Computing Bootcamp

Object-Oriented Programming - Motivation

Lecture 8

Hyung-Sin Kim

SNU Graduate School of Data Science

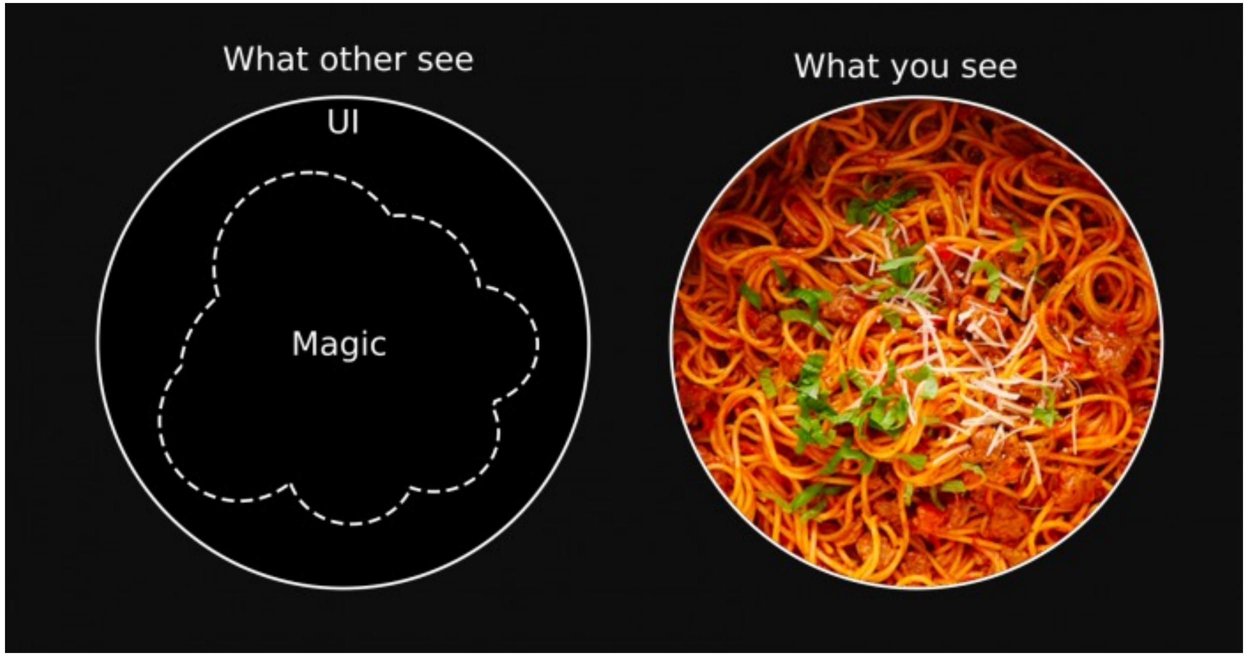
What is OOP?

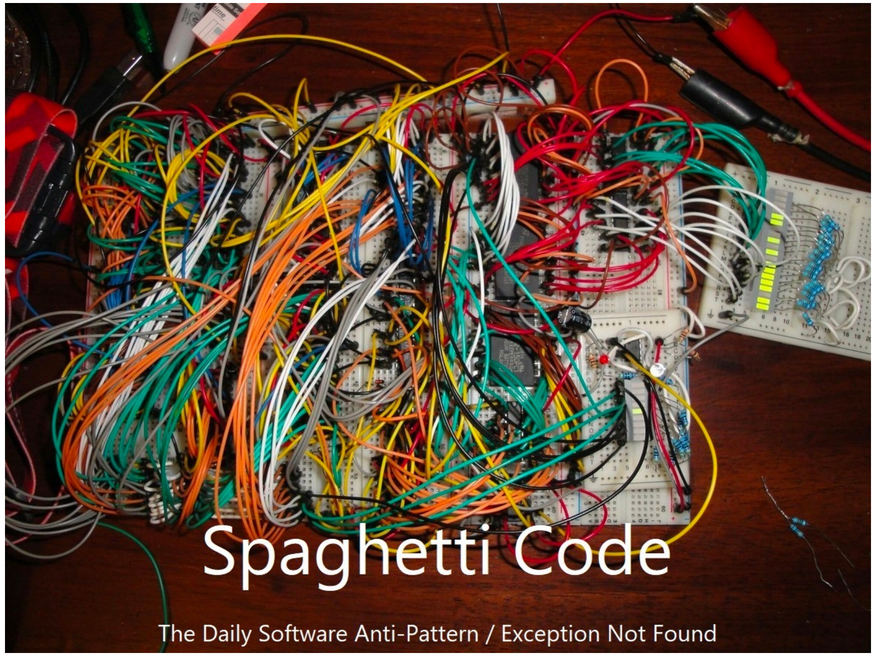
- What you learnt before using classes is called **Procedural Programming**
  - A programming paradigm that relies on **variables**, **data structures**, and **functions**
  - It breaks down a programming task into a collection of variables, data structures, and functions
  - Ex.) max(2, 4), convert_to_celsius(80)

