CS1010S Programming Methodology

Lecture 9
Object-Oriented Programming

21 Oct 2020

2^{11} Contest

11 submissions considered

2 won at least once

Today's Agenda

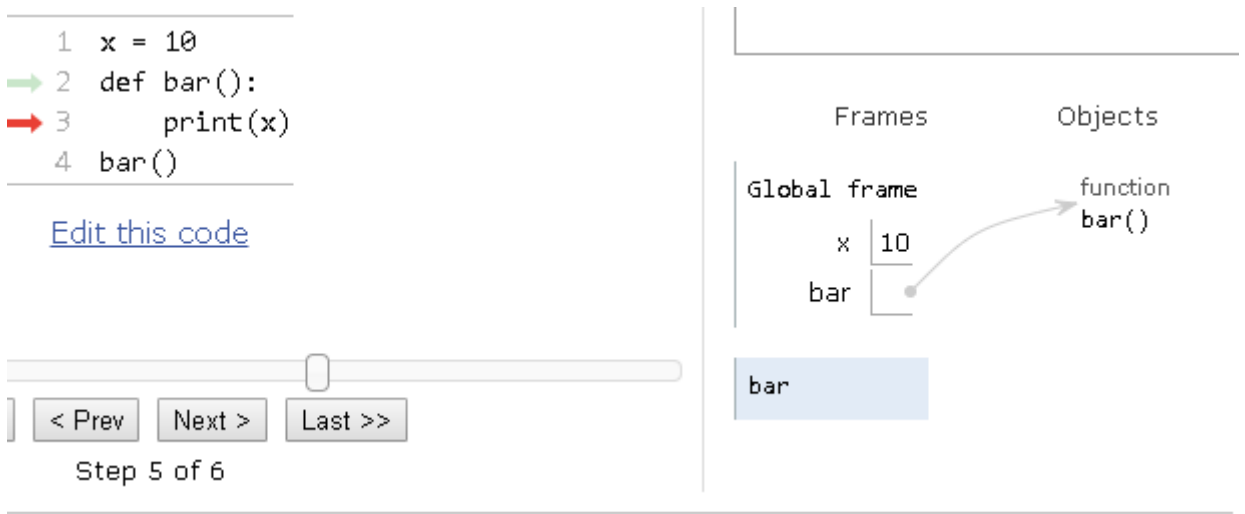
- Object-Oriented Programming (OOP)
- Inheritance
- Polymorphism

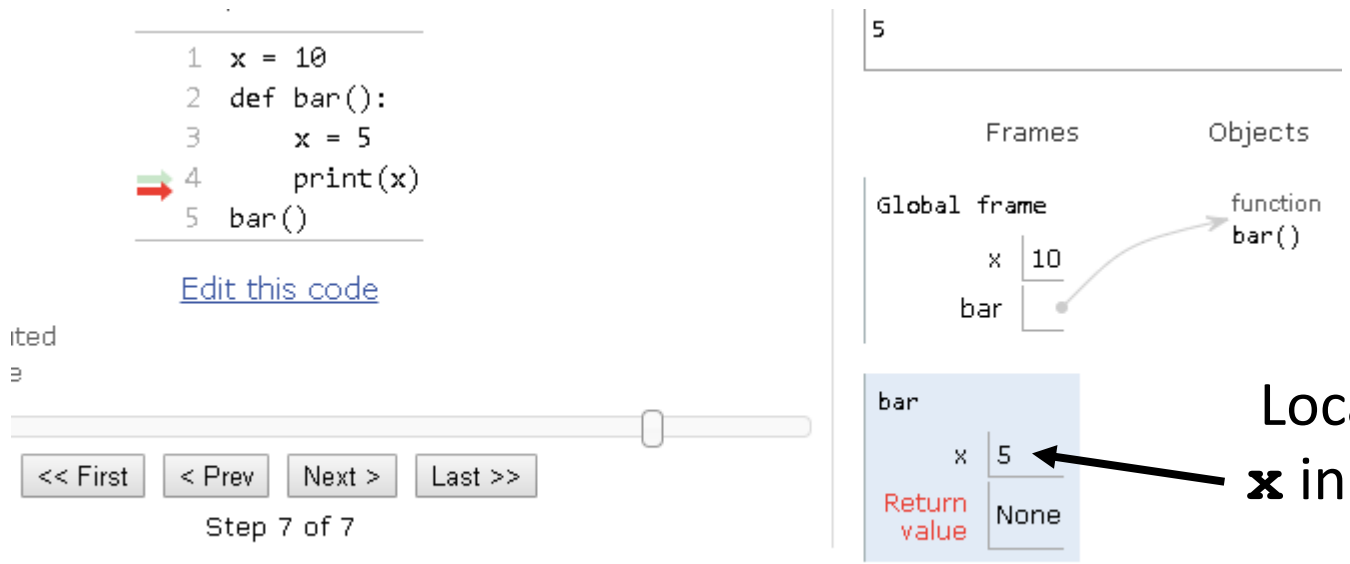
Variable Scoping

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

>>> bar()
10
```

```
x = 10
def bar():
    x = 5
    print(x)

>>> bar()
5
```

Rule of thumb: Search from innermost to outermost scope





Local version of **`x`** in stack frame for **`bar`**

UnboundLocalError?!

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

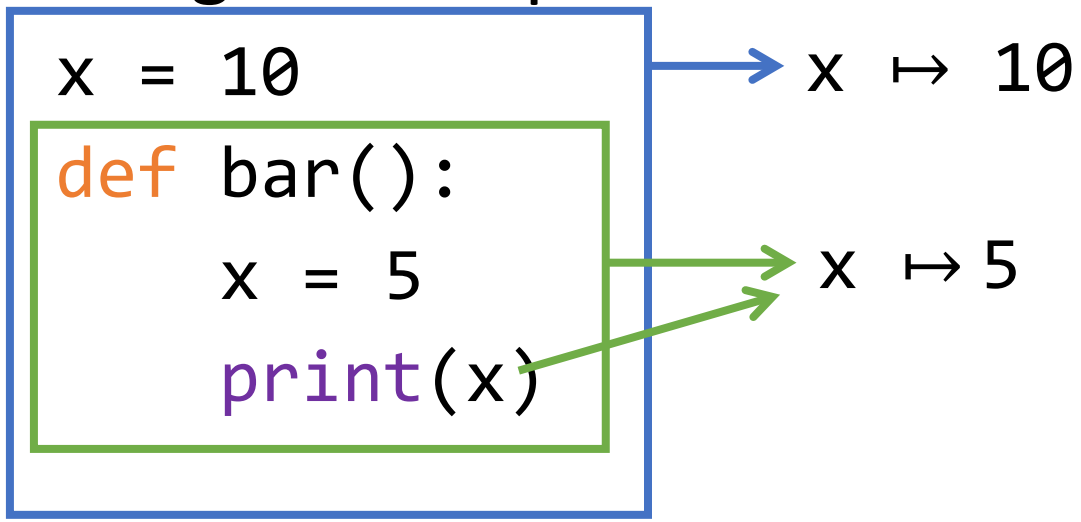
WTF?!

(World Taekwondo Foundation)

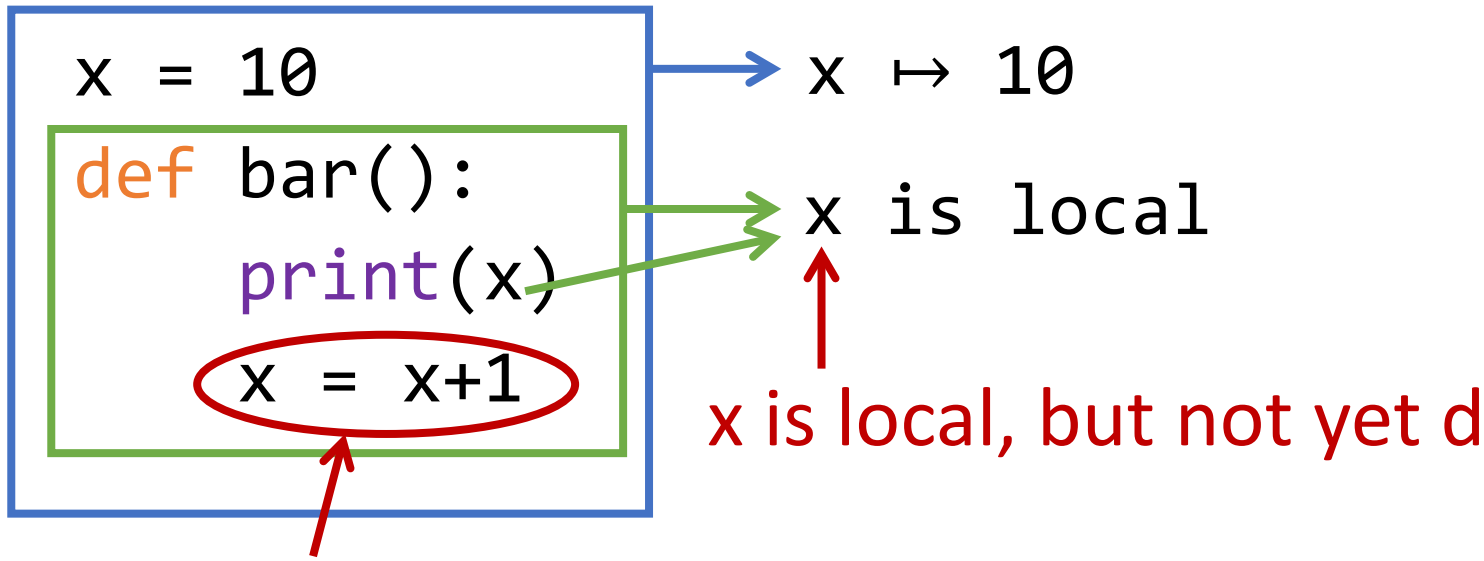
```
>>> bar()
Traceback (most recent call last):
  ...
UnboundLocalError: local variable 'x' referenced
before assignment
```

Let's figure this out

Assignment operation creates new local variable



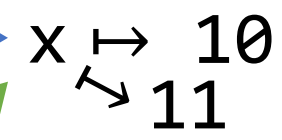
Where's the SNAFU?




