**Лекція 23. Ієрархія класів.**

**Ієрархія класів.**

Тему ієрархії класів будемо розглядати з прикладами з [9], де використовується система позначень Буча. Коротко познайомимся з нею.

**Система позначень Буча**.

Діаграми класів за Бучем представляються у вигляді "хмарки", а відношення у вигляді ліній з різними закінченнями (див. рис.1). Відношення визначаються так:

* + - асоціація позначає деякий зв’язок між класами,
    - успадкування позначає зв’язок загальне–частинне,
    - агрегація позначає зв’язок ціле–частина,
    - використання встановлює наявність зв‘язку між екземплярами класів.

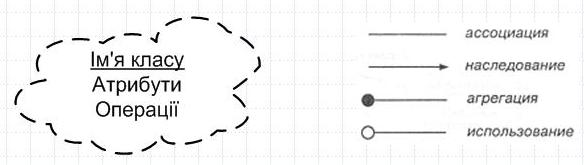
 

Рисунок 1. Діаграми класів за Бучем

На кінцях стрілок може встановлюватися потужність (кратність, множинність) відношення:

**1** – один зв’язок

**N** – необмежене число (0 або більше)

**0..N** - Нуль або більше

**1..N** - Один або більше

**0..1** - Нуль або один

**3..7** - Заданий інтервал

**1..3, 7** - Заданий інтервал або точне число

Режими доступу визначаються так: **|** - protected, **|| -** private

**Визначення ієрархії.**

***Ієрархія*** – це ранжована та упорядкована система абстракцій, тобто впорядкування абстракцій та розміщення їх по рівнях. Основними видами ієрархічних структур стосовно до складних систем є структура класів (ієрархія за номенклатурою) та структура об’єктів (ієрархія за складом). Принцип абстрагування, обмеження доступу та ієрархії не мають між собою строгої узгодженості, зокрема, якщо абстрагування полягає у встановлення жорстких меж, що захищають стан і функції об’єкту , то принцип наслідування вимагає відкрити доступ і до стану, і до функцій для майбутніх похідних об’єктів. Для будь-якого класу може існувати два види об’єктів; 1) об’єкти, які використовують операції даного класу для доступу до його елементів; 2) об’єкти, отримані за допомогою наслідування даного класу. Можна визначити три способи порушення механізму обмеження доступу через механізм наслідування: 1) похідний клас може отримати доступ до даних свого базового класу, 2) здійснити виклик захищеної функції базового класу та 3) звернутися напряму до базового класу.

**Ієрархія класів**

Наслідування (успадкування) — це ієрархія “узагальнення-спеціалізація”. Об’єкти різних класів і самі класи можуть перебувати у відношенні успадкування, за якого формується ієрархія об’єктів, що відповідає заздалегідь передбаченій ієрархії класів.

Ієрархія класів дозволяє визначати нові, похідні класи (або класи – нащадки) на основі вже існуючих базових (батьківських). Похідні класи “отримують спадок” — дані і методи своїх базових класів — і, крім того, можуть поповнюватись власними компонентами (даними і власними методами). Використання віртуальних функцій, визначених специфікатором virtual, дозволяє описати в базовому класі функції, які можна було б замінити у кожному наступному класі Ці питання детально розглядались при вивченні лекцій 12, 13.

Приклади ієрархії: одиночне спадкування — від одного класу; множинне спадкування — від багатьох (декількох класів). Відношення "узагальнення-спеціалізація" ми детально розглядали на попередніх лекціях.

На рис.2 наведено приклад одиночного успадкування - дані космічних апаратів (використана нотація Буча).

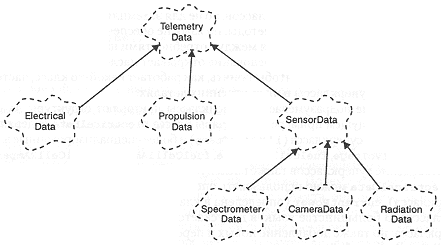


Рисунок 2. Приклад одиночного спадкування

Зокрема, **Cameradata** - це різновид класу **SensorData**, який в свою чергу є різновидом класу **TelemetryData**. Такий же тип ієрархії характерний для семантичних мереж, які часто використовуються фахівцями по розпізнаванню образів та штучному інтелекту для організації баз знаннь. Представлену на рис.2 ієрархічну структуру можна описати за таким зразком.