- While using classes and methods, you have gradually been exposed to **Object-oriented Programming**
  - A programming paradigm that relies on the concept of **classes** and **objects**
  - It breaks down a programming task into **objects** that expose behavior and data using interfaces (methods)
  - Ex.) students.append("inhoe"), students.clear()

Why OOP? – Spaghetti Code

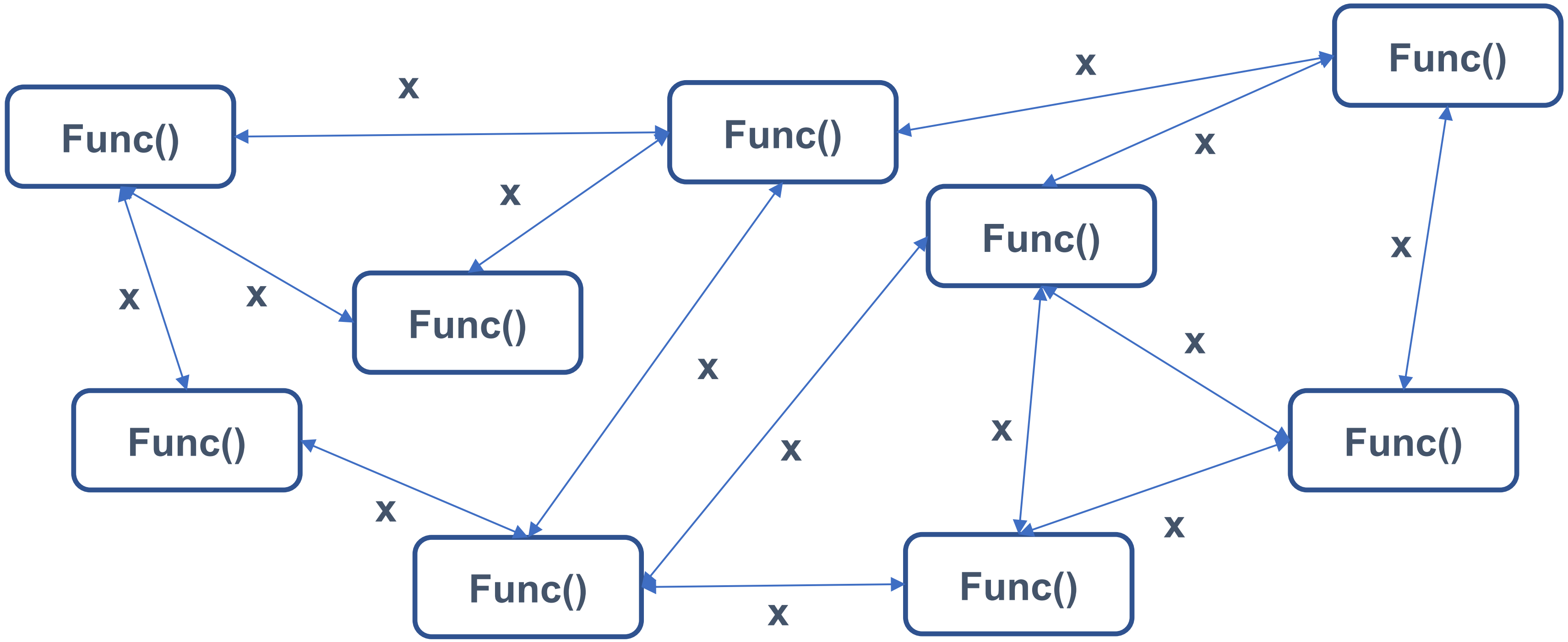
- Spaghetti code is a complex code where many modules (functions) are **super inter-dependent** to each other
  - Very hard to understand and fix
  - Fixing one function might cause another problem to several other functions





