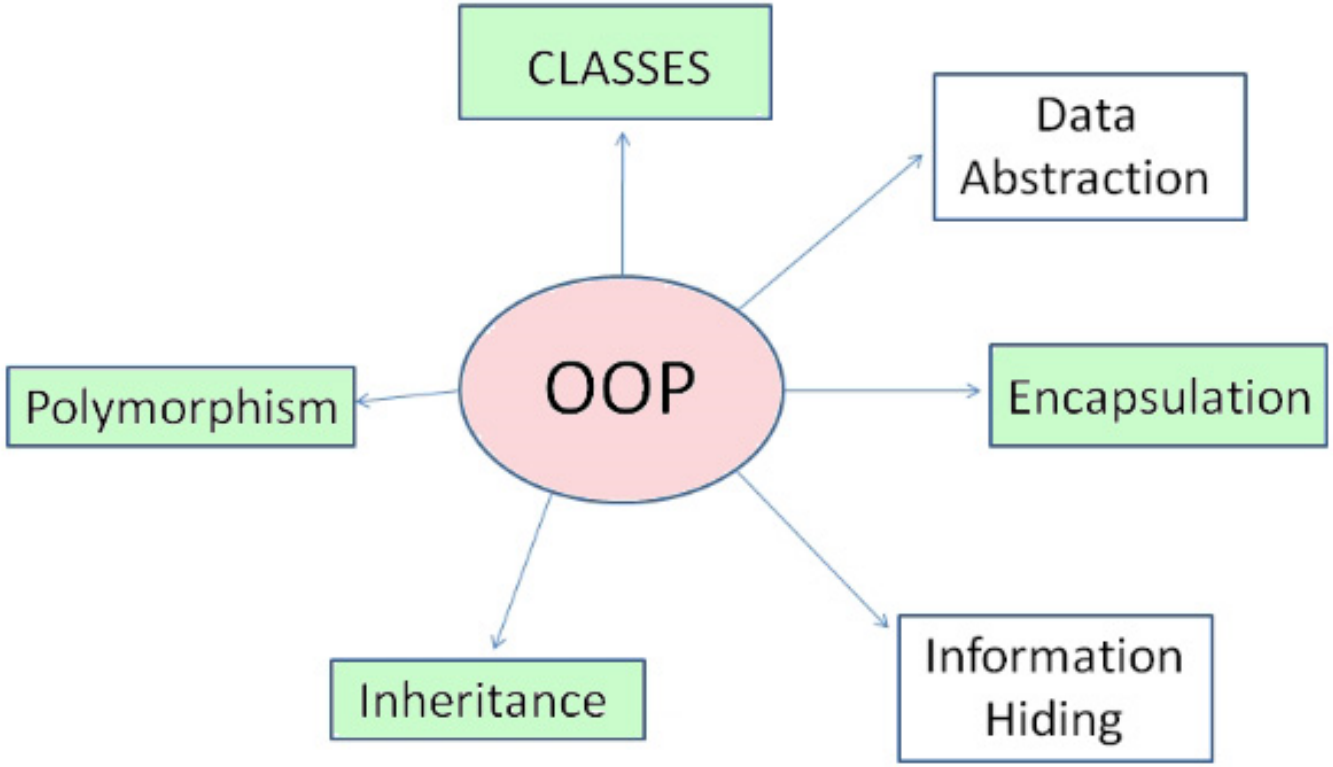
# Лекція 22

## Об'єктно-орієнтоване програмування

(продовження)

**Структура ООП, яка вперше була запропонована доктором комп'ютерних наук, страшим архітектором по розробці програмного забезпечення Sun та IBM Річардом Гебріелом.**



## ІНКАПСУЛЯЦІЯ(encapsulation)

Укладання в «капсулу», обмеження доступу до вмісту «капсули» з зоні і відсутність такого обмеження в середині «капсули»

