CS1010S Programming Methodology

Lecture 10

Object-Oriented Programming

31 Oct 2018

Happy Diwali

Tue 6 Nov 2018

Makeup Tutorials arranged by tutors
Can attend any tutorial for next week

Practical Exam

- Week 13 Saturday, 17 Nov 2018, 12 to 6pm
  - Venues:
    - Programming Labs @ COM 1
    - Embeded Systems Labs @ COM 1
    - Media Teaching Labs @ AS6
    - Workstation Labs @ I3
  - Two sessions
  - Seating plan will be posted on Coursemology
    - Note: PL1 equipped with Apple Macs
    - Survey on Coursemology

Today's Agenda

- Scope of Variables

- Object-Oriented Programming (OOP)
  - Basics
  - Inheritance
  - Polymorphism

Global Variables

```
x = 10          # x is a global variable
def bar():
    print(x)

bar()
```

Output:
10

- Global variable: a variable defined outside any function (i.e. in the global scope).
  - global variable can be accessed in functions.

Local Variables

```
x = 10          # x is a global variable
def bar():
    x = 5       # local x shadows global x
    print(x)

bar()
print(x)
```

Output:
5
10

- Local variable: a variable defined inside a function.
  - local variable can be accessed inside the function only.

UnboundLocalError

```python
x = 10
def bar():
    print(x)
    x = x + 1

bar()
```

WTF?!

(World Taekwondo Foundation)

Output:

```
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    bar()
  File "<pyshell#3>", line 2, in bar
    print(x)
UnboundLocalError: local variable 'x'
referenced before assignment
```

Let's Figure This Out

```
x = 10
def bar():
    print(x)
    x = x + 1

bar()
```

x refers to the local variable, which is defined in the next line!

First time assign a value to x! So a local variable x will be created.

- Assignment operation creates a new local variable.

Let’s Fix it

```
x = 10
def bar():
    global x
    print(x)
    x = x + 1

bar()
print(x)
```

Make it clear x is a global variable

Output:
10
11

Nonlocal Variables

```
def bar(x):
    def do():
        nonlocal x
        x = x + 1
        print(x)
    print(x)
    return do

a = bar(10)
a()
```

x is a parameter of bar() function

Output:
10
11

- Nonlocal variable: a variable used in inner function that is defined in the outer function.
  - non-examinable

Today's Agenda

- Scope of Variables

- Object-Oriented Programming (OOP)
  - Basics
  - Inheritance
  - Polymorphism

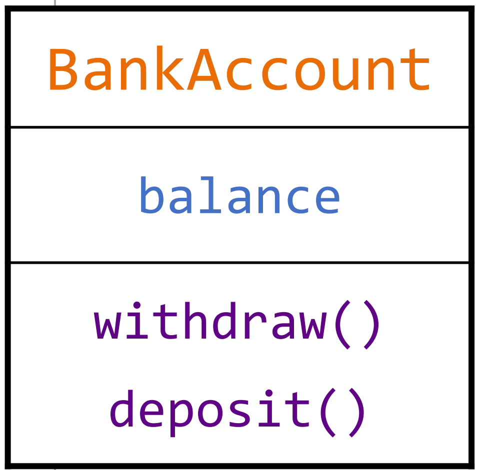
Class and Instance

- Class
  - A programming unit to model an entity.
    - e.g. a class to describe human
  - Serves as a blueprint to create real instances of the entity.
- Instance (object)
  - A particular object of a certain class.
    - e.g. a particular human called Taylor Swift

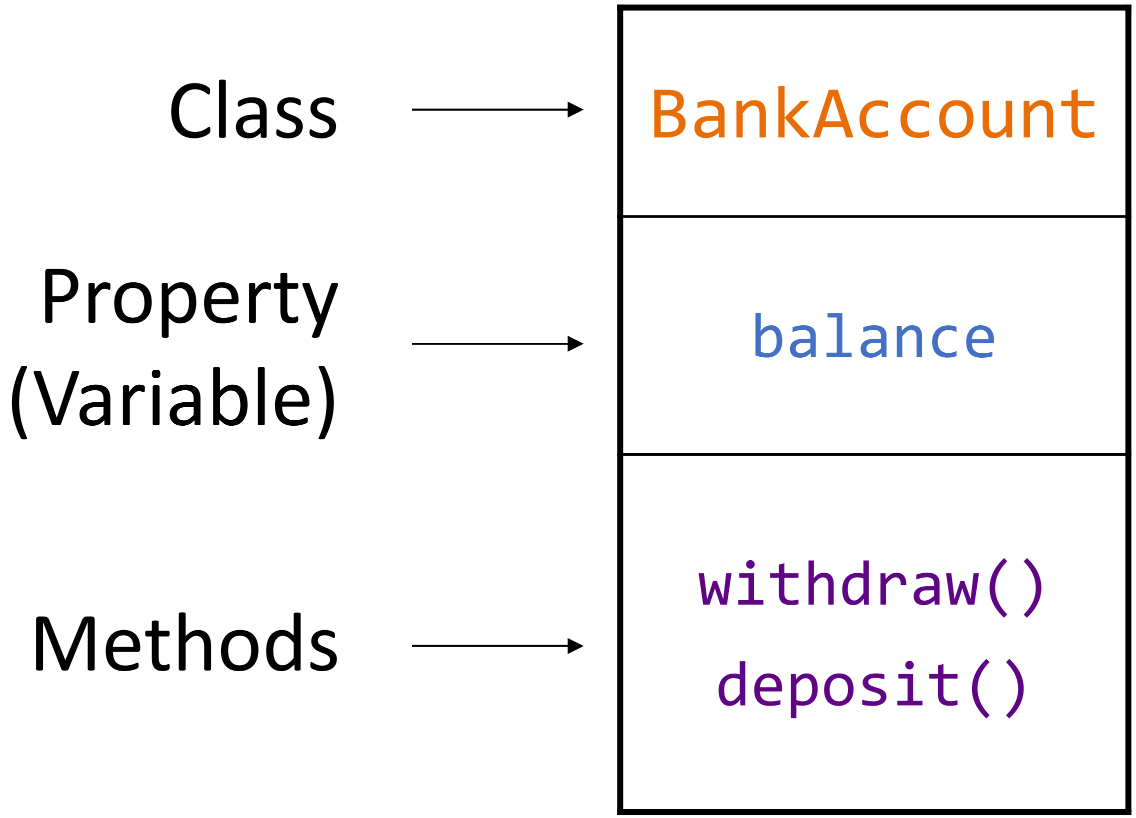
First example:

BankAccount

```python
class BankAccount(object):

    def __init__(self, initial_balance):
        self.balance = initial_balance

    def withdraw(self, amount):
        if self.balance > amount:
            self.balance -= amount
            return self.balance
        else:
            return "Money not enough"

    def deposit(self, amount):
        self.balance += amount
        return self.balance
```



Class Diagram



```python
class BankAccount(object):

  def __init__(self,
               initial_balance):
    self.balance = initial_balance

  def withdraw(self, amount):
    if self.balance > amount:
      self.balance -= amount
      return self.balance
    else:
      return "Money not enough"

  def deposit(self, amount):
    self.balance += amount
    return self.balance
```

```
>>> my_account = BankAccount(100)
>>> print(my_account)
<__main__.BankAccount object at
0x000000003AC6E80>

>>> my_account.withdraw(40)
60

>>> my_account.withdraw(200)
Money not enough

>>> my_account.deposit(20)
80
```

```python
class BankAccount(object):

  def __init__(self,
               initial_balance):
    self.balance = initial_balance

  def withdraw(self, amount):
    if self.balance > amount:
      self.balance -= amount
      return self.balance
    else:
      return "Money not enough"

  def deposit(self, amount):
    self.balance += amount
    return self.balance
```

```python
>>> ba1 = BankAccount(0)

>>> ba1.deposit(500)
500

>>> ba2 = BankAccount(400)

>>> amt = ba1.withdraw(50)
>>> amt
450

>>> ba2.deposit(amt)
850
```

Constructor

```
class BankAccount(object):
    def __init__(self, initial_balance): # constructor
        self.balance = initial_balance
    ...
```

- The constructor **`__init__`** is a special method.
  - Called automatically when a new instance is created.
  - If you never define any constructor by yourself, a default constructor will be given by Python (which does almost nothing).
  - Special methods have double underscores __ in front and behind the method name.
    - There exist other special methods such as __str__.

