

# Лекція 4

## Об'єктно-орієнтоване програмування

Лектор: *Розова Людмила Вікторівна*

# План лекції 4



**Статичні методи**

---



**Успадкування**

---

Матеріали курсу:

<https://github.com/LRozova/oop1>

# Статичні методи

- Якщо статична змінна об'явлена у розділі **private** до неї має доступ відкритий **статичний метод (static)**.

**static int Point::getCount ();**

- Статичний метод не має покажчика this, тому що статичні поля і статичні методи існують незалежно від будь-яких об'єктів класу, тобто до них не прив'язані
- Метод класу може бути оголошений як static, якщо він не має доступ до нестатичних елементів класу.
- Статичний метод викликається з додаванням перед його ім'ям **імені класу** і бінарної операції оператора розширення видимості “::” **Point::getCounter();**  
або через об'єкт класу **p1.getCounter();**
- Статичні поля класу створюються в єдиному екземплярі незалежно від кількості визначених в програмі об'єктів.
- Всі об'єкти (навіть створені динамічно) поділяють єдину копію статичних полів.

```
class Account //клас банківський рахунок
{private:
    double sum;
    static int rate; //процентна ставка
    const static int rate_default=5; //стат. КОНСТАНТА
public:
    Account(double sum) ;
    double getIncome() ;
//оголошення статичних методів
    static int getRate();
    static void setRate(int r) ;
};

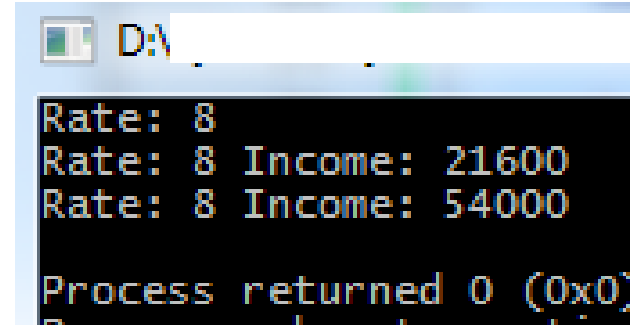
int Account::rate = 5;

Account::Account(double sum)
    {    this->sum = sum;    }

double Account::getIncome()
    {    return sum + sum * rate / 100;    }
```

//реалізація статичних методів другий раз static не вказується

```
int Account::getRate ()
{ return rate; }
void Account::setRate(int r)
{ rate = r; }
```



```
D:\...
Rate: 8
Rate: 8 Income: 21600
Rate: 8 Income: 54000
Process returned 0 (0x0)
```

```
int main()
{
    Account ac1(20000), ac2(50000);
    Account::setRate(8); //НОВЕ ЗНАЧЕННЯ rate
    cout <<"Rate:"<< Account::getRate() <<endl;
    cout <<"Rate:"<< ac1.getRate() <<"Income:"
        << ac1.getIncome() << endl;
    cout <<"Rate:"<<ac2.getRate() <<"Income:"
        << ac2.getIncome() << endl;
    return 0; }
```

# Успадкування

**Успадкування** – створення нових класів на базі існуючих.

Це дуже потужна можливість в ООП, що дозволяє створювати нові похідні класи, взявши за основу всі методи і елементи базового класу. Таким чином економиться час на написання і налагодження коду нової програми.

