ECE326

PROGRAMMING LANGUAGES

Lecture 11 : Multiple Inheritance and Mixin

Kuei (Jack) Sun
ECE
University of Toronto

Fall 2020

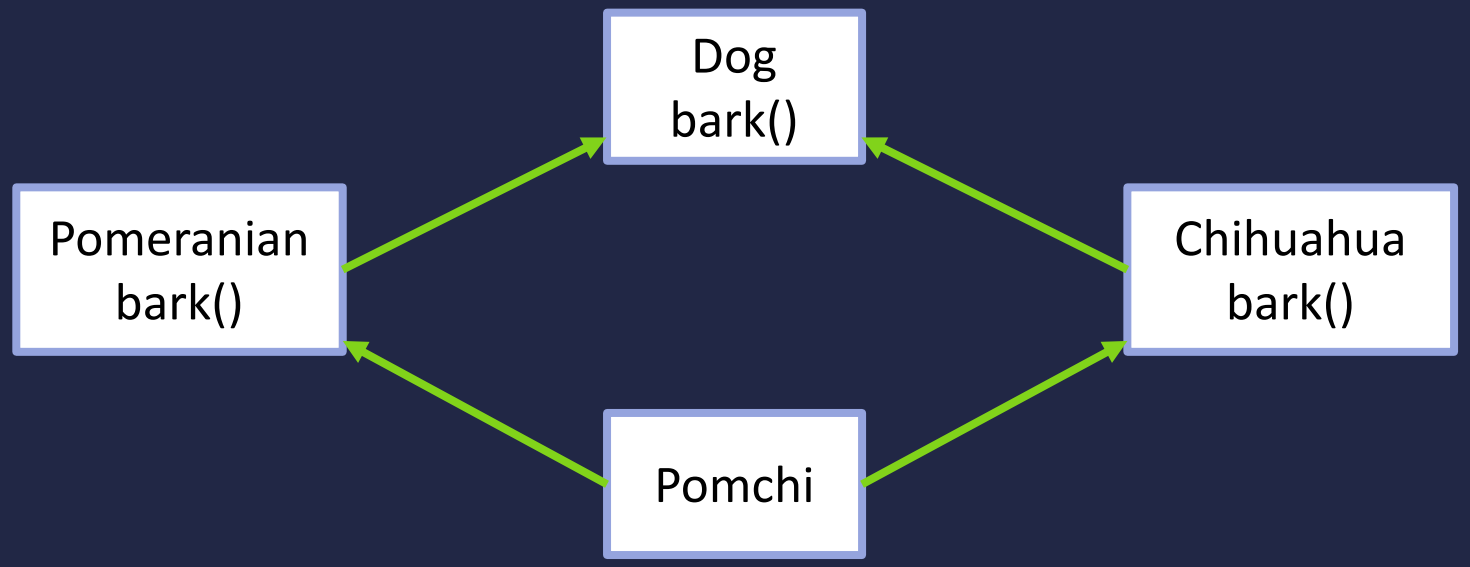
Multiple Inheritance

- When class is derived from multiple classes
- Models naturally complex relationships
- Requires deep understanding of name resolution in Python

```
class Dog:
    def bark(self):
        print("woof woof")

class Pomeranian(Dog):
    def bark(self):
        print("yap yap")

class Chihuahua(Dog):
    def bark(self):
        print("yip yip")

class Pomchi(Pomeranian, Chihuahua):
    pass
```