```
Traceback (most recent call last):
  ...
UnboundLocalError: local variable
'x' referenced before assignment
```

nonlocal

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



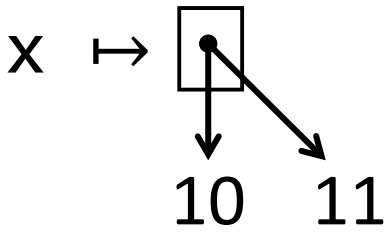

```
>>> a = adder(10)
>>> a()
11
```

Quirk of Python, nothing to do with Programming Methodology

(For knowledge only)

But why does `list` work?

```
x = [10]
def bar():
    print(x)
    x[0] += 1
```

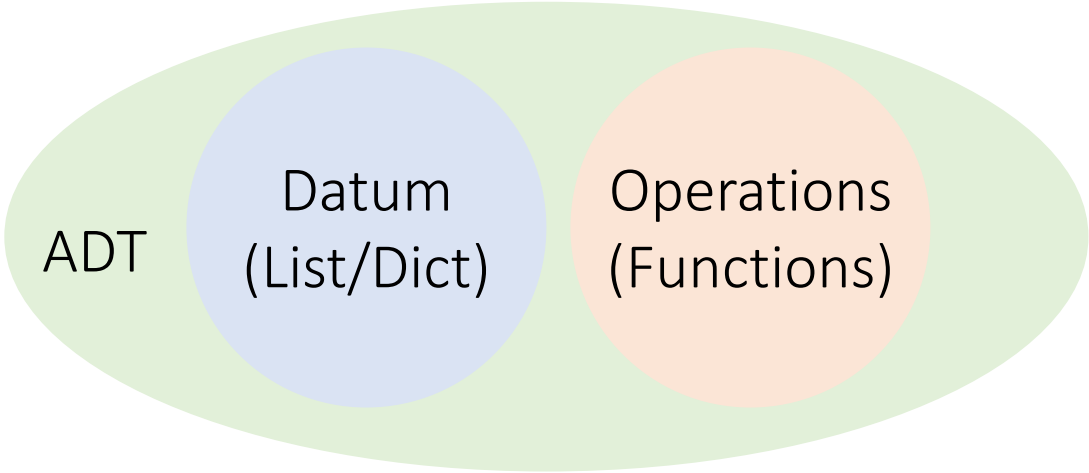



```
>>> bar()
[10]
>>> bar()
[11]
```

This does not rebind (reassign) x. It rebinds the 1st element of x

Recall: Abstract Data Type

- ADT = Datum + Operations



- So far…
  - data is loosely coupled with its operations
  - not a contract, more of a gentlemen agreement

What's the issue?

- Function names must be uniquely named
  - Otherwise will be overwritten
  - Naming convention cannot be enforced
- Functions can be applied to wrong ADT
  - Use special labelling and check

Object-Oriented Programming

- Four basic computational ideas:
  1. Encapsulation
     - Hiding functions
  2. Data Abstraction
     - Express intent over implementation
  3. Inheritance
  4. Polymorphism

Data Abstraction Example

Constructors:

```
def make_acct(name, bal):
    return (name, bal)
```

Accessors/Mutators:

```
def get_name(acct):
    return acct[0]
def get_bal(acct):
    return acct[1]
def set_bal(acct, bal):
    return make_acct(get_name(acct), bal)
```

Data Abstraction Example

Functions:

```
def deposit(acct, val):
    bal = get_bal(acct)
    set_bal(acct, bal + val)

def withdraw(acct, val):
    #Left as an exercise
```

Data Abstraction Example

Usage:

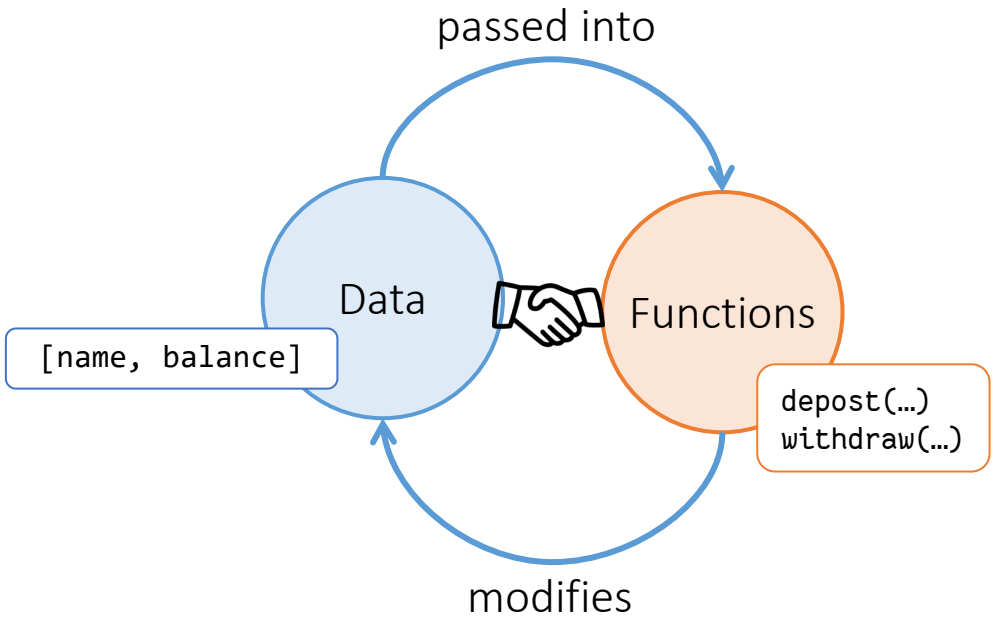
```
ben_acct = make_acct('ben', 1000)
deposit(ben_acct, 40)
print(get_bal(ben_acct))
```

Actually, this code isn't right and doesn't do the right thing. Why and how do we fix it?

Comparing Programming Paradigms

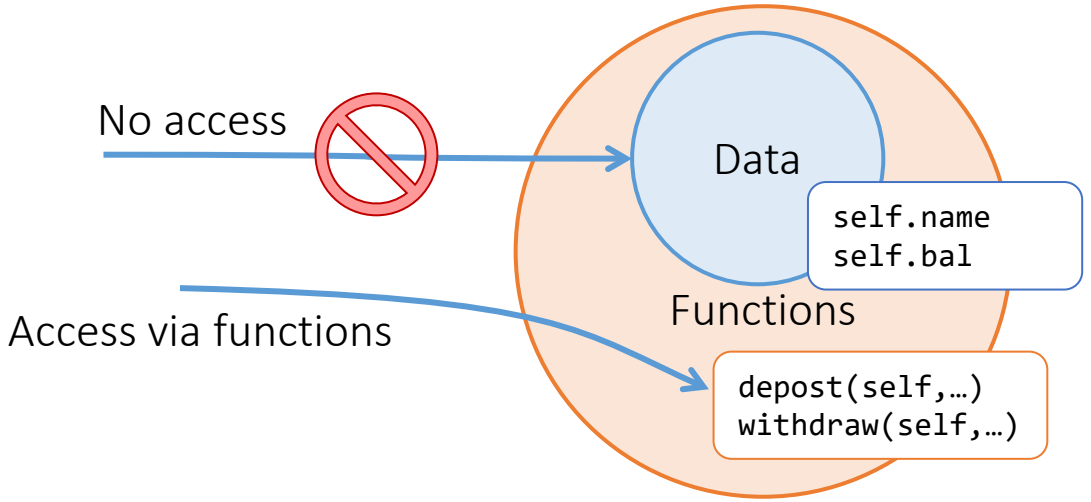
Procedural Model

- Data and operations are separate entities



Object Oriented Model

- Data is encapsulated in functions
- No direct access to data
- Only access using exposed functions



Comparing

```
def make_acct(name, bal):
    return (name, bal)

def deposit(acct, amt):
    acct[1] += amt

def withdraw(acct, amt):
    if acct[1] < amt:
        return "Money not enough"
    else:
       acct[1] -= amt
       return acct[1]
```

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

    def deposit(self, amt):
        self.bal += amt

    def withdraw(self, amt):
        if self.bal < amt:
            return "Money not enough“
        else:
        self.bal -= amt
            return self.bal
```

Terminology

- Class:
  - specifies the common behavior of entities.
  - a blueprint that defines properties and behavior of an object.
- Instance:
  - A particular object or entity of a given class.
  - A concrete, usable object created from the blueprint

What is __init__ ?

