Summary

There are four major principles of object oriented programming:

- Encapsulation
- Composition
- Inheritance
- Polymorphism

Encapsulation

In OOP objects are composed of both data and methods. Encapsulation refers to the bundling of these two together into a single object – as opposed to separating data and functions that operate on that data in non-OO programming.

Typically this also leads to implementation details of the data and methods of an object being hidden behind the methods that that function offers. This is a good way to protect data from invalid modification.

Encapsulation Example

```
# Define a class
class MyCounter:

    def __init__(self, count):
        self.__secretCount = count

    def count(self):
        self.__secretCount += 1

    def show(self):
       print self.__secretCount
```

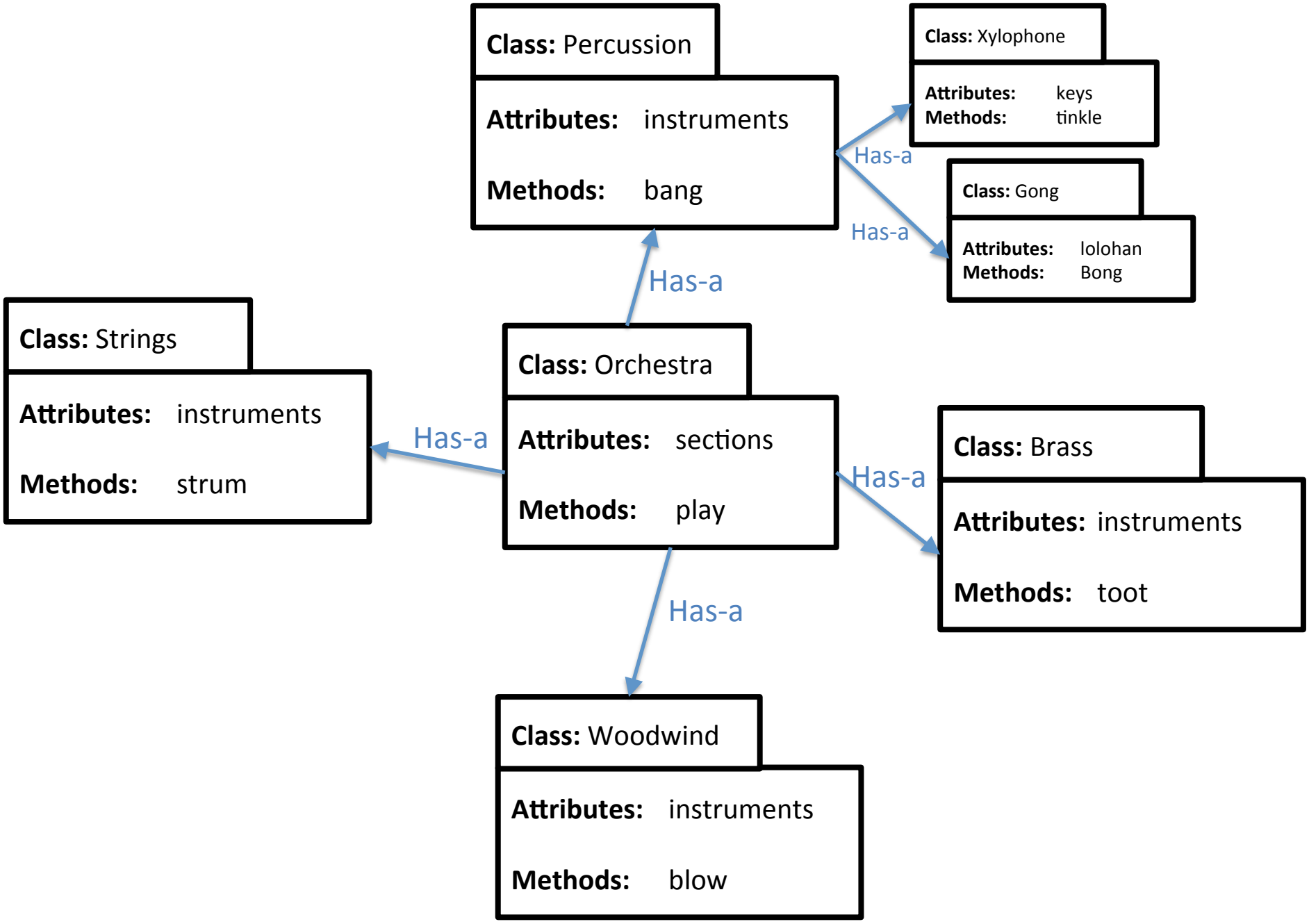
Encapsulation Example

```
# Use the class
counter = MyCounter (1)
counter.count()

print counter.__secretCount

counter.show()
```

