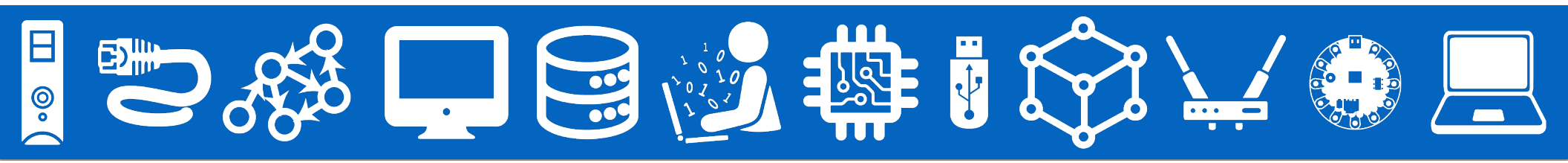
CS134:
Classes, Objects, and Inheritance



Announcements & Logistics

- **Lab 8** is a **partner lab:** focuses on using classes
  - **Must attend one lab session with your partner**
  - Mon lab due on Wed, Tue lab due on Thur
  - Make sure both partners are typing/participating
- **HW 7** due tonight (on Glow)

Do You Have Any Questions?

Last Time

- Built the **Book class** to represents book objects
- Learned about private, protected, public attributes and methods (indicate scope using underscores in Python)
- Explored accessor (getter) and mutator (setter) methods in Python
- Talked about `__init__` (aka constructor) and `__str__` methods

Today's Plan

- Look at another simple example involving classes and methods
- Begin talking about inheritance

Print Representation of an Object

```
class Book():
   __slots__ = ["_title"]

   def __init__(self, title):
      self._title = title
```

```
>>> test = Book("testing")
>>> print(test)
<__main__.Book object at 0x105eecca0>
```

By default, if we print an object, the output is not helpful

- Special method `__str__` is automatically called when we ask to print a class object in Python
- `__str__` must always return a string
- We can customize how the object is printed by writing a custom `__str__` method for our class
- *Very* useful for debugging!

`__str__` for Book class

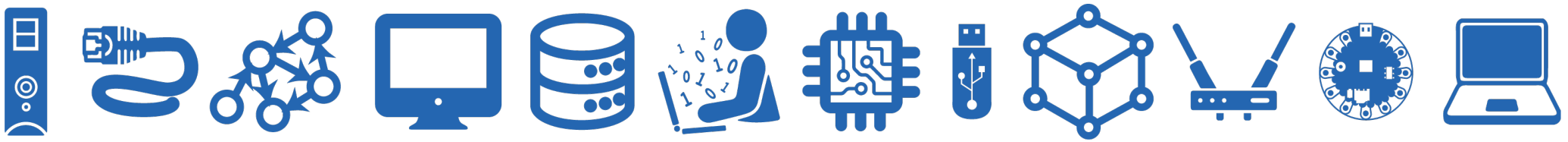
- What is a useful string representation of a **Book**?
  - Something that combines the attributes in a meaningful way
  - The `format( )` string method comes in handy here

```
# __str__ is used to generate a meaningful string representation for Book objects
# __str__ is automatically called when we ask to print() a Book object
def __str__(self):
   return "'{}', by {}, in {}".format(self._title, self._author, self._year)
```

  - Now when we ask to print a specific instance of a **Book**, we get something useful

```
>>> print(emma)

'Emma', by Jane Austen, in 1815
```

Special Methods



Special methods and attributes

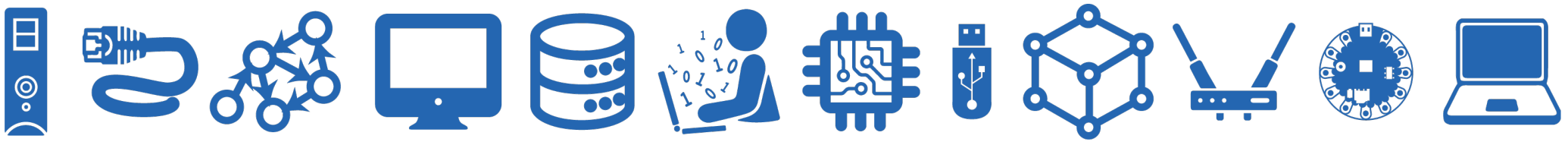
- We've seen several "special" methods and attributes in Python:
  - `__name__` special module attribute
  - `__main__` name attribute of scripts
  - `__slots__` list for attributes
  - `__init__` method
  - `__str__` method

Other Special Methods

- There are many other "special" methods in Python.
  - `__len__(self): len(x)`
  - `__contains__(self, item): item in x`
  - `__eq__ (self, other): x == y`
  - `__lt__ (self, other): x < y`
  - `__gt__ (self, other): x > y`
  - `__add__(self, other) :x + y`
  - `__sub__(self, other): x - y`
  - `__mul__(self, other): x * y`
  - `__truediv__(self, other): x / y`
  - `__pow__(self, other): x ** y`
  - There are others!

We'll come back to these in a few weeks!

Another Class Example



Another Example: Name Class

- Names of people have certain attributes
    - Almost everyone has a **first and last name**
    - Some people have a **middle name**
- We can create name objects by defining a class to represent these attributes
- Then we can define methods, e.g., getting initials of people's names, etc
- Let's practice some of the concepts using this class
    - **__str__**: how do we want the names to be printed?
    - **initials**: can we define a method that returns the initials of people's names?

Example: Name Class

```
class Name:
    """Class to represent a person's name."""
    __slots__ = ['_f', '_m', '_l']

    def __init__(self, first, last, middle=''):
        self._f = first
        self._m = middle
        self._l = last


    def __str__(self):
        # if the person has a middle name
        if len(self._m) > 0:
            return '{}. {}. {}'.format(self._f[0], self._m[0], self._l)
        else:
            return '{}. {}'.format(self._f[0], self._l)
```

```
>>> n1 = Name("Jeannie", "Albrecht", "Raye")
>>> n2 = Name("Iris", "Howley")
>>> print(n1)
J. R. Albrecht
>>> print(n2)
I. Howley
```

intials() method

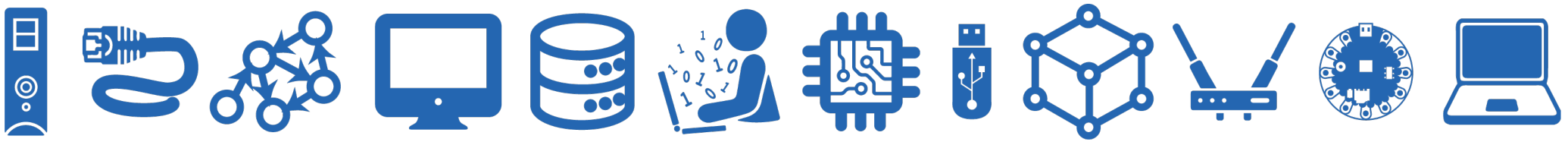
- Suppose we want to write a method that returns the person's initials as a string?
- How would we do that?