- `def __init__(self, balance):`
  - called when the object is first initialized
  - `self` argument is a reference to the object calling the method.
  - It allows the method to reference properties and other methods of the class.
- Are there other special methods?
  - Yes! Special methods have `__` in front and behind the name

Example: Bank Account

```
>>> my_account = BankAccount(100)
>>> my_account.withdraw(40)
60
>>> my_account.withdraw(200)
Money not enough
>>> my_account.deposit(20)
80
```

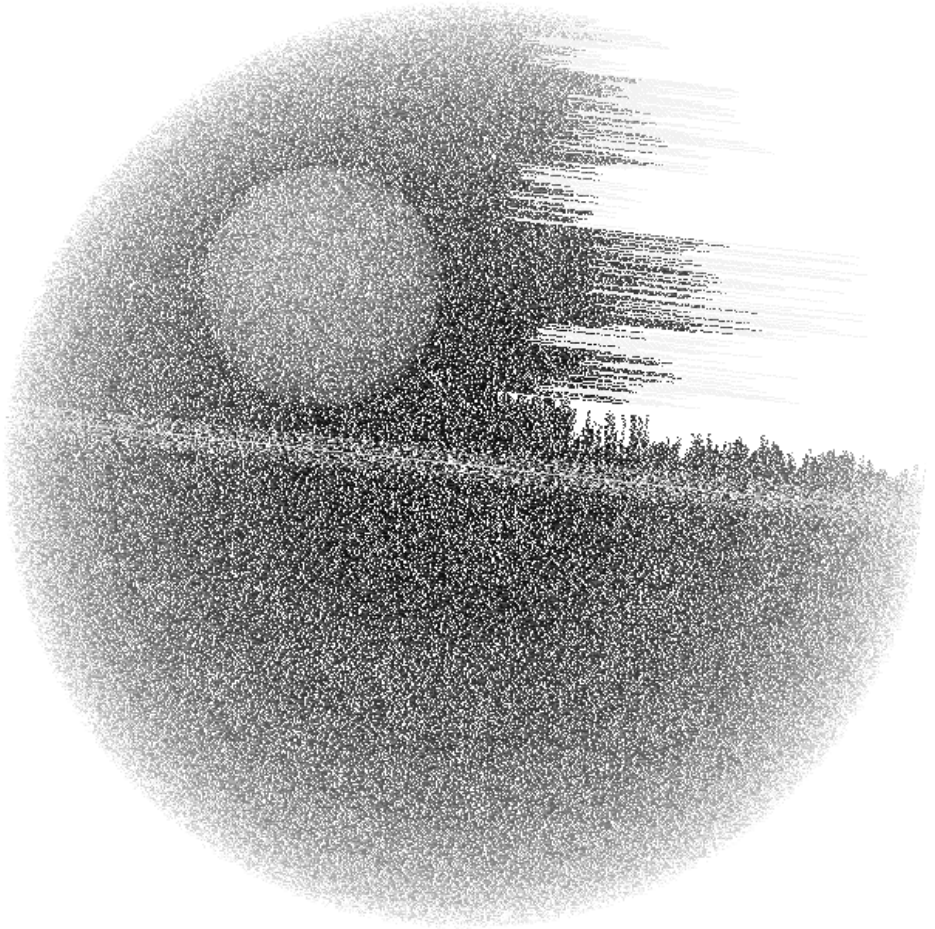
Is it a *really* a new thing?

- Recall your previous lectures…

```
lst = [1, 2, 3]
lst.append(4)
lst  →[1, 2, 3, 4]
```

- Conceptually, append is a method defined in the `List` class.
- Just like withdraw is a method defined in the `BankAccount` class

Suppose we want to build a “space wars” simulator



Using Classes & Instances to Design a System

- Start by thinking about what kinds of objects we want (what classes, their state information, and their interfaces)
  - ships
  - space stations
  - other objects

Using Classes & Instances to Design a System

- We can then extend to thinking about what particular instances of objects are useful
  - Enterprise
  - Millenium Falcon
  - Death Star

Defining the Ship Class

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

    def move(self):
        self.position = …

    def fire_torps(self):
        if num_torps > 0:
            …
```

How to implement?

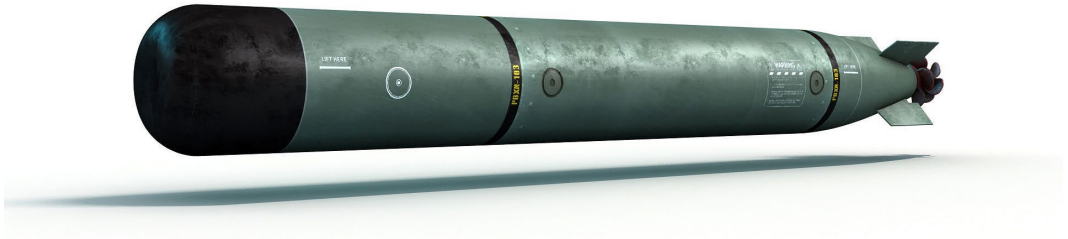
- Objects have:
  - State
  - Methods
- Starship example:
  - State: position, velocity, num_torps
  - Methods: move, attack

Instances of Objects

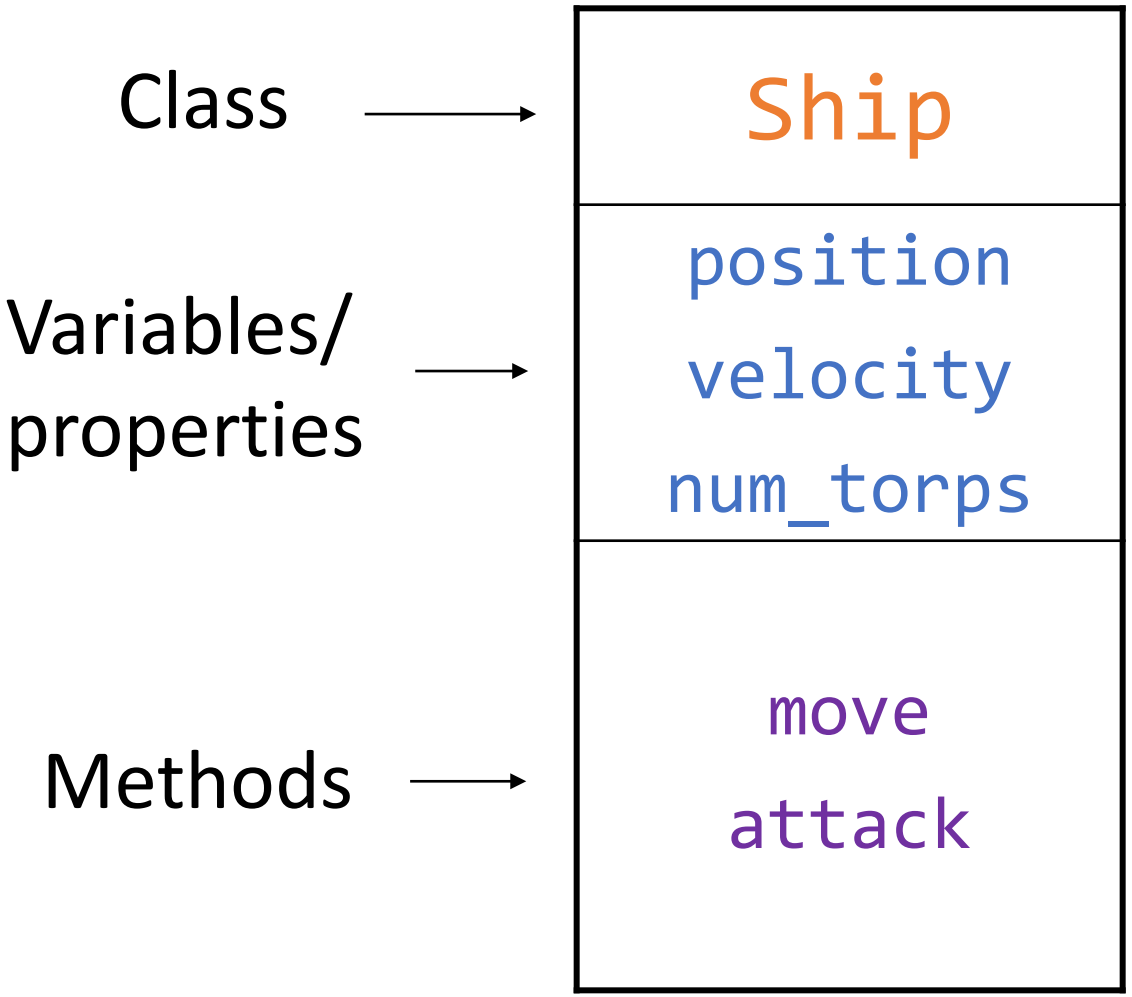
```
>>> enterprise = Ship((10,10), (5,0), 3)
>>> falcon = Ship((-10,10), (10,0), 8)
>>> print(enterprise)
<__main__.Ship object at 0x109b2fd90>

>>> print(falcon)
<__main__.Ship object at 0x109b2ff10>
```

Torpedo

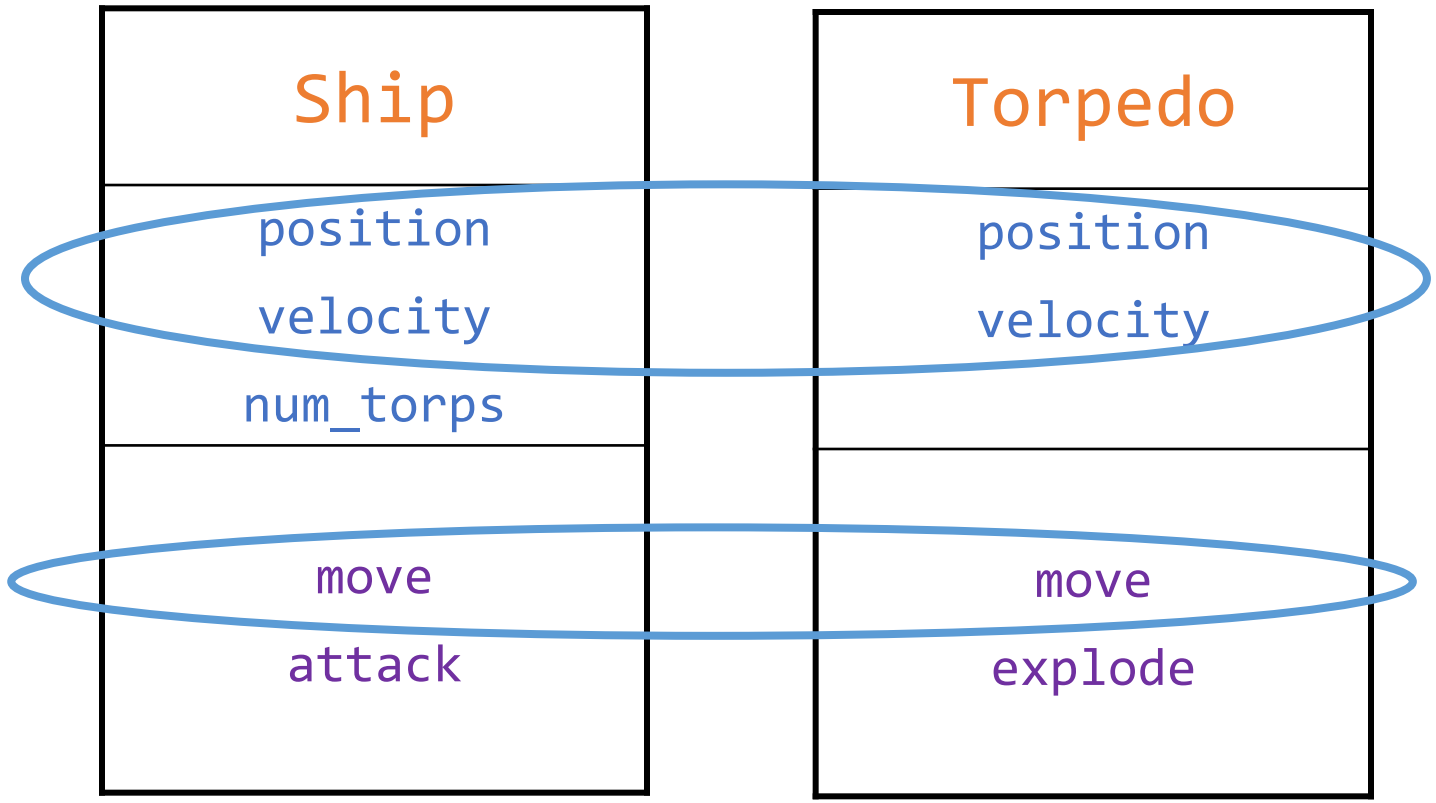
```
class Torpedo(object):
    def __init__(self, p, v):
        self.position = p
        self.velocity = v

