Object-Oriented Programming

Basic OOP Priciples

- Encapsulation: the data inside the object should only be accessed through the object's method.
  - In Python 3, the attribute whose name starts with __ but does not end with __ is private and not visible outside the class.
  - In Python, setter and getter methods can be used to control the access of attributes.

    ```
    @property, @property_name.setter, AND @property_name.deleter
    ```

- In Python, there are two main types of relationships between classes:
  - Inheritance is an is-a relation between classes and allows new classes to be defined as extensions or refinements of existing classes.
    - Multiple inheritance, mix-in
  - Composition is a has-a relation between classes, and is a way of combining simple objects into more complicated objects.

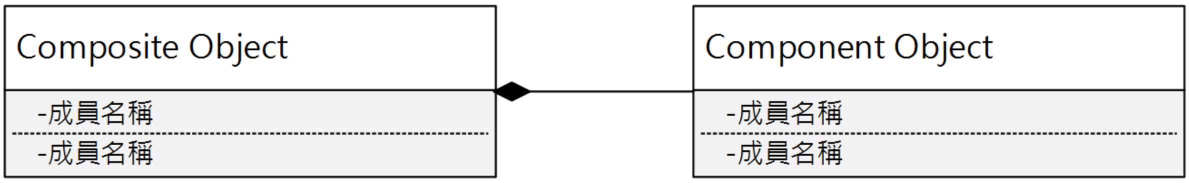
In [5]:

```python
class A:
    def __init__(self):
        self.__a   = 0
        self.__b__ = 1
a = A()
try:
    print(a.__a)
except BaseException as ae:
    print('Exception:{}'.format(ae))
```

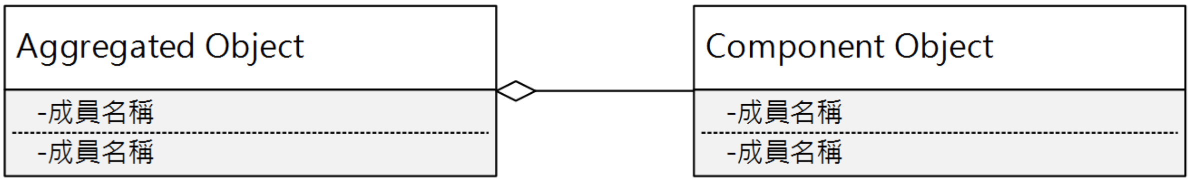
```
Exception:'A' object has no attribute '__a'
```

Composition and Aggregation

- Composition implies two objects linked strongly. The owner object has the responsibility for destroying the component object.
- Aggregation implies two objects linked weakly. The component object can be accessed through other objects and may outlive the owner object.



composition



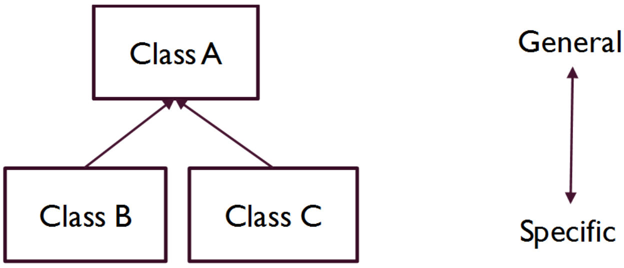
aggregation

Unified Modeling Language (UML)

Inheritance

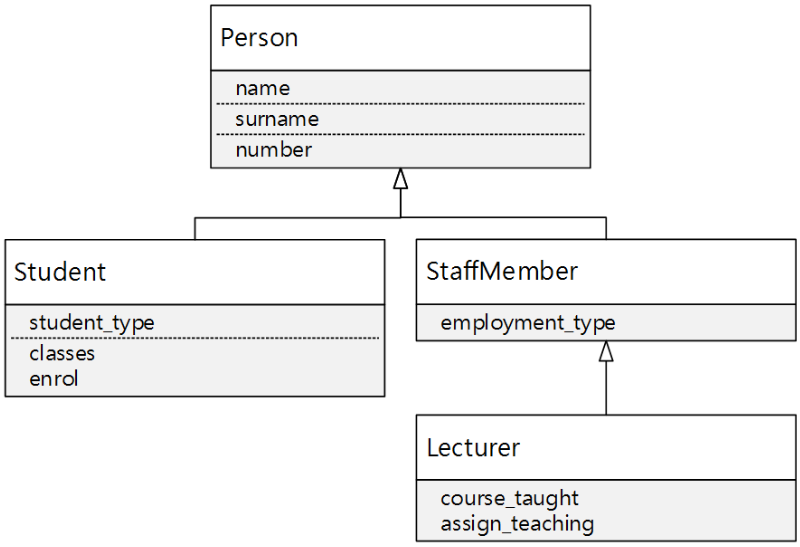
Inheritance can be used to arrange objects from the most general to the most specific in a hierarchy.