Example: Name Class

```
class Name:
    """Class to represent a person's name."""
    __slots__ = ['_f', '_m', '_l']

    def __init__(self, first, last, middle=''):
        self._f = first
        self._m = middle
        self._l = last

    def initials(self):
        if len(self._m) > 0:
            return '{}. {}. {}.'.format(self._f[0], self._m[0], self._l[0]).upper()
        else:
            return '{}. {}.'.format(self._f[0], self._l[0]).upper()

    def __str__(self):
        # if the person has a middle name
        if len(self._m) > 0:
            return '{}. {}. {}'.format(self._f[0], self._m[0], self._l)
        else:
            return '{}. {}'.format(self._f[0], self._l)
```

```
>>> n1 = Name('Jeannie', 'Albrecht', 'Raye')
>>> n1.initials()
'J. R. A.'
>>> n2 = Name('Lida', 'Doret')
>>> n2.initials()
'L. D.'
```

Inheritance Example

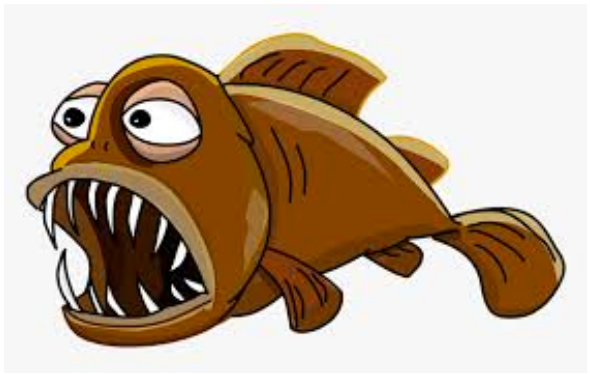


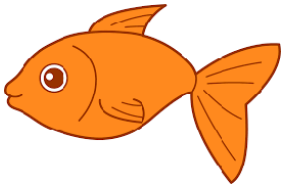
Introduction to Inheritance

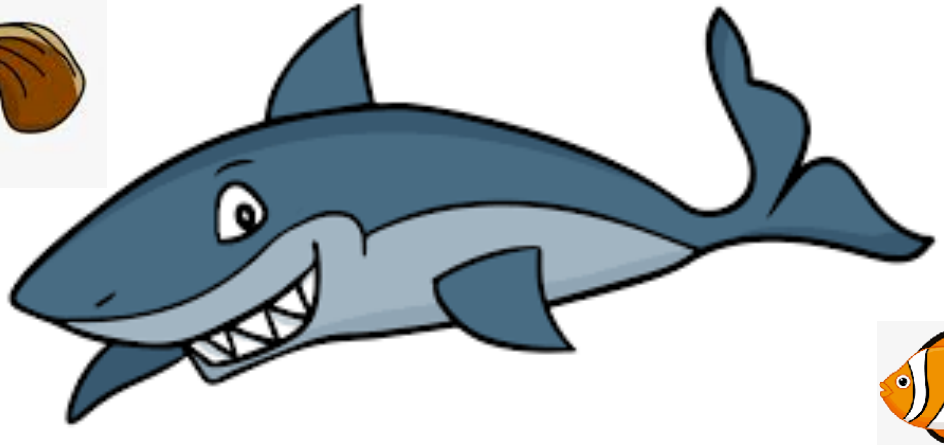
- **Inheritance** is the capability of one class to derive or ***inherit*** the properties from another class
- The benefits of inheritance are:
  - Often represents real-world relationships well
  - Provides **reusability of code**, so we don't have to write the same code again and again
  - Allows us to add more features to a class without modifying it
- Inheritance is **transitive** in nature, which means that if class B inherits from class A, then all the subclasses of B would also automatically inherit from class A
- When a class inherits from another class, all methods and attributes are accessible to subclass, ***except private attributes*** (indicated with __)

Inheritance Example

- Suppose we have a base (or parent) class `Fish`
- `Fish` defines several methods that are common to all fish:
  - `eat()`, `swim()`
- `Fish` also defines several attributes with default values:
  - `_length`, `_weight`, `_lifespan`

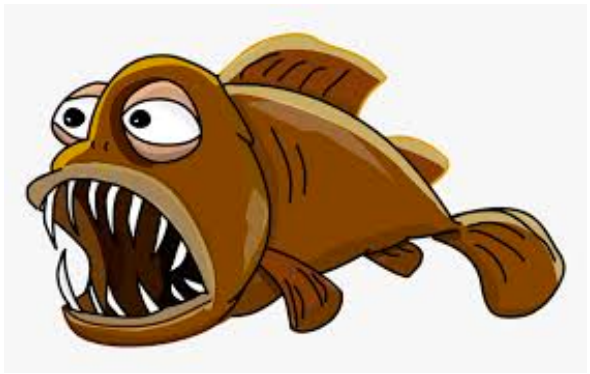


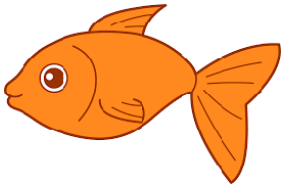


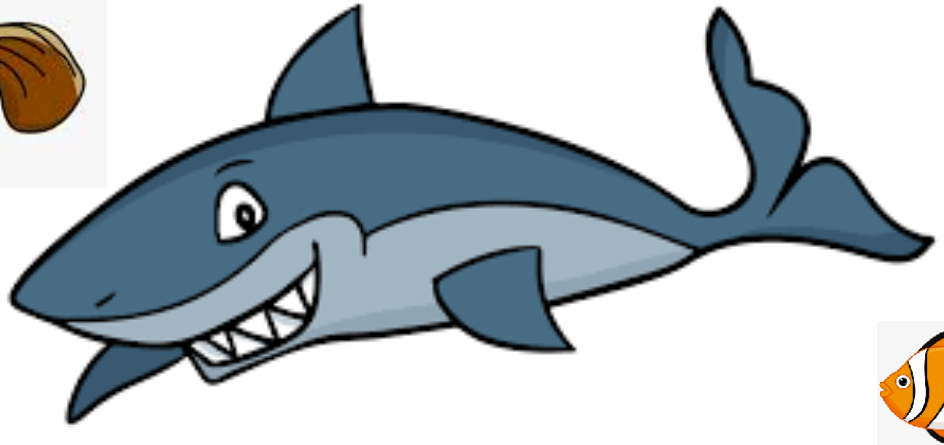


Inheritance Example

- All fish have some features in common
    - But not all fish are the same!
- Each `Fish` instance will specify different values for attributes (`_length`, `_weight`, `_lifespan`)
- Some fish may still need extra functionality!