Method vs. Function

```
class BankAccount(object):
  def __init__(self,
               initial_balance):
    self.balance = initial_balance

  def withdraw(self, amount):
    if self.balance > amount:
      self.balance -= amount
      return self.balance
    else:
      return "Money not enough"

  def deposit(self, amount):
    self.balance += amount
    return self.balance
```

- A method is a function defined inside a class.
  - The first parameter of a method always refers to the object calling the method.
  - This parameter can be named anything, but traditionally it is named **`self`**.

Binding self to Object

```
class BankAccount(object):
  def __init__(self,
               initial_balance):
    self.balance = initial_balance

  def withdraw(self, amount):
    if self.balance > amount:
      self.balance -= amount
      return self.balance
    else:
      return "Money not enough"

  def deposit(self, amount):
    self.balance += amount
    return self.balance
```

- **self** is bound to the object calling the method.

```
>>> ba = BankAccount(100)

>>> ba.withdraw(50)
```

Dot Reference

```python
class BankAccount(object):
  def __init__(self,
               initial_balance):
    self.balance = initial_balance

  def withdraw(self, amount):
    if self.balance > amount:
      self.balance -= amount
      return self.balance
    else:
      return "Money not enough"

  def deposit(self, amount):
    self.balance += amount
    return self.balance
```

- We can access the properties/methods of an object using dot notation.

```python
>>> ba = BankAccount(100)
>>> ba.withdraw(50)
```

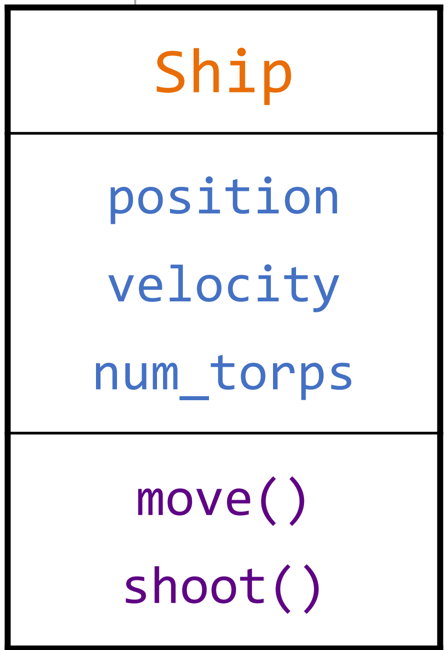
Second example:

Space Wars Simulator

* we focus on OO constructs, not game logic

Ship Class

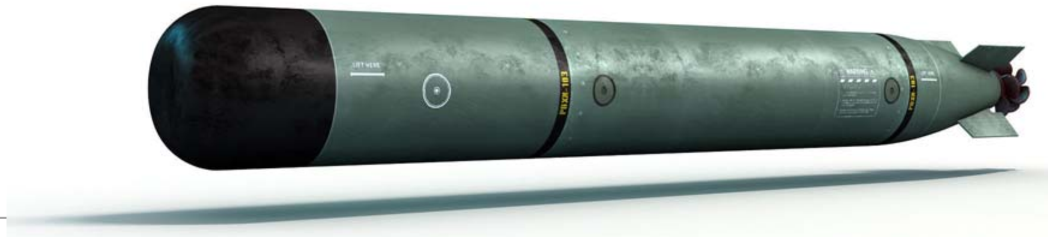
```
class Ship(object):
    def __init__(self, p, v, num_trops):
        self.position = p
        self.velocity = v
        self.num_torps = num_torps

    def move(self):
        # move to a random position
        pass

    def shoot(self, ship_name):
        # fire!
        pass
```



Torpedo Class

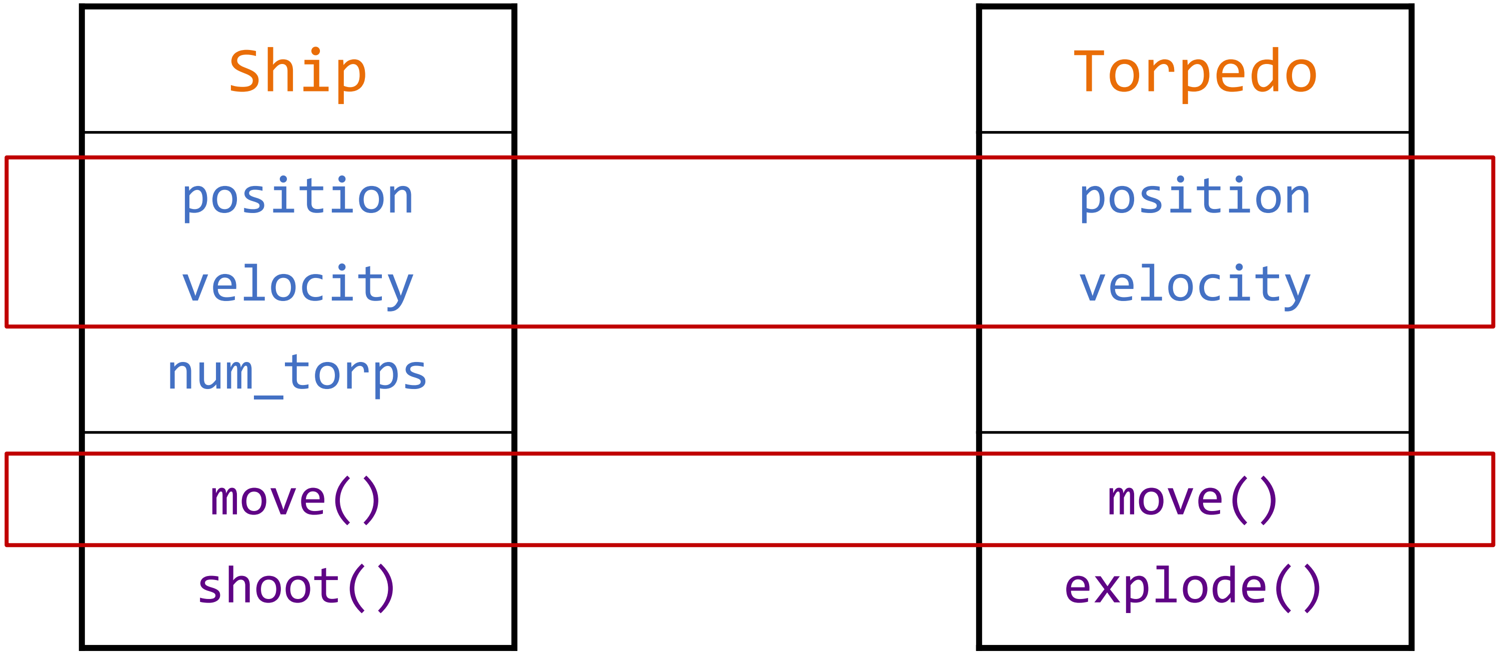
```
class Torpedo(object):
    def __init__(self, p, v):
        self.position = p
        self.velocity = v


    def move(self):
        # move to somewhere
        pass

    def explode(self):
        # bomb!
        pass
```

Torpedo
position velocity
move() explode()

