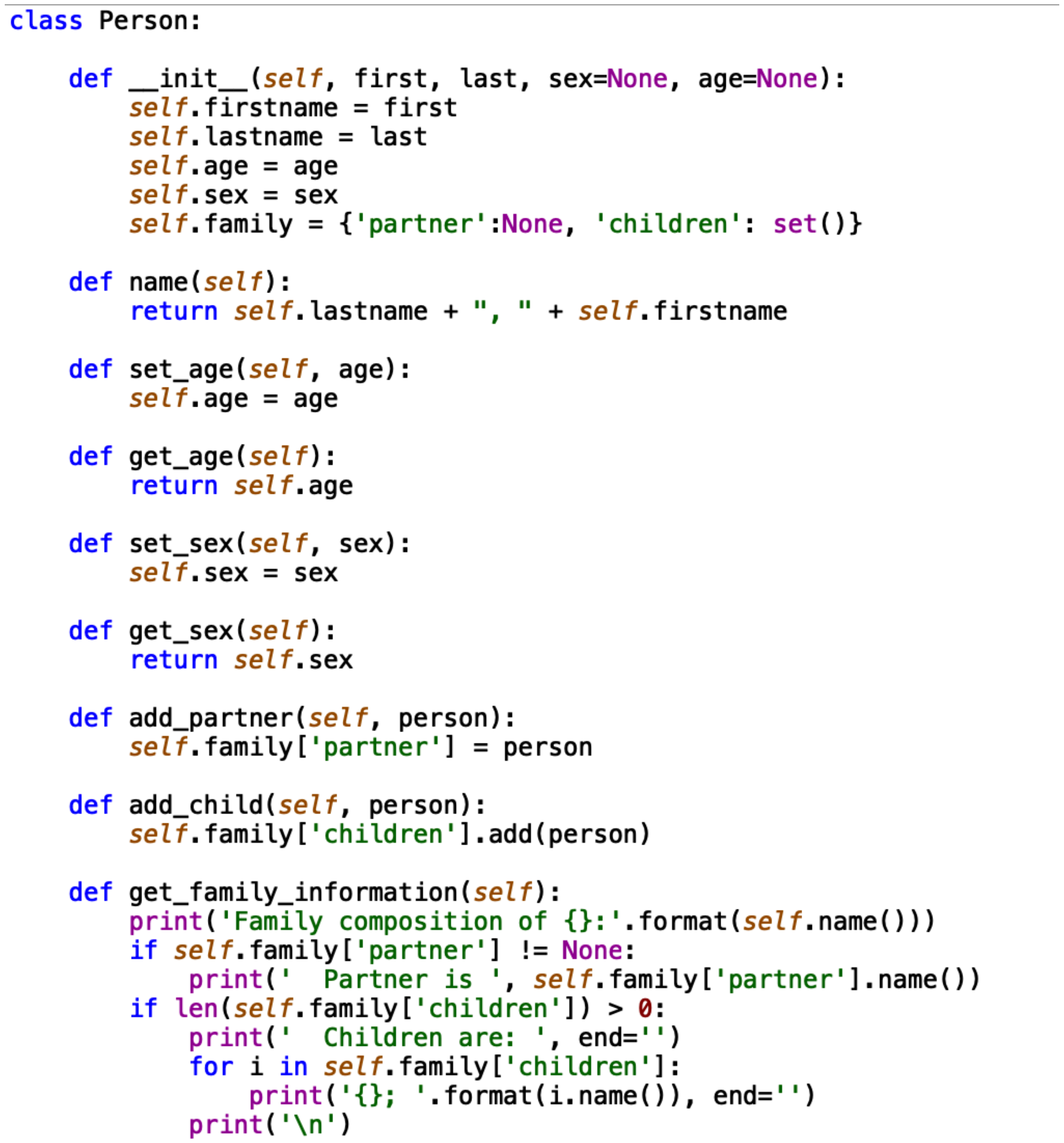
Dispose of the object (not the class!)

