Interface for a course

- Each courses has:
  - a number (e.g., 42)
  - a name (e.g., Principles and Practices of Computer Science)
- We can:
  - create a course (initializing it with its number and name)
  - access / change a course's number
  - access / change a course's name
  - determine if a course is intro-level
    true if the course's number is ≤ 100
  - print a course

Creating, accessing, and modifying a course

create an instance by "calling" the class (calls __init__)

```
cs42 = Course(42, 'Principles and Practices of Computer Science')

print(cs42.number)
```

data attribute access

```
print(cs42.isIntro())
```

method call

```
cs42.number = 1000
```

data attribute modification

```
print(cs42)
```

calls __str__

Python code for a course

```
class Course:
  '''Represents a course at Ivy Tech State (go Platypuses!)'''

  HIGHEST_INTRO_LEVEL = 100

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

  def isIntro(self):
    '''Returns True if this is an introductory-level course'''
    return self.number <= Course.HIGHEST_INTRO_LEVEL

  def __str__(self):
    return '{}: {}'.format(self.number, self.name)
```

class attribute

If there are no restrictions on the data attributes, they can be public; we don't need "getters" and "setters".

Interface for a student

- Each student has:
  - a name (e.g., Zhi)
  - an ID number (e.g., 101010101)
  - a collection of courses that the student has registered for

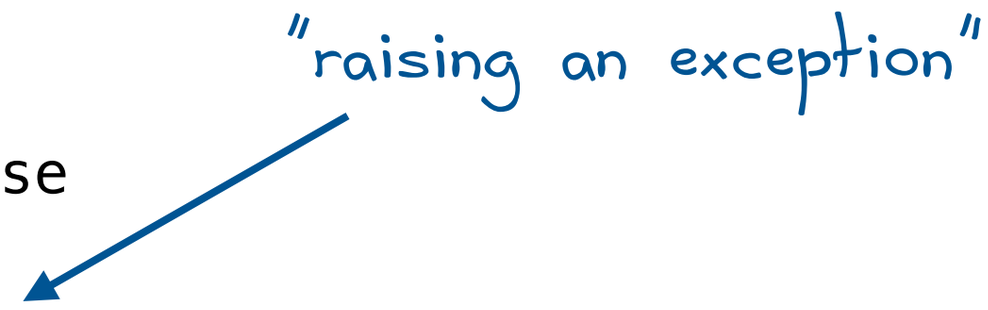
- We can:
  - create / initialize a student instance
  - access / change a student's name
  - access / change a student's ID number
  - access a list of a student's courses
  - register the student for a course
    only if the student is registered for < 5 courses
  - drop a student from a course

Interface for a student

- Each student has:
  - a name (e.g., Zhi) — string
  - an ID number (e.g., 101010101) — int or string
  - a collection of courses that the student has registered for — list of courses or dictionary of number → course
- We can:
  - create / initialize a student instance — constructor
  - access / change a student's name — N/A?
  - access / change a student's ID number — N/A?
  - access a list of a student's courses — courses()
  - register the student for a course — register(course)
    only if the student is registered for < 5 courses
  - drop a student from a course — drop(course)

Python code for a student

```
class Student:
  '''...'''

  def __init__(self, number, name):
    self.number = number
    self.name = name
    self._courses = {}

  def courses(self):
    '''...'''
    return self._courses.values()

def add(self, course):
  '''...'''
  if (len(self.courses()) < 5):
    self._courses[course.number] = course
  else:
    raise ValueError("can't add more than four courses")

def drop(self, course):
  '''...'''
  return self._courses.pop(course.number, None)
```



Object-oriented programming

objects interacting with each other

```
from course import Course
from student import Student

# create a course
cs42 = Course(42, 'Principles and Practices of CS')
```

```
# create a student
ben = Student(101010, ‘Ben')

# register student for class
ben.add(cs42)
```

Aside: exception handling

a common feature in languages

```
from course import Course
from student import Student

# create a course
cs42 = Course(42, 'Principles and Practices of CS')
```

```
# create a student
ben = Student(101010, ‘Ben')

# (try to) register student for class
try:
  ben.add(cs42)
except ValueError as error:
  print(error)
```

Interface for a course

How can we extend the interface, ideally without modifying the existing one or knowing its implementation?