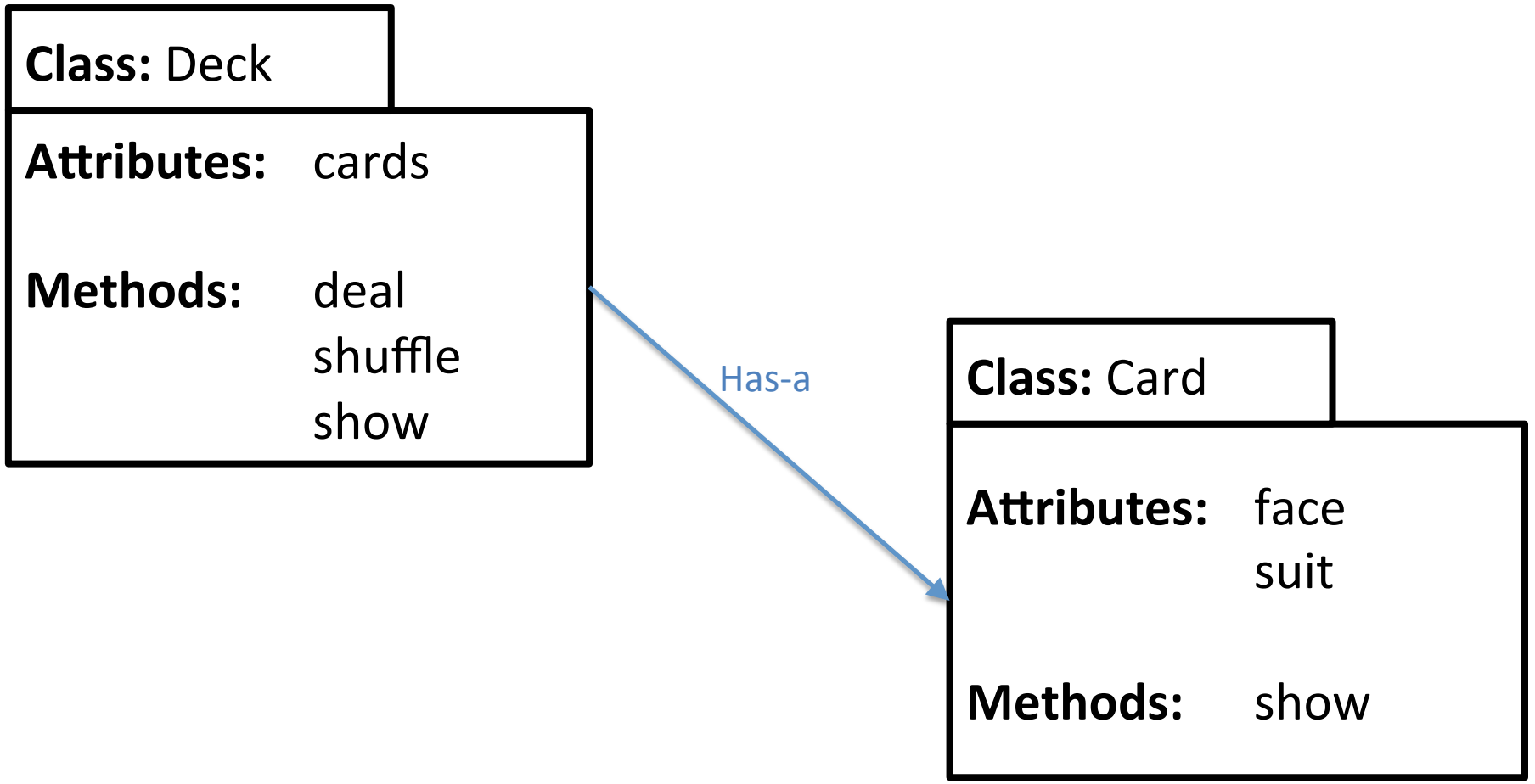
Composition

In OOP composition refers to the fact that to build sophisticated objects we compose them out of simpler objects

This is one of the big advantages of OOP as it promotes code reuse, leads to easy to follow code, and reduces the chances of errors creeping into code


Class: Percussion
Attributes: instruments
Methods: bang
Class: Xylophone
Attributes: keys
Methods: tinkle
Class: Gong
Attributes: lolohan
Methods: Bong
Has-a
Has-a
Has-a
Class: Strings
Attributes: instruments
Methods: strum
Class: Orchestra
Attributes: sections
Methods: play
Has-a
Has-a
Class: Brass
Attributes: instruments
Methods: toot
Has-a
Class: Woodwind
Attributes: instruments
Methods: blow

A simpler Composition Example



Composition Example

```
# The card class
class Card:
        # A constructor called when an object of
        # the class is instantiated.
        def __init__(self, suit, face):
            self.suit = suit
            self.face = face
        # A class method that prints a card
        def show(self):
            print(self.face + " of " + self.suit)
```

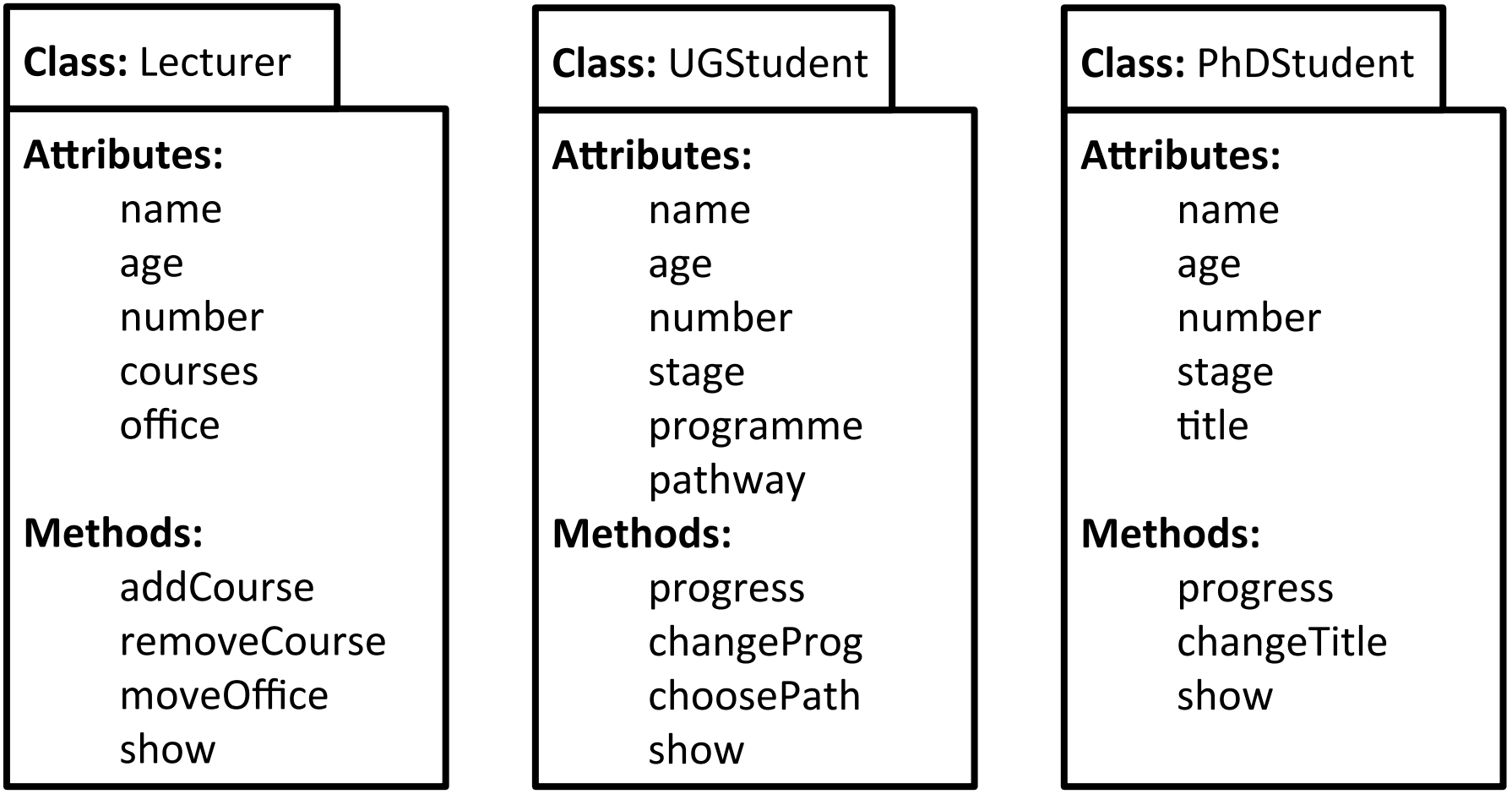
Composition Example

```
import random

# The deckclass
class Deck:

    # A constructor called when an object of the class is instantiated.
    def __init__(self):
        self.cards = list()
        for suit in ['Hearts', 'Diamonds', 'Spades', 'Clubs']:
            for face in ['2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K', 'A']:
                self.cards.append(Card(suit, face))

    # A class method that prints the deck
    def show(self):
        for c in self.cards:
            c.show()

    # A class method that shuffles the deck
    def shuffle(self):
        random.shuffle(self.cards)

    # A class method that deals a card from the deck
    def deal(self):
        return self.cards.pop()
```

Inheritance

In OOP we can use inheritance to build hierarchies of object definitions.