__del__ (**destructor**)

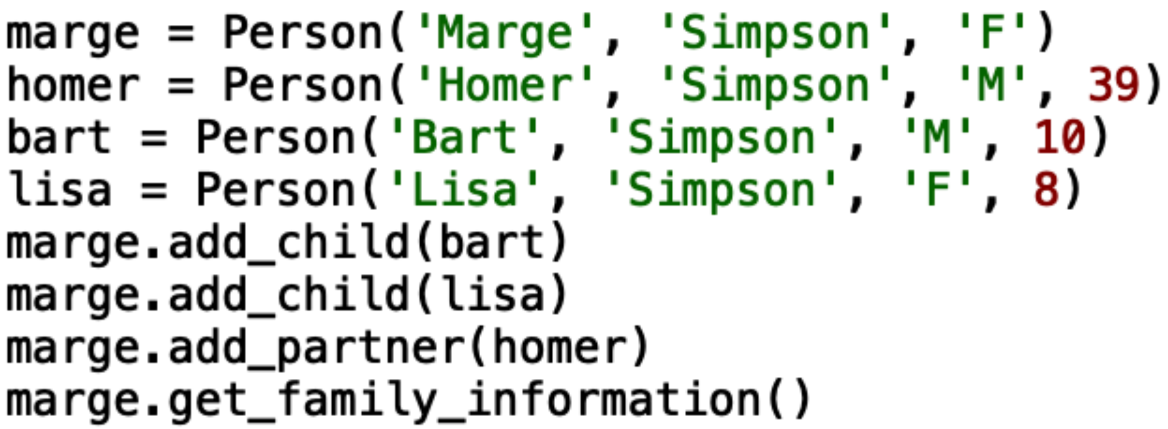
Partner and children of a person

```python
class Person:

    def __init__(self, first, last, sex=None, age=None):
        self.firstname = first
        self.lastname = last
        self.age = age
        self.sex = sex
        self.family = {'partner':None, 'children': set()}


    def add_partner(self, person):
        self.family['partner'] = person

    def add_child(self, person):
        self.family['children'].add(person)

    def get_family_information(self):
        print('Family composition of {}:'.format(self.name()))
        if self.family['partner'] != None:
            print('  Partner is ', self.family['partner'].name())
        if len(self.family['children']) > 0:
            print('  Children are: ', end='')
            for i in self.family['children']:
                print('{}; '.format(i.name()), end='')
            print('\n')
```

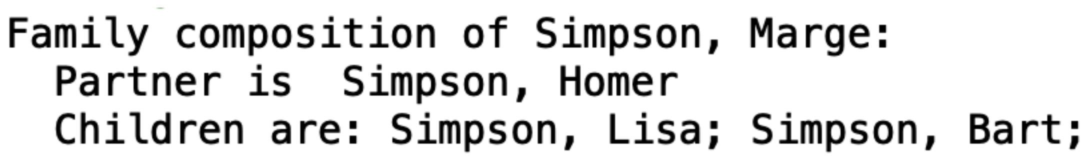
Partner and children of a person



```python
class Person:

    def __init__(self, first, last, sex=None, age=None):
        self.firstname = first
        self.lastname = last
        self.age = age
        self.sex = sex
        self.family = {'partner':None, 'children': set()}

    def name(self):
        return self.lastname + ", " + self.firstname

    def set_age(self, age):
        self.age = age

    def get_age(self):
        return self.age

    def set_sex(self, sex):
        self.sex = sex

    def get_sex(self):
        return self.sex

    def add_partner(self, person):
        self.family['partner'] = person

    def add_child(self, person):
        self.family['children'].add(person)

    def get_family_information(self):
        print('Family composition of {}:'.format(self.name()))
        if self.family['partner'] != None:
            print('  Partner is ', self.family['partner'].name())
        if len(self.family['children']) > 0:
            print('  Children are: ', end='')
            for i in self.family['children']:
                print('{}; '.format(i.name()), end='')
            print('\n')
```



```python
marge = Person('Marge', 'Simpson', 'F')
homer = Person('Homer', 'Simpson', 'M', 39)
bart = Person('Bart', 'Simpson', 'M', 10)
lisa = Person('Lisa', 'Simpson', 'F', 8)
marge.add_child(bart)
marge.add_child(lisa)
marge.add_partner(homer)
marge.get_family_information()
```