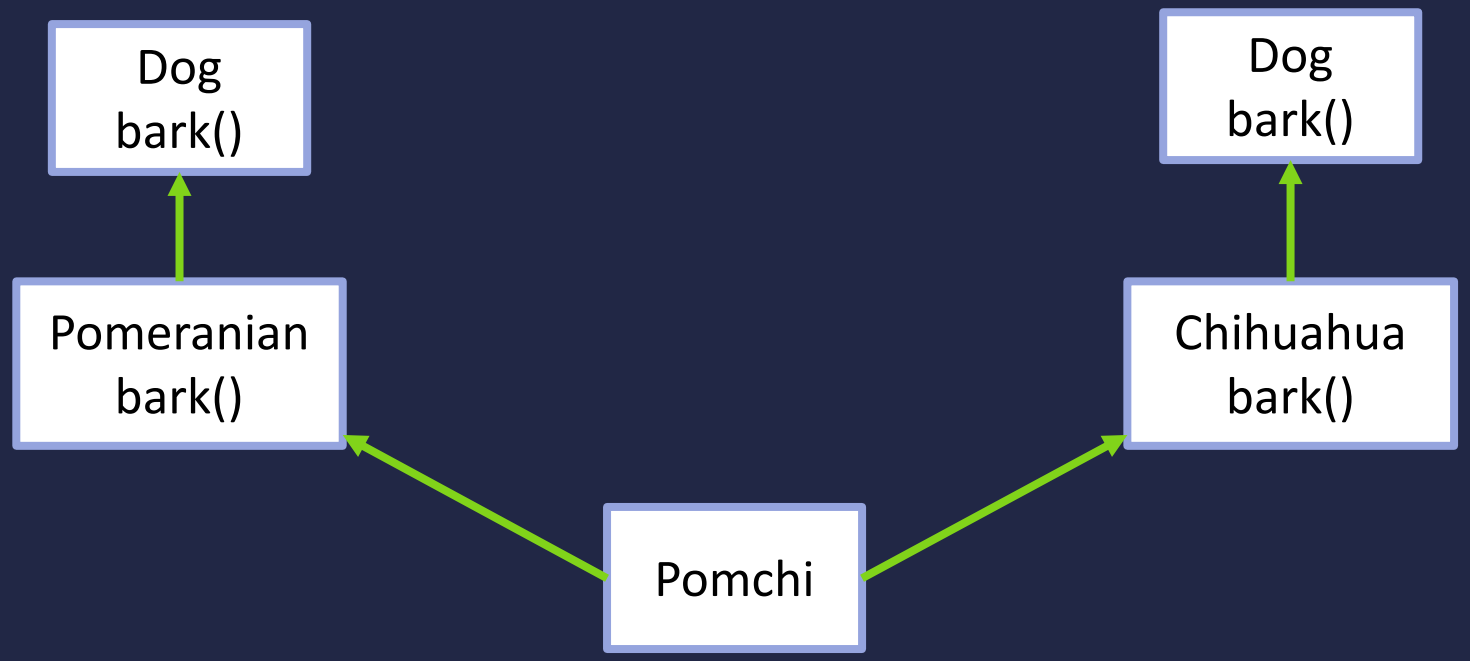
Diamond Problem

1. Inheriting from a class more than once
   - A problem in C++
   - Example: shared base class



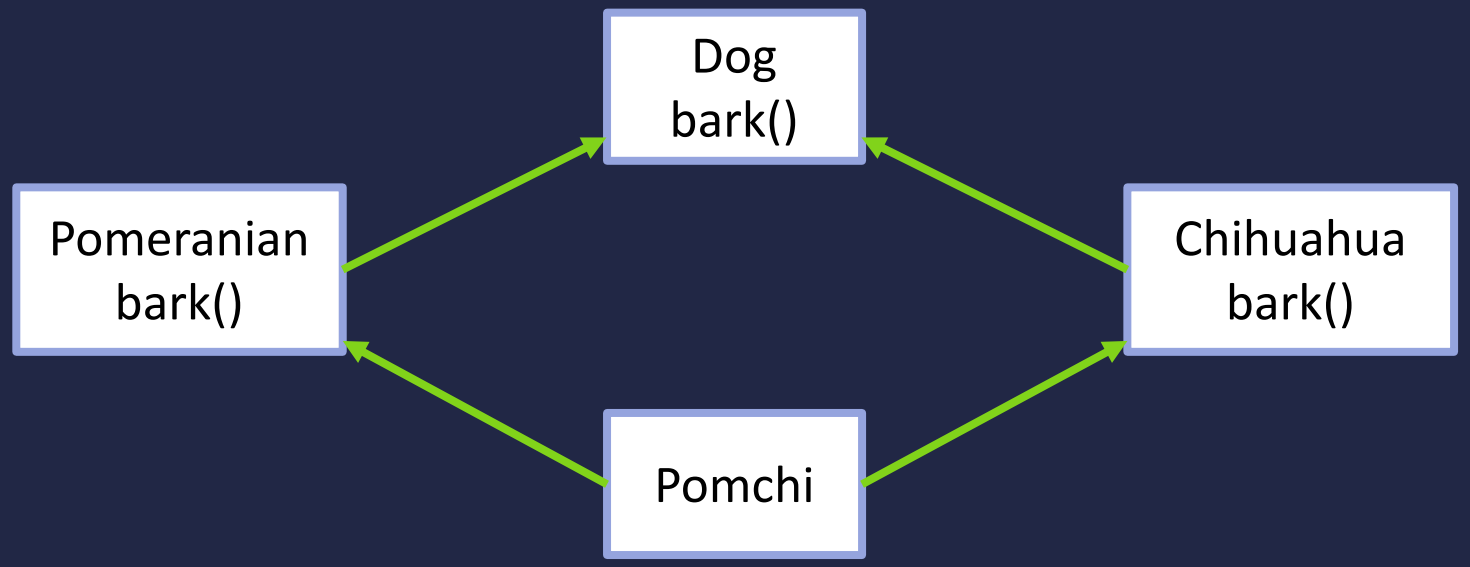
Diamond Problem

1. Inheriting from a class more than once
   - A problem in C++
   - Example: repeated base class



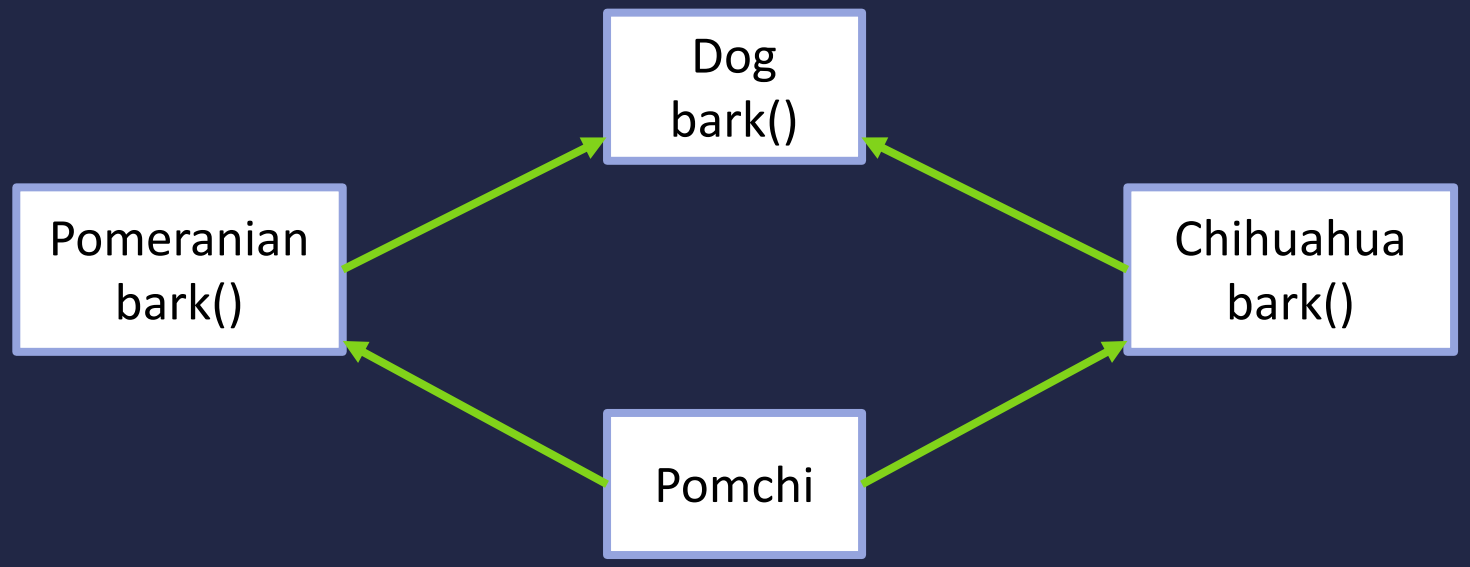
Diamond Problem

1. Inheriting from a class more than once
   - Not a problem in Python
   - A class can only be inherited at most one time
     - Object cannot contain multiple instances of a single class



Diamond Problem

2. Choosing which attribute a name resolves to
   - Method resolution order (MRO)
     - The order *attributes* are resolved name binding
     - Python uses linearization to unambiguously determine this order



help

- Provides inheritance information on an object
- Used for debugging purposes only

```
mishu = Pomchi()

# which bark() is called?
mishu.bark()

# use help() to see the MRO
help(mishu)
```

```
class Dog:
    def bark(self):
        print("woof woof")

class Pomeranian(Dog):
    def bark(self):
        print("yap yap")

class Chihuahua(Dog):
    def bark(self):
        print("yip yip")

class Pomchi(Pomeranian, Chihuahua):
    pass
```