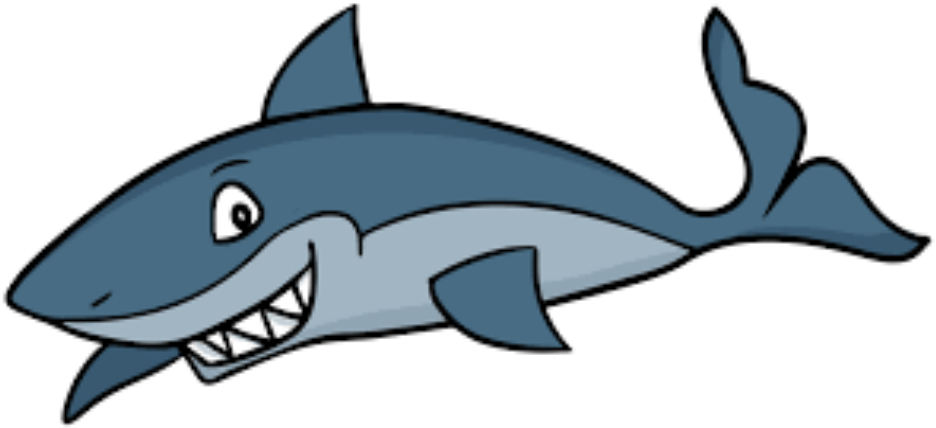


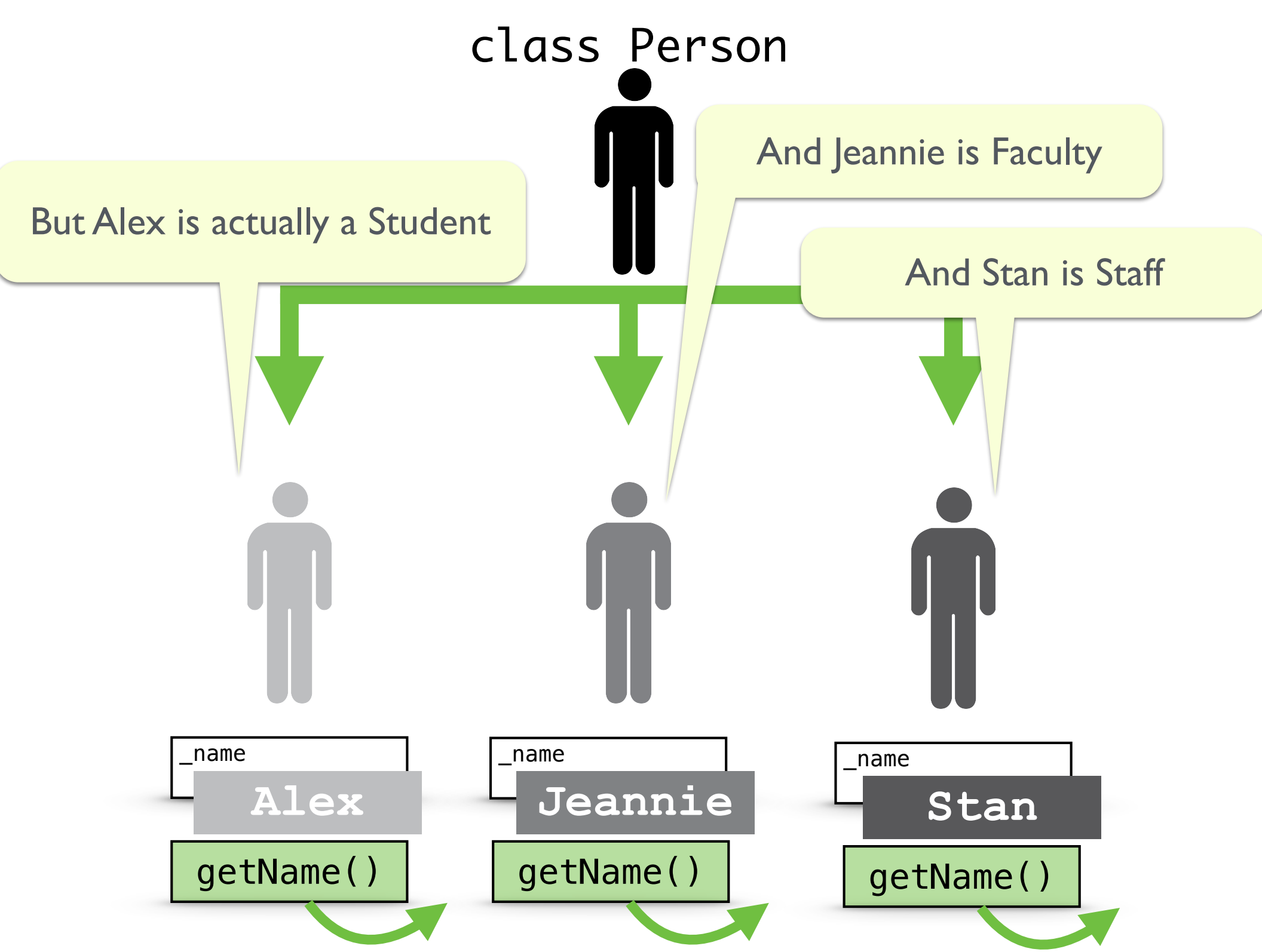


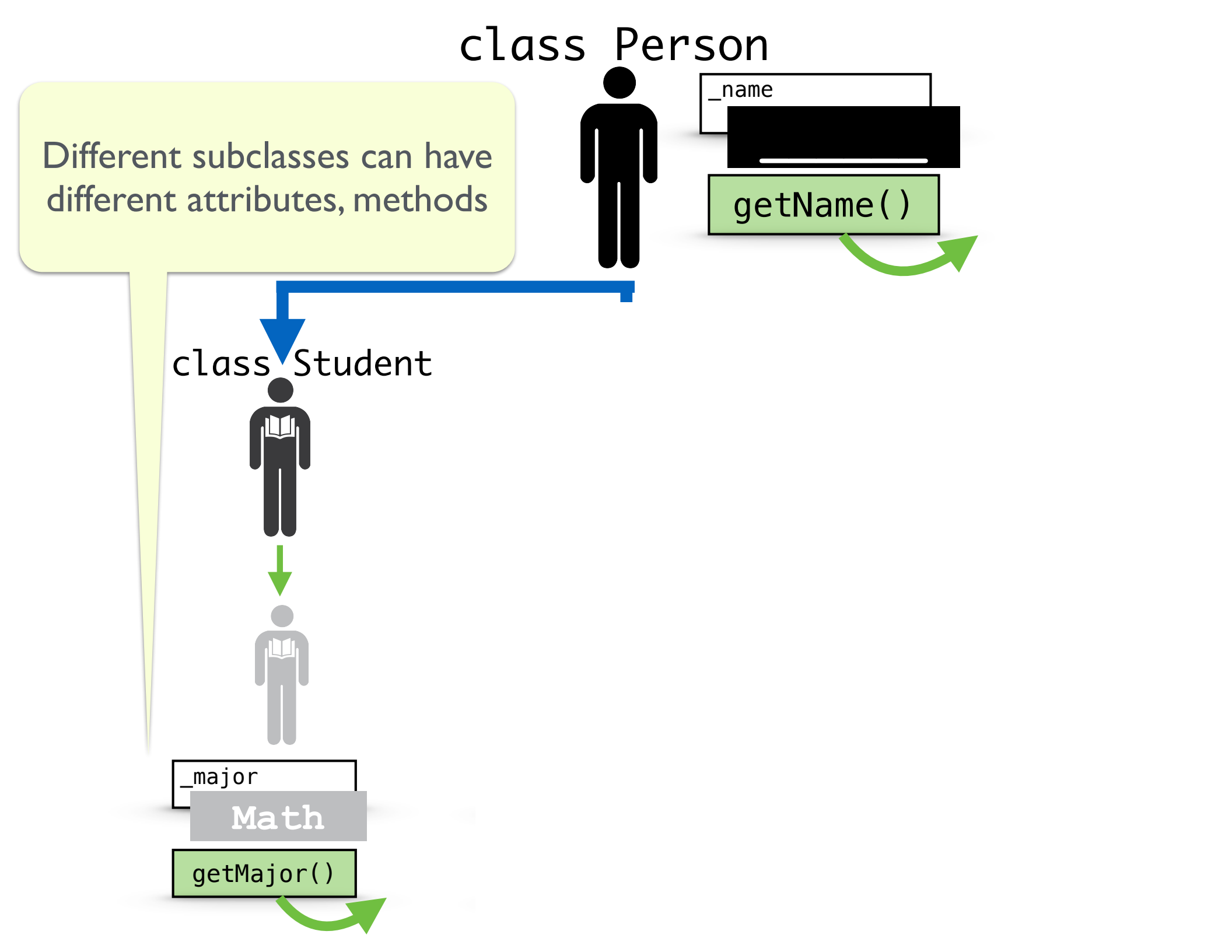


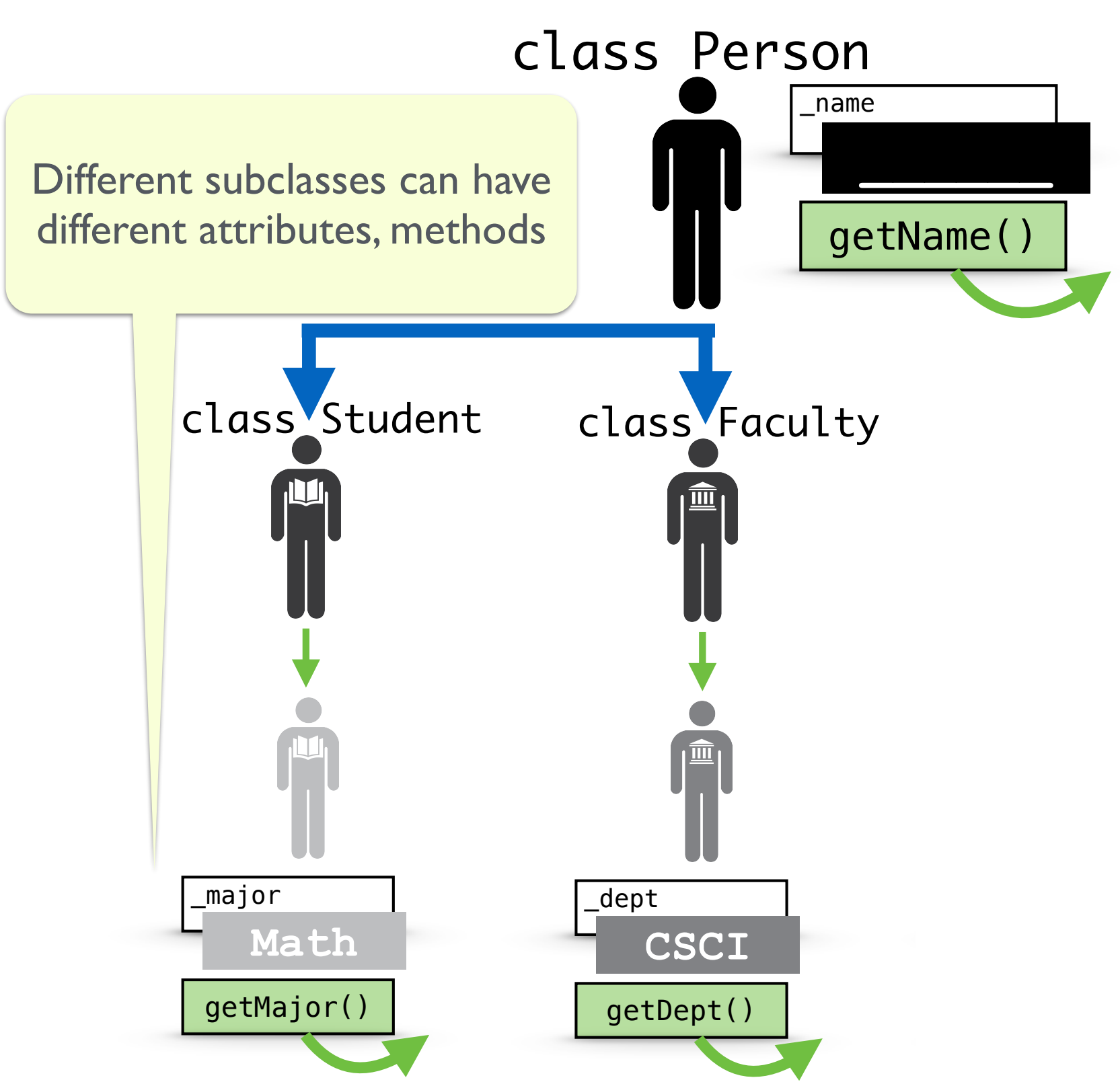
Inheritance Example

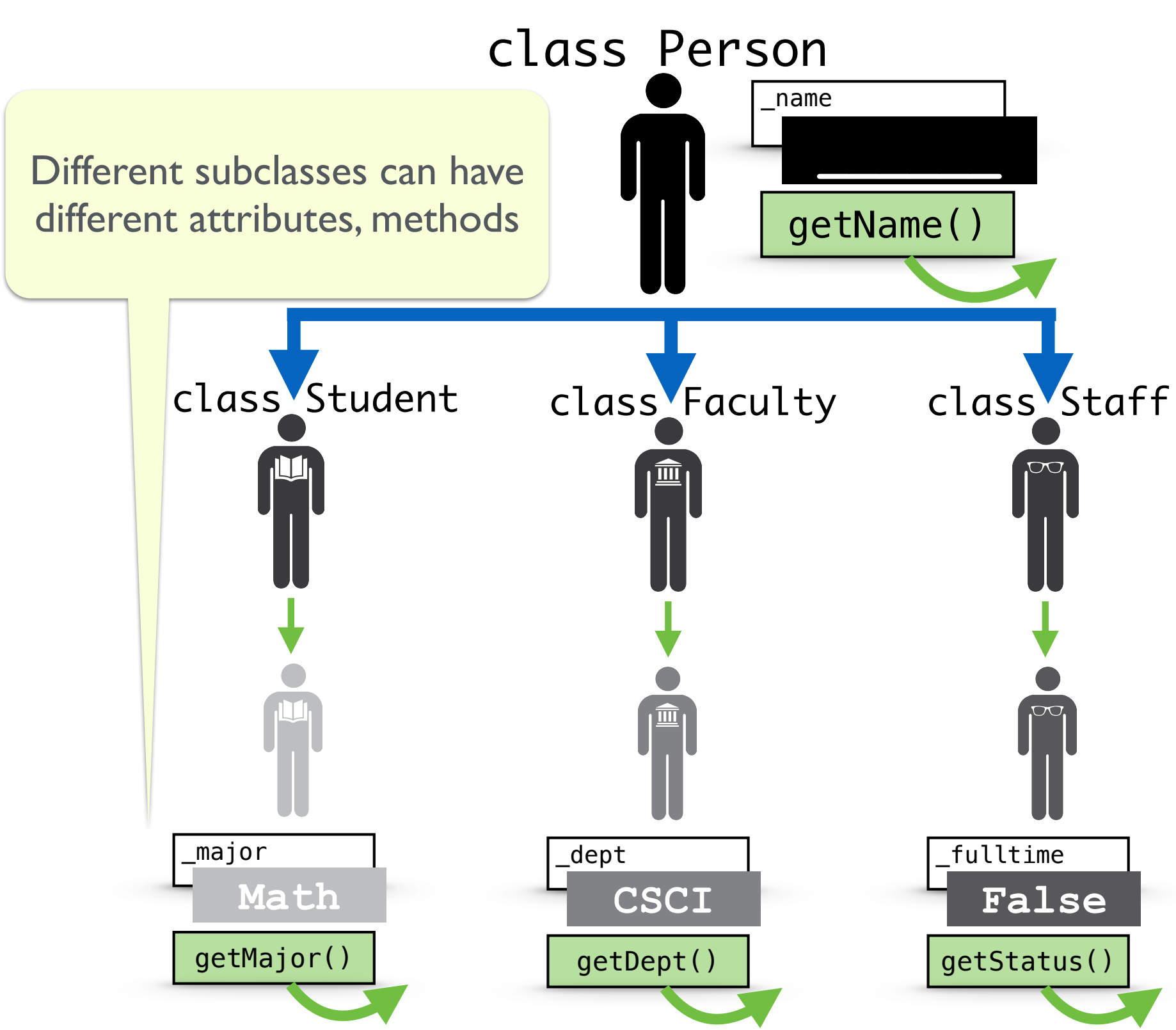