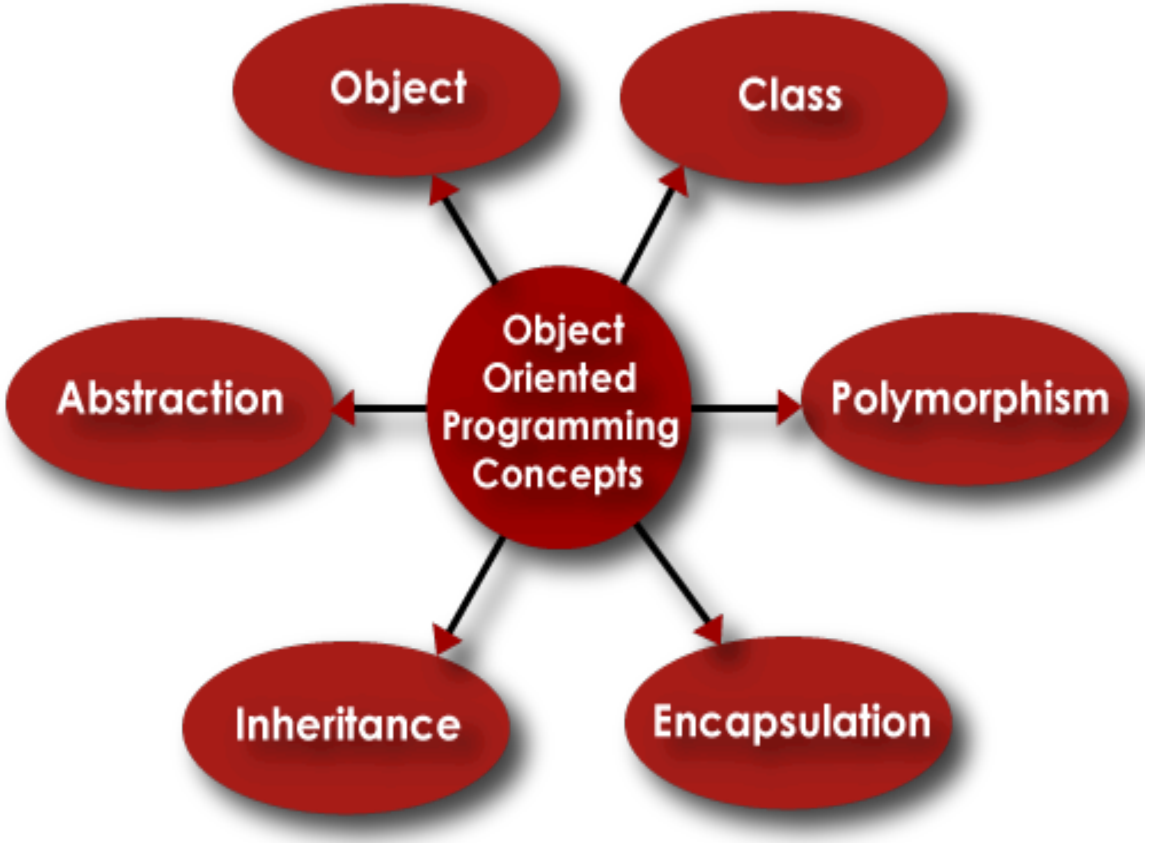


```
Family composition of Simpson, Marge:
  Partner is  Simpson, Homer
  Children are: Simpson, Lisa; Simpson, Bart;
```

Person class

- ✓ Check the code in `Person.py` for playing with additional methods and attributes
- ✓ Check the class `Employee` in file `Employee.py` as an example of **inheritance** and **polymorphism**