__init__

- How to initialize all immediate super classes?
  1. Fully qualified name
     - Advantage
       - Very clear what is happening
     - Disadvantage
       - Cannot deal with the diamond problem
  2. super()
     - Advantage
       - Handles the diamond problem by using method resolution order
     - Disadvantage
       - Must use variable argument in method signature

Fully Qualified Name

```
class Person:
    def __init__(self, name, age):
        … set name and age …

class Student(Person):
    def __init__(self, name, age, id):
        self.id = id # print("In Student")
        Person.__init__(name, age)

class Teacher(Person):
    def __init__(self, name, age, resume):
        self.resume = resume # print("In Teacher")
        Person.__init__(name, age)

class TA(Student, Teacher):
    def __init__(self, name, age, id, resume):
        Student.__init__(name, age, id)
        Teacher.__init__(name, age, resume)
```

```
>> TA(“joey",
.. 21, 63490483,
.. “good bye")

In Student
In Person
In Teacher
In Person
```

Person’s initializer got called twice…

Cooperative Inheritance

- Take the arguments you need and pass the rest on
  - Power of Python's variable keyword arguments
  - super() is used to chain the __init__ calls in different classes

```
class Student(Person):
    def __init__(self, student_id, **kwargs):
        print("In Student")
        self.student_id = student_id
        super().__init__(**kwargs)
```

Automatically removed from kwargs

Keyword argument unpacking. Passes unused arguments to the next class

Cooperative Inheritance

```
class Person:
    def __init__(self, name, age):
        self.name = name # print("In Person")
        self.age = age

class Student(Person):
    def __init__(self, id, **kwargs):
        self.id = id # print("In Student")
        super().__init__(**kwargs)

class Teacher(Person):
    def __init__(self, resume, **kwargs):
        self.resume = resume # print("In Teacher")
        super().__init__(**kwargs)

class TA(Student, Teacher):
    def __init__(self, name, age, id, resume):
        super().__init__(name=name, age=age, id=id, resume=resume)
```

```
>> TA("jack",
.. 13, 48957257,
.. "hello world")

In Student
In Teacher
In Person
```

pass everything as keyword argument

Class Scope

- Each instance of class has its own scope
  - E.g. C++
  - Base and derived class can have same name
- Python
  - No class scope
  - Object has one big namespace
  - Trouble if multiple classes in inheritance use same name for different purposes

Mixin

- Code reuse without becoming the parent class
  - Inclusion rather than inheritance
- Provides functionality to another class
  - Should not be used as a standalone object
- Can contain states (e.g. have fields)
- Python Mixin
  - Use inheritance to enable code reuse
  - Convention: do not use mixin as a base class
    - `isinstance(obj, mixin)` is semantically meaningless

Type Slot

- A table of built-in (magic) methods
  - Operator overloading methods
    - E.g. __add__, __str__
  - Attribute interception methods
    - E.g. __getattr__, __setattr__
  - Attribute descriptors
- Look up for these methods go through *type slots*
  - Much simpler and faster
  - Not all built-in methods go through type slots
    - E.g. __prepare__

Mixin

- Python Comparable

```
# Comparable Mixin, you must supply __lt__ to enable these
class Comparable:
    def __eq__(self, other):
        return not (self < other) and not (other < self)
    def __ge__(self, other):
        return not (self < other)
    def __ne__(self, other):
        return self < other or other < self
    def __le__(self, other):
        return self < other or not (other < self)
    def __gt__(self, other):
        return not (self < other) and other < self
```

Mixin

```
class Student(Person, Comparable):
    def __init__(self, name, score):
        Person.__init__(self, name)
        self.score = score

    def __lt__(self, other):
        return self.score < other.score

>> a = Student(“Alice”, 50)
>> b = Student(“Bob”, 60)
>> a == b, a != b, a >= b, a > b, a <= b, a < b
(False, True, False, False, True, True)
>> c = Student(“Clive”, 50)
>> a == c, a != c, a >= c, a > c, a <= c, a < c
(True, False, True, False, True, False)
```

Receives all the other comparison operators just by implementing the less than operator