    def move(self):
        self.position = ...

    def explode(self):
        print("torpedo goes off!")
        # remove torpedo from the world
```



A Tale of Two Objects

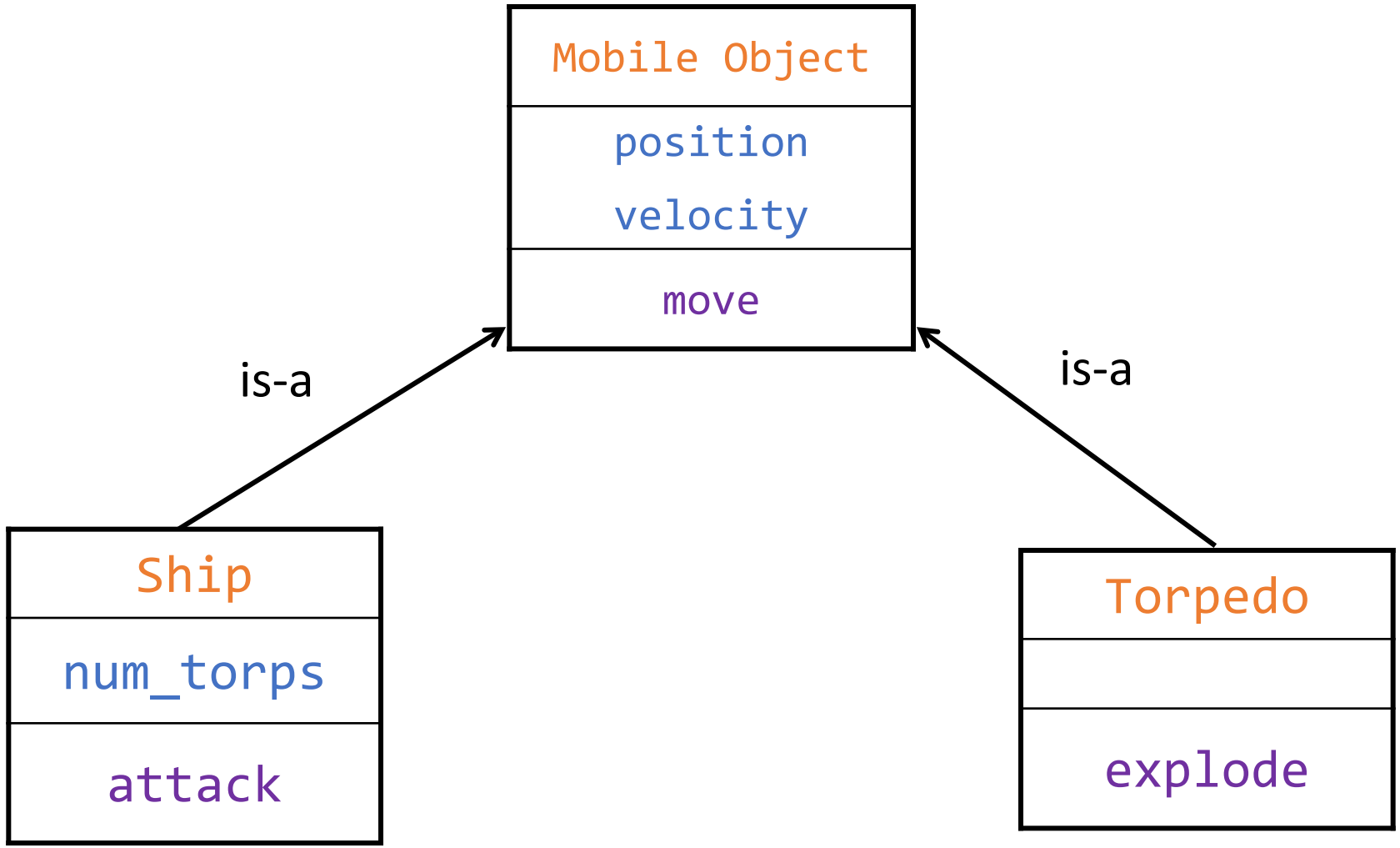


A Tale of Two Objects



What do you notice about the two objects?

Inheritance



Inheritance

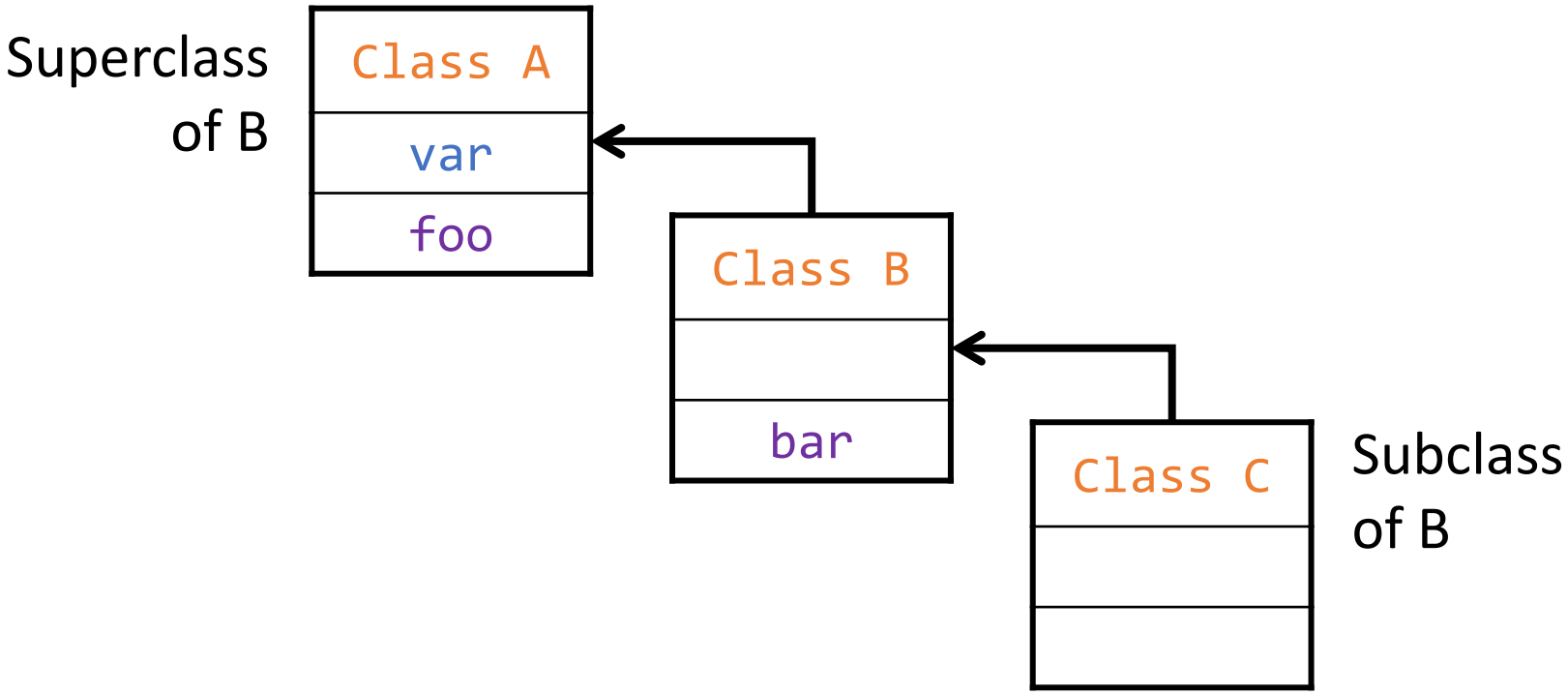
Exploit commonality to share structure and behaviour

Inheritance

- Objects that exhibit similar functionality should “inherit” from the same base object, called the superclass.
- An object that inherits from another is called the subclass.

Inheritance

- Superclass vs Subclass
  - Subclass specializes the superclass by extending state/behavior



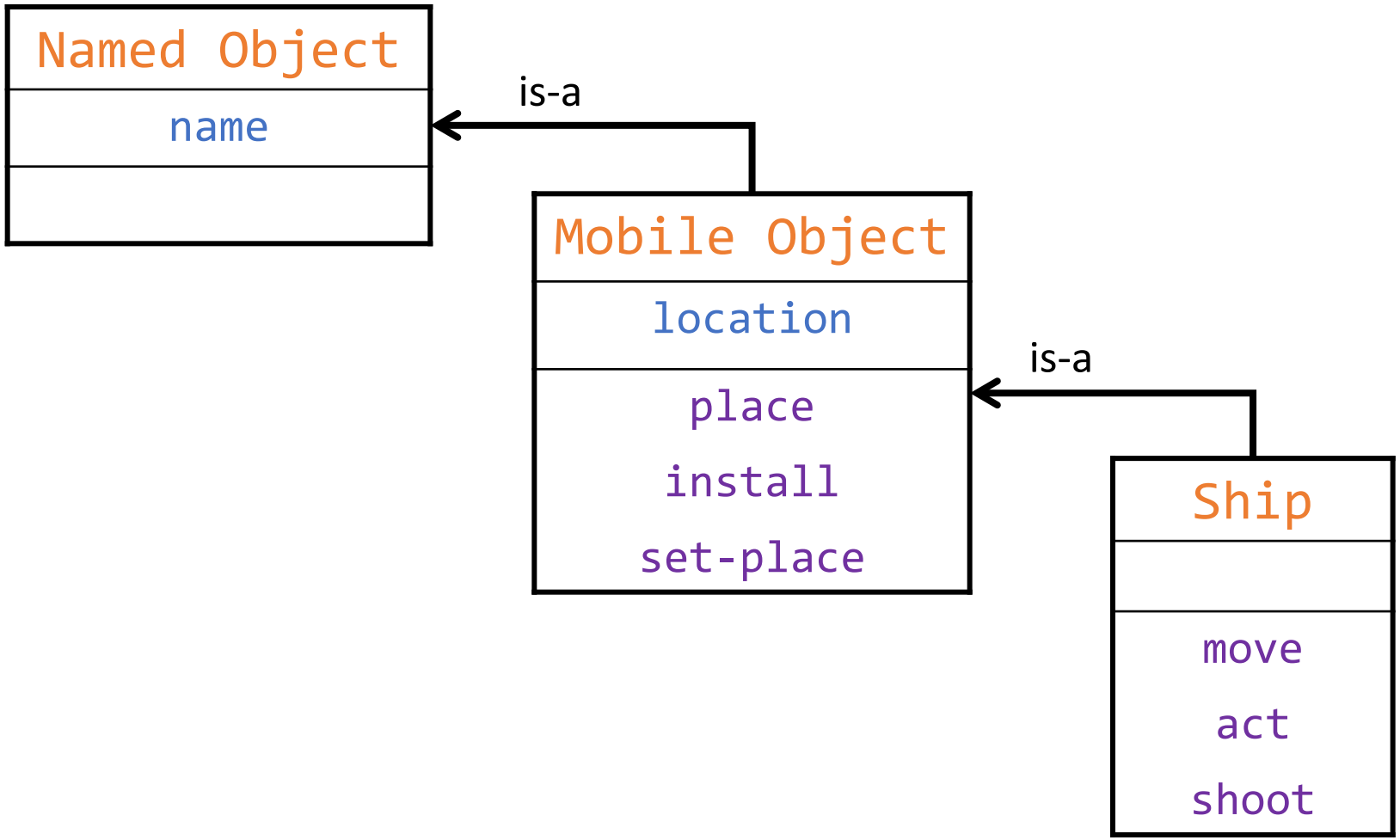
Inheritance

- Classes have an "is-a" relationship with their superclasses
  - Establishes a natural type hierarchy
  - When did we last see this??

Overview

- Class
  - Defines what is common to all instances of that class
    - Provides local state variables
    - Provides a message handler to implement methods
    - Specifies what superclasses and methods are inherited
  - Root class: All user defined classes should inherit from either root-object class or from some other superclass

Example: Star Trek Simulation



Example: Star Trek Simulation

The basic (root) object

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

A “self” variable?

- Every class definition has access to a `self` variable
- `self` is a reference to the entire instance

User View: Why a “self” variable?

- Why need this? How or when use self ?
  - When implementing a method, sometimes you “ask” a part of yourself to do something
  - However, sometimes we want to ask the whole instance to do something

- This mostly matters when we have subclass methods that shadow superclass methods, and we want to invoke one of those shadowing methods from inside the superclass

Example: Star Trek Simulation

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

    def install(self):
        self.location.add_thing(self)
```