- For example, Sharks might need an `attack()` method
- Pufferfish might need a `puff()` method
- We might even want to ***override*** an existing method with a different (more specialized) implementation
    - Inheritance allows for all of this!

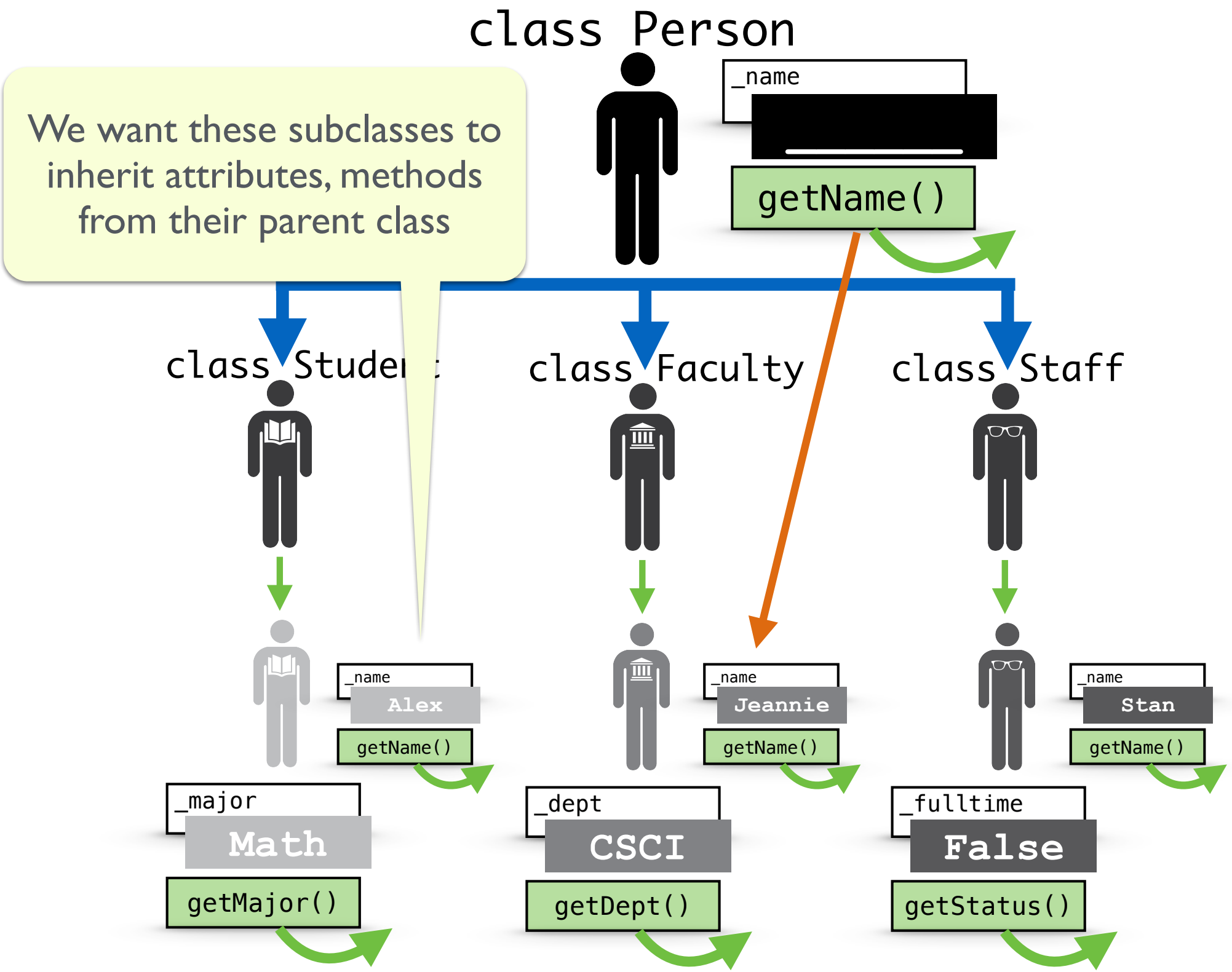



class Person
But Alex is actually a Student
And Jeannie is Faculty
And Stan is Staff
_name
Alex
getName()
_name
Jeannie
getName()
_name
Stan
getName()