- Each courses has:
  - a number (e.g., 42)
  - a name (e.g., Principles and Practices of Computer Science)
  - a campus
- We can:
  - create a course (initializing it with its number and name)
  - access / change a course's number
  - access / change a course's name
  - determine if a course is intro-level
    true if the course's number is ≤ 100
  - print a course

interface

what a piece of code can do

implementation

how a piece of code works

type

describe a set of supported operations

class

implement a type's operations

subtype

add more operations to an existing type

subclass

re-use/modify an existing implementation

inheritance

usually extends interface *and* implementation

Inheritance in Python

`CampusCourse` inherits from `Course`

subclass

superclass

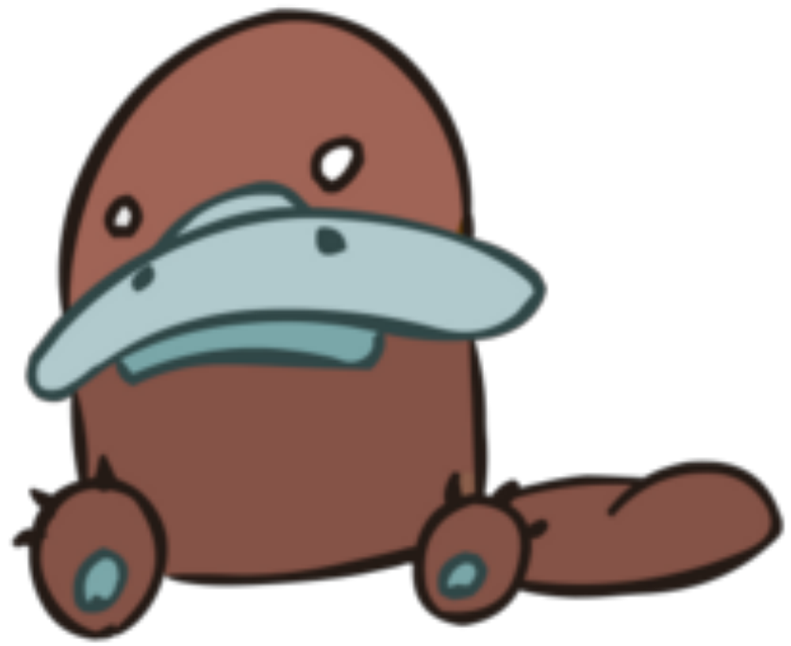
```
import course

class CampusCourse(course.Course):
    '''   '''
```

```
  def __init__(self, number, name, campus):
    self.campus = campus
    super().__init__(number, name)
```

extend the interface

reuse existing code

```
  def __str__(self):
    return self.campus + ' ' + super().__str__()
```



https://www.mysoti.com/img/user/tsswitch/product/web/1161649/1161649_show_default.png

Reusable components: modules

python

file / module / session

```
from course import Course
from student import Student


# create a course
cs42 = Course(42, '…')


# create a student
ben = Student(101010, 'Ben')

# register student for class
ben.add(cs42)
```

scopes

2

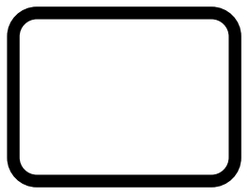
built-in

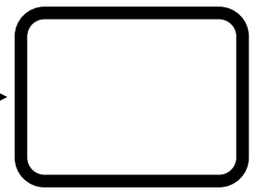
print ⟼ 

(and others)