A **child class** inherits the properties (data and methods) of a **parent class**


Class: Lecturer
Attributes:
name
age
number
courses
office
Methods:
addCourse
removeCourse
moveOffice
show
Class: UGStudent
Attributes:
name
age
number
stage
programme
pathway
Methods:
progress
changeProg
choosePath
show
Class: PhDStudent
Attributes:
name
age
number
stage
title
Methods:
progress
changeTitle
show

Class: Lecturer

Attributes:

- **name**
- **age**
- **number**
- courses
- office

Methods:

- addCourse
- removeCourse
- moveOffice
- **show**

Class: UGStudent

Attributes:

- **name**
- **age**
- **number**
- stage
- programme
- pathway

Methods:

- progress
- changeProg
- choosePath
- **show**

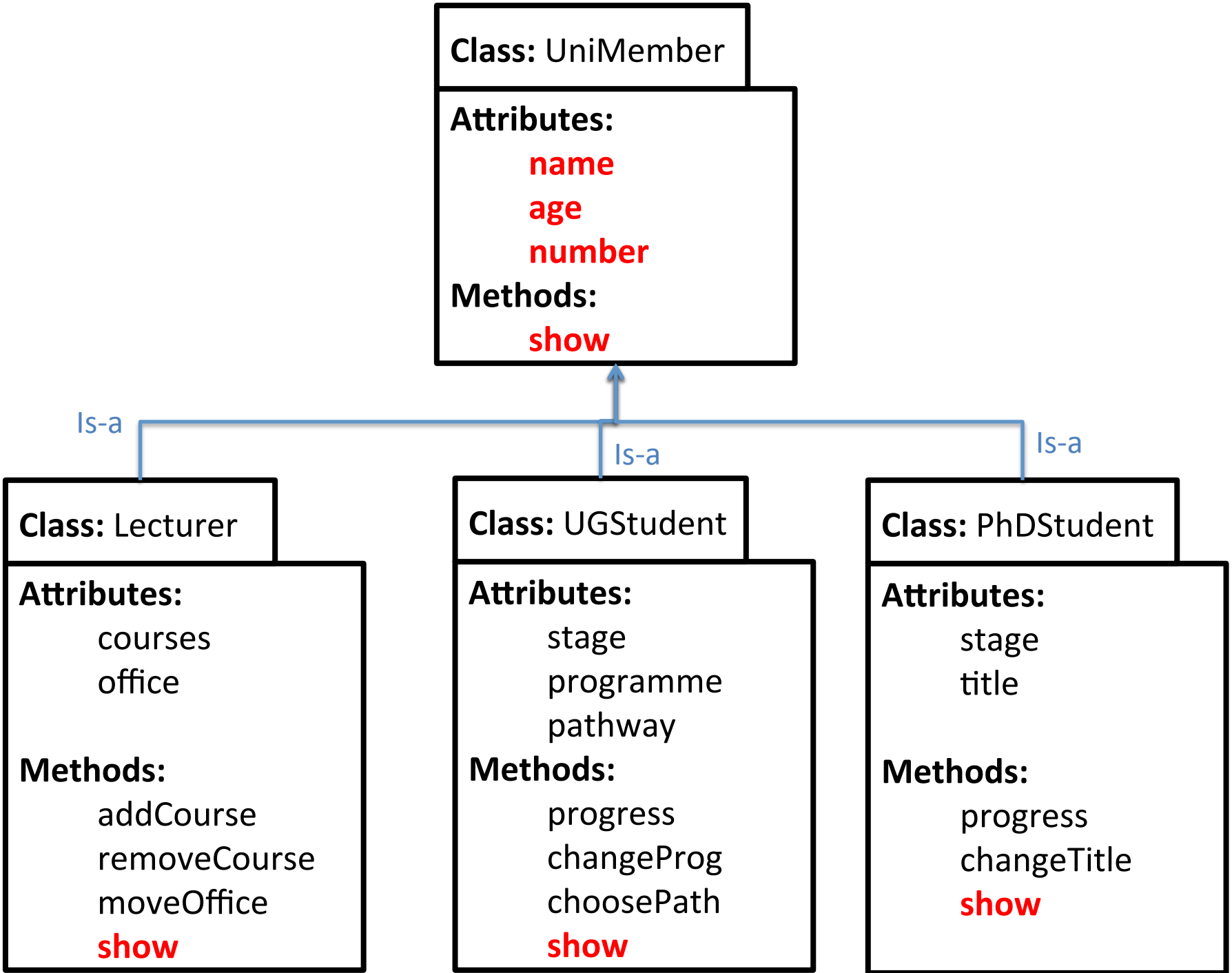
Class: PhDStudent

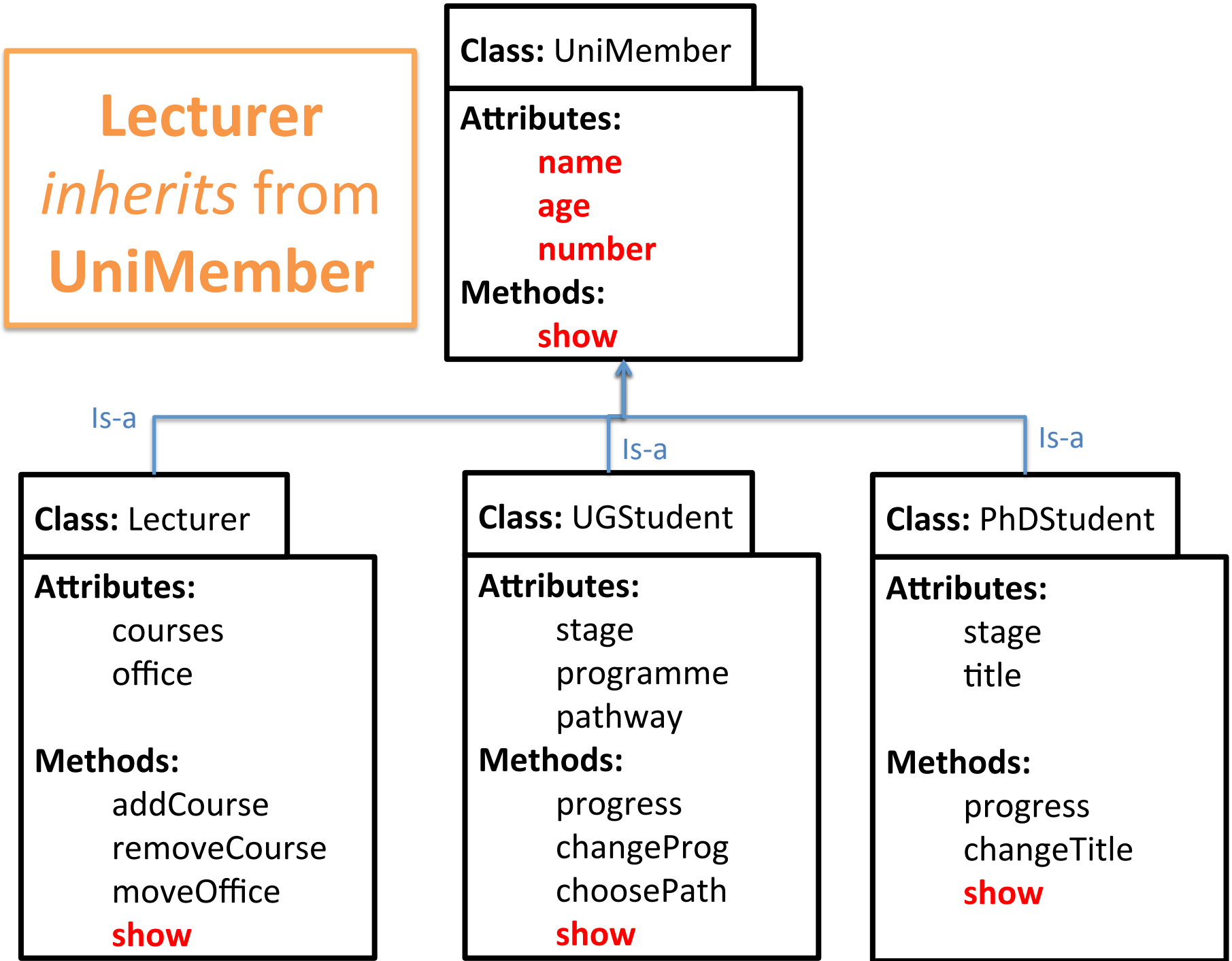
Attributes:

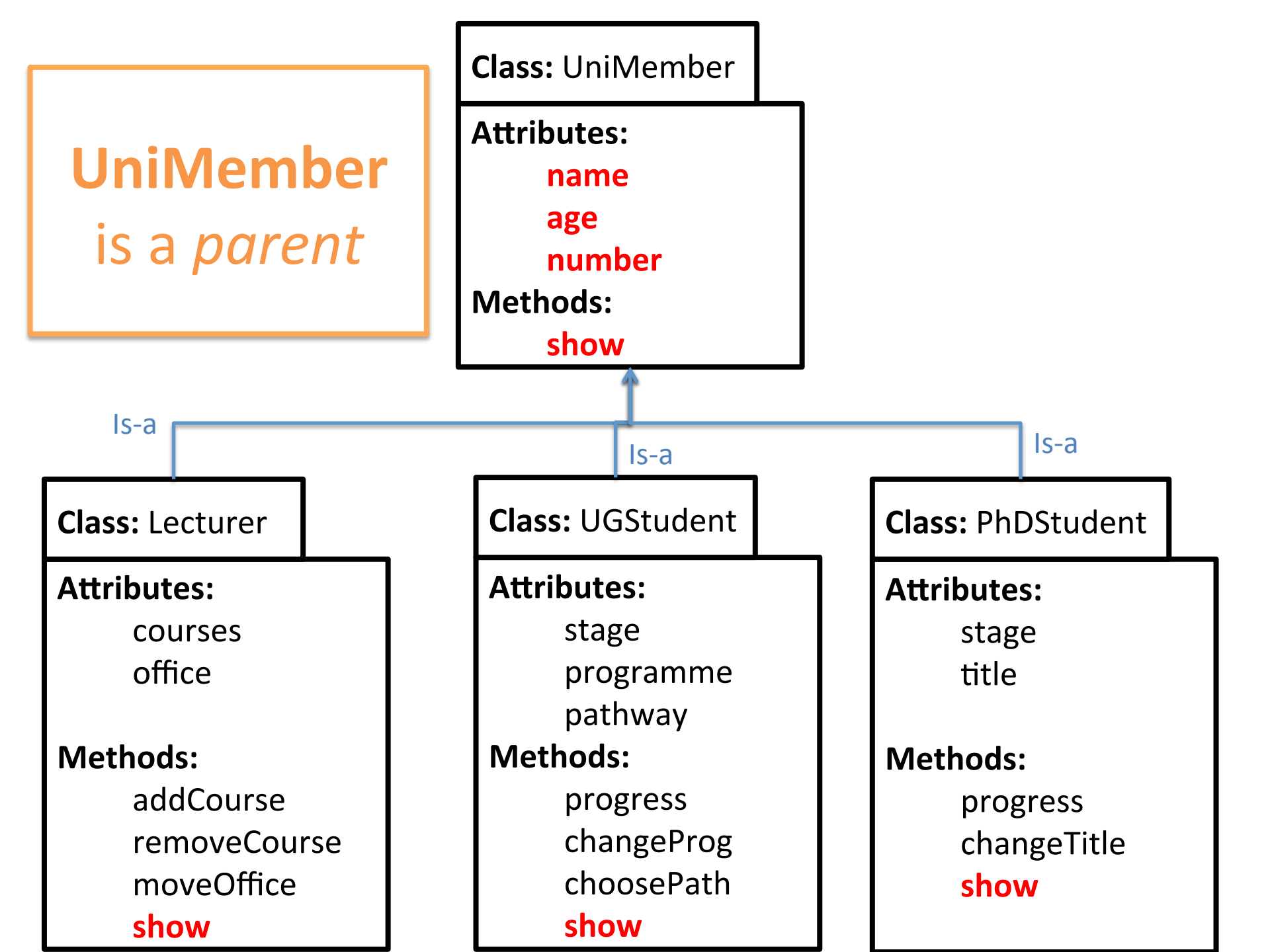
- **name**
- **age**
- **number**
- stage
- title