- The base class is the most general class in a class hierarchy.
- The subclass inherits the properties of the superclass.
- The subclass can override the method of the superclass and add new properties.



Class A is the superclass or parent class of classes B and C.
Classes B and C are subclasses of class A.

Example



In [7]:

```
class Person:
    def __init__(self, name, surname, number):
        self.name = name
        self.surname = surname
        self.number = number

class Student(Person):
    UNDERGRADUATE, POSTGRADUATE = range(2)
    def __init__(self, student_type,*args, ** kwargs):
        self.student_type = student_type
        self.classes = []
        super(Student, self).__init__( *args,** kwargs)
    def enrol(self, course):
        self.classes.append(course)

class StaffMember(Person):
    PERMANENT, TEMPORARY = range(2)
    def __init__(self, employment_type, *args, ** kwargs):
        self.employment_type = employment_type
        super(StaffMember, self).__init__( *args, ** kwargs)

class Lecturer(StaffMember):
    def __init__(self, *args, ** kwargs):
        self.courses_taught = []
        super(Lecturer, self).__init__( *args,** kwargs)
    def assign_teaching(self, course):
        self.courses_taught.append(course)
```

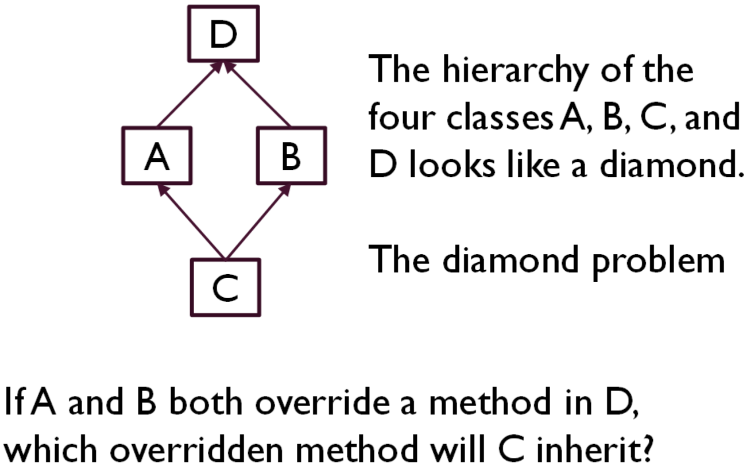
In [9]:

```
jane = Student(Student.POSTGRADUATE, "Jane", "Smith", "SMTJNX045")
jane.enrol('Machine Learning')
bob = Lecturer(StaffMember.PERMANENT, "Bob", "Jones", "123456789")
bob.assign_teaching('Python')
```

- How many properties does jane have?
- How many properties does bob have?

Multiple Inheritance

- In Python, a class can inherit from multiple classes. Due to multiple inheritance, an ambiguity known as the diamond problem can arise.



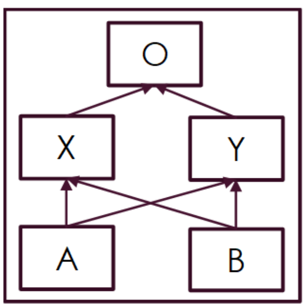
- Python creates a list of classes by the C3 method. This algorithm enforces two constraints:
  - local precedence order, and
  - the monotonicity criterion.

```
If C1 precedes C2 in the linearization of C, then C1 precedes C2 in the linearization of any subclass of C.
```

Method Resolution Order

- The C3 superclass linearization of a class is the merge of parents' linearizations and parents list.
- The C3 method:
  - The first head of the lists which does not appear in the tail (all elements of a list except the first) of any of the lists is selected.
  - If no good head can be selected, then no linearization of the original class exists due to cyclic dependencies in the hierarchy and break.
  - The selected element is removed from all the lists where it appears as a head and appended to the output list.
  - The above two steps are repeated until all remaining lists are exhausted.

Example 1.