1

global

Course ⟼ 

Student ⟼ 

namespaces

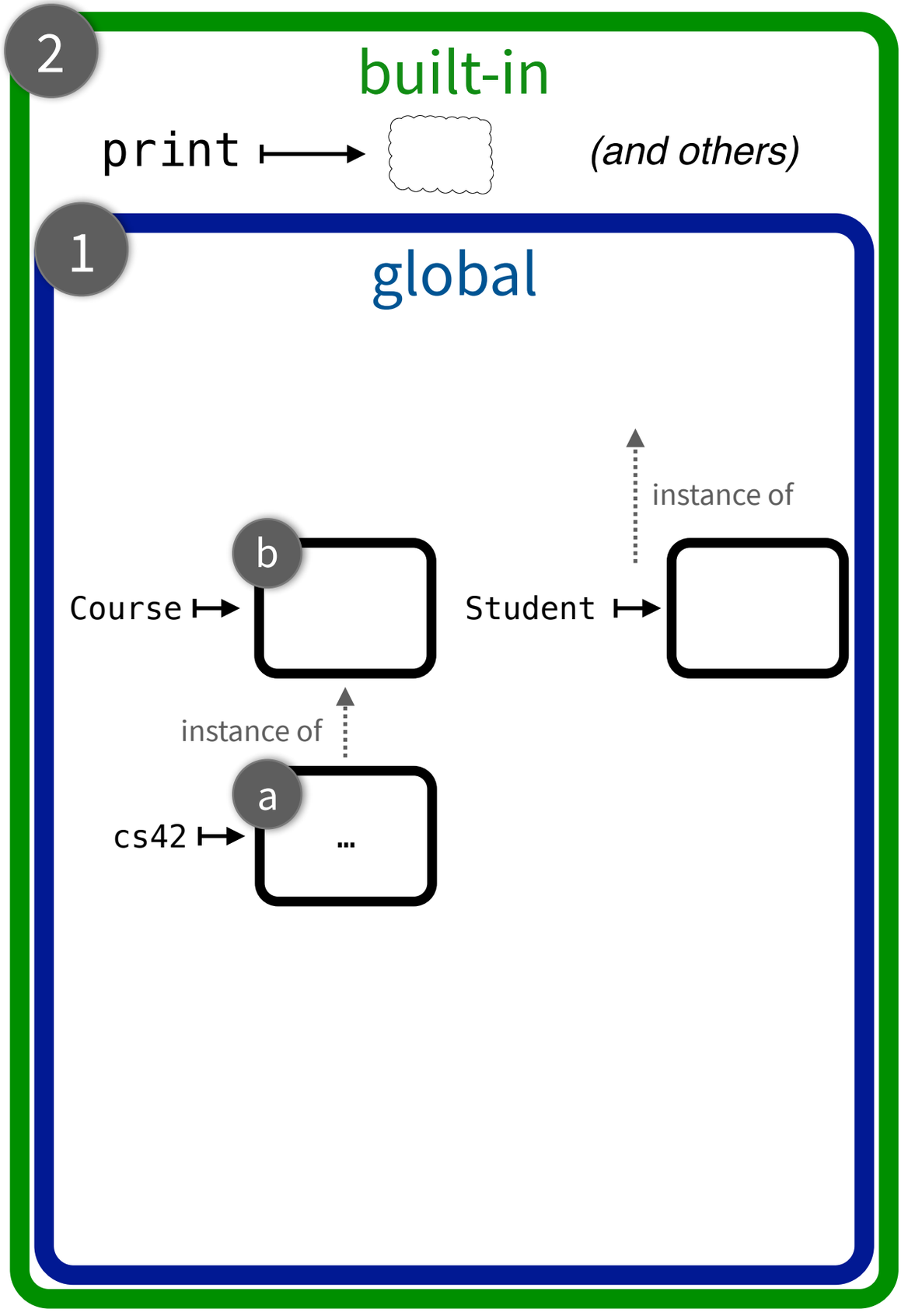
Reusable components: objects

python

file / module / session

```
from course import Course
from student import Student

# create a course
cs42 = Course(42, '…')

# create a student
ben = Student(101010, 'Ben')

# register student for class
ben.add(cs42)
```