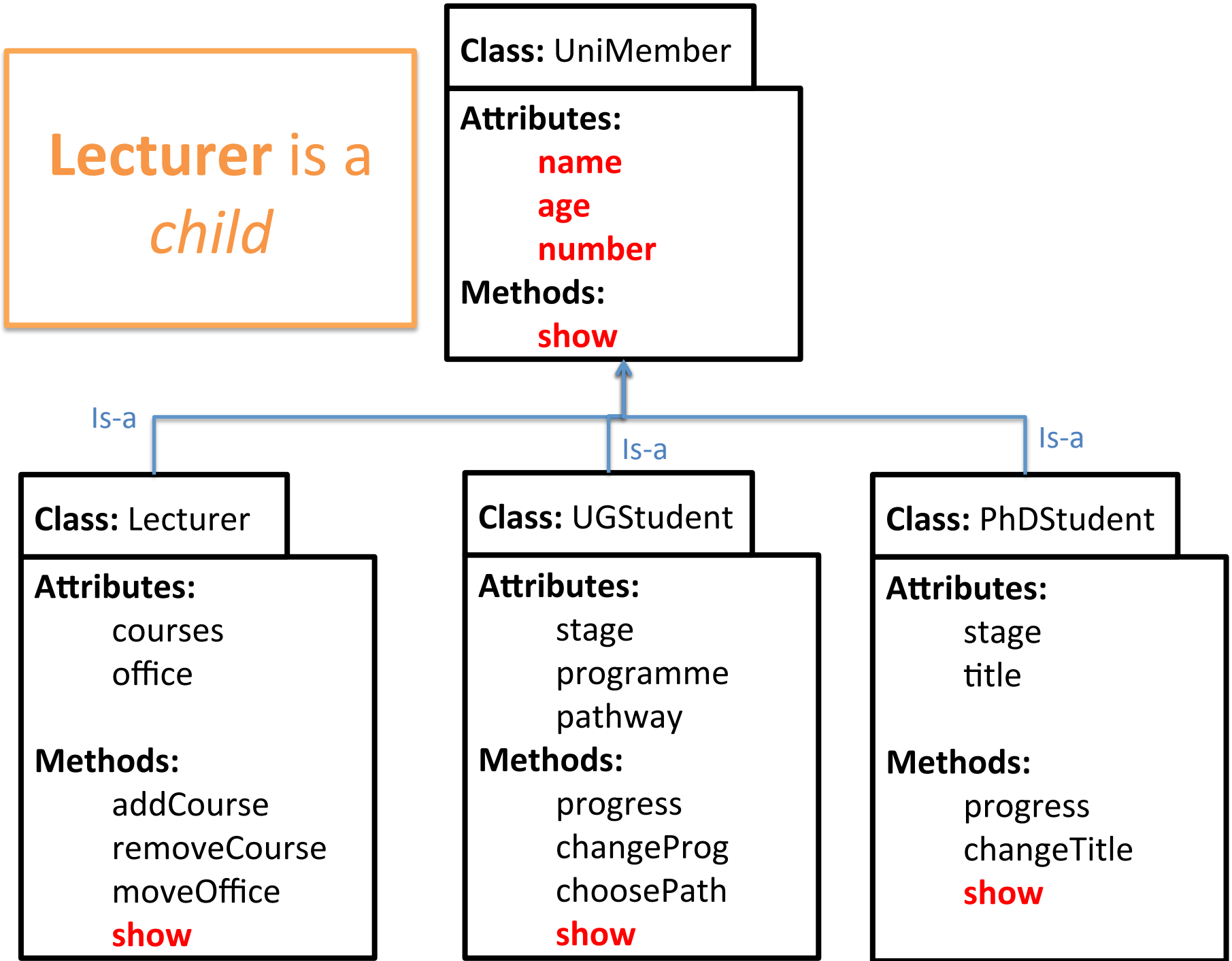
Methods:

- progress
- changeTitle
- **show**

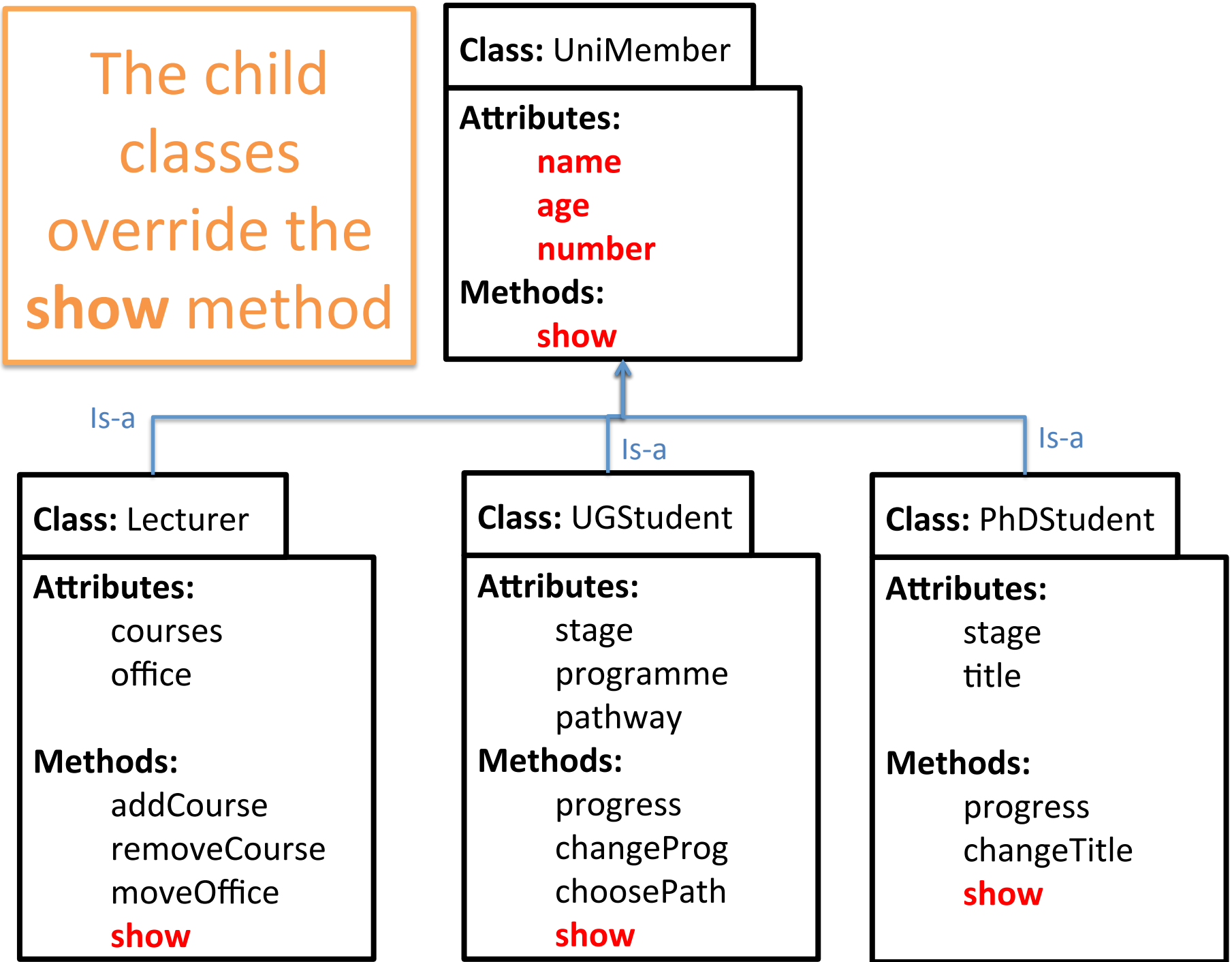

Class: UniMember
Attributes:
name
age
number
Methods:
show
Is-a
Is-a
Is-a
Class: Lecturer
Attributes:
courses
office
Methods:
addCourse
removeCourse
moveOffice
show
Class: UGStudent
Attributes:
stage
programme
pathway
Methods:
progress
changeProg
choosePath
show
Class: PhDStudent
Attributes:
stage
title
Methods:
progress
changeTitle
show


Lecturer
inherits from
UniMember
Class: UniMember
Attributes:
name
age
number
Methods:
show
Is-a
Is-a
Is-a
Class: Lecturer
Attributes:
courses
office
Methods:
addCourse
removeCourse
moveOffice
show
Class: UGStudent
Attributes:
stage
programme
pathway
Methods:
progress
changeProg
choosePath
show
Class: PhDStudent
Attributes:
stage
title
Methods:
progress
changeTitle
show


UniMember
is a parent
Class: UniMember
Attributes:
name
age
number
Methods:
show
Is-a
Is-a
Is-a
Class: Lecturer
Attributes:
courses
office
Methods:
addCourse
removeCourse
moveOffice
show
Class: UGStudent
Attributes:
stage
programme
pathway
Methods:
progress
changeProg
choosePath
show
Class: PhDStudent
Attributes:
stage
title
Methods:
progress
changeTitle
show


Lecturer is a child
Class: UniMember
Attributes:
name
age
number
Methods:
show
Is-a
Is-a
Is-a
Class: Lecturer
Attributes:
courses
office
Methods:
addCourse
removeCourse
moveOffice
show
Class: UGStudent
Attributes:
stage
programme
pathway
Methods:
progress
changeProg
choosePath
show
Class: PhDStudent
Attributes:
stage
title
Methods:
progress
changeTitle
show

The child classes override the **show** method



Inheritance Syntax

```
class Parent:

    def __init__(self, attr1, attr2):
        self.attr1 = attr1
        self.attr2 = attr2


class Child(Parent):

    def __init__(self, attr1, attr2, attr3):
        Parent.__init__(self, attr1, attr2)
        self.attr3 = attr3
```

Inheritance Syntax

```
class Parent:

    def __init__(self, attr1, attr2):
        self.attr1 = attr1
        self.attr2 = attr2
```

Parent class is defined as normal - nothing special about it.

```
class Child(Parent):

    def __init__(self, attr1, attr2, attr3):
        Parent.__init__(self, attr1, attr2)
        self.attr3 = attr3
```

Inheritance Syntax

```
class Parent:

    def __init__(self, attr1, attr2):
        self.attr1 = attr1
        self.attr2 = attr2


class Child(Parent):

    def __init__(self, attr1, attr2, attr3):
        Parent.__init__(self, attr1, attr2)
        self.attr3 = attr3
```

To specify the parent-child relationship we use the parent class name in brackets after the child class definition

Inheritance Syntax

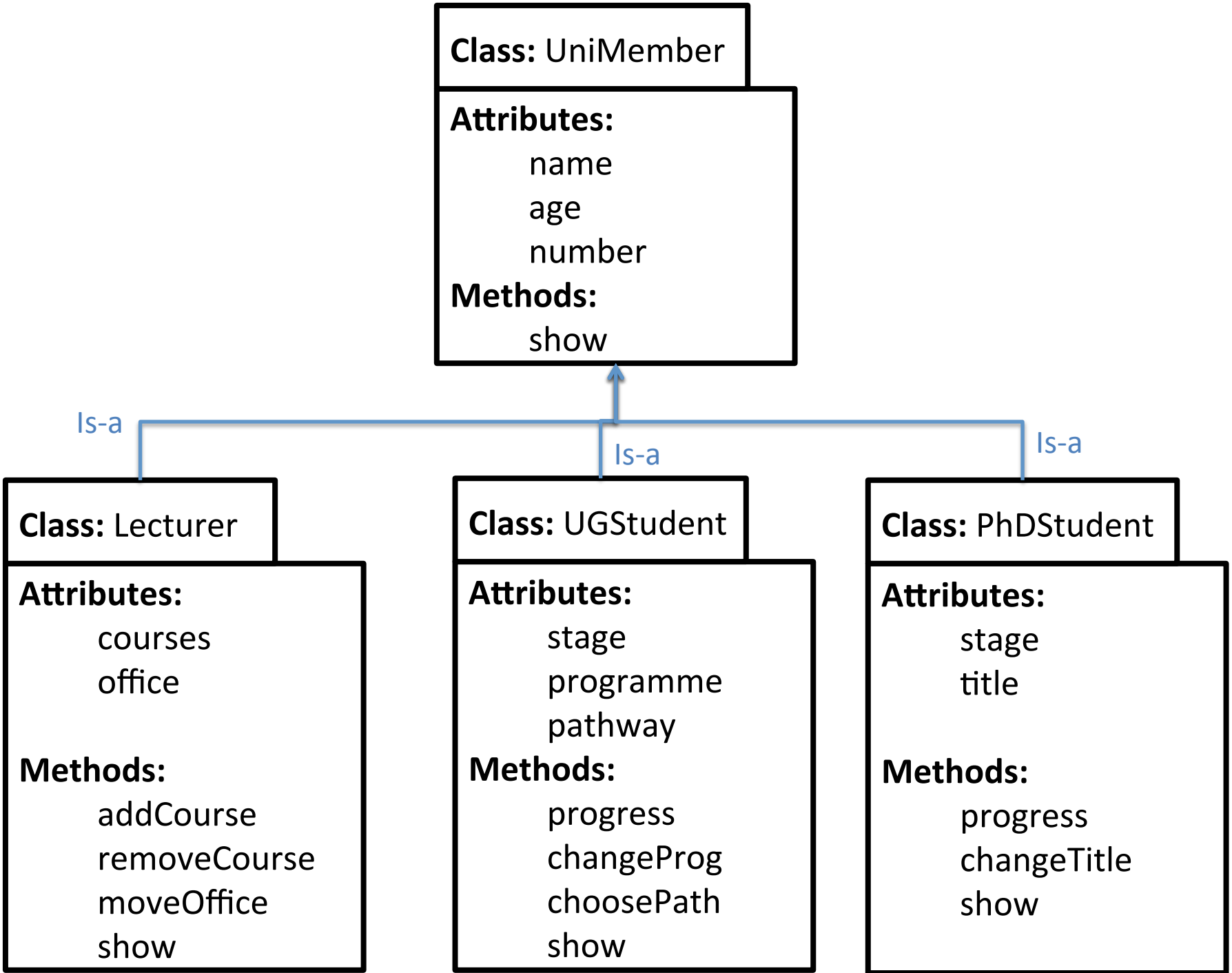
```
class Parent:

    def __init__(self, attr1, attr2):
        self.attr1 = attr1
        self.attr2 = attr2


class Child(Parent):

    def __init__(self, attr1, attr2, attr3):
        Parent.__init__(self, attr1, attr2)
        self.attr3 = attr3
```