Did you notice repeated code?

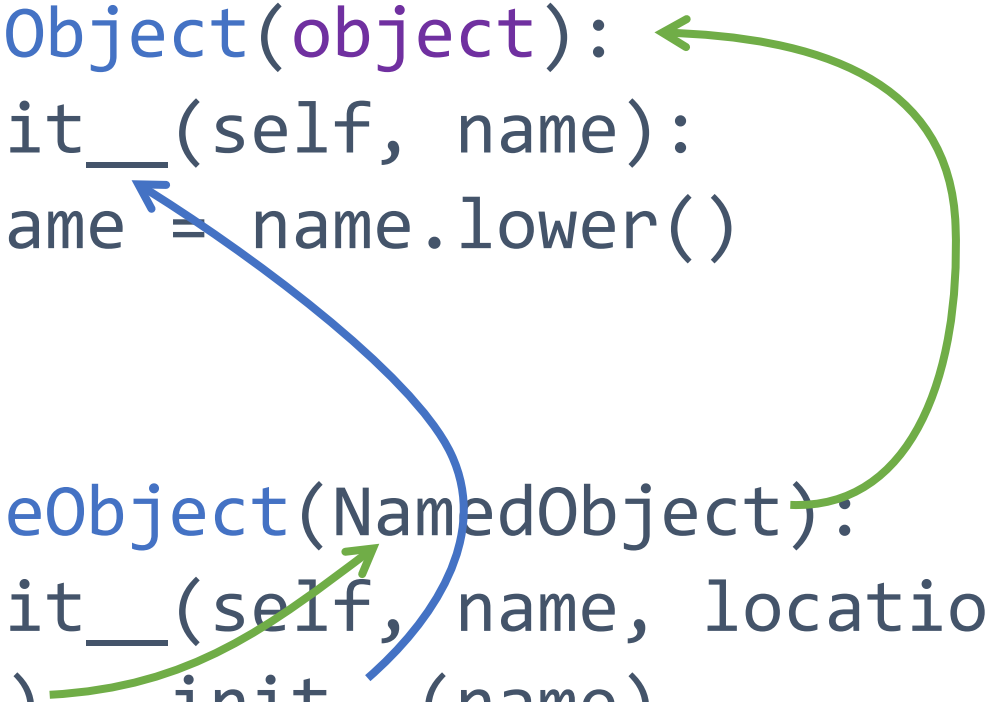
```
class NamedObject(object):
    def __init__(self, name):
        self.name = name


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

The ‘super()’ method

- What happens if a new directive states that all names must be in lowercase?
- Do we have to manually change all the declarations in all the methods in the class hierarchy?
  - Doesn’t sound very reusable right?
- We need a way to access the next higher class in the class hierarchy – the `super()` method

The ‘super()’ method

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


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



Example: Star Trek Simulation

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

    def move(self):
        if self.threshold < 0:
            pass
        elif random.randint(0, self.threshold) == 0:
            self.act()
```

Example: Star Trek Simulation

Artificial Intelligence....

```
def act(self):
    new_location =
        self.location.random_neighbor()
    if new_location:
        self.move_to(new_location)
```

Example: Star Trek Simulation

```
def move_to(self, new_location):
  if self.location == new_location:
      print(self.name, "is already at", new_location.name)
  elif new_location.accept_ship():
      print(self.name, "moves from", self.location.name,
            "to", new_location.name)
      self.location.remove_thing(self)
      new_location.add_thing(self)
      self.location = new_location
  else:
      print(self.name, "can't move to", new_location.name))
```

Example: Star Trek Simulation

```
def go(self, direction):
  new_place = self.location.neighbors_towards(direction)
  if new_place:
    self.move_to(new_place)
  else:
    print("You cannot go", direction,
          "from", self.location.name)
```

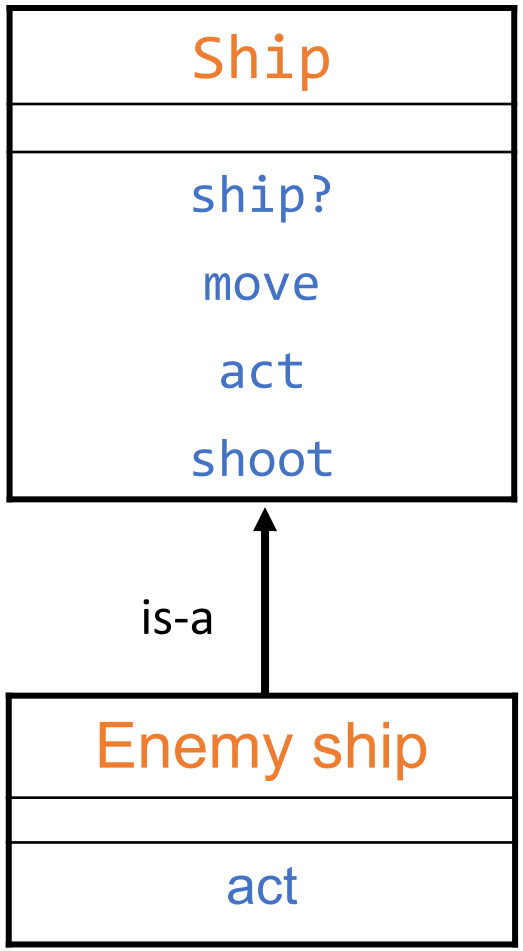
Example: Star Trek Simulation

```
def shoot(self, ship_name):
  target_ships = list(filter(
                 lambda ship: ship.name == ship_name,
                 self.location.things))
  if len(target_ships) > 0:
      target_ship = target_ships[0]
      print(self.name, "opens fire at", target_ship.name)
      target_ship.destroy()
```

Example: Star Trek Simulation

```
    def destroy(self):
        print(self.name, "destroyed!")
        self.is_alive = False
        self.move_to(HEAVEN)

ENTERPRISE = Ship("enterprise", EARTH, -1)
```

Now for the bad guys .....