**Інкапсуляція** – один з основних принципів ООП

**Інкапсуляція** – властивість мови програмування, яка дозволяє користувачеві не задумуватися про складність реалізації програмного забезпечення, яке використовується (не думати про внутрішні принципи його роботи), а взаємодіяти з ним шляхом використання інтерфейсу

## **Інкапсуляція** (продовження)

Інкапсуляція - це концепція розробки програмного забезпечення

Ідея інкапсуляції полягає в тому, щоб сховати логіку функціонування від зовнішнього доступу, а користувачеві даного фрагмента коду надати тільки інтерфейс для його використання.

1. При цьому виникає можливість звертатися до інкапсульованого фрагмента коду з різних областей програми, що заощаджує розмір коду.
2. Якщо необхідно модифікувати інкапсульований фрагмент, то це не вплине на працездатність всієї програми.

В Python інкапсуляція виконується за допомогою методів класу.

## Приклад розширеного класу Person

Розглянемо приклад, що дозволяє зробити клас `Person` таким, що не тільки вміє зберігати дані, але й виконувати деякі дії над ними.

**Приклад 1.** Опис розширеного класу

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self): # Метод виводу прізвища
                return self.name.split()[-1]
                # Повертає останнє слово рядка
        def giveRaise(self, percent):
                self.pay = int(self.pay + (self.pay/100)*percent)
```

# Створення екземплярів класу Person

**Приклад 2**

```
petro = Person("Petro Petrenko","Python developer", 10000)
ivan = Person("Ivan Ivanov")
print(petro.lastName(), ivan.lastName())
# Використали новий метод lastName()
petro.giveRaise(10)
print(petro.name, petro.pay)
print(ivan.name, ivan.pay)
```

Результат виконання:

```
Petrenko Ivanov
Petro Petrenko 11000
Ivan Ivanov 0
```

## Перезавантаження операторів

В Python є можливість:

1. Перехоплювати за допомогою спеціальних вбудованих методів виконання стандартних операцій
2. Застосовувати нові варіанти цих операцій до екземпляра класу.

Розглянемо виконання перезавантаження операторів на прикладі методу **`__str__`**.

Цей метод належить типу `class` і викликається, якщо атрибут екземпляра класу необхідно перетворити у рядок для друку.

Розглянемо перезавантаження операторів, модифікувавши клас Person.

## Приклад 3. Перезавантаження методу `__str__`.

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self): # Метод виводу прізвища
        return self.name.split()[-1]
    def giveRaise(self, percent):
                self.pay = int(self.pay+ (self.pay/100)*percent)
    def __str__(self): # Added method
        return '[Person: %s, %s]' % (self.name, self.pay)
petro = Person("Petro Petrenko","Python developer", 10000)
ivan = Person("Ivan Ivanov")
petro.giveRaise(10)
print(petro)
print(ivan)
```

Результат виконання: [Person: Petro Petrenko, 11000]

[Person: Ivan Ivanov, 0]

## Спадкування

Спадкування, як і інкапсуляція є концепцією розробки.