Why OOP? – Spaghetti Code

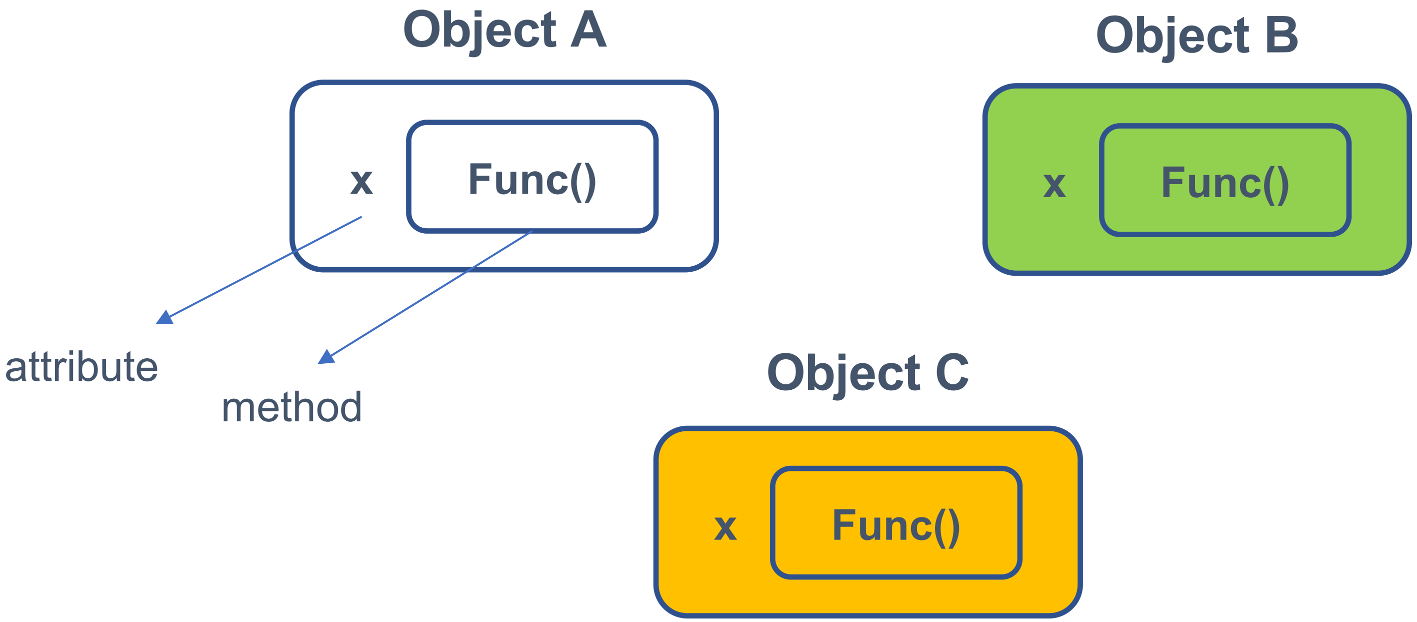
- With procedural programming, it is easy to generate a spaghetti code
  - Various functions and variables become dependent on each other



Do you have a bug here?
Good luck with that ^^

Why OOP? – Toward More Modular Coding

- Combine a group of related variables (attributes) and functions (methods) into a unit, which is called **object** (encapsulation)
    - If object A's method does not work as intended, you just need to fix object A!



Object-Oriented Programming - Principles

Lecture 8

Hyung-Sin Kim

SNU Graduate School of Data Science

Four Principles

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Encapsulation

- Contain related information and behaviors (attributes and methods) in an **object**
  - Example of grading

PP: 4 lists, name, hw, midterm, and final, each of which has 100 elements

name_lst

midterm_lst

hw_lst

final_lst

```
def get_total(midterm, final):
        return midterm * 0.3 + final * 0.4
```

Variables and methods are <u>decoupled</u>

```
>>> get_total(midterm[0], final[0])
```

Function with many parameters

OOP: A list of 100 student objects

Student object

name | hw | midterm | final

```
def get_total(self):
        return self.midterm * 0.3
                + self.final * 0.4
```

Variables and methods are <u>encapsulated</u> in an object

```
>>> students[0].get_total()
```

Function with no parameter!