- Let's create a new class, enemy ship, that will fire at us.
- Enemy ship is a subclass of ship since it behaves like a ship
- Define a new version of the method act in the subclass
- When act is called, the version in the subclass will be used instead.
- This is known as overriding.

  (polymorphism)

Now for the bad guys .....

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

To venture where no man has gone before

.....

Captain's Log: Stardate 10677.5

The Enterprise was back on Earth for routine maintenance when suddenly we received a distress signal from deep space in the vicinity of a black hole. We assembled the crew to investigate ....

isinstance vs type

```
class Vehicle:
    ...

class Truck(Vehicle):
    ...

isinstance(Vehicle(), Vehicle)  # returns True
type(Vehicle()) == Vehicle      # returns True
isinstance(Truck(), Vehicle)    # returns True
type(Truck()) == Vehicle        # returns False
type(Truck()) == Truck          # returns True
```

Another Example: A Speaker

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

What does the speaker do?

Example: A Speaker in action

```
>>> ah_beng = Speaker()
>>> ah_beng.say("Hello World")
Hello World


>>> ah_beng.dance()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Speaker' object has no attribute
'dance'
```

More about Inheritance

- We can define an object type to be a more “specialized” kind of some other object type
- Example:
  - A lecturer is a kind of speaker
  - The lecturer also has a method called lecture
  - To lecture something, the lecturer says it and then says: “You should be taking notes”

More about Inheritance

- Observations:
  - A lecturer can do anything a speaker can (i.e. say things), and also lecture
  - Lecturer inherits the “say” method from speaker
  - Lecturer is a subclass of speaker
  - Speaker is a superclass of lecturer

Making a Lecturer

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

Python would go through up in the class hierarchy if a method definition is not found in the class

Example: A Lecturer in action

```
>>> seth = Lecturer()

>>> seth.lecture("Java is easy")
Java is easy
You should be taking notes

>>> seth.say("You have a quiz today")
You have a quiz today
```

Making an Arrogant Lecturer

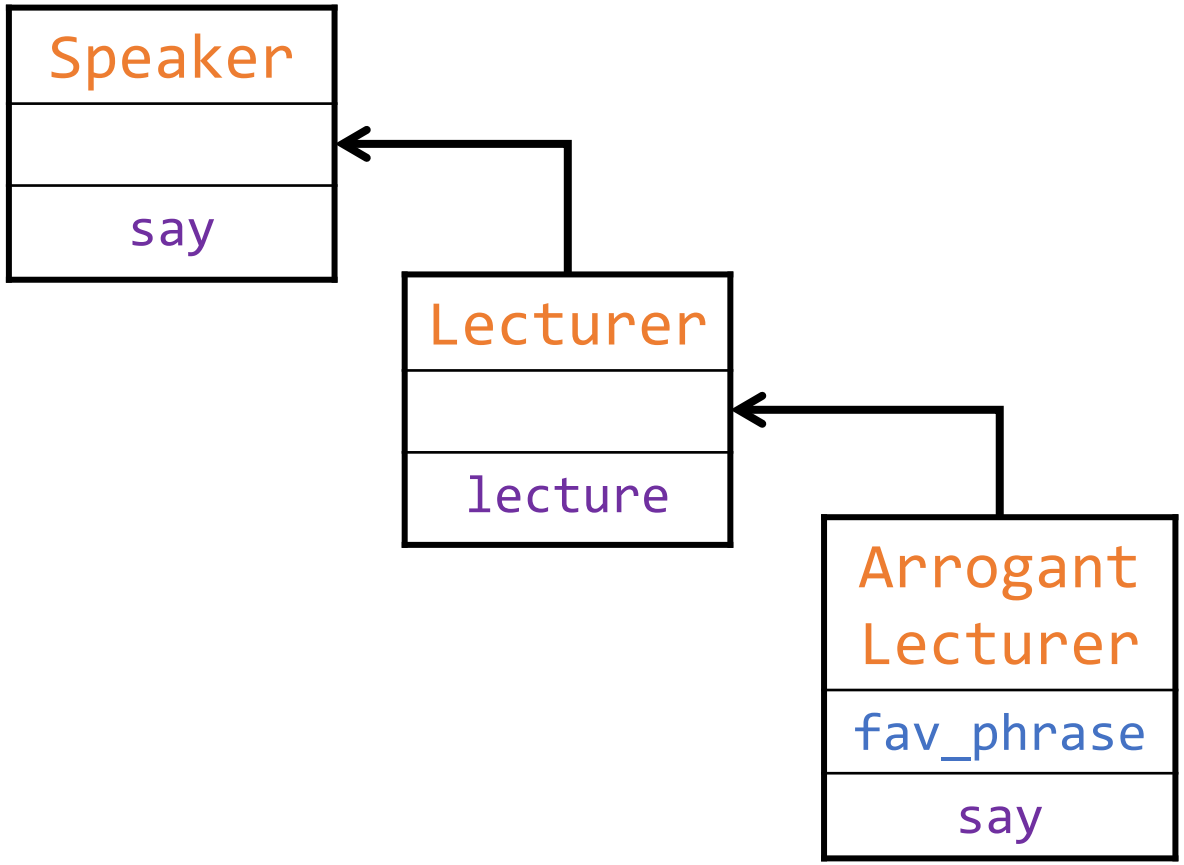
- Define an arrogant lecturer to be a kind of lecturer
- Whenever an arrogant lecturer says anything, she or he will say it as an ordinary lecturer would, but he will also add some favourite phrase of his/hers at the end.

Making an Arrogant Lecturer

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

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

`super()` allows us to access methods in the superclass.

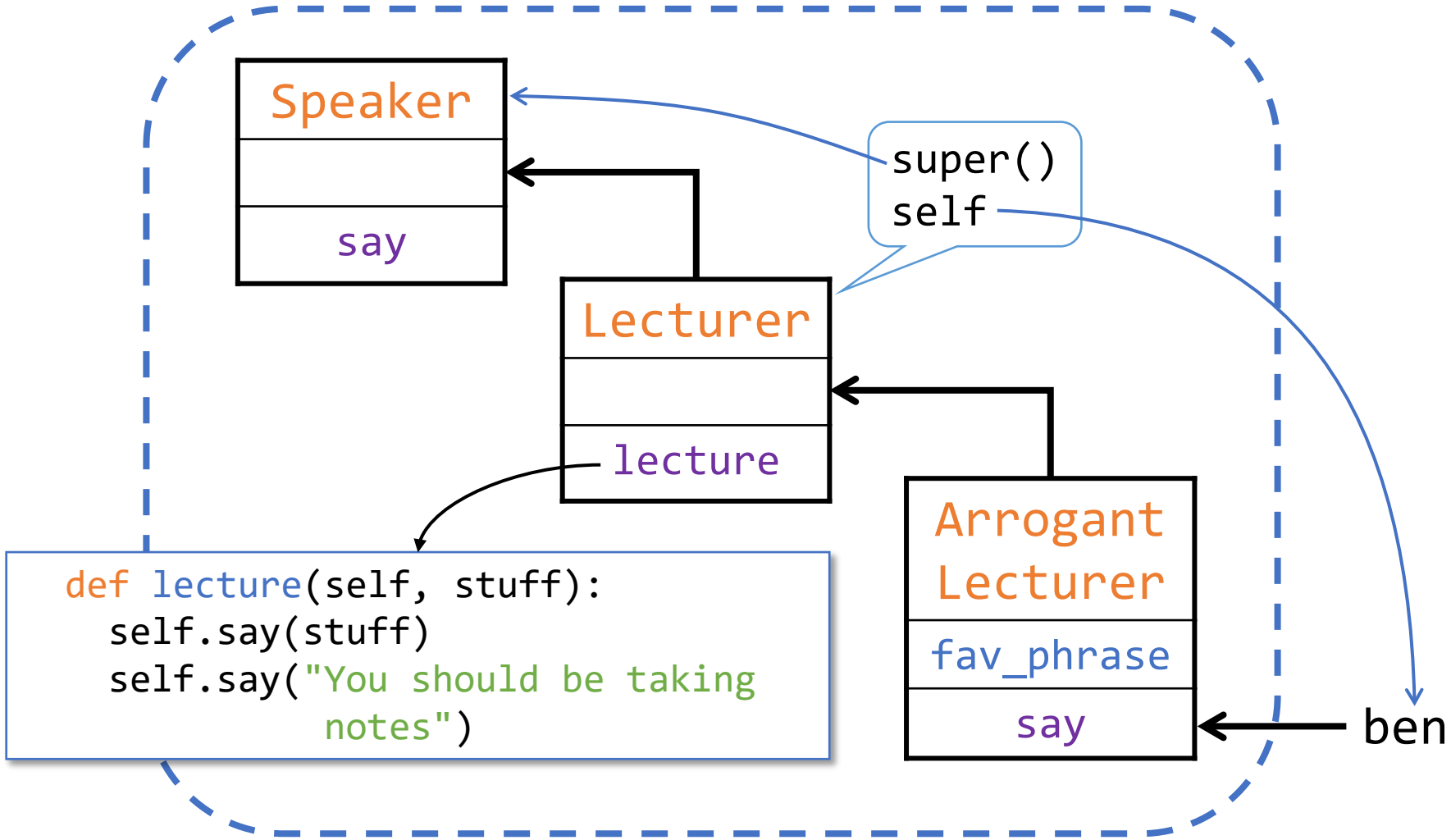
Object Hierarchy



Example: An Arrogant Lecturer in action

```
>>> ben = ArrogantLecturer(" ... How cool is that?")

>>> ben.say("We'll have a PE tomorrow")
We'll have a PE tomorrow ... How cool is that?

>>> ben.lecture("Python is cool")
Python is cool ... How cool is that?
You should be taking notes ... How cool is that?
```