scopes



namespaces

Reusable components: composition

python

file / module / session

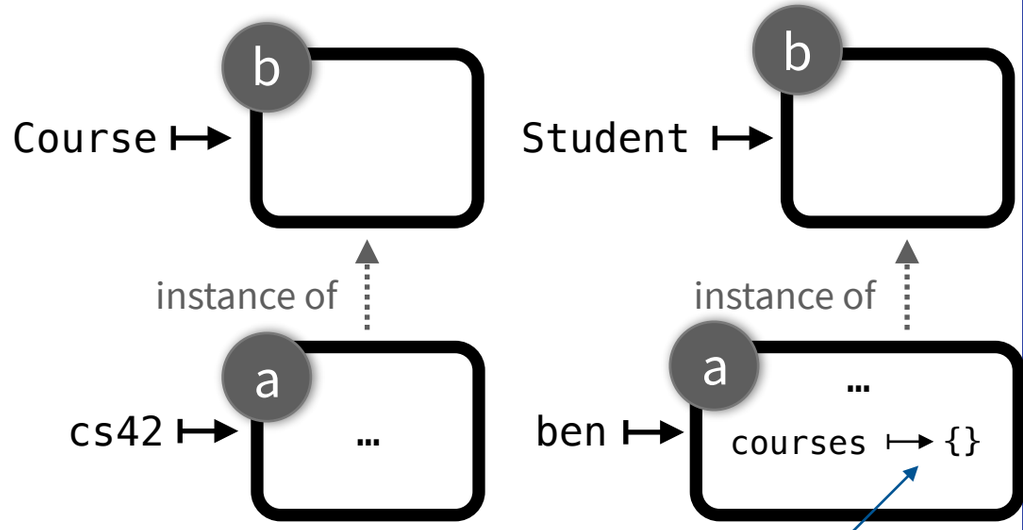
```
from course import Course
from student import Student

# create a course
cs42 = Course(42, '…')

# create a student
ben = Student(101010, 'Ben')

# register student for class
ben.add(cs42)
```

scopes

2 built-in

print ⟼  (and others)

1 global



Student "has" Courses

namespaces

Extensible components: inheritance

python

file / module / session

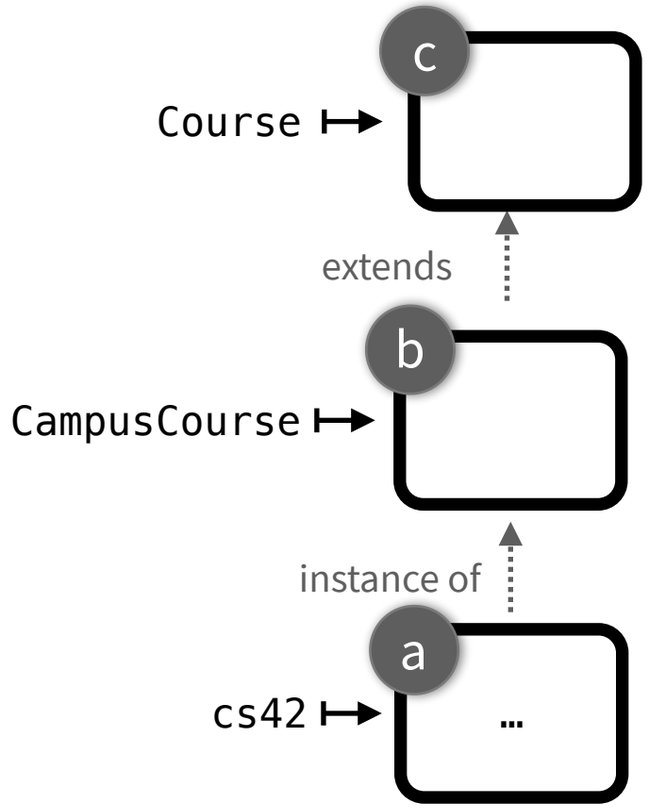
```
from course import CampusCourse
from student import Student


# create a course
cs42 = CampusCourse(42, '…', 'HMC')

# create a student
ben = Student(101010, 'Ben')

# register student for class
ben.add(cs42)
```

scopes

2 built-in

print ⟼  *(and others)*

1 global



namespaces

Let’s practice design

composition, inheritance, other, or “I don’t know”

Implement a new data structure: a “queue”

- stores a sequence of values
- **create** an empty queue
- get the **size** of a queue
- **enqueue**: add an element to the back of the queue
- **dequeue**: remove an element from the front of a queue
- **print** a queue