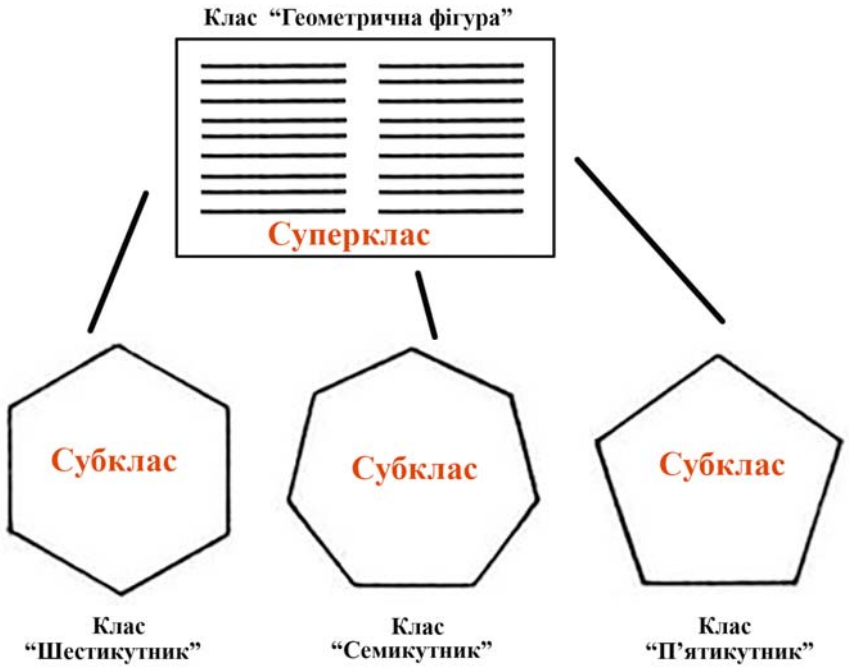
Це однин з найголовніших принципів ООП.

Основна мета спадкування: можливість створення ієрархій класів.

Існування ієрархій класів дозволяє істотно скоротити код програмного забезпечення за рахунок запозичення певних властивостей у класів, які є предками даного класу.

Отже, скорочення опису класів за рахунок використання їх подібності відбувається за рахунок створення **ієрархії**.

У корені цієї ієрархії знаходиться базовий клас, від якого походять всі інші класи ієрархії. Ці класи **успадковують** свої атрибути, уточнюючи й розширюючи поведінку нащадків класу.




Клас «Геометрична фігура» є предком класів «П'ятикутник, шестикутник та семикутник»

Класи «П'ятикутник, шестикутник та семикутник» є нащадками класу «Геометрична фігура»

«Геометрична фігура» **суперклас** (надклас)

«П'ятикутник, шестикутник та семикутник» **- субкласи** (підкласи)

## Ієрархія спадкування

Нехай у нас є клас (наприклад, `OldClass`). За допомогою спадкування ми можемо створити новий клас (наприклад, `NewClass`), у якому буде реалізований доступ до всіх атрибутів і методів класу `OldClass`.

```
class OldClass:
    <Атрибути OldClass>
    <Методи OldClass>

class NewClass(OldClass):
    <Атрибути NewClass>
    <Методи NewClass>
N=NewClass()
N.OldClass_attribute
N.NewClass_attribute
```

**Приклад 4**. Спадкування

```
class Class1: # Базовий клас
    def func1(self):
        print("Метод func1() класу Class1")
    def func2 (self) :
        print("Метод func2() класу Class1")
class Class2(Class1):# Клас Class2 успадковує клас Class1
    def func3 (self) :
        print("Метод func3() класу Class2")
c = Class2 ()   # Створюємо екземпляр класу Class2
c.func1()       # Виведе: Метод func1() класу Class1
c.func2()       # Виведе: Метод func2() класу Class1
c.func3()       # Виведе: Метод func3 () класу Class2
Результат виконання:
Метод func1() класу Class1
Метод func2() класу Class1
Метод func3() класу Class2
```

## Синтаксис спадкування

Клас `OldClass` вказують всередині круглих дужок у визначенні класу `NewClass`.

**Приклад 5.**

```
class OldClass: pass
class NewClass(OldClass):pass
c = NewClass
```

Таким чином, клас `NewClass` успадковує всі атрибути й методи класу `OldClass`.

Клас `OldClass` називають базовим або суперкласом, а клас `NewClass` – похідним або підкласом.

Якщо ім'я методу в класі `NewClass` збігається з ім’ям методу класу `OldClass`, то буде використовуватися метод з класу `NewClass`.

Щоб викликати однойменний метод з базового класу, слід указати перед методом назву базового класу. Крім того, у першому параметрі методу необхідно явно вказати посилання на екземпляр класу. Розглянемо це на прикладі.