Encapsulation

- In procedural programming, variables and functions are all decoupled. There is no explicit relationship between them
  - Therefore, a function needs to take all variables that it needs as parameters

- In object-oriented programming, highly related variables (attributes) and functions (methods) are in **one** object as a **group**
  - Therefore, most variables that a method needs are **already** part of one unit (in one object)
  - A method does not need to have many parameters

Oops! I forgot to include hw score!

Encapsulation

- Contain related information and behaviors (attributes and methods) in an **object**
  - Example of grading

PP: 4 lists, name, hw, midterm, and final, each of which has 100 elements

name_lst

midterm_lst

hw_lst

final_lst

```
def get_total(midterm, final, hw):
        return midterm * 0.3 + final * 0.4
                + hw * 0.3
```

```
>>> get_total(midterm[0], final[0], hw[0])
```

Change all the function calls

OOP: A list of 100 student objects

Student object

name | hw | midterm | final

```
def get_total(self):
        return self.midterm * 0.3
                + self.final * 0.4
```

```
>>> students[0].get_total()
```

Encapsulation

- Contain related information and behaviors (attributes and methods) in an **object**
  - Example of grading

PP: 4 lists, name, hw, midterm, and final, each of which has 100 elements

name_lst

midterm_lst

hw_lst

final_lst

```
def get_total(midterm, final, hw):
    return midterm * 0.3 + final * 0.4
        + hw * 0.3
```

```
>>> get_total(midterm[0], final[0], hw[0])
```

Change all the function calls

```
>>> get_total(midterm[1], final[1], hw[1])
```

```
>>> get_total(midterm[2], final[2], hw[2])
```

```
>>> get_total(midterm[72], final[72], hw[72])
```

```
>>> get_total(midterm[9], final[9], hw[9])
```

```
>>> get_total(midterm[101], final[101], hw[101])
```

You can see only several hundreds of errors ☺

Encapsulation

- Contain related information and behaviors (attributes and methods) in an **object**
  - Example of grading

PP: 4 lists, name, hw, midterm, and final, each of which has 100 elements

name_lst

midterm_lst

hw_lst

final_lst

```
def get_total(midterm, final, hw):
    return midterm * 0.3 + final * 0.4
        + hw * 0.3
```

```
>>> get_total(midterm[0], final[0], hw[0])
```

Change all the function calls

OOP: A list of 100 student objects

Student object

name

hw

midterm

final

```
def get_total(self):
    return self.midterm * 0.3 + self.hw* 0.3
        + self.final * 0.4
```

```
>>> students[0].get_total()
```

Change only function definition and DONE!

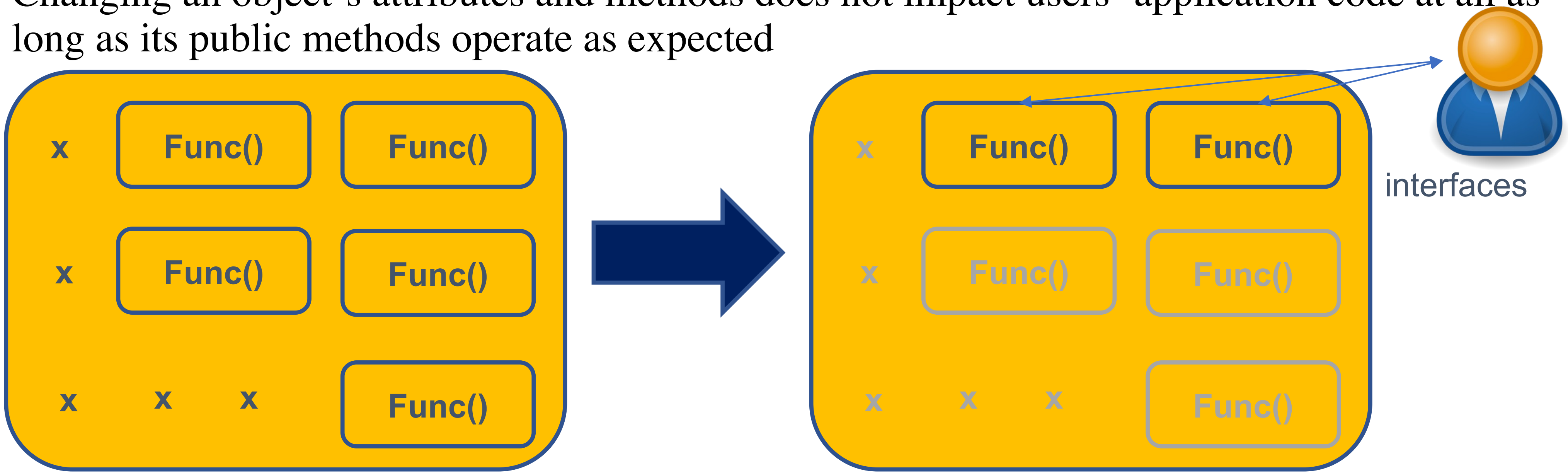
"The best functions are those with no parameters!"