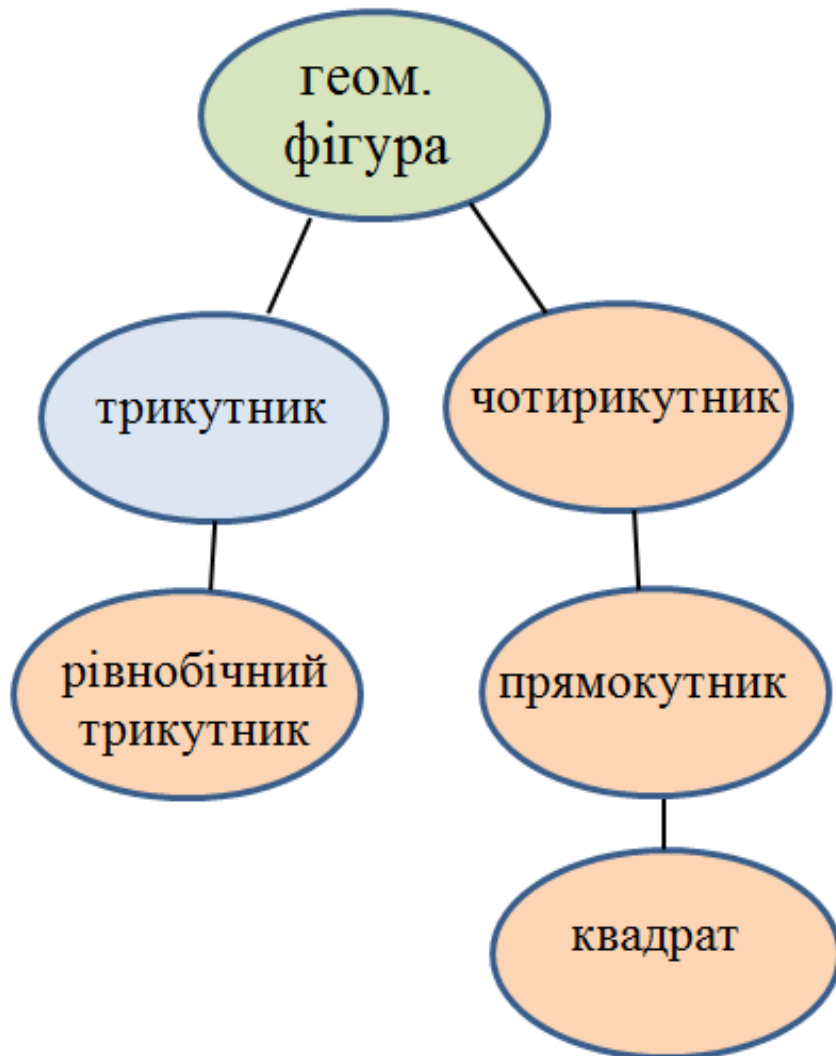


Клас, який успадковується, називається **базовим** (або **предком, суперкласом**).

Клас, який успадковує базовий клас, називається **похідним** (або **нащадком, підкласом**).

Похідний клас можна використовувати в якості базового для іншого похідного класу. Таким чином вбудовується багаторівнева ієрархія класів.

**Успадкування** включає в себе створення нових об'єктів шляхом безпосереднього збереження властивостей і поведінки інших об'єктів, а потім їх розширення або навпаки - конкретизації.



Кожен об'єкт похідного класу є також об'єктом відповідного базового класу. Однак, зворотне невірно: об'єкт базового класу не є об'єктом класів, породжених цим базовим класом.

У нащадка можна описувати *нові поля і методи*, а також *перевизначити існуючі методи*.

Види успадкування: **просте і множинне**.

**Просте успадкування** - кожен клас має тільки один батьківський клас найближчого рівня.

**Множинне спадкування** - клас-нащадок створюється з використанням декількох базових класів-батьків



**Успадкування** застосовується для таких взаємопов'язаних цілей:

- виключення з програми повторюваних фрагментів коду;
- спрощення модифікації програми;
- спрощення створення нових програм на основі існуючих.

Крім того, успадкування є єдиною можливістю використовувати об'єкти, вихідний код яких недоступний, але в які потрібно внести зміни.