Class Hierarchy



Polymorphism

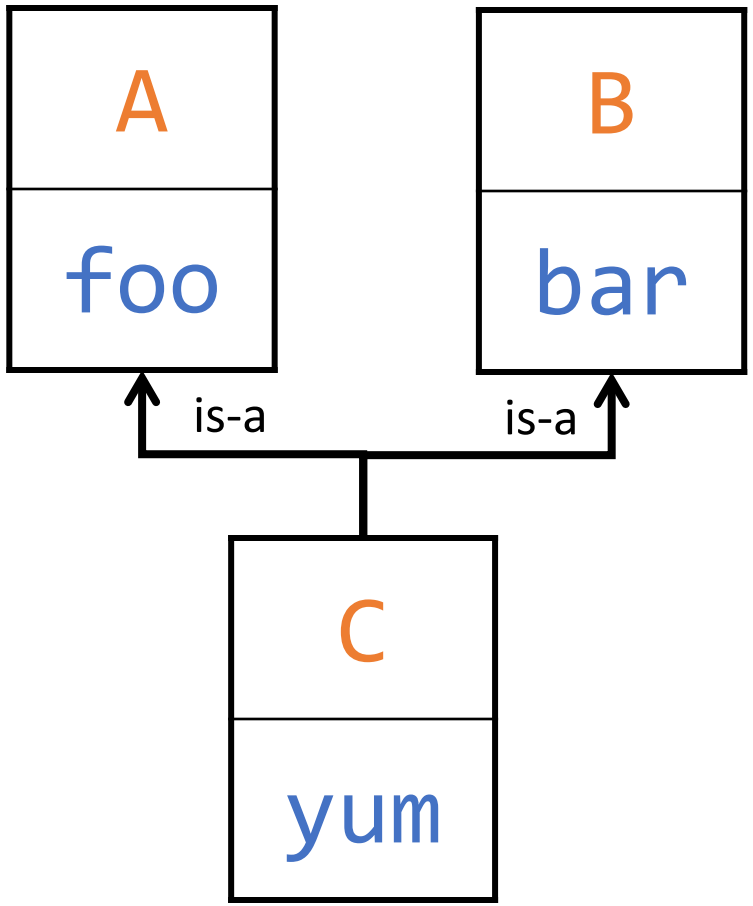
- Poly = many; Morphism = form
- Object-oriented programming provides a convenient means for handling polymorphic functions (overloading)
  - Functions that take different types of arguments
- The same message can be sent to different types of objects and handled by different methods that perform the proper actions based on the object class (overriding)
  - e.g. ask a speaker, lecturer, or arrogant-lecturer to “say” something

How would you implement overloading?

Polymorphism

- Benefits for programmer:
  - does not need to worry about the type of the object
  - can focus on the message

Multiple Inheritance



- A class can inherit from multiple classes
- C is subclass of both A and B
- A class inherits both its state and methods from superclasses
  - C has methods: `foo, bar, yum`
- Multiple inheritance has issues:
  - Not all languages support this
  - Resolution order issues

Multiple Inheritance

```
class Singer(object):
    def say(self, stuff):
        print("tra-la-la -- " + stuff)

    def sing(self):
        print("tra-la-la")
```

What does the singer do?

What is the singer a subclass of?

Singer Sings

```
>>> taylor_swift = Singer()
>>> taylor.say("I like the way you sound in the morning")
tra-la-la -- I like the way you sound in the morning

>>> taylor_swift.sing()
tra-la-la
```

Moonlighting.... shhhhh

Suppose Ben decides to moonlight as a singer....

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

Note the order of the super class!

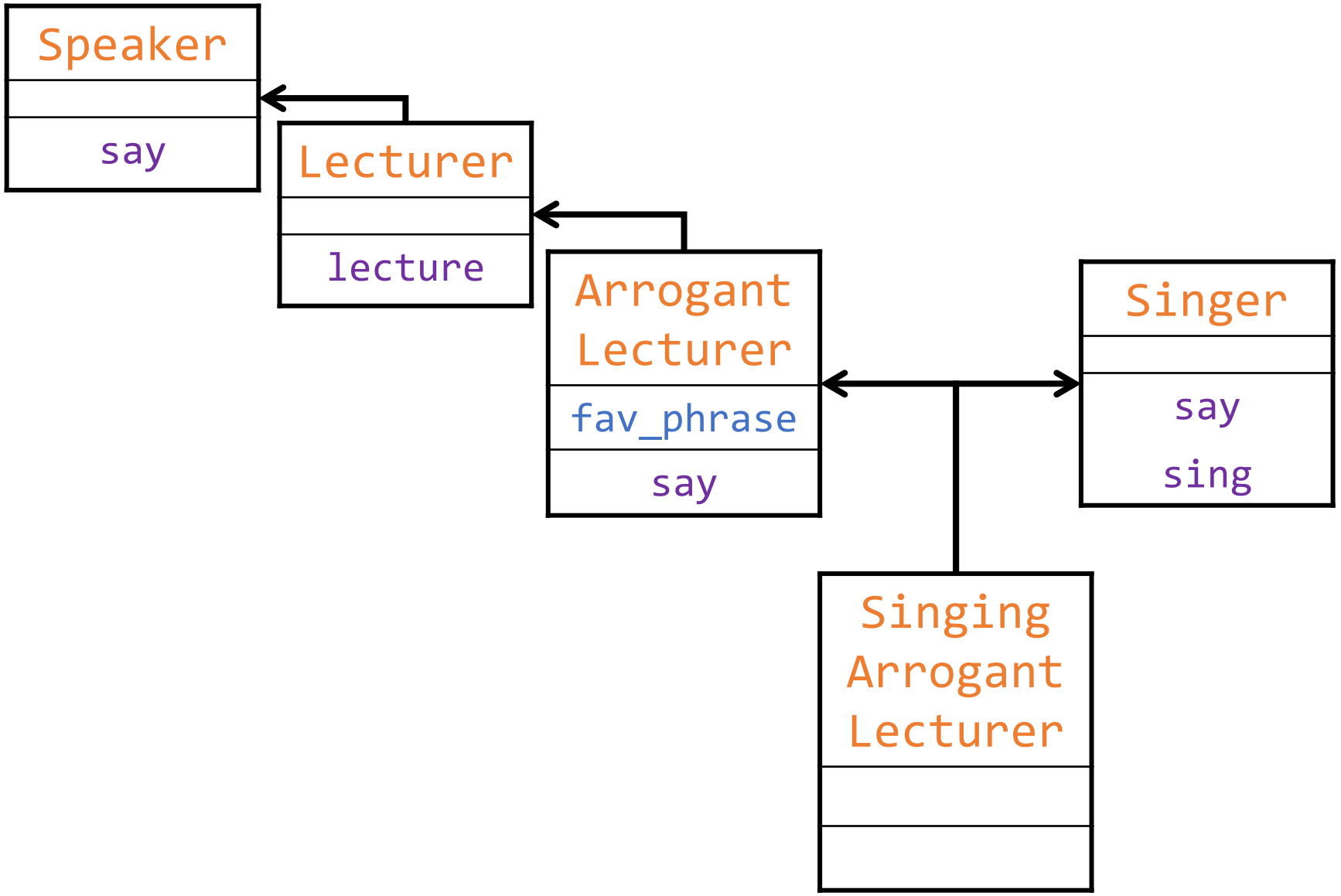
Ben showing off his hidden talents

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

>>> ben.say("We'll have a PE tomorrow")
We'll have a PE tomorrow ... How cool is that?
```

Ben showing off his hidden talents

```
>>> ben.lecture("Python is cool")
Python is cool ... How cool is that?
You should be taking notes ... How cool is that?