Uncle Bob – Robert C Martin

A function with fewer parameters is easier to use and maintain…

Abstraction

- Hide details (many attributes and methods) from outside and expose only high level methods to the outside world
  - Simpler interface
    - However complex an object is, users can understand and use it by studying only a few methods
  - Isolated impact of change
    - Changing an object's attributes and methods does not impact users' application code at all as long as its public methods operate as expected



Abstraction

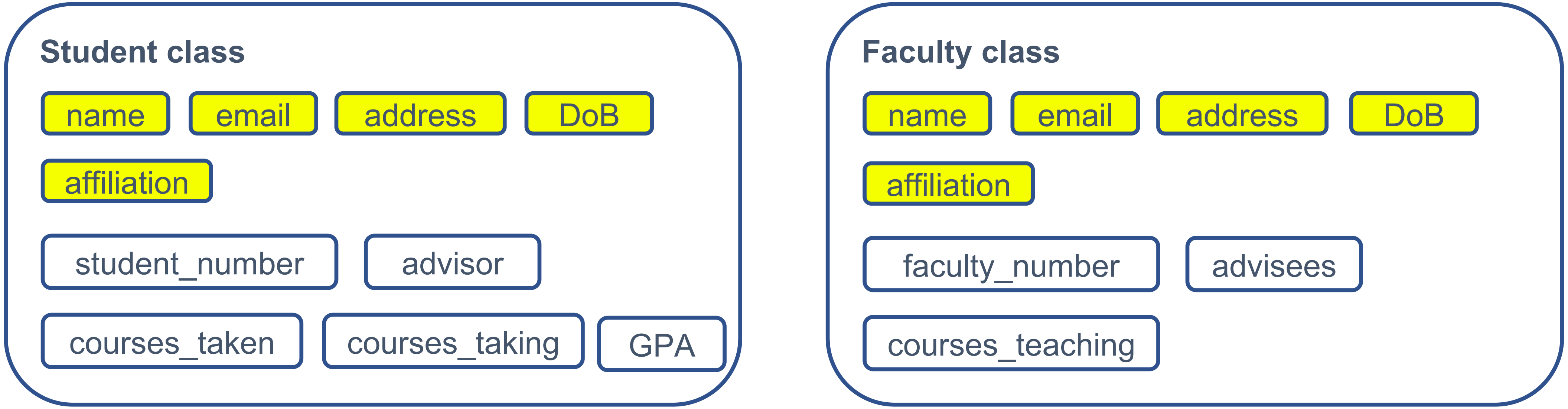
- Jupyter notebook
  - We don't know how its underlying codes work and how it interacts with an operating system (implementation details)
  - But we know that if we put a line of python code on a Jupyter cell and press [CTRL+Enter], we will see a corresponding result (interface)
  - When Jupyter version is updated, we don't worry about studying it from scratch again, because we already know how to use its interfaces

```
In [1]: temp_celsius = 31.0
        difference = 10.0
        temp_celsius = temp_celsius - 2*difference
        difference = 5.0
        temp_celsius

Out[1]: 11.0
```

Inheritance

- Eliminate redundant code by making child classes **inherit** data and behaviors from parent class



Yellow attributes are overlapped.
It is redundant to type these again…

Inheritance

- Eliminate redundant code by making child classes **inherit** data and behaviors from parent class

Member class

name | email | address | DoB | affiliation

Student class (Member)

Inherit Member's attributes and methods

student_number | advisor

courses_taken | courses_taking | GPA

Faculty class (Member)