**class TelemetryData {  
public:  
 TelemetryData();  
 virtual ~TelemetryData();  
 virtual void transmit();  
 Time currentTime() const;  
protected:  
 int id;  
 Time timeStamp;  
};**

**class ElectricalData : public TelemetryData {  
public:  
 ElectricalData(float v1, float v2, float a1, float a2);  
 ~ElectricalData();  
 void transmit();  
 float currentPower() const;  
protected:  
 float fuelCell1Voltage, fuelCell2Voltage;   
 float fuelCell1Amperes, fuelCell2Amperes;  
};**

**Класифікація типів відношень між класами**.

Між класами можливі два типи відношень:

1. Відношення ***типу is-a*** (є, являється) при якому один клас є підвидом іншого класу. При такому співвідношенні один клас розширює (деталізує) можливості іншого класу. Розширення можливостей класу здійснюється завдяки використанню спадковості (ці питання розглядалися в лекціях15-16)*.*

2. Відношення, при якому існує взаємозв’язок між двома класами. Тут виділяють два підвиди взаємозв’язку між класами:

2.1. Відношення ***типу has-a*** (клас містить інший клас). У цьому випадку в класі оголошується один або декілька екземплярів іншого класу. При даному відношенні можливі два випадки взаємодії. Перший випадок, це коли об’єкт (екземпляр), що оголошений в класі, не є складовою частиною класу (агрегація) і його використання не впливає на функціональну роботу класу. Другий випадок, коли об’єкт, що оголошений в класі, є складовою частиною цього класу (композиція).

2.2. Відношення ***типу uses*** (клас “використовує інший клас). У цьому випадку клас містить програмний код іншого вкладеного класу, до якого він має доступ.

**Агрегація**

***Агрегація*** виражає ***відношення цілого та частини (типу is-a)*** .

***Агрегація*** — включення одного класу до іншого — відповідає *агрегації* між екземплярами. Агрегація як співвідношення "*ціле–частина*" є *спрямованою*. Вона ***не вимагає обов‘язкового фізичного включення*** (наприклад, акціонер *володіє* акціями, але не складається з них). Якщо (і тільки якщо) існує відношення "ціле–частина" між об‘єктами, класи повинні знаходитися у співвідношенні *агрегації*. Приклад агрегації наведено на рис.3, де *Heater* – нагрівач, *TemperatureController* – контролер температури.

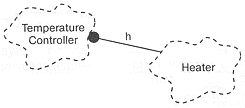


Рисунок 3. Приклад агрегації.

Представлений на рис.3 зв’язок можна представити таким чином.

**class TemperatureController {  
public:  
 TemperatureController(Location);  
 ~TemratureController();  
 void process(const TemperatureRamp&);  
 Minute schedule(const TemperatureRamp&) const;  
private:  
 Heater h;  
};**

В агрегації час життя об’єктів може бути різним. Початком часу існування будь-якого об’єкту є момент його створення (виділення області пам’яті), а закінченням –повернення виділеної області назад у систему.

**Композиція**

Іншим типом відношення цілого та частини є ***композиція,*** колиоб'єкти класів можна робити елементами даних інших класів (це відношення "***Has-A*** Relationship" чи "***Part Of*** Relationship"). Коли ми створюємо класи з членами, то ми, по суті створюємо складний об’єкт з більш простих частин, що і є композицією об’єкту. Саме тому структури та класи ще називають **складеними типами даних.**

Конструктори класів, об'єкти яких розміщені у зовнішньому класі, викликаються до виконання конструктору зовнішнього класу. Виклик конструкторів без параметрів здійснюється автоматично. Конструктори завжди викликаються у порядку, у якому об'єкти оголошені у класі. Деструктори викликаються у зворотному порядку.

Іноді для внутрішніх об'єктів необхідно викликати конструктори з параметрами. Це можна зробити з використанням *списку ініціалізації,* який розміщується перед тілом конструктору (див. лекцію 12). Наприклад:

**class** X

{

**public**:

X(**int** j) { ... }

. . .

};

**class** Y

{

X x;

**public**:

Y(**int** k) : x(k) { ... }

. . .

};

За допомогою списку ініціалізації можна також встановлювати початкові значення для елементів вбудованих типів:

**class** X

{

**int** k;

**double** d;

**public**:

X(**int** j, **double** h) : k(j), d(h) { ... }

. . .