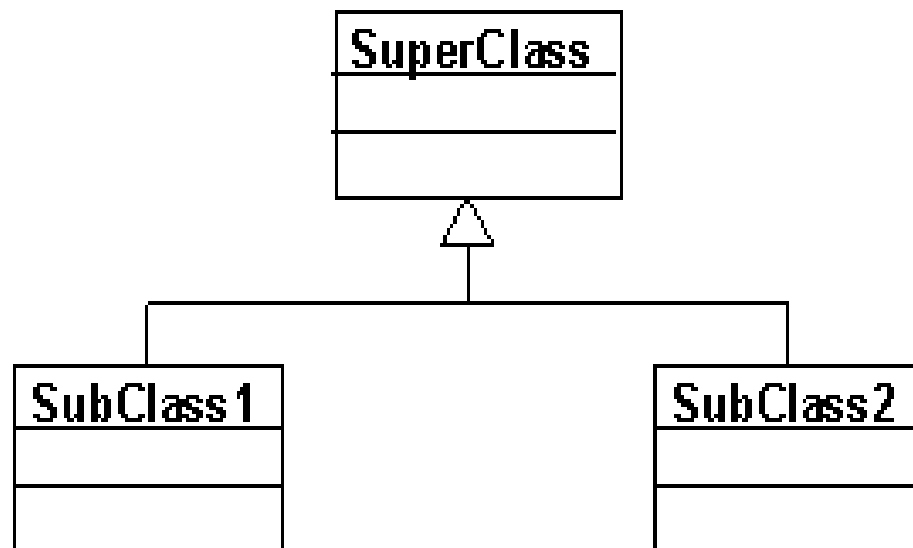
Успадкування встановлює між класами **відношення «є»**: похідний клас є частиною базового класу.

# Синтаксис успадкування

Ключи доступу

**class ім'я : [private | protected | public] базовий\_клас**  
**{ тіло класу };**

```
class A { ... };  
class B { ... };  
class C { ... };  
class D: public C  
{ ... };  
class B: protected A  
{ ... };
```



# Правило успадкування

Режим доступу до елемента в базовому класі	Ключ доступу при успадкуванні класу	Режим доступу до елемента в похідному класі
private		недоступний
protected	public	protected
public		public
private		недоступний
protected	protected	protected
public		protected
private		недоступний
protected	private	private
public		private

```
class A
{
public:
    int x;
protected:
    int y;
private:
    int z;
};
```

//B успадковує A

```
class B : public A
{
    // x - public
    // y - protected
    // z - не має
    доступу
};
```

//C успадковує A

```
class C : protected A
{
    // x - protected
    // y - protected
    // z не має доступу
};
```

//D успадковує A

```
class D : private A
{
    // x - private
    // y - private
    // z - не має
    доступу
};
```

Якщо базовий клас успадковується з ключем **private**, можна вибірково зробити деякі його елементи доступними в похідному класі:

```
class Base{  
    ...  
    public: void f();  
};  
class Derived : private Base{  
    ...  
    public: Base::void f();  
};
```

## Виклик конструкторів класів при успадкуванні

Конструктори не успадковуються, тому похідний клас повинен мати власні конструктори.

Порядок виклику конструкторів при створенні об'єкту похідного класу:

- Конструктор базового класу викликається автоматично. При цьому, якщо в похідному класі явний виклик конструктора базового класу відсутній, автоматично викликається конструктор базового класу за замовчуванням.
- Потім виконується відповідний **конструктор похідного класу**.
- При багаторівневому успадкуванні конструктори викликаються по черзі походження класів.

# Виклик конструкторів класів при успадкуванні<sup>15</sup>

```
#include <iostream>
using namespace std;
class base {
public:
base() { cout <<" Створення base-об'єкту.\n"; }
~base() { cout <<" Знищення base-об'єкту.\n"; }
};
class derived1 : public base {
public:
derived1() { cout <<"Створення derived1-
об'єкту.\n"; }
~derived1() { cout <<"Знищення derived1-
об'єкту.\n"; }
};
```

```
class derived2: public derived1 {  
public:  
    derived2() { cout <<"Створення derived2-  
    об'єкту\n"; }  
    ~derived2() { cout <<"Знищення derived2-  
    об'єкту.\n"; }  
};  
int main()  
{  
    derived2 ob;  
    return 0;  
}
```

```
Створення base-об'єкту.  
Створення derived1-об'єкту.  
Створення derived2- об'єкту  
Знищення derived2- об'єкту.  
Знищення derived1- об'єкту.  
Знищення base- об'єкту.
```

```
Process returned 0 (0x0)   execution time : 0.049 s  
Press any key to continue.
```



# Передача параметрів конструкторам базового класу

17

Якщо необхідно передати параметри конструктору **базового класу**, використовується розширена форма оголошення конструктора **похідного класу**, в якій передбачена можливість передачі аргументів одному чи декільком конструкторам базового класу.

```
Конструктор_похідного_класу(список_аргументів)
:конструктор_базового_класу(список_аргументів)
{
тіло конструктора похідного класу
}
```