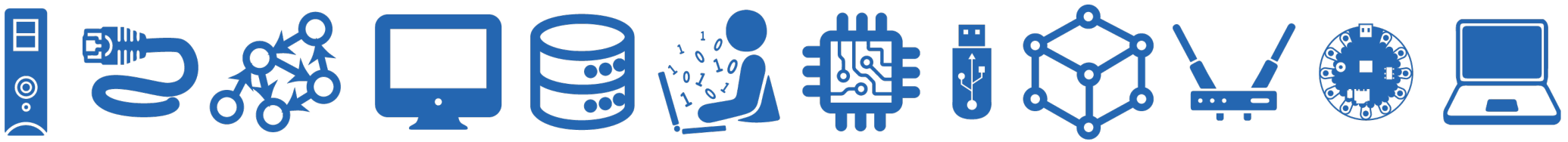

class Person
_name
getName()
Different subclasses can have different attributes, methods
class Student
_major
Math
getMajor()


class Person
Different subclasses can have different attributes, methods
_name
getName()
class Student
class Faculty
_major
Math
getMajor()
_dept
CSCI
getDept()


class Person
_name
getName()
Different subclasses can have different attributes, methods
class Student
class Faculty
class Staff
_major
Math
getMajor()
_dept
CSCI
getDept()
_fulltime
False
getStatus()


class Person
_name
getName()
We want these subclasses to inherit attributes, methods from their parent class
class Student
class Faculty
class Staff
_name
Alex
getName()
_name
Jeannie
getName()
_name
Stan
getName()
_major
Math
getMajor()
_dept
CSCI
getDept()
_fulltime
False
getStatus()

Inheritance

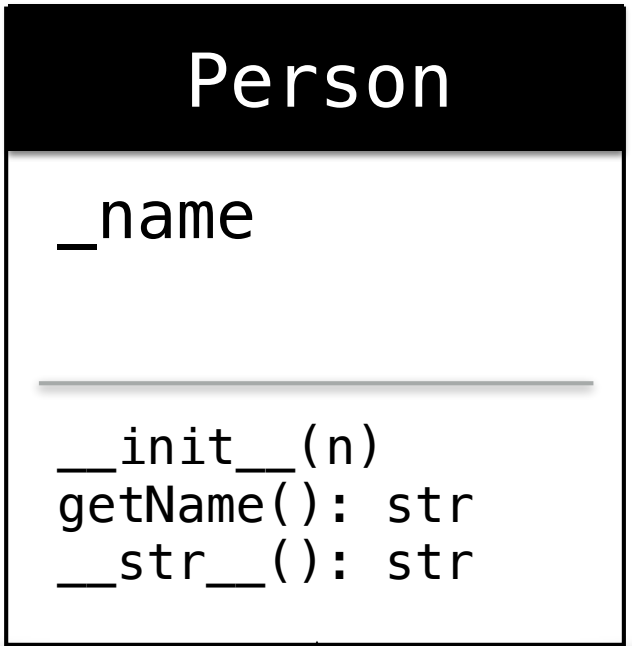


Inheritance

- When defining **super/parent** classes, think about the common features and methods that all subclasses will have
- In subclasses, inherit as much as possible from parent class, and add and/or override attributes and methods as necessary
- Consider an simple example:
  - **Person** class: defines common attributes for all people on campus
  - **Student** subclass: inherits from `Person` and adds additional attributes for student's ***major*** and ***year***
  - **Faculty** subclass: inherits from `Person` and adds additional attributes for ***department*** and ***office***
  - **Staff** subclass: inherits from `Person` and adds additional attributes for type/status of employee (***full-time, part-time***)

Person Class

```
class Person:
    __slots__ = ['_name']

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

    def getName(self):
        return self._name

    def __str__(self):
        return self._name
```


Person
_name
__init__(n)
getName(): str
__str__(): str

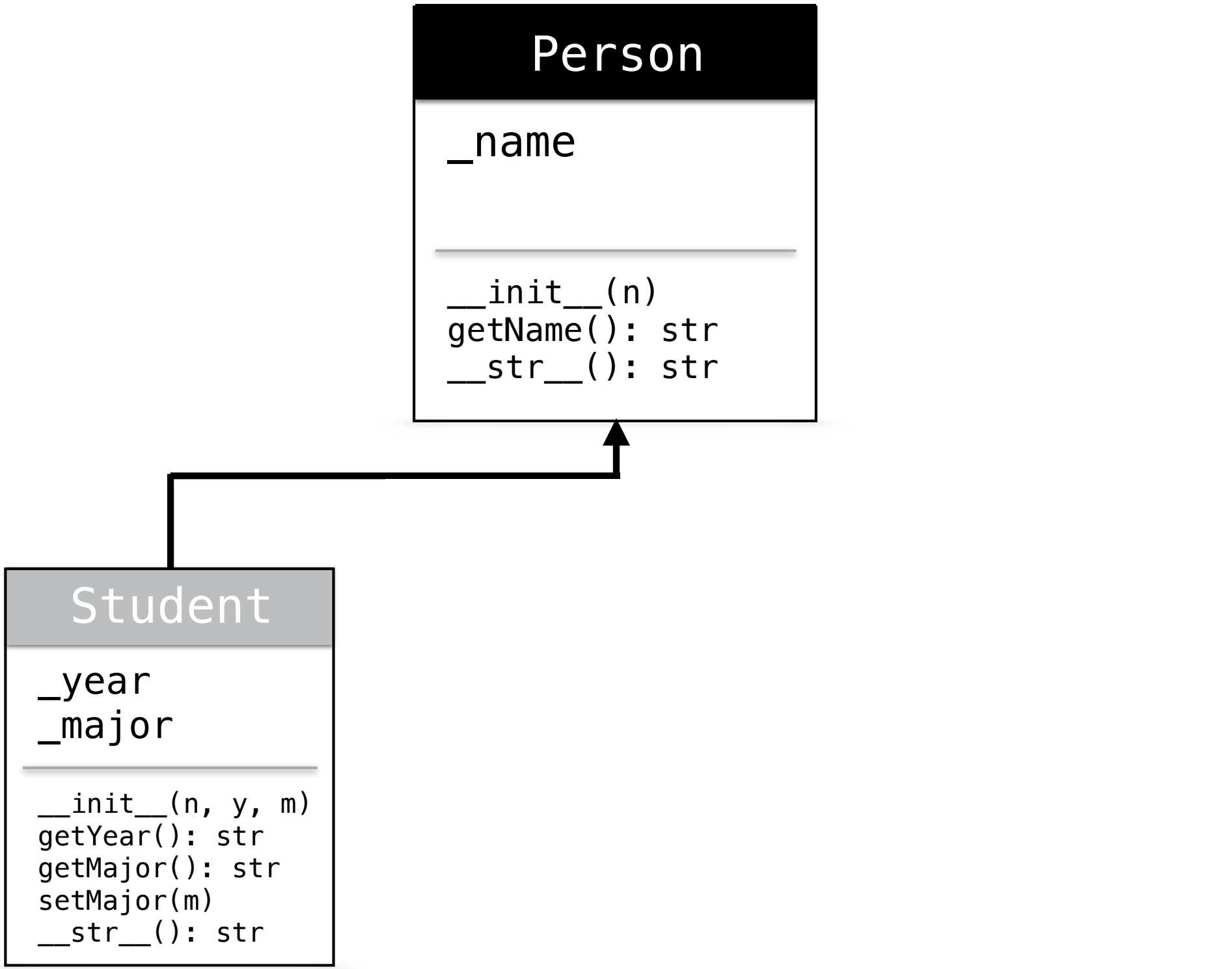
Student Class

Our Student class inherits from Person

Notice this does not include the inherited attribute _name since that is already provided in Person