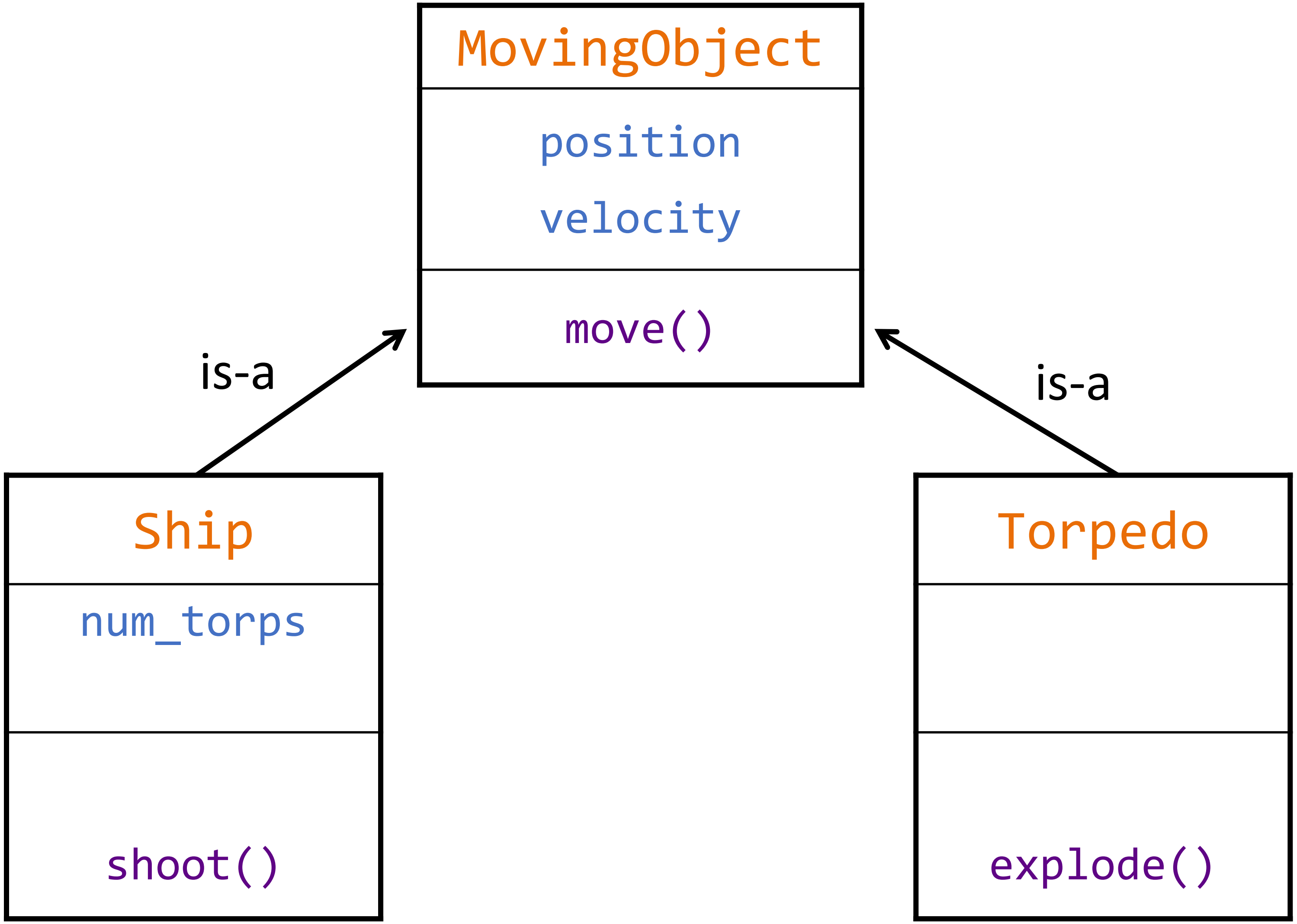


A Tale of Two Classes

- Two common properties
- One common behavior
- use inheritance to capture commonality



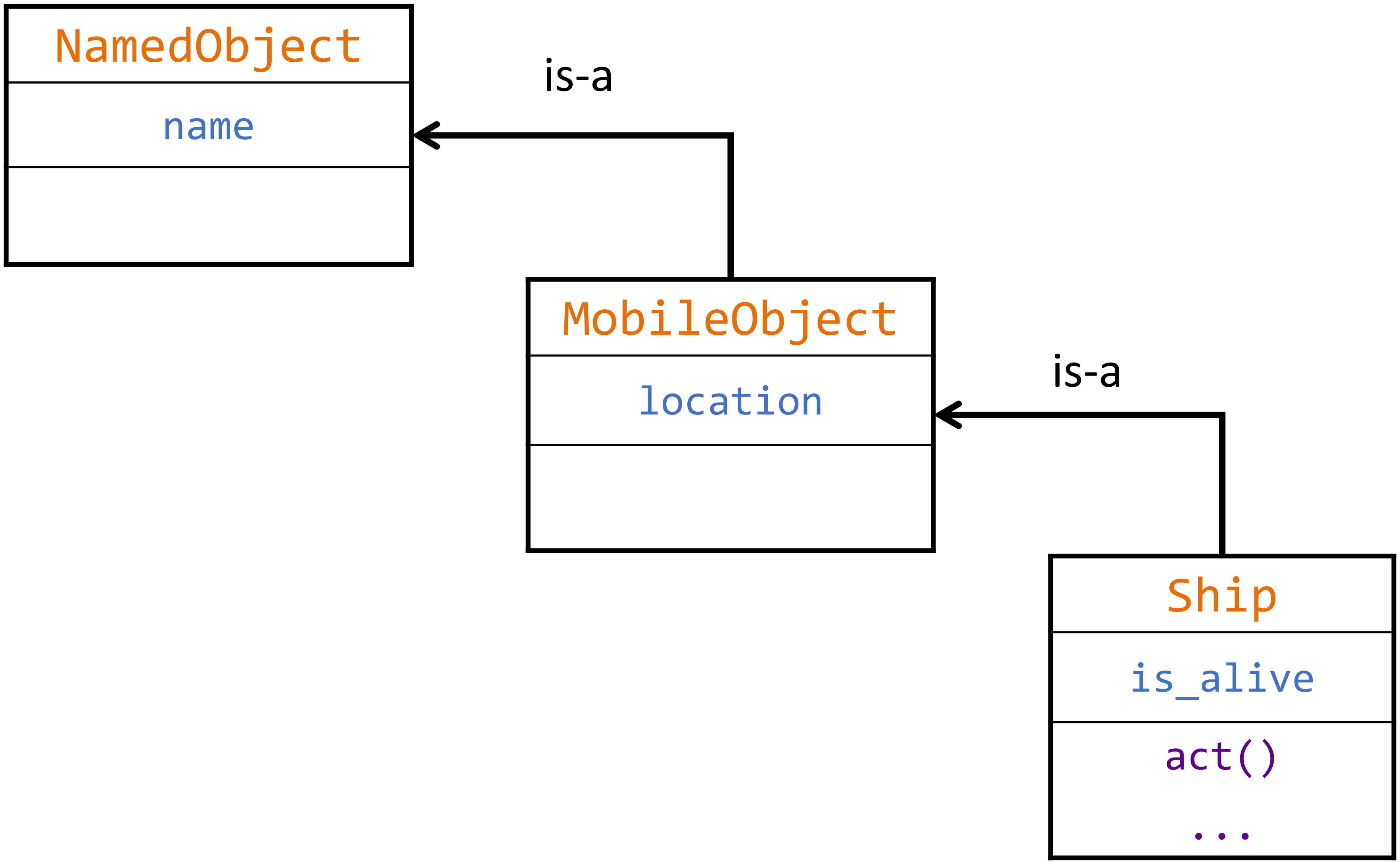
Inheritance



Superclass vs. Subclass

- Subclass is a kind of superclass.
  - Establishes a natural type hierarchy
- Subclass (child class) inherits methods and properties from superclass (parent class).
  - No need to rewrite the same code in subclass again.
  - Improve code reusability.