**!** Якщо базовий клас містить тільки конструктори з параметрами, то похідний клас має викликати в своєму конструкторі один з конструкторів базового класу.

```
#include <iostream>
using namespace std;
class base
{protected:
    int i;
public:
    base (int x)
    { i = x;
      cout << "Створення base-об'єкту.\n";
    }
    ~base() { cout<<"Знищення base-
об'єкту.\n"; }
};
```

```
class derived: public base
```

```
{ int j;
```

```
public:
```

```
// Клас derived використовує параметр x, а параметр y
```

```
//конструктору класу base.
```

```
derived(int x, int y) : base(y)
```

```
{ j = x; cout << "Створення derived1-  
об'єкту.\n"; }
```

```
~derived()
```

```
{ cout <<"Знищення derived1- об'єкту.\n"; }
```

```
void show() { cout <<i<< " " << j << "\n"; }
```

```
};
```

```
int main()
```

```
{ derived ob(3, 4);
```

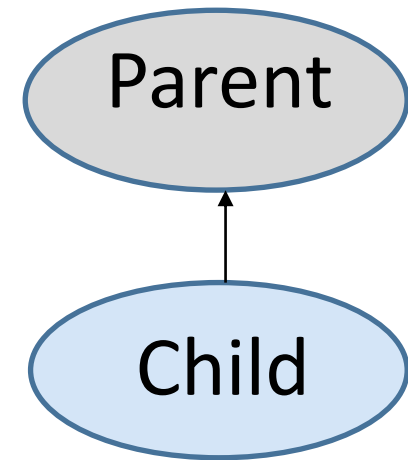
```
ob.show();
```

```
return 0;
```

```
}
```

```
Створення base-об'єкту.  
Створення derived1-об'єкту.  
4 3  
Знищення derived1- об'єкту.  
Знищення base- об'єкту.
```

```
#include <iostream>
#include <string>
using namespace std;
class Parent //базовий клас
{
protected: //захищені елементи
    int age;
    string name;
    int getAge() {return age;}
public:
    Parent(string N,int A)
    {name=N; age=A;
    }
    void identify() { cout << "I am a Parent: «
        <<name<<" I am: "<<age<<endl; }
};
```



```
class Child : public Parent
```

```
{private:
```

```
    string ch_name;
```

```
    int ch_age;
```

```
public:
```

```
    Child(string CN, int CA, string N, int A)
```

```
        : Parent(N,A)
```

```
    {ch_name=CN; ch_age=CA; }
```

//заміщення або розширення методів базового класу

```
void identify()
```

```
    {Parent::identify(); // спочатку виконається для
```