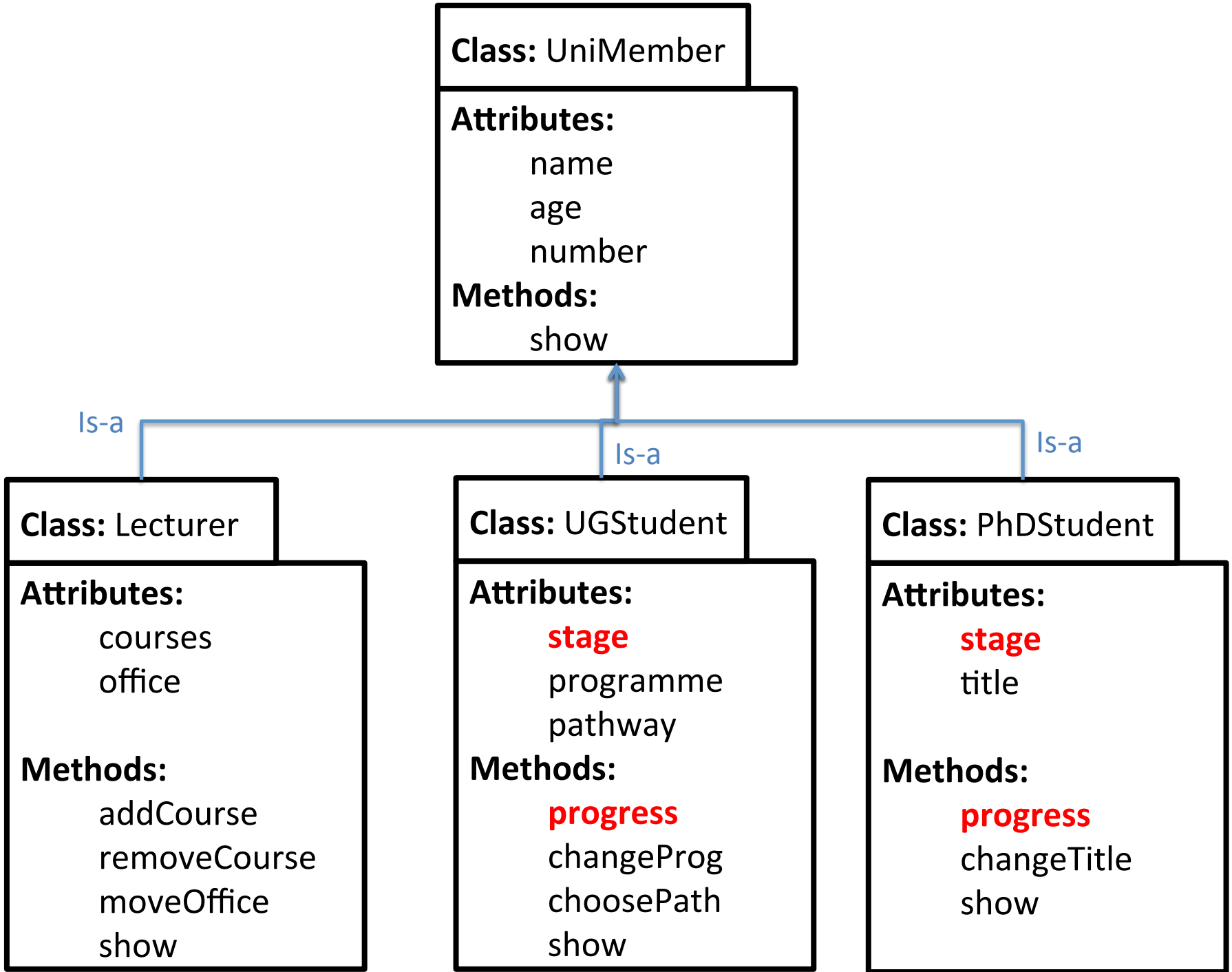
The child constructor takes the same parameters as the parent constructor, plus whatever extra is required