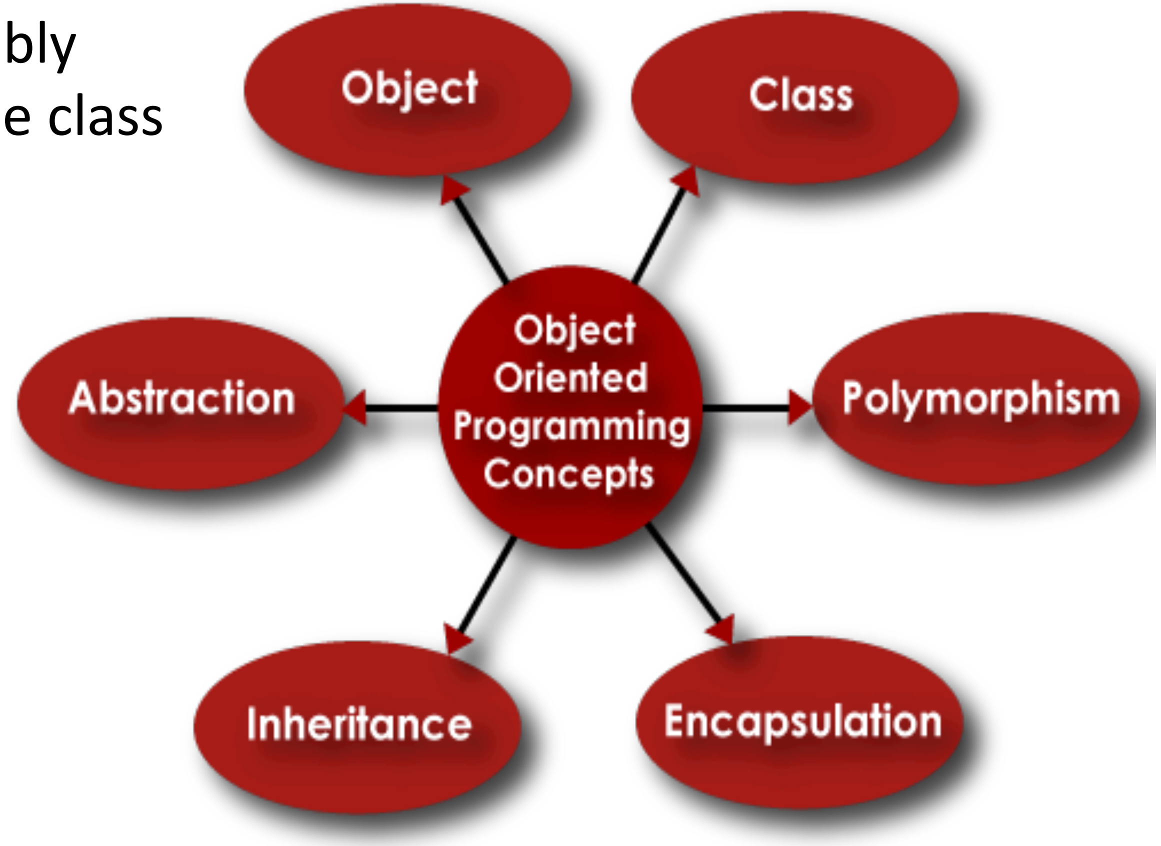
Object-Oriented (OO) design and programming



- **class = *attributes + methods***
- It provides the *blueprint* for creating multiple independent instances of class objects