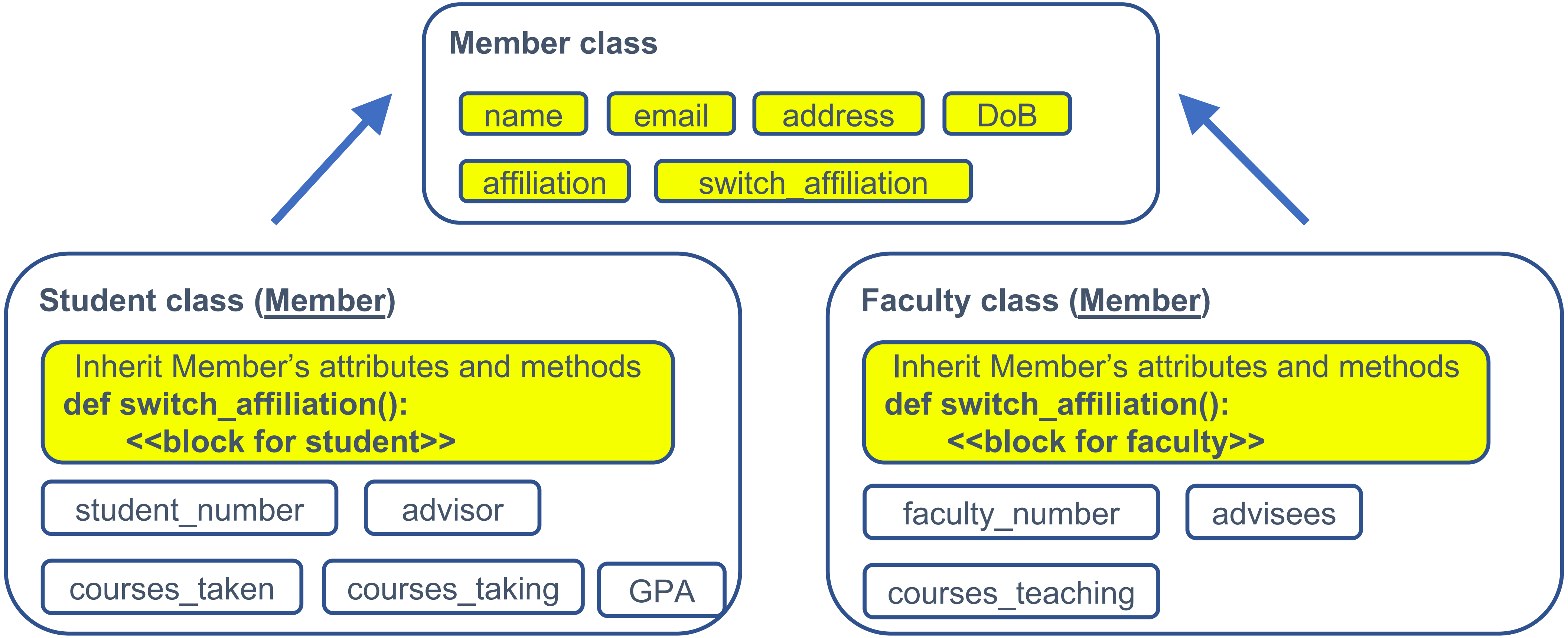
Inherit Member's attributes and methods

faculty_number | advisees

courses_teaching

Polymorphism (many forms)

- Allow a single method to do different things depending on what object it is included in
  - studentA.switch_affiliation() and facultyA.switch_affiliation() do different things



Polymorphism (many forms)

- Allow a single method to do different things depending on what object it is included in
  - studentA.switch_affiliation() and facultyA.switch_affiliation() do different things
  - If we write the function by using procedural programming, there will be ugly if/else statements

```
def switch_affiliation(member):
    if type(member) == faculty:
        <<block for faculty>>
    elif type(member) == student:
        <<block for student>>
    …
```

Summary

- **Encapsulation**: Contain related information in an object
  - Reduce complexity and increase reusability
- **Abstraction**: Expose only high level interfaces to the outside world
  - Reduce complexity and isolate impact of changes
- **Inheritance**: Child classes inherit data and behaviors from parent class
  - Eliminate redundant code
- **Polymorphism:** A single method acts in a different way depending on objects
  - Escape from complex if/else statements

Thanks!