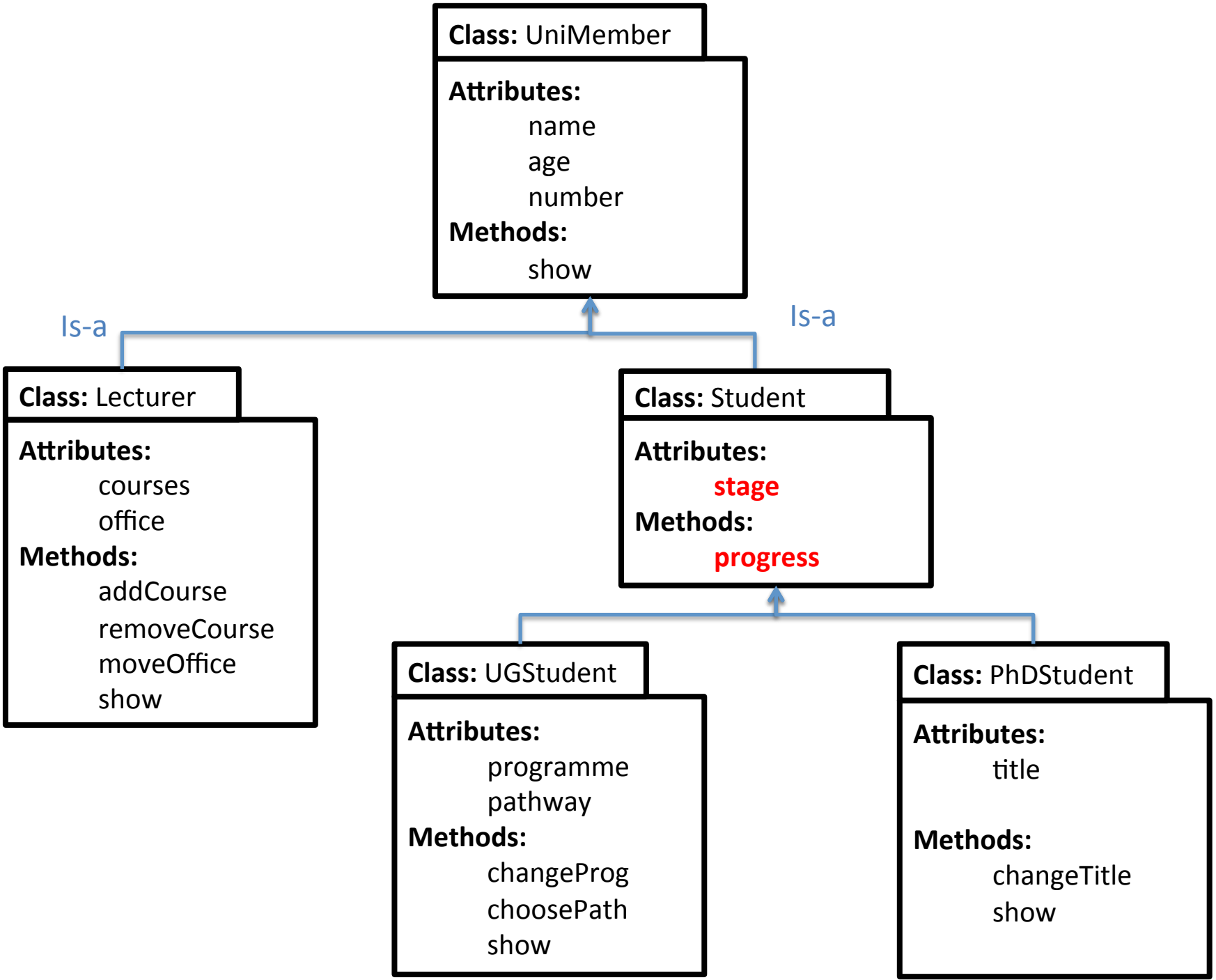
Inheritance Syntax

```
class Parent:

    def __init__(self, attr1, attr2):
        self.attr1 = attr1
        self.attr2 = attr2


class Child(Parent):

    def __init__(self, attr1, attr2, attr3):
        Parent.__init__(self, attr1, attr2)
        self.attr3 = attr3
```

We call the parent constructor to deal with the parent class attributes


Class: UniMember
Attributes:
name
age
number
Methods:
show
Is-a
Is-a
Is-a
Class: Lecturer
Attributes:
courses
office
Methods:
addCourse
removeCourse
moveOffice
show
Class: UGStudent
Attributes:
stage
programme
pathway
Methods:
progress
changeProg
choosePath
show
Class: PhDStudent
Attributes:
stage
title
Methods:
progress
changeTitle
show


Class: UniMember
Attributes:
name
age
number
Methods:
show
Is-a
Is-a
Is-a
Class: Lecturer
Attributes:
courses
office
Methods:
addCourse
removeCourse
moveOffice
show
Class: UGStudent
Attributes:
stage
programme
pathway
Methods:
progress
changeProg
choosePath
show
Class: PhDStudent
Attributes:
stage
title
Methods:
progress
changeTitle
show


Class: UniMember
Attributes:
name
age
number
Methods:
show
Is-a
Is-a
Class: Lecturer
Attributes:
courses
office
Methods:
addCourse
removeCourse
moveOffice
show
Class: Student
Attributes:
stage
Methods:
progress
Class: UGStudent
Attributes:
programme
pathway
Methods:
changeProg
choosePath
show
Class: PhDStudent
Attributes:
title
Methods:
changeTitle
show

Using Parent Methods Syntax

```
class Parent:

    def __init__(self, attr1, attr2):
        self.attr1 = attr1
        self.attr2 = attr2

    def show( )
        print(self.attr1 + " " + self.attr2)
```

Using Parent Methods Syntax

```
class Parent:

    def __init__(self, attr1, attr2):
        self.attr1 = attr1
        self.attr2 = attr2

    def show( )
        print(self.attr1 + " " + self.attr2)
```

Parent class with a **show** method

Inheritance Syntax

```
class Child(Parent):

    def __init__(self, attr1, attr2, attr3):
        Parent.__init__(self, attr1, attr2)
        self.attr3 = attr3

    def show( )
        print(self.attr1 + " " + self.attr2)
        print(self.attr3)
```

Inheritance Syntax

```
class Child(Parent):

    def __init__(self, attr1, attr2, attr3):
        Parent.__init__(self, attr1, attr2)
        self.attr3 = attr3

    def show( )
        print(self.attr1 + " " + self.attr2)
        print(self.attr3)
```

Child class overrides the parent class method with a new version

Inheritance Syntax

```
class Child(Parent):

    def __init__(self, attr1, attr2, attr3):
        Parent.__init__(self, attr1, attr2)
        self.attr3 = attr3

    def show( )
        Parent.show(self)
        print(self.attr3)
```

Inheritance Syntax

```
class Child(Parent):

    def __init__(self, attr1, attr2, attr3):
        Parent.__init__(self, attr1, attr2)
        self.attr3 = attr3

    def show( )
        Parent.show(self)
        print(self.attr3)
```

Use the **Parent** version of **show** instead of repeating this code

Polymorphism

Polymorphism refers to the ability to treat different child classes the same way as long as their parent class provides a common interface

For example, given a base class *shape,* polymorphism enables the programmer to define different *area* methods for any number of derived classes, such as circles, rectangles and triangles. No matter what shape an object is, applying the *area* method to it will return the correct results.

Polymorphism Example

```
# Make some objects
lec1 = Lecturer("Albert", 40, 450312, "K023")
lec2 = Lecturer("Sarah", 56, 451234, "D024")

stu1 = UGStudent("Mary", 22, 321005, 3, "Science")
stu2 = UGStudent("Simon", 18, 321005, 1, "Arts")

phd1 = PhDStudent("Niels", 32, 400120, 7,
"Deliberations on the Economy")
phd2 = PhDStudent("Jane", 23, 400323, 1, "Big
Idea")
```

Polymorphism Example

```
# Put the objects in a list and use their show methods
members = [lec1, lec2, stu1, stu2, phd1, phd2]
for m in members:
    m.show()
    print("------")
```