**Приклад 6**. Перевизначення методів

```
class OldClass:          # Базовий клас
    def __init__(self):
        print("Конструктор базового класу")
    def func(self):
        print("Метод func() класу OldClass")
class NewClass(OldClass): # Клас NewClass успадковує клас OldClass
    def __init__(self):
       print("Конструктор підкласу")
       OldClass.__init__(self)# Конструктор базового класу
    def func(self):
        print("Метод func() класу NewClass")
        OldClass.func(self)# Викликаємо метод базового класу
c = NewClass()     # Створюємо екземпляр класу NewClass
c.func()   # Викликаємо метод func ()
Результат виконання:
Конструктор підкласу
Конструктор базового класу
Метод func() класу NewClass
Метод func() класу OldClass
```

УВАГА!

Конструктор базового класу автоматично не викликається, якщо він перевизначений у підкласі.

Щоб викликати однойменний метод з базового класу, можна також скористатися функцією **super().**

Формат функції:

```
super([<Класс>, <self>])
```

За допомогою функції super():

```
OldClass.__init__(self)# Викликаємо конструктор базового класу
```

інструкцію можна записати так:

```
super().__init__()  # Викликаємо конструктор базового класу
```

або так:

```
super(NewClass, self).__init__()
#Викликаємо конструктор базового класу
```

При використанні функції `super()` не потрібно явно передавати вказівник `self` у викликуваний метод.

```
super().__init__()
```

Крім того, у першому параметрі функції `super()` вказують похідний клас (підклас), а не базовий.

```
super(NewClass, self).__init__()
```

Пошук ідентифікатора буде проводитися у всіх базових класах. Результатом пошуку стане перший знайдений ідентифікатор у ланцюжку спадкування.

## Множинне спадкування

У визначенні класу в круглих дужках можна вказати відразу кілька базових класів через кому. Розглянемо множинне спадкування на прикладі.

**Приклад 7**.

```
class Class1: # Базовий клас для класу Class2
    def func1 ( self) :
        print("Метод func1() класу Class1")
class Class2(Class1): # Class2 успадковує Class1
    def func2(self):
        print("Метод func2() класу Class2")
class Class3(Class1): # Class3 успадковує Class1
    def func1(self):#-----------
       print( "Метод func1 () класу Class3")
    def func2(self):
        print("Метод func2() класу Class3")
```

```
    def func3(self):
        print("Метод func3() класу Class3")
    def func4(self):
        print("Метод func4() класу Class3")
class Class4(Class2, Class3):#Множ. спадкування
    def func4 (self):
        print("Метод func4() класу Class4")
c = Class4() # Створюємо екземпляр класу Class4
c. func1()   # Виведе: Метод func1 () класу Class3
c. func2()   # Виведе: Метод func2 () класу Class2
c. func3()   # Виведе: Метод func3 () класу Class3
c. func4()   # Виведе: Метод func4 () класу Class4
Результат роботи:
Метод func1() класу Class3
Метод func2() класу Class2
Метод func3() класу Class3
Метод func4() класу Class4
```

Метод **func1()** визначений у двох класах: **Class1** і **Class3.**

Оскільки спочатку відбувається перегляд всіх базових класів, безпосередньо зазначених у визначенні поточного класу, метод func1() буде знайдений у класі Class3 (оскільки він зазначений у числі базових класів у визначенні Class4

Метод **func2()** також визначений у двох класах: Class2 і Class3. Оскільки клас Class2 розміщений першим у списку базових класів, то метод буде знайдений саме в ньому. Щоб одержати доступ до методу із класу Class3, слід указати це явно.

Змінимо визначення класу Class4 з попереднього прикладу й успадковуємо метод func2() з класу Class3.