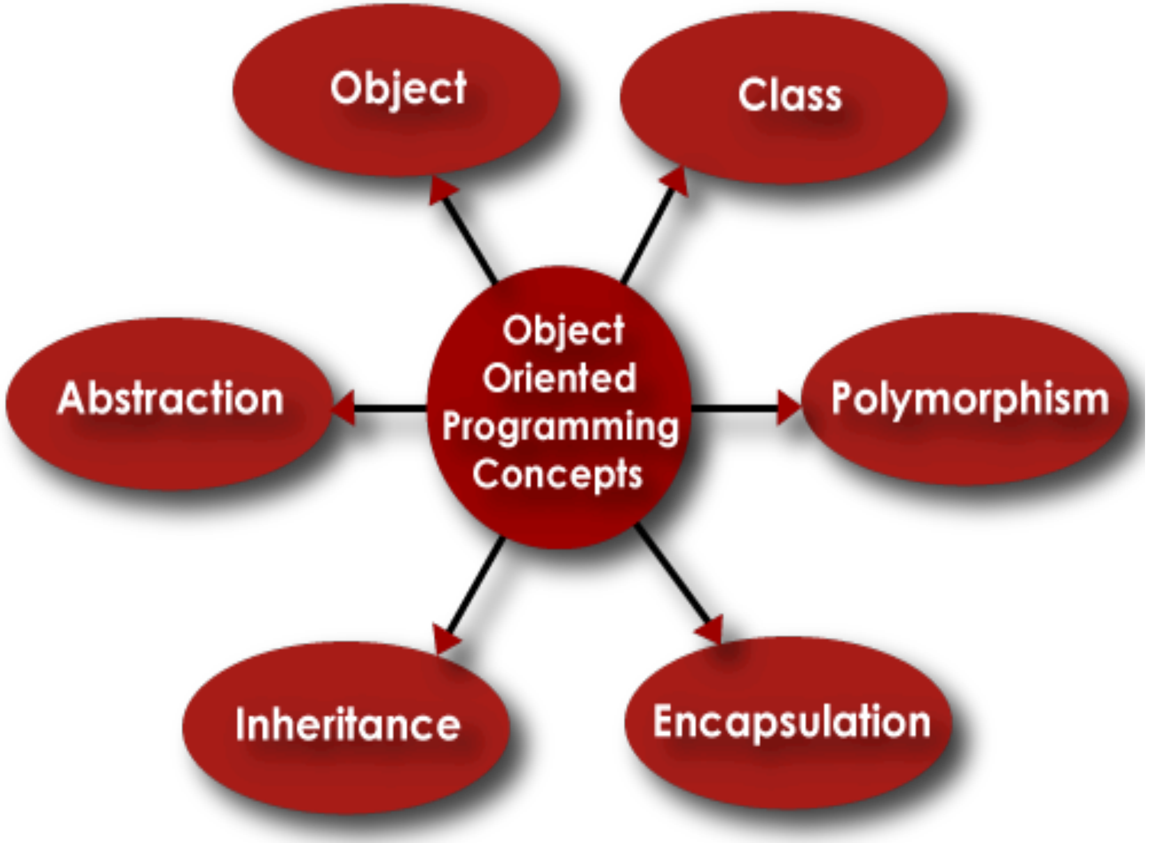
Object-Oriented (OO) design and programming

Object: An instance of the class, with the same methods and its own values for the attributes, and which “lives” independently from (but possibly interacting with) other objects from the same class