In [14]:

```
class X(object): pass
class Y(object): pass
class A(X,Y): pass
class B(Y,X): pass
print(A.mro())
print(B.mro())
```

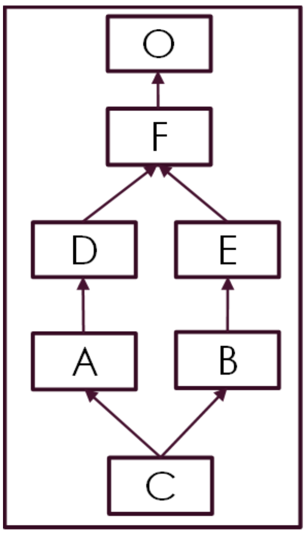
```
[<class '__main__.A'>, <class '__main__.X'>, <class '__main__.Y'>, <class 'o
bject'>]
[<class '__main__.B'>, <class '__main__.Y'>, <class '__main__.X'>, <class 'o
bject'>]
```

MRO

```
L(O)=[O]
L(X)=[X]+merge(L(O),[O])=[X,O]
L(Y)=[Y,O]
L(A)=[A]+merge(L(X),L(Y),[X,Y])
      =[A]+merge([X,O],[Y,O],[X,Y]
      =[A,X]+merge([O],[Y,O],[Y])
      =[A,X,Y]+merge([O],[O])
      =[A,X,Y,O]

A.mro(): [__main__.A, __main__.X, __main__.Y, object]
B.mro(): [__main__.B, __main__.Y, __main__.X, object]
```

Example 2.



```
L(C)=[C]+merge(L(A),L(B),[A,B])
       =[C]+merge([A,D,F,O],[B,E,F,O],[A,B])
       =[C,A]+merge([D,F,O],[B,E,F,O],[B])
       =[C,A,D]+merge([F,O],[B,E,F,O],[B])
       =[C,A,D,B]+merge([F,O],[E,F,O])
       =[C,A,D,B,E]+merge([F,O],[F,O])
       =[C,A,D,B,E,F]+merge([O],[O])
       =[C,A,D,B,E,F,O]

C.mro(): [__main__.C,  __main__.A,  __main__.D,  __main__.B,  __main__.E,  __main_
_.F,  object]
```

---

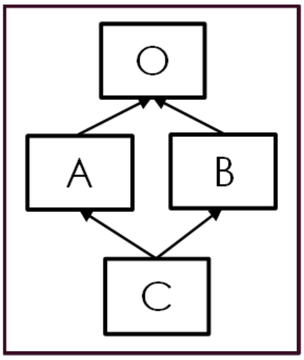
super()

```
help(super)
 super() -> same as super(__class__, <first argument>)
 super(type) -> unbound super object
 super(type, obj) -> bound super object; requires isinstance(obj, type)
 super(type, type2) -> bound super object; requires issubclass(type2, type)
```

- To resolute super(X,self).__init__(), Python starts from the class next to X in self.__class__.__mro__.

Example 1

The class hierarchy of the following classes:



The mro of class C is [__main__.C, __main__.A, __main__.B, object].

In [3]:

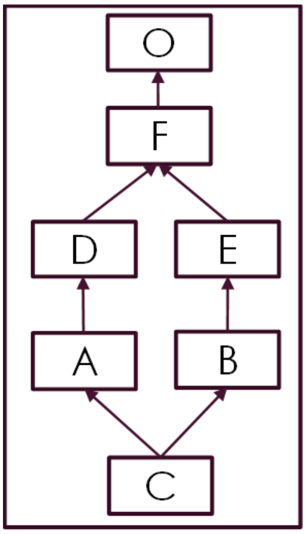
```
class A:
    def __init__(self):
        print('initA')
        self.x = 0
    def f(self):
        print('fA')

class B:
    def __init__(self):
        print('initB')
        self.y = 1
    def f(self):
        print('fB')

class C(A,B):
    def __init__(self):
        print('C')
        # C.mro() [__main__.C, __main__.A, __main__.B, object]
        super(C,self).__init__() # __main__.A
        super(A,self).__init__() # __main__.B
c = C();
c.f() # fA
super(A,c).f() # fB
```

```
C
initA
initB
fA
fB
```

Example 2.