**Приклад 8**.

```
class Class4(Class2, Class3):# Множ. Спадкування
# Успадковуємо func2() із класу Class3, а не із класу Class2
    func2 = Class3.func2
    def func4(self):
        print("Метод func4() класу Class4")
```

Метод **func3()** визначений тільки в класі `Class3`, тому метод успадковується від цього класу. Метод `func4()`, визначений у класі `Class3`, перевизначається в підкласі.

Якщо ж шуканий метод знайдений у підкласі, то вся ієрархія спадкування переглядатися не буде.

Для одержання переліку базових класів можна скористатися атрибутом `__bases__`.

Як значення атрибут повертає кортеж. Як приклад виведемо базові класи для всіх класів з попереднього прикладу:

**Приклад 9**.

```
print(Class1.__bases__)
print(Class2.__bases__)
print(Class3.__bases__)
print(Class4.__bases__)
```

Результат виконання:

```
(<class 'object'>,)
(<class '__main__.Class1'>,)
(<class '__main__.Class1'>,)
(<class '__main__.Class2'>, <class
'__main__.Class3'>)
```

Розглянемо порядок пошуку ідентифікаторів при складній ієрархії множинного спадкування

**Приклад 10**.

```
class Class1: x = 10
class Class2(Class1): pass
class Class3(Class2): pass
class Class4(Class3): pass
class Class5(Class2): pass
class Class6(Class5): pass
class Class7(Class4, Class6): pass
c = Class7 ()
print(c.x)
Результат виконання: 10
```

Послідовність пошуку атрибутів буде такою:
Class7->Class4->Class3->Class6->Class5->Class2->Class1

Одержати весь ланцюжок спадкування дозволяє атрибут __mro__:

```
class Class1: x = 10
class Class2(Class1): pass
class Class3(Class2): pass
class Class4(Class3): pass
class Class5(Class2): pass
class Class6(Class5): pass
class Class7(Class4, Class6): pass
c = Class7 ()
print(Class7. __mro__)
```

Результат виконання:

```
(<class '__main__.Class7'>, <class '__main__.Class4'>, <class '__main__.Class3'>, <class '__main__.Class6'>, <class '__main__.Class5'>, <class '__main__.Class2'>, <class '__main__.Class1'>, <class 'object'>)
```

# Поліморфізм

Парадигма об'єктно-орієнтованого програмування крім спадкування включає ще одну важливу особливість — поліморфізм.

Слово «поліморфізм» можна перевести як «багато форм».

В ООП (об’єктно-орієнтованому програмуванні) цим терміном позначають можливість використання того самого імені операції або методу до об'єктів різних класів, при цьому дії, виконувані з об'єктами, можуть суттєво різнитися. Тому можна сказати, що в одного слова багато форм.

**Приклад 11. Метод в одному класі**

```
class First:
    def func(self,a,b):
        return a+b
m=First()
result=m.func(25,75)
print(result)
result=m.func("При", "віт")
print(result)
result=m.func(True, True)
print(result)
Результат:100
Привіт
2
```

Використовуємо один і той же метод, який дає різну реакцію при вводі різних даних

## Однойменні методи в різних класах

Два різні класи можуть містити метод з назвою **`total`**. Інструкції у цих методах можуть передбачати зовсім різні операції:

Розглянемо приклад двох класів:

Клас FClass містить метод total, який виконує додавання до вхідного параметру числа 10.

Клас SClass містить метод **`total`**, який виконує підрахунок кількості символів у вхідному параметрі.

Залежно від того, до об'єкта якого класу застосовується метод **`total`**, виконуються ті або інші інструкції.

**Приклад 12.**

```
class FClass:
      n=10
      def total(self,N):
            self.total = int(self.n) + int(N)

class SClass:
      def total(self,s):
            self.total = len(str(s))
f = FClass()
s = SClass()
f.total(45)
s.total(45)
print (f.total) # Вивід: 55
print (s.total) # Вивід: 2
Результат виконання: 55 2
```