>>> ben.sing()
tra-la-la
```

Object Hierarchy



Multiple Inheritance

- Complication arises when the same method is available in two distinct superclasses
- Ben is both a singer and a lecturer, but primarily a lecturer
- If his internal arrogant lecturer has a method with the name given by the message, then that method is returned
- If the singer has no method with that name, then the message is passed to the internal singer.

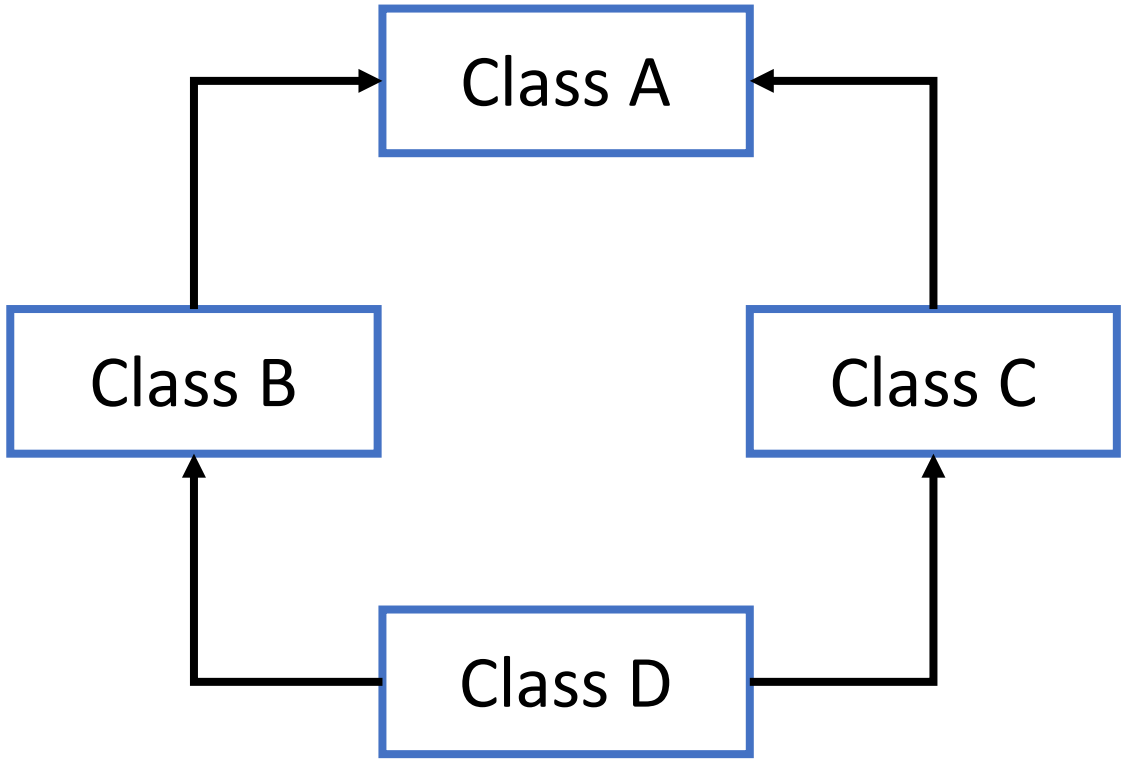
Diamond Inheritance

Suppose Singer inherits Speaker

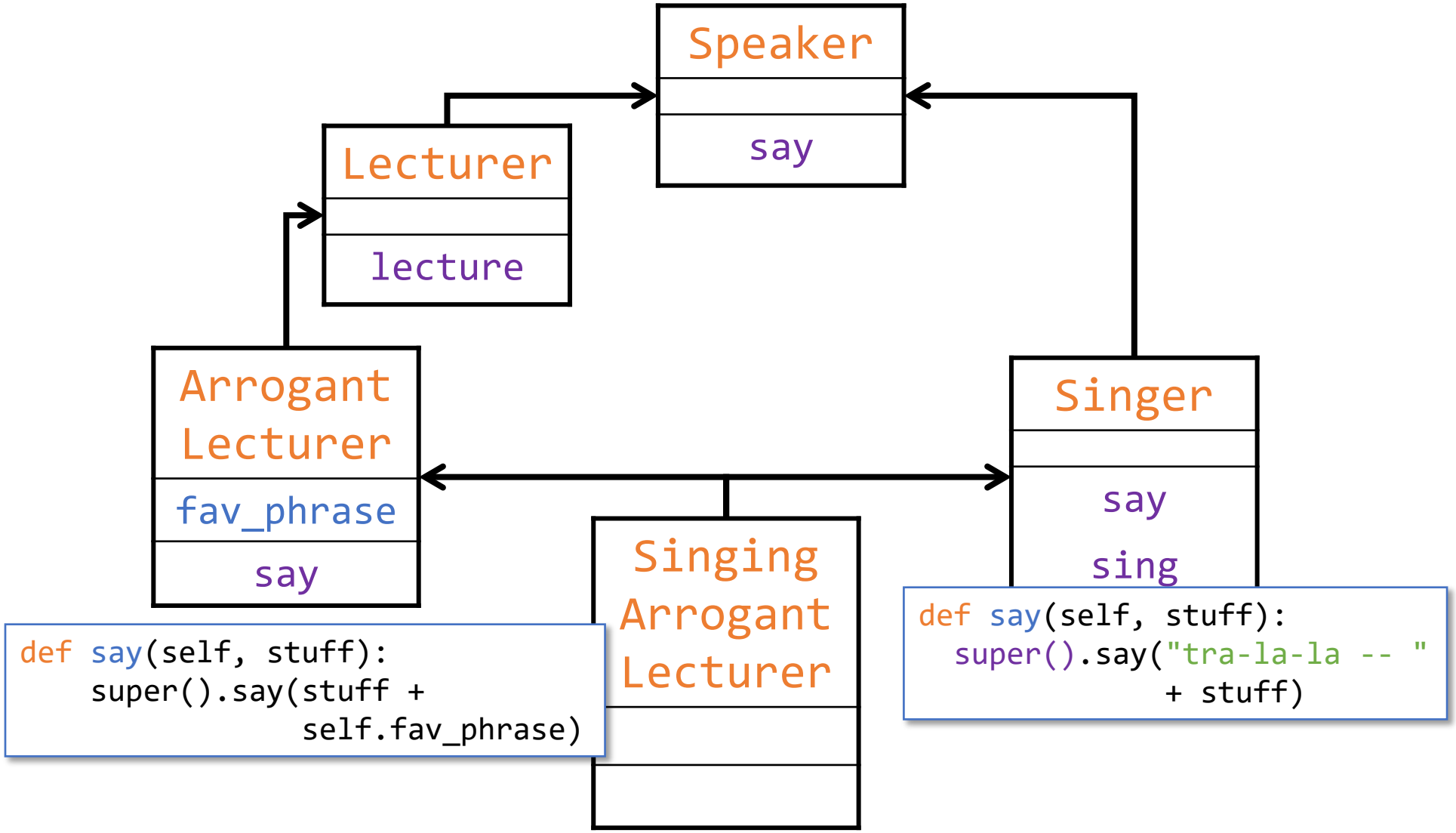
```
class Singer(Speaker):
    def say(self, stuff):
        super().say("tra-la-la -- " + stuff)

    def sing(self):
        print("tra-la-la")
```

Diamond Problem

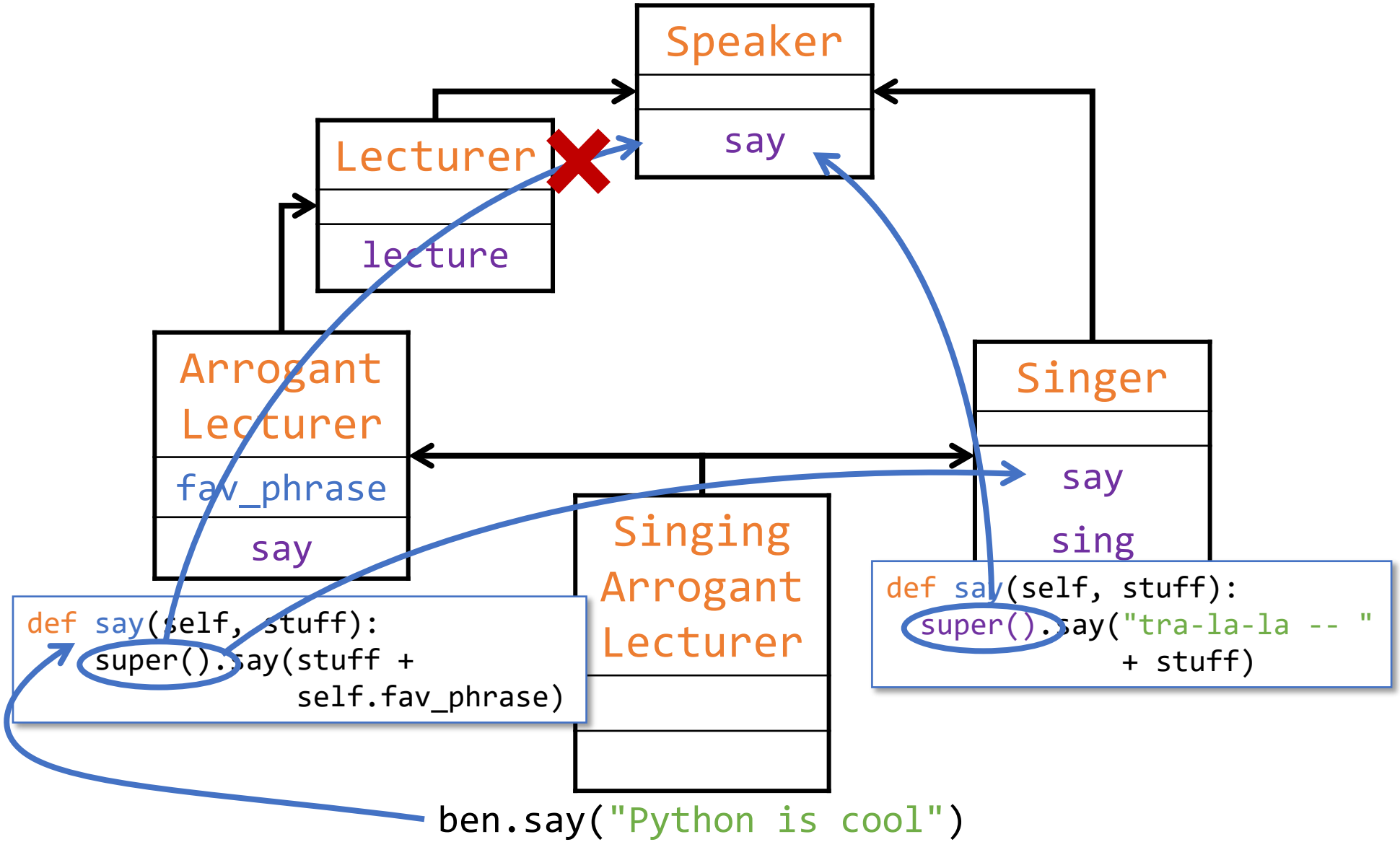
If all classes have same method.
Which class's method to call?



Diamond Hierarchy



Diamond Hierarchy



Benefits of OOP

- Simplification of complex, possibly hierarchical structures
- Easy reuse of code
- Easy code modifiability
- Intuitive methods
- Hiding of details through message passing and polymorphism

Costs of OOP

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++, Smalltalk

Python??

Which is the best paradigm?

- 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: capture common behavior
- Instances: unique identity with own local state
- Hierarchy of classes
  - Inheritance of state and behavior from superclass
  - Multiple inheritance: rules for finding methods
- Polymorphism : override methods with new functionality