C.mro(): [__main__.C, __main__.A, __main__.D, __main__.B, __main__.E, __main__.F, object]

In [7]:

```
class F:
    def __init__(self):
        print('initF')

class D(F):
    def __init__(self):
        print('initD')
        super().__init__()
        self.d = 0;
class E(F):
    def __init__(self):
        print('initE')
        super().__init__()
        self.e = 0;
class A(D):
    def __init__(self):
        print('initA')
        self.x = 0
        super(A,self).__init__()

    def f(self):
        print('fA')
class B(E):
    def __init__(self):
        print('initB')
        super(B,self).__init__()
        self.y = 1
    def f(self):
        print('fB')

class C(A,B):
    def __init__(self):
        print('initC')
        print('-'*10)
        super(C,self).__init__()
        print('-'*10)
        super(A,self).__init__()
        print('-'*10)
        super(B,self).__init__()
c = C()
```

```
initC
----------
initA
initD
initB
initE
initF
----------
initD
initB
initE
initF
----------
initE
initF
```

Mix-Ins

- Mix-ins can be used to avoid the inheritance ambiguity due to multiple inheritance. A mix-in can be regarded as an interface with implemented methods.
- A mix-in is responsible for providing a specific piece of optional functionality.

In [1]:

```
class Person:
    def __init__(self, name, surname, number):
        self.name = name
        self.surname = surname
        self.number = number

class TeacherMixin:
    def __init__(self):
        self.courses_taught = []
    def assign_teaching(self, course):
        self.courses_taught.append(course)

class LearnerMixin:
    def __init__(self):
        self.classes = []
    def enrol(self, course):
        self.classes.append(course)

class Tutor(Person, LearnerMixin, TeacherMixin): # LearnerMixin, TeacherMixin: More functic
    def __init__(self,* args, ** kwargs):
        super(Tutor, self).__init__( * args,** kwargs)
        super(Person,self).__init__()
        super(LearnerMixin,self).__init__()

jane = Tutor("Jane", "Smith", "SMTJNX045")
jane.enrol('Machine Learning')
jane.assign_teaching('Python')
```

Abstract Class and Interfaces

- In C++, an abstract class, which cannot be instantiated, can be defined and be used for creating subclasses.
    - The abstract class can have a set of methods that is required for some tasks, and is an interface definition. The subclass must implement these methods.
- In Python, any class can be instantiated. The abstract class can be defined in the following two manners.
    - The abstract method raises the NotImplementedError exception.

In [13]:

```
class Shape2D:
    def area(self):
        raise NotImplementedError()
class Shape3D:
    def volume(self):
        raise NotImplementedError()
```

The abc module can also be used to define abstract classes.

Example

In [ ]:

```
from abc import ABCMeta, abstractmethod

class Ordering(metaclass=ABCMeta):
     @abstractmethod
    def __eq__(self,other):
        pass
     @abstractmethod
    def __le__(self,other):
        pass

    def __ge__(self, other):
                return other<=self
    def __lt__(self, other):
        return self<=other and not self == other
    def __gt__(self, other):
        return not self<=other
    def __ne__(self,other):
        return not self == other

class Integer(Ordering):
    def __init__(self, i):
        self.i = i
    def __le__(self, other):
        return hasattr(other,"i") and self.i <= other.i
    def __eq__(self, other):
        return hasattr(other,"i") and self.i == other.i
```

We can also define relational operators through @functools.total_ordering

In [ ]:

```
import functools

@functools.total_ordering
class Integer(object):
    def __init__(self, i):
        self.i = i
    def __le__(self, other):
        return hasattr(other,"i") and self.i <= other.i
    def __eq__(self, other):
        return hasattr(other,"i") and self.i == other.i
```

Avoid Inheritance

- In Python, a function can work on objects if they both have the appropriate attributes and methods even if these objects don't share a parent class, and are completely unrelated.
  - In Python, inheritance is not required for polymorphism.
  - Check for the presence of the methods and attributes that the function requires to use. It is not necessary to check if the object is in the same class hierarchy.
- Sometimes, similar results can be obtained by replacing inheritance with composition.

In [12]:

```
class Person:
    def __init__(self, name, surname, number, learner=None, teacher=None):
        self.name = name
        self.surname = surname
        self.number = number
        self.learner = learner
        self.teacher = teacher

class Learner:
    def __init__(self):
        self.classes = []
    def enrol(self, course):
        self.classes.append(course)

class Teacher:
    def __init__(self):
        self.courses_taught = []
    def assign_teaching(self, course):
        self.courses_taught.append(course)
#----------------------------------------------------------------------
jane = Person("Jane", "Smith", "SMTJNX045", Learner(), Teacher())
jane.learner.enrol('Machine Learning')
jane.teacher.assign_teaching('Python')
```