Example Class Hierarchy



```python
class NamedObject(object):
    def __init__(self, name):
        self.name = name
```

NamedObject is a superclass of MobileObject

```python
class MobileObject(NamedObject):
    def __init__(self, name, location):
        self.name = name
        self.location = location
```

MobileObject is a superclass of Ship

```python
class Ship(MobileObject):
    def __init__(self, name, birthplace, threshold):
        self.name = name
        self.location = birthplace
        self.threshold = threshold
        self.is_alive = True
    ...
```

```
class NamedObject(object):
    def __init__(self, name):
        self.name = name

class MobileObject(NamedObject):
    def __init__(self, name, location):
        self.name = name
        self.location = location

class Ship(MobileObject):
    def __init__(self, name, birthplace, threshold):
        self.name = name
        self.location = birthplace
        self.threshold = threshold
        self.is_alive = True
    ...
```

Same code as in superclass

Same code as in superclass

```
class NamedObject(object):
    def __init__(self, name):
        self.name = name

class MobileObject(NamedObject):
    def __init__(self, name, location):
        super().__init__(name)
        self.location = location

class Ship(MobileObject):
    def __init__(self, name, birthplace, threshold):
        super().__init__(name, birthplace)

        self.threshold = threshold
        self.is_alive = True
    ...
```

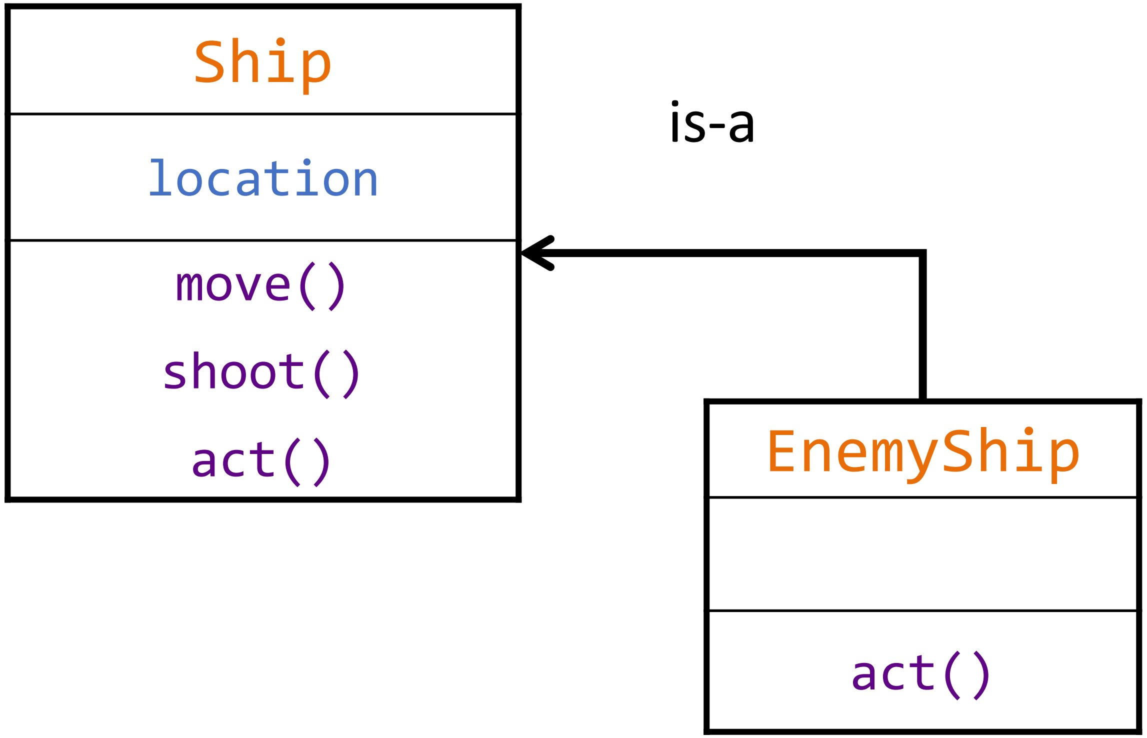
The super() Function

- super() refers to the direct superclass.
- One benefit of using super():
  - Improves code reusability
    - e.g. all names must be in lower case -> only need to revise the constructor of NamedObject class.

```
class NamedObject(object):
    def __init__(self, name):
        self.name = name.tolower()
```

Another Class Hierarchy

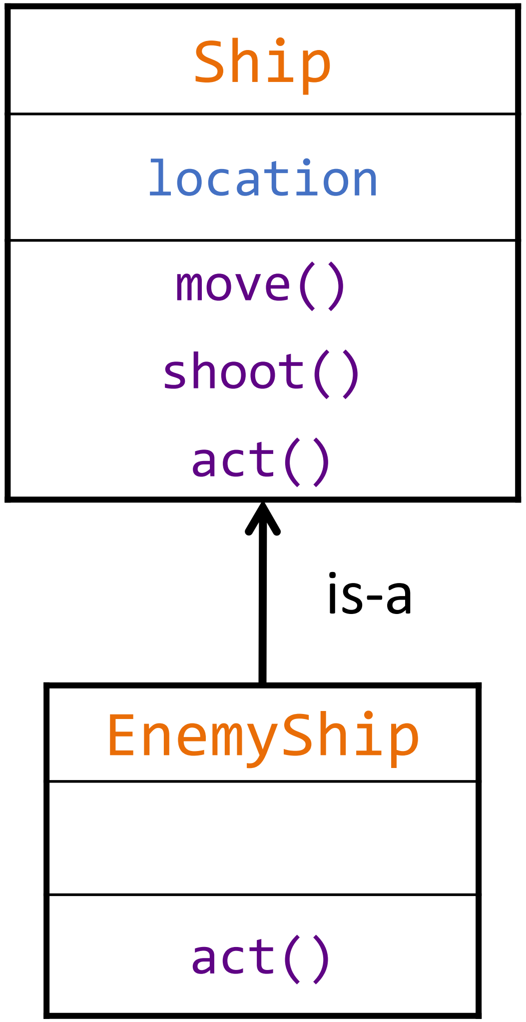
- EnemyShip is a subclass of Ship.
  - But it has its own version of method `act()`.



```python
class Ship(MobileObject):
    ...
    def act(self):
        new_location = self.location.random_neighbor()
        if new_location:
            self.move_to(new_location)
    ...

class EnemyShip(Ship):
    def act(self):
        ...
        if len(other_ships) == 0:
            super().act()
        else:
            ...
            self.shoot(random.choice(ship_names))
```

```
>>> enemy = EnemyShip()

>>> enemy.act()
```

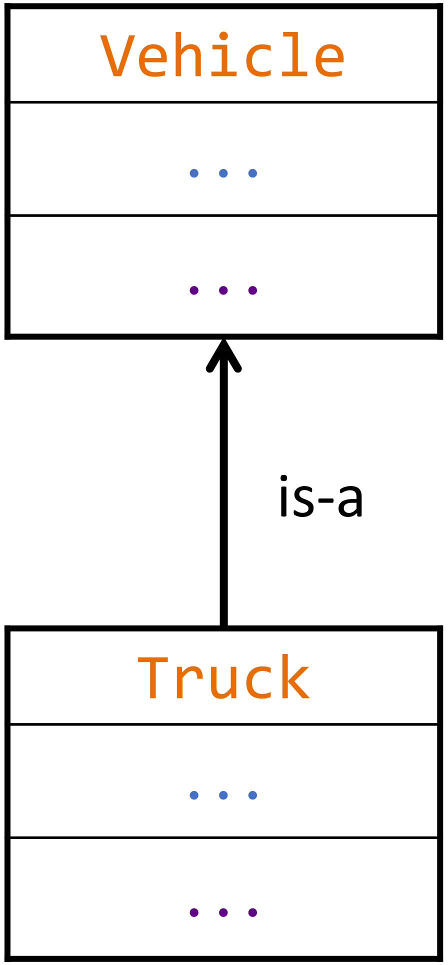
Method Overriding



- When `act()` is called by an EnemyShip instance, the version defined in the EnemyShip class will be invoked.
  - This is known as method overriding.

```python
class Ship(MobileObject):
    ...

class EnemyShip(Ship):
    def act(self):
        ships = list(filter(
                        lambda thing: isinstance(thing, Ship),
                        self.location.things))
        ...
        if len(other_ships) == 0:
            super().act()
        else:
            ...
            self.shoot(random.choice(ship_names))
```