This calls the __init__ method of Person

```
class Student(Person):
    __slots__ = ['_year', '_major']

    def __init__(self, name, year, major):
        # call __init__ of Person (the super class)
        super().__init__(name)
        self._year = year
        self._major = major

    def getYear(self):
        return self._year

    def getMajor(self):
        return self._major

    def setMajor(self, major):
        self._major = major

    def __str__(self):
        return '{}, {}, {}'.format(self._name, self._major, self._year)
```


Person
_name
__init__(n)
getName(): str
__str__(): str
Student
_year
_major
__init__(n, y, m)
getYear(): str
getMajor(): str
setMajor(m)
__str__(): str

Using the Student Class

```
>>> alex = Student("Alex", 2026, "Math")

>>> # inherited from Person
>>> alex.getName()
'Alex'
>>> # defined in Student
>>> alex.getMajor()
'Math'
>>> alex.setMajor("CS")
>>> alex.getMajor()
‘CS'

>>> print(alex)
‘Alex, CS, 2026’
```

This calls `__str__` of the Student class

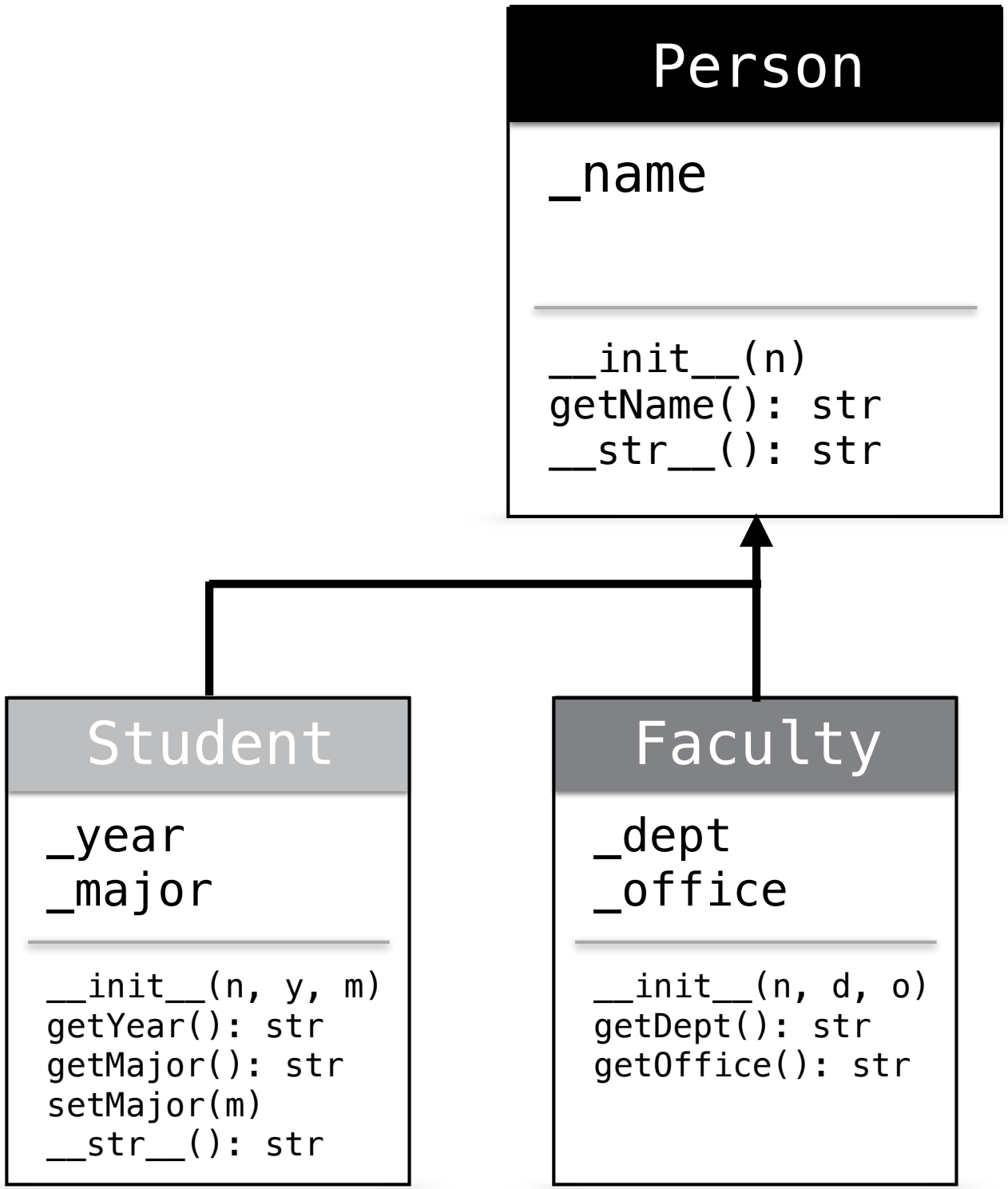
Faculty Class

Faculty inherits from Person

Does not include the inherited attribute _name from Person

```
class Faculty(Person):
    __slots__ = ['_dept', '_office']

    def __init__(self, name, dept, office):
        # call __init__ of Person (the super class)
        super().__init__(name)
        self._dept = dept
        self._office = office

    def getDept(self):
        return self._dept

    def getOffice(self):
        return self._office
```

Calls the __init__ method of Person


Person
_name
__init__(n)
getName(): str
__str__(): str
Student
_year
_major
__init__(n, y, m)
getYear(): str
getMajor(): str
setMajor(m)
__str__(): str
Faculty
_dept
_office
__init__(n, d, o)
getDept(): str
getOffice(): str

Using the Faculty Class

```
>>> jeannie = Faculty("Jeannie","CS","TCL 305")

>>> # inherited from Person
>>> jeannie.getName()
'Jeannie'
>>> # defined in Faculty
>>> jeannie.getDept()
'CS'
>>> print(jeannie)
Jeannie
>>> jeannie.getMajor()
AttributeError: 'Faculty' object has no
attribute 'getMajor'
```