};

Для реалізації композиції об’єкт і частина **повинні мати такі відношення**:

* Частина (член) є частиною об’єкта (класу).
* Частина (член) може належати тільки одному об’єкту (класу) за раз.
* Частина (член) існує, керована об’єктом (класом).
* Частина (член) не знає про існування об’єкту (класу).

Прикладом композиції є серце, яке є частиною тіла людини, яке не може бути частиною тіла ще однієї людини одночасно. У відношеннях композиції об’єкт несе відповідальність за існування частин. Частіше за все це означає, що частина створюється при створенні об’єкту та знищується при його знищенні. Це означає, що об’єкт керує часом життя частини таким чином, що користувач, який використовує об’єкт, не повинен приймати участь в цьому. Наприклад, при створенні тіла створюється і серце, коли тіло людини знищується, його серце також знищується. Частина не знає про існування цілого: серце працює цілодобово, не знаючи, що воно є частиною більш великої структури. Це називається односпрямованим відношенням, Оскільки тіло знає про серце, а серце про тіло — ні.

Частини в композиції можуть бути як сингулярними (єдиними в своєму роді), так и мультиплікативними (таких частин може бути декілька). Наприклад, в тілі людини є тільки одне серце (серце є сингулярним), але також 20 пальців (пальці є мультиплікативними і можуть бути реалізовані у вигляді масиву).

**Реалізація композицій**

Композиції є одними з самих простих типів відносин для реалізації в C++. Це звичайні структури або класи зі звичайними членами. Оскільки члени існують безпосередньо як частини структур/класів, то їхня тривалість життя напряму залежить від тривалості життя об’єктів цих структур/класів.

Композиції, в яких виконується динамічне виділення та звільнення пам'яті, можуть бути реалізовані з використанням ***покажчиків*** у вигляді членів цих структур або класів. В цьому випадку керування пам'яттю повністю накладається на композицію.

Якщо ви можете створити клас, використовуючи композицію, то ви повинні створювати клас, використовуючи композицію. ***Класи з реалізованою композицією є простими, гнучкими та надійними***.

**Варіації композиції**

Хоч у більшості композицій створення/вилучення частин виконується безпосередньо при створенні/вилученні самої композиції, є варіації композиції, де правила декілька змінені. Приклади.

Композиція може відкласти створення деяких зі своїх частин, доки вони не знадобляться. Наприклад, рядковий клас може не створювати динамічний масив символів, доки користувач не надасть дані, які цей клас має зберігати.

Композиція може використовувати часть, яка була надана їй як вхідні дані, а не створювати цю частину самостійно.

Композиція може делегувати знищення своїх частин іншому об’єкту (наприклад, процедурі збирання "сміття").

Ключовим моментом є те, що композиція повинна керувати своїми частинами самостійно, без втручання користувача композиції.

**Переваги композиції**

**1**: Кожен окремий клас можна зберігати як відносно простий/зрозумілий та сфокусуватися на виконанні однієї конкретної задачі. В такий спосіб писати класи легше і розуміти їх простіше.

**2**: Кожен підклас може бути автономним, що робить його багаторазовим, наприклад, для використання в іншій програмі.

**3**: Базовий/батьківський клас може покласти виконання більшої частини складної роботи на підкласи, а сам зосередитися на координації потоку даних між підкласами. Це допоможе знизити загальну складність батьківського об’єкту, оскільки батьківський об’єкт делегує виконання роботи своїм дочірнім/ підлеглим елементам, які вже знають, як виконувати ці завдання.

Хорошим правилом є те, що **кожен клас повинен виконувати одне конкретне завдання**. Таким завданням може бути зберігання, маніпулювання даними або координація підкласів. Те, як потрібно виконувати конкретні завдання залежить вже від кожного підкласу окремо.

**Різниця між композицією та агрегацією**

***В композиції****:*

* Використовуються звичайні змінні-класи
* Використовуються покажчики, якщо клас реалізує власне керування пам'яттю (виконується динамічне виділення/ звільнення пам’яті).
* Клас відповідальний за створення/вилучення своїх частин.

***В агрегації****:*

* Використовуються покажчики /посилання, які вказують/посилаються на частини поза класом.
* Клас не несе відповідальності за створення/вилучення своїх частин.

Ідеї композиції та агрегації не є взаємовиключними і можуть змішуватися в одному класі. Можна реалізувати клас, який відповідає за створення/вилучення тільки певних частин. Наприклад, клас Відділ може мати своє Ім’я та Робітника. Ім’я додається до класу через композицію та створювалось/вилучалось разом з об’єктами класу Відділ. А Робітник додається до класу Відділ через агрегацію та створювався/вилучався незалежно/окремо.