isinstance vs. type



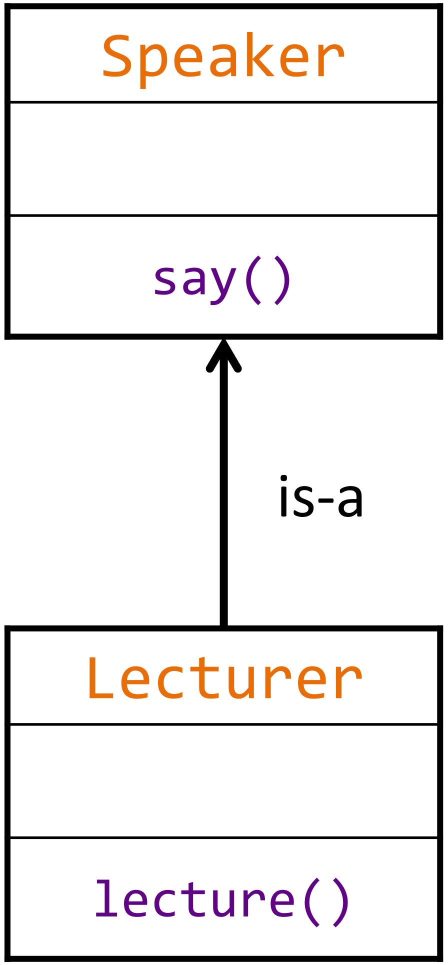
- Both are Python built-in functions.

```
>>> type(Vehicle()) == Vehicle
True

>>> isinstance(Vehicle(), Vehicle)
True

>>> type(Truck()) == Vehicle
False

>>> isinstance(Truck(), Vehicle)
True
```

Last example:

Singing Arrogant Speaker

Class Diagram



- Lecturer inherits the `say()` method from Speaker.
- A lecturer is a (kind of) speaker.
  - A lecturer can do anything a speaker can (i.e. say things) plus lecture.

```python
class Speaker(object):
    def say(self, stuff):
        print(stuff)

ah_beng = Speaker()
ah_beng.say("Hello World")  # print: Hello World

class Lecturer(Speaker):
    def lecture(self, stuff):
        self.say(stuff)
        self.say("You should be taking notes")

way_kay = Lecturer()
way_kay.say("Quiz today")
```

Q: What is the output?

- Definition of `say()` not found in `Lecturer` class.
  - Python will go through the class hierarchy to look for it in the superclass.

```
class Speaker(object):
    def say(self, stuff):
        print(stuff)

class Lecturer(Speaker):
    def lecture(self, stuff):
        self.say(stuff)
        self.say("You should be taking notes")

way_kay = Lecturer()
way_kay.say("Quiz today")  # print: Quiz today
```

- Invoke `lecture()` defined in `Lecturer` class, which invoke `say()` method of superclass.

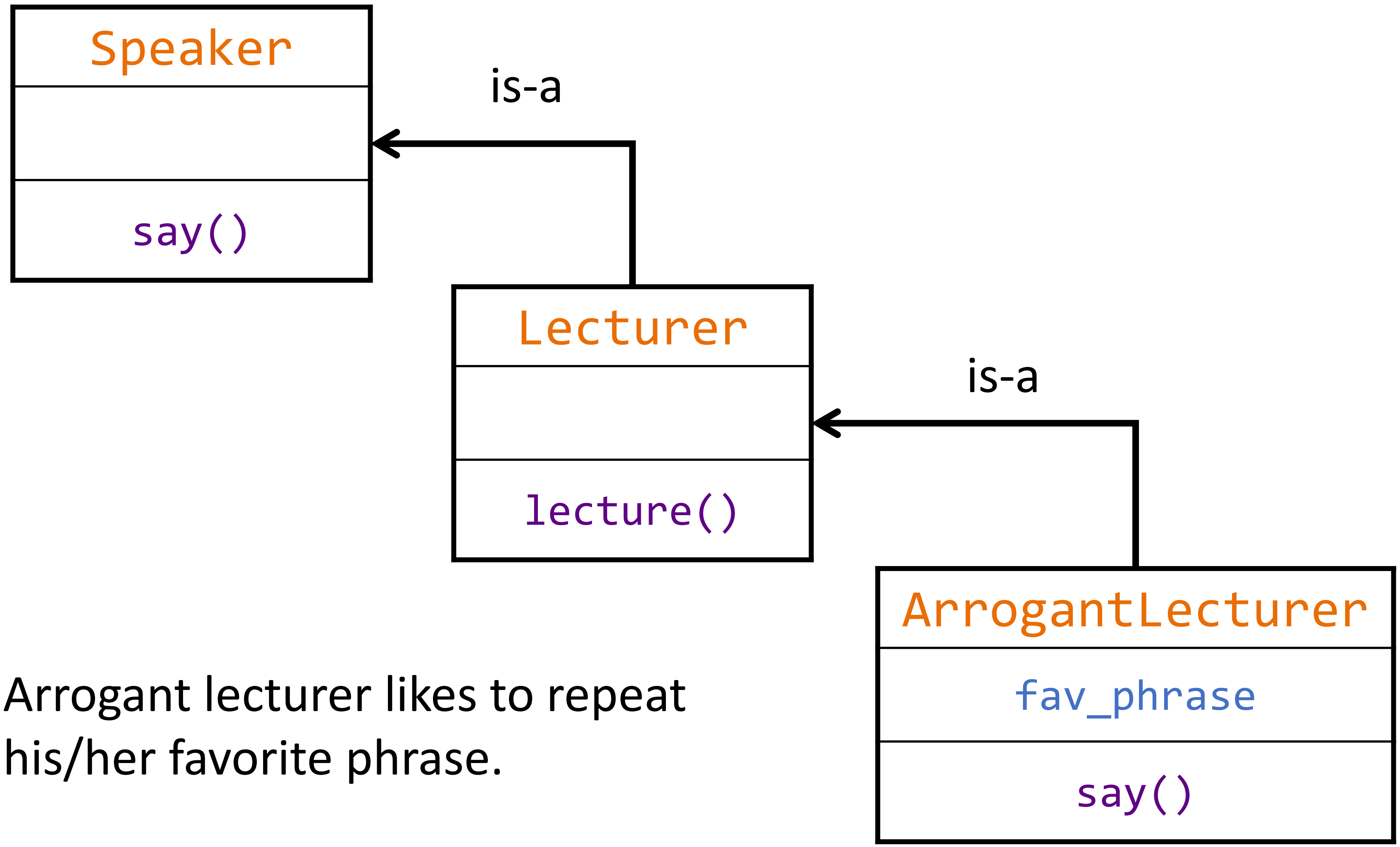
```
class Speaker(object):
    def say(self, stuff):
        print(stuff)

class Lecturer(Speaker):
    def lecture(self, stuff):
        self.say(stuff)
        self.say("You should be taking notes")

way_kay = Lecturer()
way_kay.lecture("Python is easy")
```

Output:

```
Python is easy
You should be taking notes
```

Q: What is the output?