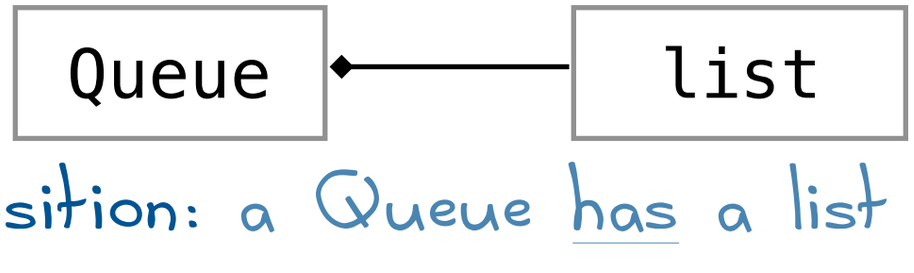
slido.com

#B425

Composition in Python

we've seen this before, with the `Stack` class

```
class Queue:
  '''A FIFO data structure'''

  def __init__(self):
    '''Creates a new Queue'''
    self._values = []

  def enqueue(self, item):
    '''Add an item to the end of the Queue'''
    self._values.append(item)

def dequeue(self):
  '''Removes and returns the item from the front of the Queue'''
  return self._values.pop(0)

def __len__(self):
  return len(self._values)

def __str__(self):
  return ', '.join(map(str, self._values))
```



Let's practice design

composition, inheritance, other, or "I don't know"

Implement some classes for a drawing program

- All shapes
  - have a **color**
  - have a **width**
- Some shapes are rectangles. All rectangles
  - have a **height**
- Some shapes are circles. All circles
  - have a **radius**

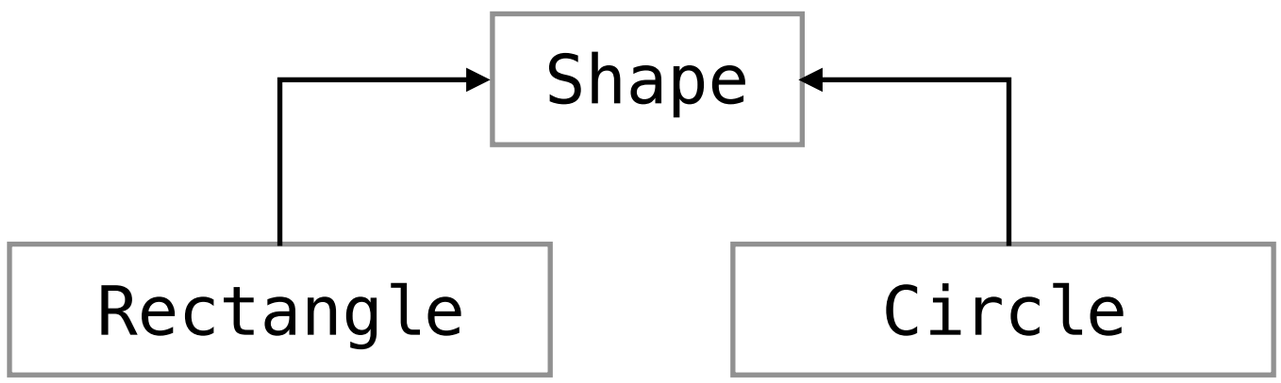
slido.com

#B425

Composition in Python

we've seen this before, with the `Stack` class

```
class Shape:
  def __init__(self, color, width):
    self.color = color
    self.width = width

class Rectangle(Shape):
  def __init__(self, color, width, height):
    super().__init__(color, width)
    self.height = height

class Circle(Shape):
  def __init__(self, color, radius):
    super().__init__(color, radius * 2)
    self.radius = radius
```