Хоч агрегації можуть бути дуже корисними, вони також містять потенційну загрозу, оскільки в агрегації автоматично не здійснюється звільнення пам'яті, яку можуть займати частини, то це повинен контролювати програміст. Якщо він забув виконати очищення (звільнити пам’ять), то тоді відбувається виток пам'яті.

##### Приклад агрегації для типу відношення has-a (композиції)

У випадку агрегації клас містить множини (один або декілька) об’єктів інших класів, які не є складовою частиною цього класу (не містять код, що доповнює роботу самого класу).

##### *Клас Figures містить екземпляри класів Triangle (трикутник) та Circle (коло).* У даному скороченому коді демонструється агрегація на прикладі класів Triangle, Circle, Figures. Клас Figures може містити різну кількість різних фігур (навіть 0). При можливості до класу Figures може бути додано масиви інших фігур, наприклад, Rectangle (прямокутник). У будь-якому випадку клас Figures буде повноцінним функціонально, отже, це є агрегація.

**// Клас трикутник**

**class Triangle**

**{**

**// Методи та поля класу Triangle**

**// ...**

**};**

**// Клас, що реалізує коло**

**class Circle**

**{**

**// Методи та поля класу Circle**

**// ...**

**};**

**// Клас, що реалізує різні геометричні фігури.**

**// Використовується тип відношення - агрегація.**

**class Figures**

**{**

**Triangle tr[10]; // масив трикутників**

**unsigned int n\_tr; // к-сть трикутників у масиві tr**

**Circle cr[10]; // масив кіл**

**unsigned int n\_cr; // к-сть кіл у масиві cr**

**// Інші поля та методи класу**

**// ...**

**};**

##### Приклад композиції для типу відношення has-a (агрегація)

##### *Клас Car (автомобіль), який містить екземпляри класів Vehicle (Двигун) та Wheel (Колесо).* У прикладі продемонстровано композицію для класів Vehicle, Wheel, Car. В автомобіль (Car) входять двигун (клас Vehicle) та колесо (Wheel), які є його складовою частиною – це є композиція (об’єднання).

// Клас Двигун

class Vehicle

{

// Поля та методи класу

// ...

};

// Клас Колесо

class Wheel

{

// Поля та методи класу

// ...

};

// Клас Автомобіль - містить обов'язкові елементи,

// які є складовими Автомобіля

class Car

{

// Екземпляри обов'язкових класів,

// що є частиною даного класу - це є композиція

private:

Vehicle veh; // автомобіль містить двигун (обов'язково)

Wheel whl[4]; // колеса - 4 штуки (обов'язково)

// Поля та методи класу

// ...

};

**Використання - відношення** **типу *uses***

***Використання*** (***uses*** )— відношення між класами, якщо клас є частиною сигнатури функції-члена іншого класу (параметром).

При розробці проектів важливим є визначення того, як клас використовує інші класи. Різні залежності між класами з’являються неявно у мовах програмування.

Згідно із засновником мови C++ Б.Страуструпом [10] між двома класами A, B можна встановити наступні способи використання класом A класу B:

* Клас A використовує ім’я класу B.
* Клас A використовує клас B.
  + Клас A викликає функцію-член класу B.
  + Клас A читає поле даних класу B.
* Клас A записує поле даних класу B.
* Клас A створює об’єкт типу B.
  + Клас A виділяє пам’ять для автоматичних об’єктів типу B.
  + Клас A створює об’єкт типу B з допомогою оператора new.
* Клас A отримує розміри класу B.

Приклади реалізації відношення ***uses*** надано після лекції.

Приклад відношення використання представлено на рис.4, де ***TemperatureController*** – контролер температури, ***TemperatureRamp*** – визначає функцію часу від температури .

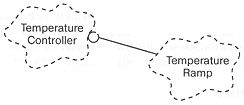


Рисунок 4. Приклад відношення використання представлено

Цей приклад використання можна проілюструвати таким кодом**.**

**class TemperatureController {  
public:  
 TemperatureController(Location);  
 ~TemratureController();  
 void process(const TemperatureRamp&);  
 Minute schedule(const TemperatureRamp&) const;  
private:  
 Heater h;  
};**

**Вкладені класи та локальні класи**

У випадку, коли при оголошенні одного класу в середині можуть бути оголошені інші класи, такі класи називають вкладеними (*nested classes*), а клас, що їх об’єднує – клас з об’єктною ідентичністю (*object identity'*). Приклад.