Arrogant Lecturer



Arrogant lecturer likes to repeat his/her favorite phrase.

```python
class Speaker(object):
    def say(self, stuff):
        print(stuff)

class Lecturer(Speaker):
    def lecture(self, stuff):
        self.say(stuff)
        self.say("You should be taking notes")

class ArrogantLecturer(Lecturer):
    def __init__(self, fav_phrase):
        self.fav_phrase = fav_phrase

    def say(self, stuff):
        super().say(stuff + self.fav_phrase)

ah_beng = ArrogantLecturer("... How cool is that?")
ah_beng.say("PE in week 13")
```

Output:

```
PE in week 13... How cool
is that?
```

Q: What is the output?

```python
class Speaker(object):
    def say(self, stuff):
        print(stuff)

class Lecturer(Speaker):
    def lecture(self, stuff):
        self.say(stuff)
        self.say("You should be taking notes")

class ArrogantLecturer(Lecturer):
    def __init__(self, fav_phrase):
        self.fav_phrase = fav_phrase

    def say(self, stuff):
        super().say(stuff + self.fav_phrase)

ah_beng = ArrogantLecturer("... How cool is that?")
ah_beng.lecture("Python is easy")
```

Output:

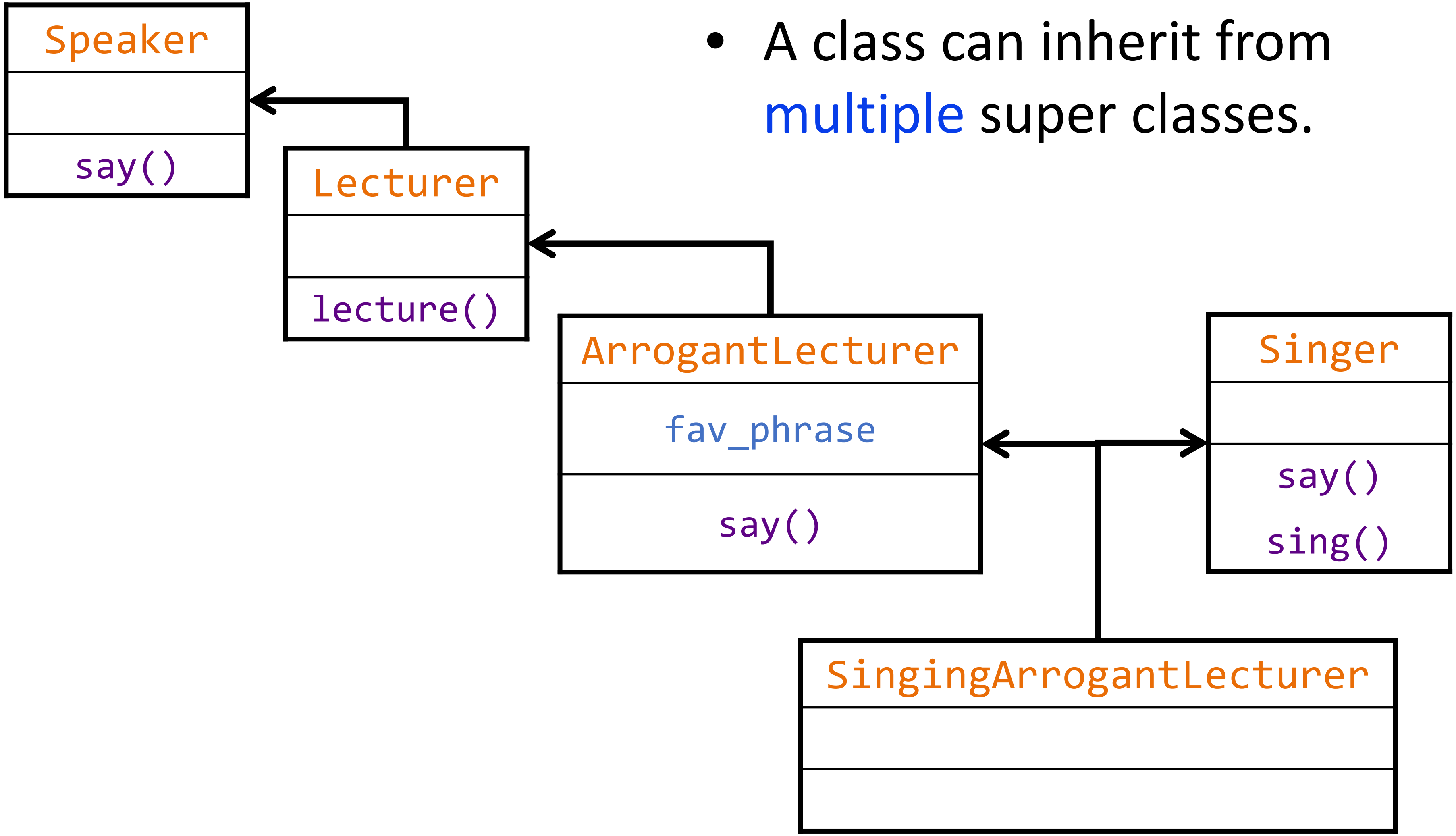
```
Python is easy... How cool
is that?
You should be taking
notes... How cool is that?
```

Q: What is the output?

Polymorphism

- Poly = many; Morphism = form
- Method overriding: The same message can be sent to different types of objects and handled differently based on the type of objects, e.g.
  - `Speaker().say("OK")`
  - `Lecturer().say("OK")`
  - `ArrogantLecturer("cool").say("OK")`
- Method overloading: same method name different number of arguments.

Multiple Inheritance



- A class can inherit from multiple super classes.

```python
class Speaker(object):
    def say(self, stuff):
        print(stuff)

class Lecturer(Speaker):
    def lecture(self, stuff):
        self.say(stuff)
        self.say("You should be taking notes")

class Singer(object):
    def say(self, stuff):
        print("tra-la-la -- " + stuff)
    def sing(self):
        print("tra-la-la")
# to continue next page
```

```
class ArrogantLecturer(Lecturer):
    def __init__(self, fav_phrase):
        self.fav_phrase = fav_phrase
    def say(self, stuff):
        super().say(stuff + self.fav_phrase)

class SingingArrogantLecturer(ArrogantLecturer, Singer):
    def __init__(self, fav_phrase):
        super().__init__(fav_phrase)

ben = SingingArrogantLecturer(" ... How cool is that?")
ben.sing()
ben.say("PE tomorrow")
```