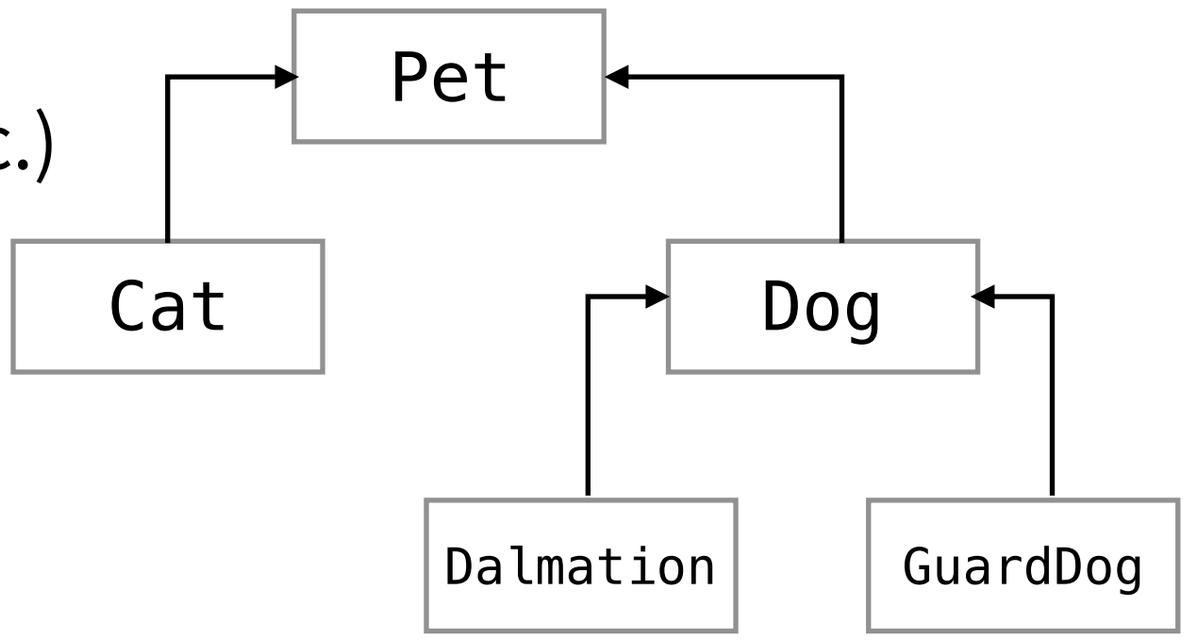

```
In [22]: import shape

In [23]: c = shape.Circle('red', 3)

In [24]: c.color
Out[24]: 'red'

In [25]: c.radius
Out[25]: 3

In [26]: c.width
Out[26]: 6

In [27]: isinstance(c, shape.Circle)
Out[27]: True

In [28]: isinstance(c, shape.Shape)
Out[28]: True

In [29]: isinstance(c, shape.Rectangle)
Out[29]: False
```

Let's practice design

composition, inheritance, other, or "I don't know"

Implement a bunch of classes for a game about a pet shelter

- All pets
  - have a **name**
  - have an **age**
  - have a **kind** (e.g., dog, cat, etc.)
  - can **speak**
- Some pets are cats
  - When cats **speak**, they meow
- Some pets are dogs
  - When dogs **speak**, they woof
  - Some dogs are Dalmatians
    - Dalmatians have **spots**
  - Some dogs are guard dogs
    - When dogs **speak**, they growl



slido.com

#B425

```
class Pet:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def speak(self):
    raise NotImplementedError

class Cat(Pet):
  def speak(self):
    print('Meow!')

class Dog(Pet):
  def speak(self):
    print('Woof!')

class GuardDog(Dog):
  def speak(self):
    print('Grrr!')

class Dalmation(Dog):
  def __init__(self, name, age, spots):
    super().__init__(name, age)
    self.spots = spots
```

subclasses should implement this method

no need to override the constructor; we're not changing it

Let's practice design

composition, inheritance, other, or "I don't know"

Implement a bunch of classes for a game about a pet shelter

- All pets
  - have a **name**
  - have an **age**
  - have a **kind** (e.g., dog, cat, etc.)
  - can **speak**
- Some pets are cats
  - When cats **speak**, they meow
- Some pets are dogs
  - When dogs **speak**, they woof
  - Some dogs are Dalmatians
    - Dalmatians have **spots**
  - Some dogs are guard dogs
    - When dogs **speak**, they growl

slido.com

#B425

Let's practice design

composition, inheritance, other, or "I don't know"

Implement some classes for a drawing program

- All shapes
  - have a **color**
  - have a **width**
- Some shapes are rectangles. All rectangles
  - have a **height**
  - Some rectangles are squares. All squares
    - have a **size** (the length of a side)
- Some shapes are circles. All circles
  - have a **radius**

slido.com

#B425

Let's practice design

composition, inheritance, other, or "I don't know"

Implement some classes for a drawing program

- All shapes
  - have a **color**
  - have a **width**
  - can have their **width stretched**
- Some shapes are rectangles. All rectangles
  - have a **height**
  - can have their **width stretched without modifying their height**
  - Some rectangles are squares. All squares
    - have a **size** (the length of a side)
- Some shapes are circles. All circles
  - have a **radius**

slido.com

#B425