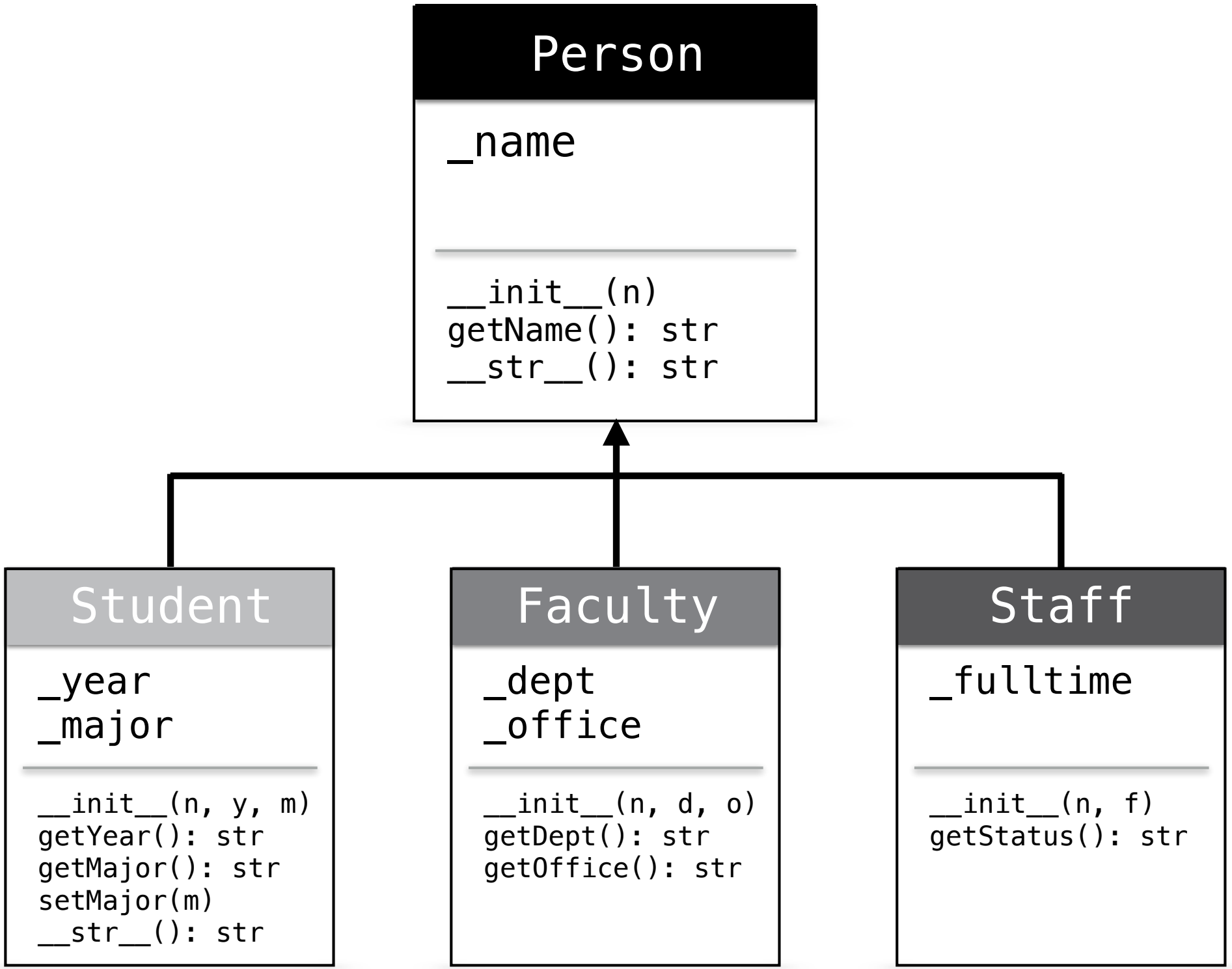
This calls `__str__` of the Person class

`getMajor` is a method of Student, not Person, and it is not defined in Faculty. This will not work.

Staff Class

```
class Staff(Person):
    # fulltime is a Boolean
    __slots__ = ['_fulltime']

    def __init__(self, name, fulltime):
        # call __init__ of super class
        super().__init__(name)
        self._fulltime = fulltime

    def getStatus(self):
        if self._fulltime:
            return "fulltime"
        return "partime"
```

Notice that getter methods can do more than just return an attribute directly


Person
_name
__init__(n)
getName(): str
__str__(): str
Student
_year
_major
__init__(n, y, m)
getYear(): str
getMajor(): str
setMajor(m)
__str__(): str
Faculty
_dept
_office
__init__(n, d, o)
getDept(): str
getOffice(): str
Staff
_fulltime
__init__(n, f)
getStatus(): str

Using the Staff Class

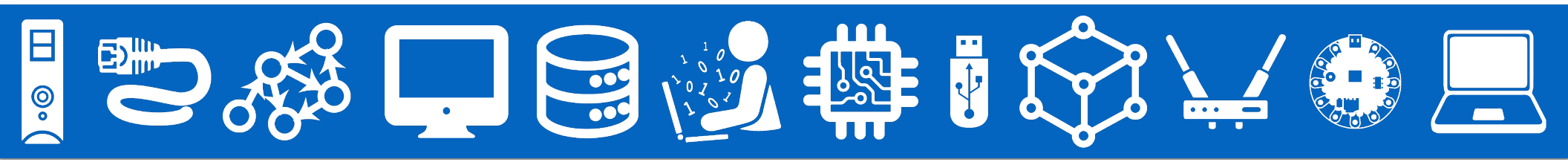
```
>>> stan = Staff("Stan", False)

>>> print(stan)
Stan

>>> stan.getStatus()
'parttime'
```

This calls __str__ of the Person class

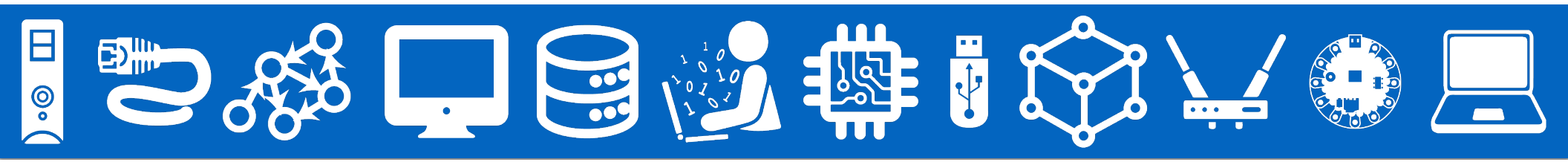
Summary

- Inheritance is a very useful feature of OOP
- Supports code reusability
- One superclass can be used for any number of subclasses in a hierarchy
- Can change the parent class without changing the subclasses
- More next time!

The end!



CS134:
Lab 8



Lab 8 Overview

- **User-defined Types!**
  - But not inheritance
- Review Lecture Materials from Wednesday & Friday!