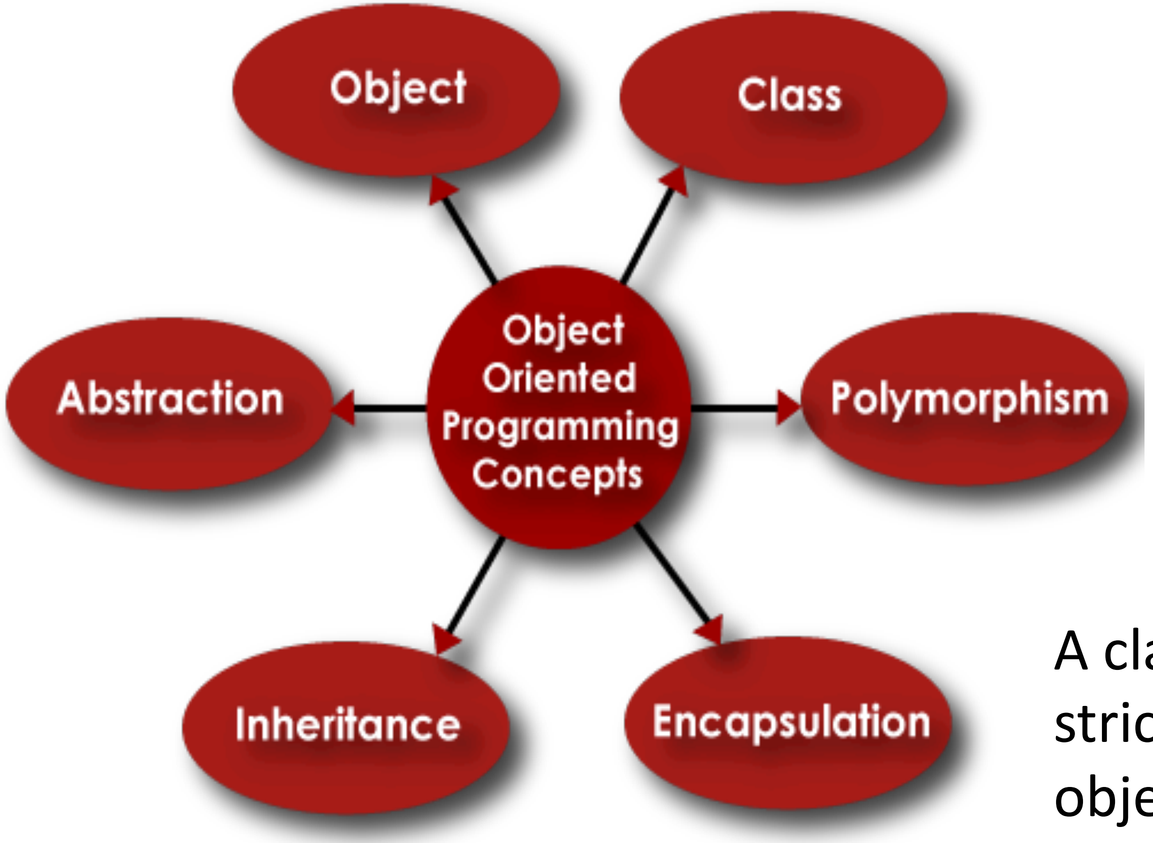


Object-Oriented (OO) design and programming

A class **abstracts** away implementation details, it provides the *interface* to make use of potentially complex entities and procedures, not bound to any specific object