// класу Parent

```
    cout <<"And I am a Child: "<<ch_name<<
```

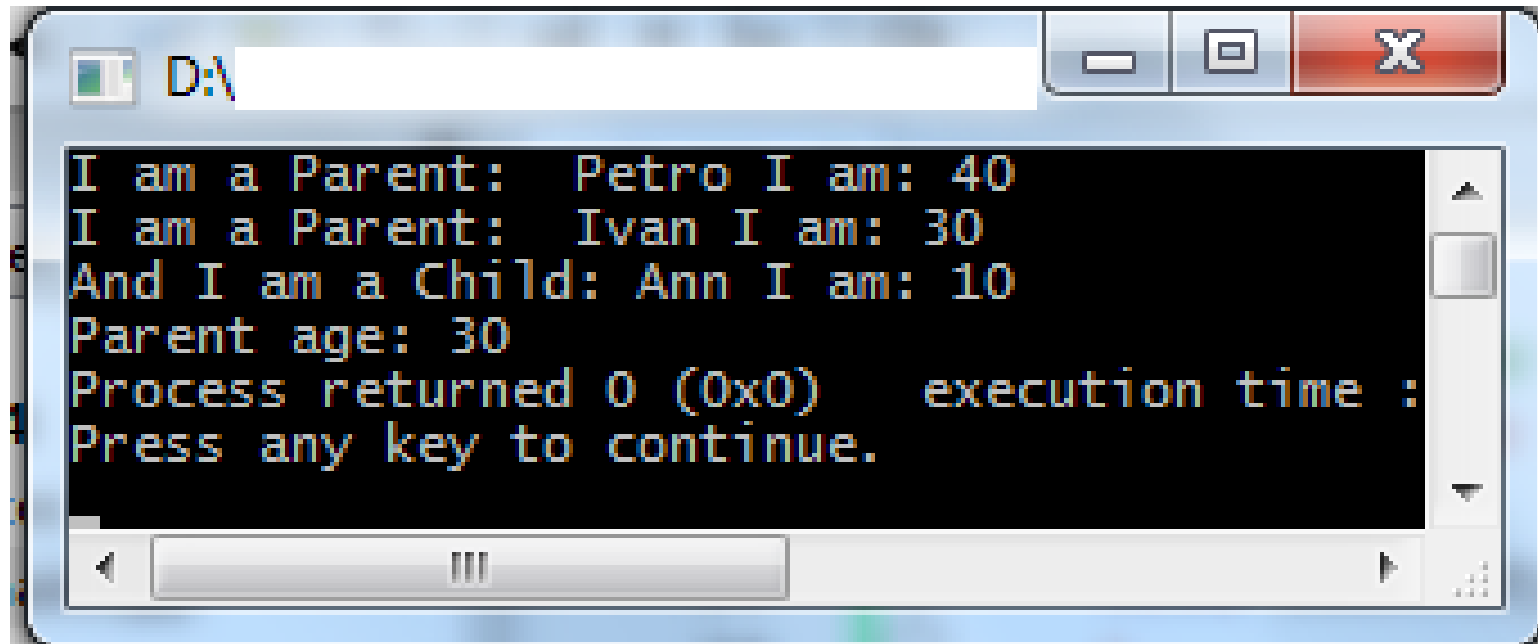
```
    "I am: "<<ch_age<<endl; // потім цей текст
```

```
    cout <<"Parent age: "
```

```
        <<getAge(); } // викликається для класу Parent
```

```
};
```

```
int main()  
{  
    Parent parent("Petro", 40);  
    parent.identify();  
    Child child("Ann", 10, "Ivan", 30);  
    child.identify();  
    return 0;  
}
```



```
I am a Parent: Petro I am: 40  
I am a Parent: Ivan I am: 30  
And I am a Child: Ann I am: 10  
Parent age: 30  
Process returned 0 (0x0) execution time :  
Press any key to continue.
```

```
class X {  
protected:  
    int x;  
public:  
    X(int i) // Конструктор X  
    {x=i;   cout<< "Constructor X \n";}  
    ~X() //Деструктор X  
    {cout<< "Destructor X \n"; }  
    void show ()  
    {cout<< "  x =" << x << "\n";}  
};  
  
class Y {  
protected:  
    int y;  
public:  
    Y (int j) // Конструктор Y  
    {y=j;   cout<< "Constructor Y \n"; }  
};
```

Базовий клас X



Базовий клас Y



Похідний клас Z

```
~Y() //Деструктор Y
```

```
{cout<< "Destructor Y \n";}
```

```
void show ()
```

```
{cout<< " y = "<< y << "\n";}
```

```
};
```

```
//похідний клас від X і Y
```

```
class Z: public X, public Y
```

```
{protected:
```

```
int z;
```

```
public:
```

```
Z (int i, int j):X(i), Y( j)
```

```
// Конструктор класу Z передає параметри конструкторам X і Y
```

```
{cout<< "Constructor Z \n";}
```

```
~Z() //Деструктор Z
```

```
{cout<< "Destructor Z \n";}
```

```
void show ()
```

```
{cout<< x*y<< " ="<< x << "*" <<y<< "; \n";}
```

```
};
```