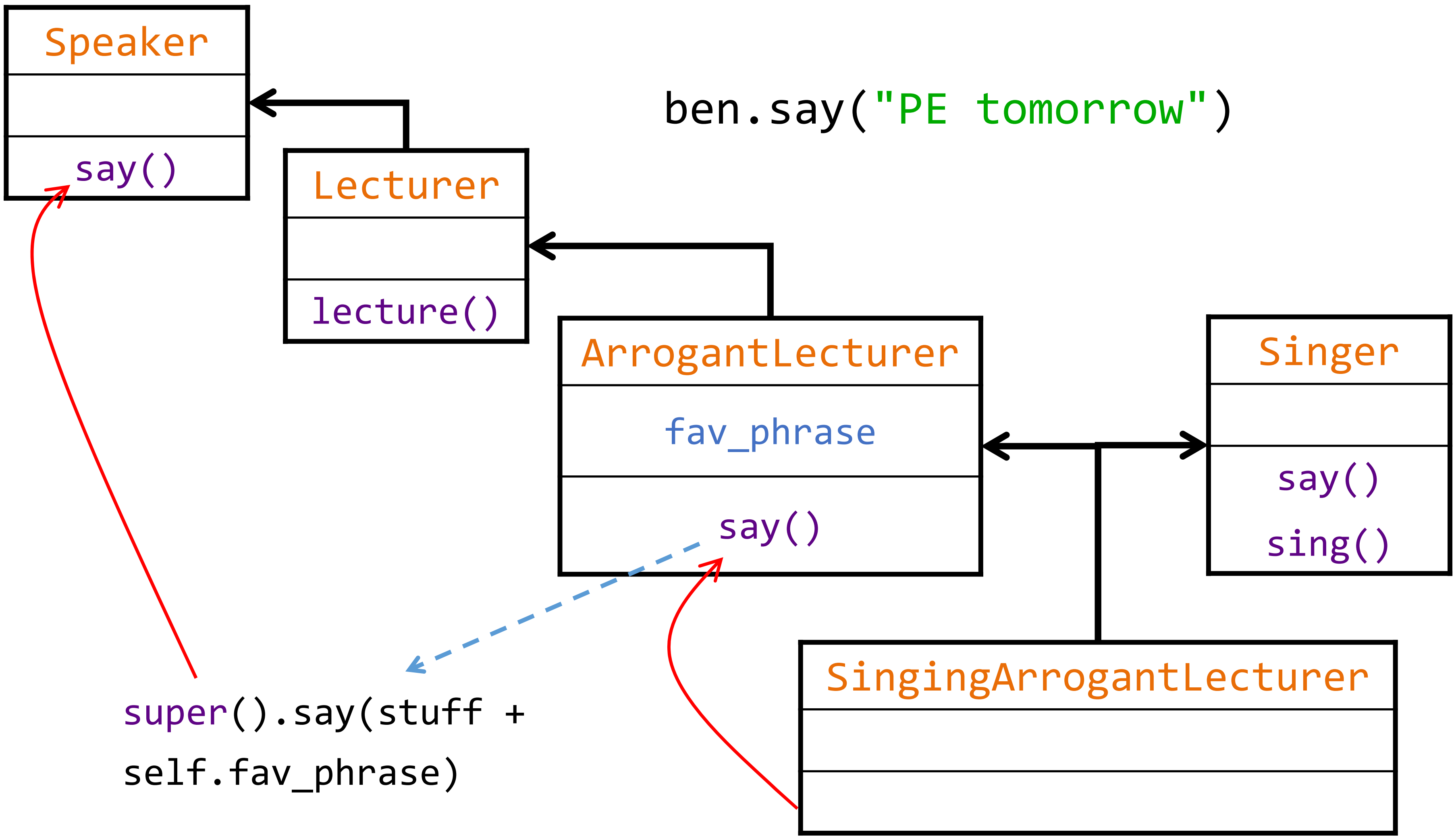
Note the order of super classes

Q: What is the output?

Output:

```
tra-la-la
PE tomorrow ... How cool is that?
```

Multiple Inheritance



Multiple Inheritance

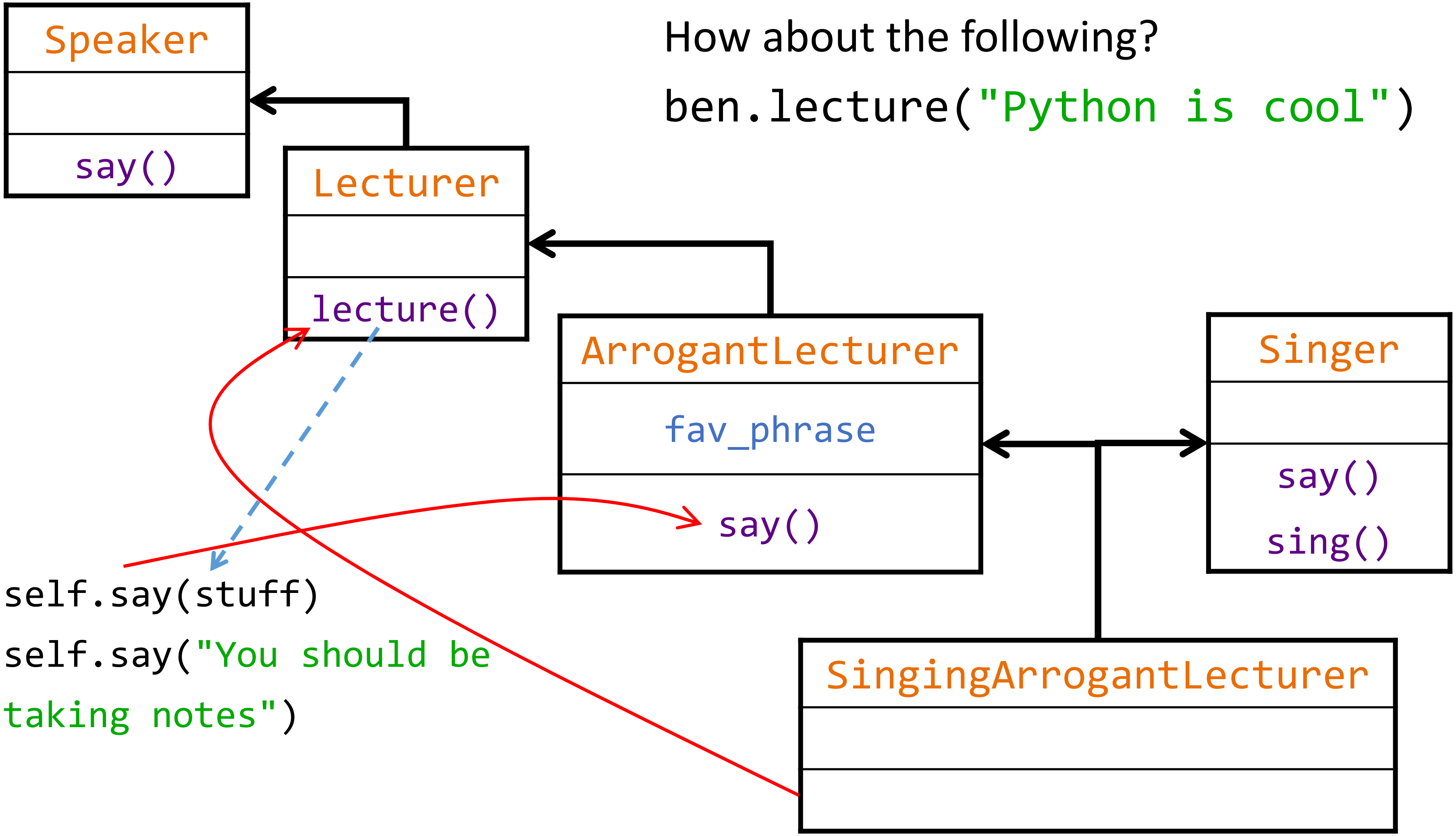
- Complication arises when the same method, e.g. `say()`, is available in two distinct super classes.

- Ben is both an ArrogantLecturer and a Singer, but primarily an ArrogantLecturer.
  - If ArrogantLecturer class defines a `say()`, that method will be used.
  - Otherwise, check Singer class for `say()`.

Multiple Inheritance

How about the following?