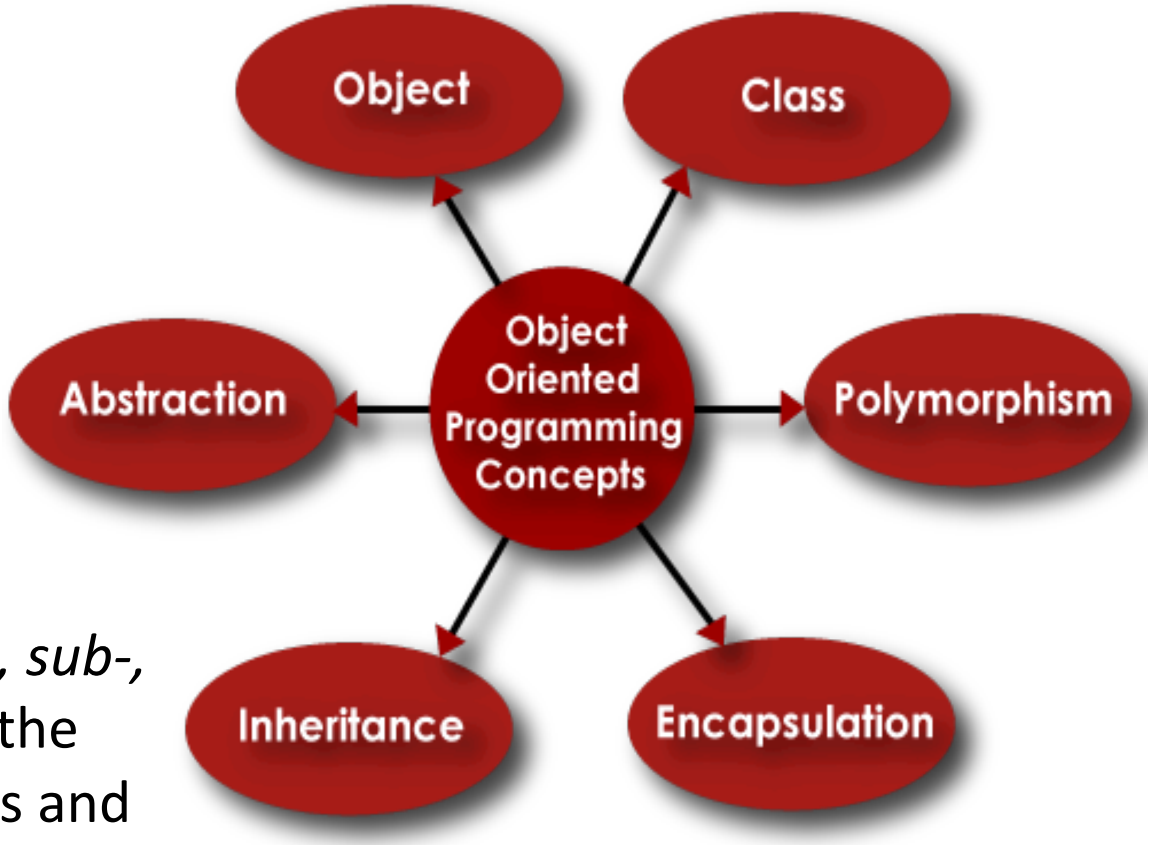


Object-Oriented (OO) design and programming



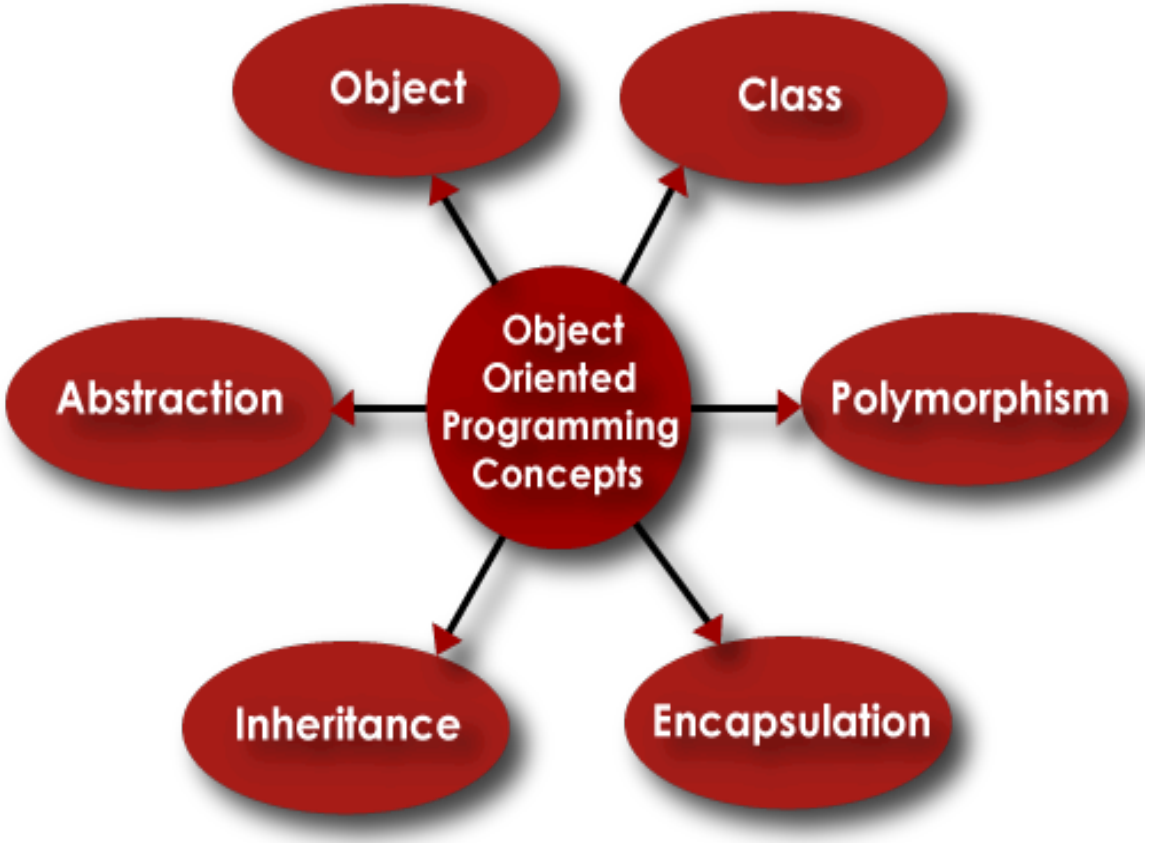
A class can *expose* only data and methods data are strictly necessary to use and interact with the class object, internal data structures are ***encapsulated*** inside the class: the external user shouldn't mess up with them!

Object-Oriented (OO) design and programming



Inheritance: it allows to create a new class (a *child, sub-, derived* class) that is based upon an existing class (the *parent, base, super* class), by adding new attributes and methods on top of the existing class, (and/or by specializing existing methods). The child class *inherits* attributes and methods of the parent class.

Object-Oriented (OO) design and programming



Polymorphism: taking different forms. The same method (same name, same interface) can process different input objects (forms) and produces different outputs accordingly, transparently to whom is invoking the method.

Operator overloading is an expression of polymorphic behavior. Functions like sorted(), max(), min() are polymorphic.
A subclass can make use of the same method of its base class, but specialized