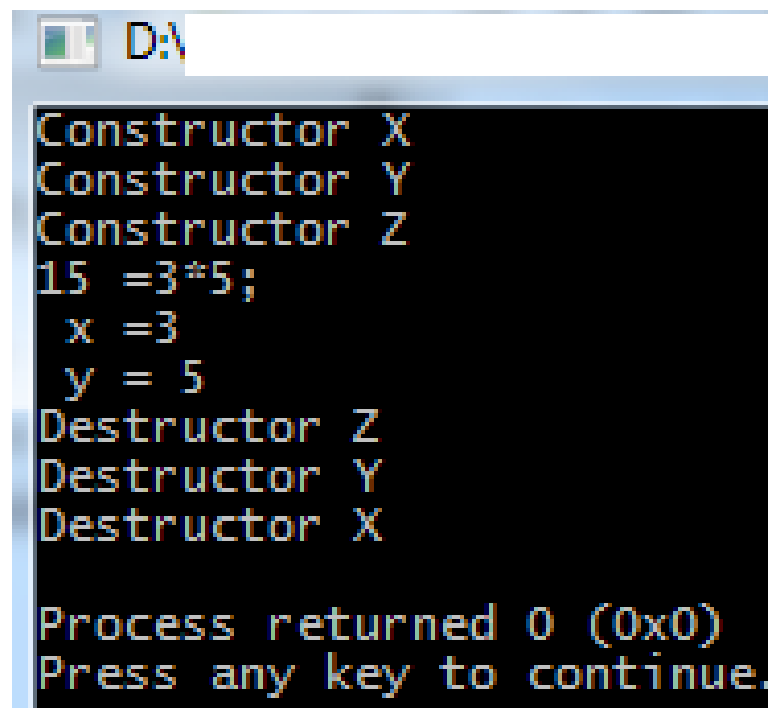
```

int main()
{
Z zobj (3, 5); //Створення об'єкта класу Z та передача
//параметрів конструкторам класів X і Y
zobj.show(); //Виклик методу show() класу Z
zobj.X::show(); //Виклик методу show() класу X
zobj.Y::show(); //Виклик методу show() класу Y
return 0;
}

```

## Увага!

Успадкування надає безліч переваг, але має бути ретельно спроектовано щоб уникнути проблем, можливість для яких воно відкриває.



```

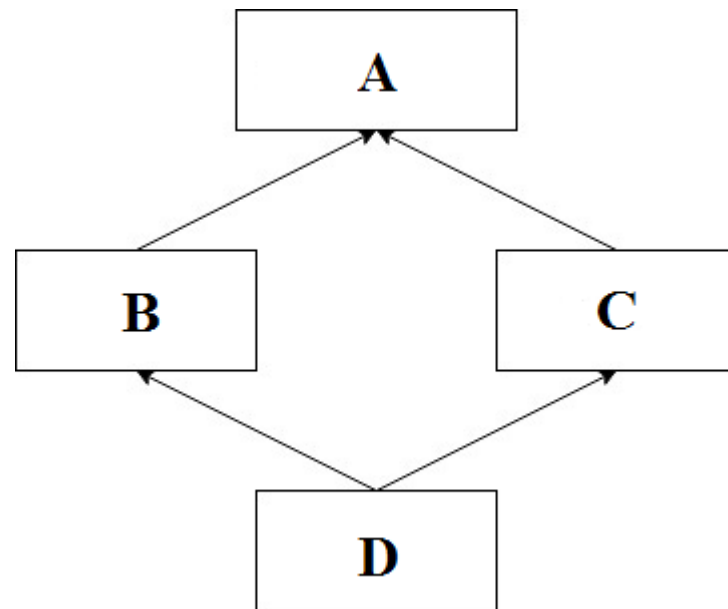
DA
Constructor X
Constructor Y
Constructor Z
15 =3*5;
  x =3
  y = 5
Destructor Z
Destructor Y
Destructor X

Process returned 0 (0x0)
Press any key to continue.

```

# Проблеми, які можуть виникнути при множинному успадкуванні

- Виникнення неоднозначності, коли кілька батьківських класів мають метод з одним і тим же ім'ям, а в дочірньому класі цей метод не перевизначений.



- «Проблема ромба»

# Успадкування деструкторів

- Деструктори не успадковуються.
- Якщо деструктор в похідному класі не описаний, він формується **автоматично** і викликає деструктори всіх базових класів.
- Не потрібно явно викликати деструктори базових класів, це буде зроблено автоматично.
- Для ієрархії, що складається з декількох рівнів, деструктори викликаються в порядку, **строого зворотному виклику конструкторів**: спочатку викликається деструктор класу, потім - деструктори елементів класу, а потім деструктори базового класу.

Дякую за увагу!