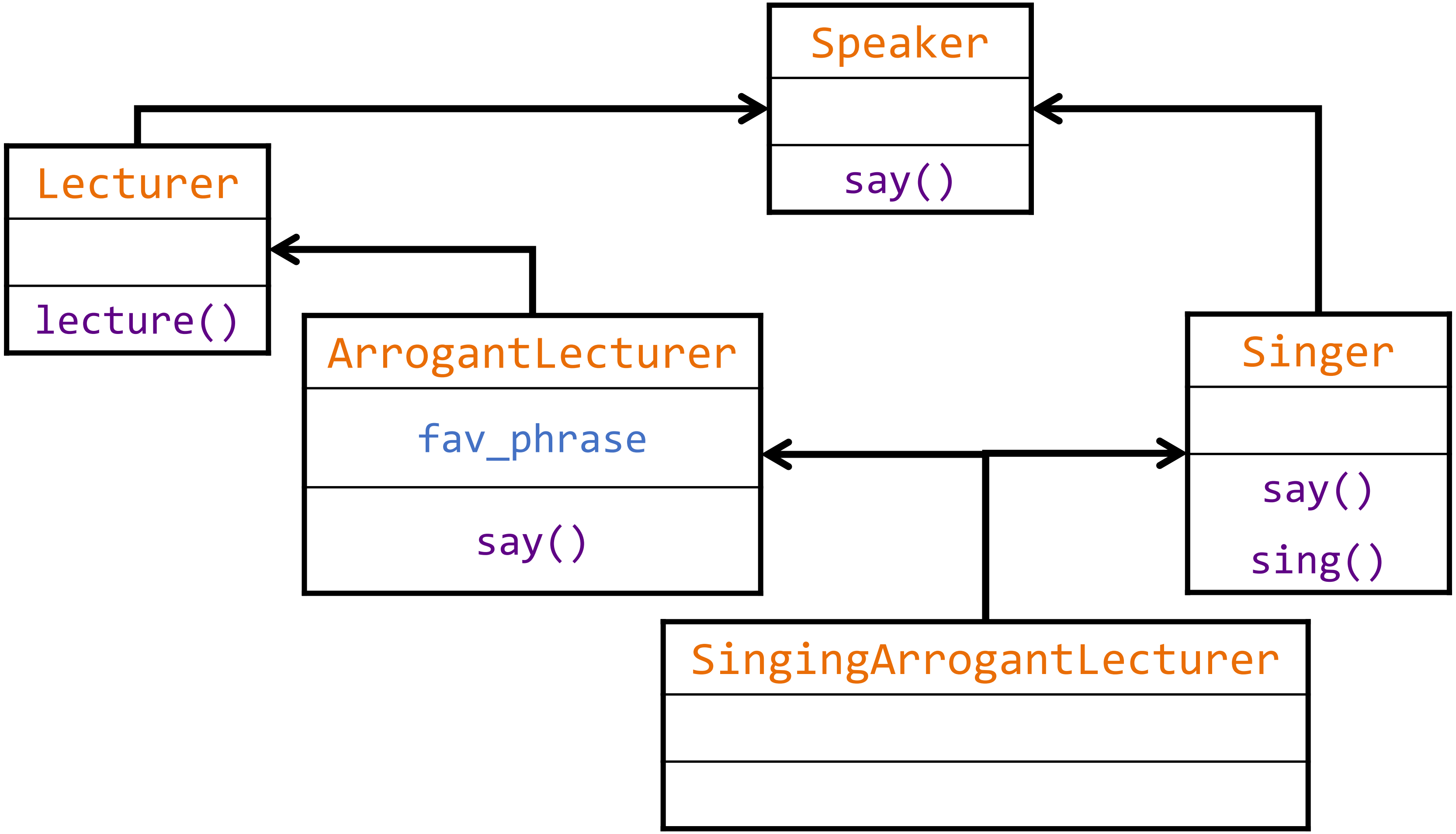
```
ben.lecture("Python is cool")
```




```
self.say(stuff)
self.say("You should be
taking notes")
```

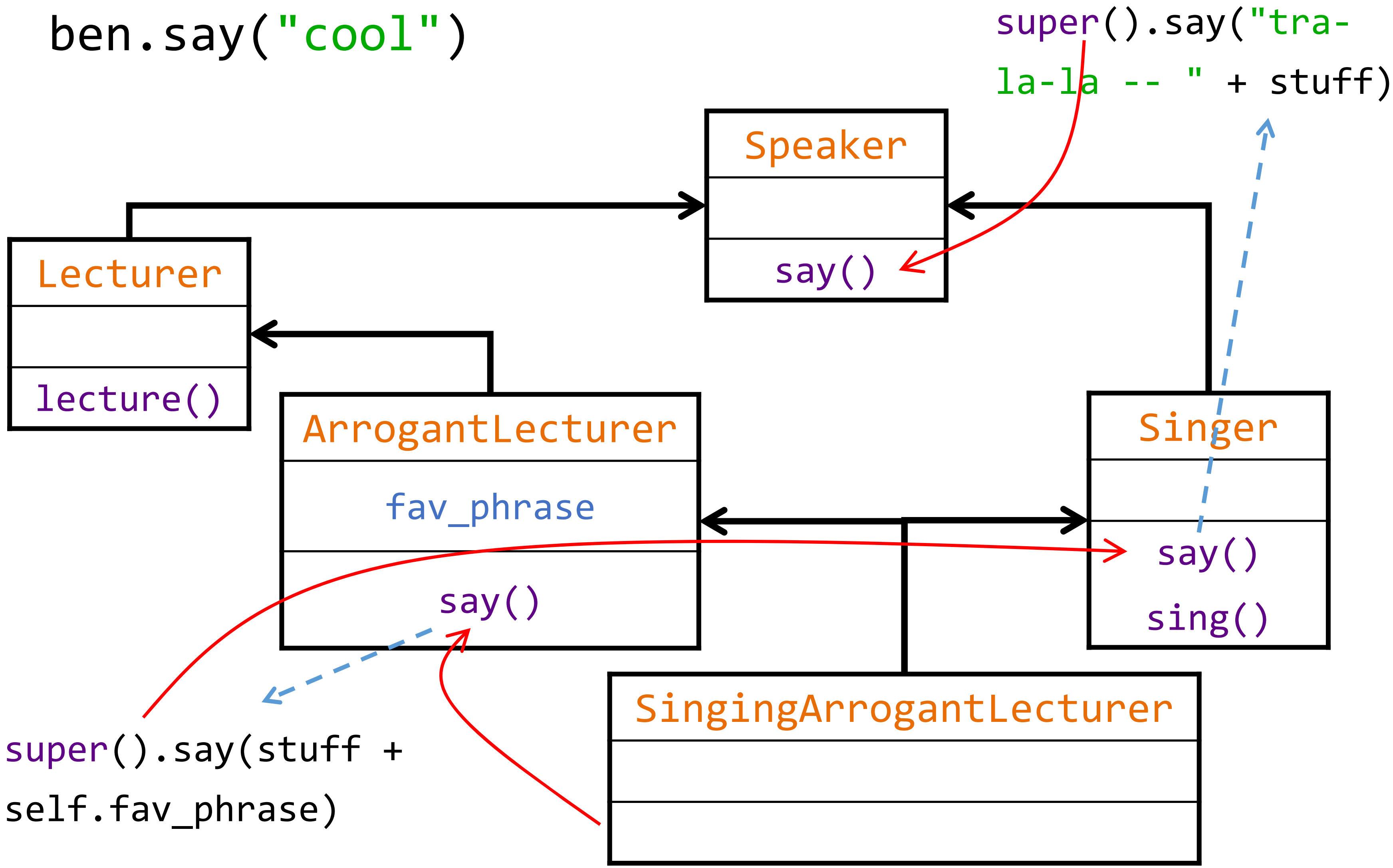
Diamond Inheritance

- Suppose Singer inherits Speaker.



```python
class Speaker(object):
    def say(self, stuff):
        print(stuff)

class Singer(Speaker):
    def say(self, stuff):
        super().say("tra-la-la -- " + stuff)
    def sing(self):
        print("tra-la-la")

# no change to the following three classes
# Lecturer, ArrogantLecturer and SingingArrogantLecturer

ben = SingingArrogantLecturer(" ... How cool is that?")
ben.say("cool")
```

Diamond Problem



OOP : Pros and Cons

- Pros:
  - Simplification of complex, possibly hierarchical structures
  - Easy reuse of code, easy code modifiability
  - Hiding of details through message passing and polymorphism
- Cons:
  - Overhead associated with the creation of classes, methods and instances

Major Programming Paradigms

- Imperative Programming
  - C, Pascal, Algol, Basic, Fortran
- Functional Programming
  - Scheme, ML, Haskell
- Logic Programming
  - Prolog, CLP
- Object-oriented programming
  - Java, C++, C#

Python?

Which Paradigm Is The Best?

- Certain tasks may be easier using a particular style.
- Any style is general enough such that a problem written in one style could be rewritten in another style.
- Choice of paradigm is context dependent and subjective.

Summary

- Classes: template to capture common behavior
- Instances: objects creates from classes; each has own local state (variable)
- Hierarchy of classes
  - Inheritance of state and behavior from superclass
  - Multiple inheritance: rules for finding methods
- Polymorphism: override methods with new functionality