**#include<iostream.h>**

**class C**

**{**

**class Nested**

**{**

**int who;**

**public:**

**Nested(int a);**

**~Nested(void);**

**};**

**public:**

**C(int b)**

**{**

**Nested (b\*b);**

**cout<<"Constructor C\n";**

**}**

**~C(void)**

**{**

**cout<<"Destructor C\n";**

**}**

**};**

**C::Nested::Nested(int a)**

**{**

**who=a;**

**cout<<"Constructor of Nested class \n";**

**}**

**C::Nested::~Nested(void)**

**{**

**cout<<"Destructor of Nested class\n";**

**}**

**int main(int argc, char\* argv[])**

**{**

**C object(3);**

**return 0;**

**}**

**В результаті роботи отримаємо:**

**Constructor of Nested class**

**Destructor of Nested class**

**Constructor C**

**Destructor C**

Потрібно зробити декілька зауважень

* У класу з об’єктною ідентичністю нема прав доступу до закритих членів вкладеного.
* Друзі вкладеного класу автоматично не вважаються друзями класу з об’єктною ідентичністю. Якщо вони також повинні бути друзями класу з об’єктною ідентичністю, це повинно бути оголошено окремо.
* Та навпаки, оскільки клас з об’єктною ідентичністю автоматично не вважається другом вкладеного класу, друзі класу з об’єктною ідентичністю не будуть вважатися друзями вкладеного класу.

Клас може бути оголошений в середині блоку, наприклад, в середині [визначення](https://intellect.icu/obespechenie-kachestva-osnovnye-ponyatiya-i-opredeleniya-6106) функції. Такий клас називається ***локальним***. [Локалізація](https://intellect.icu/lokalizatsiya-1887) [класу](https://intellect.icu/1-obektno-orientirovannyj-podkhod-k-razrabotke-po-1-1-ponyatiya-obekta-i-klassa-3307) передбачає недоступність його компонентів поза області визначення класу (поза блоком).

Локальний клас не може мати статичних даних, оскільки компоненти локального класу не можуть бути  визначені поза текстом класу. В середині  локального класу  дозволено використовувати із охоплюючей його області тільки імена типів, статичні (static) змінні, зовнішні(extern) змінні, зовнішні функції та елементи перелічень. В  локальних класах їх компонентні функції можуть бути лише inline.

В середині класу дозволяється визначати типи, відповідно, один клас може бути описаний всередині іншого, тобто бути вкладеним. Вкладений клас є локальним для класу, в рамках якого він описаний, і на нього поширюються визначені вище правила використання локального класу. Вкладений клас не має ніякого спеціального права доступу до членів охоплюєчого класу, тобто ви може звертатися до них тільки через об’єкт типу цього класу (також охоплюючий клас не має будь-яких спеціальних прав доступу к вкладеного класу).

  Вкладені класи широко застосовується в технології об’єктів *COM* (*Component Object Model*), основна ідея якої полягає у відокремленні інтерфейсу класу від його реалізації.

Приклад локального класу.

**#include<iostream.h>**

**void f(void);**

**int main(void)**

**{**

**// Виклик функції, що містить локальний клас**

**f();**

**return 0;**

**}**

**void f(void)**

**{**

**class Local**

**{**

**int who;**

**public:**

**Local(int a)**

**{**

**who=a;**

**cout<<"Consr Local"**

**<<who<<"\n";**

**}**

**~Local(void)**

**{**

**cout<<"Destr Local"**

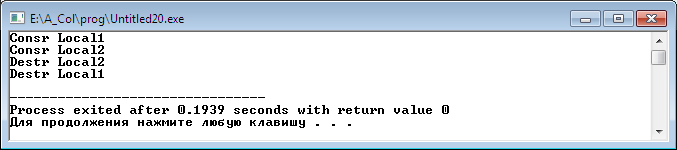
**<<who<<"\n";**

**}**

**} loc\_ob(1), loc\_ob1(2);**

**}**

Результат



Безпосередньо у тілі функції створюються два об’єкти локального класу *loc\_ob* та *loc\_ob1*. Після того, як функція завершує своє існування, обидва об’єкти автоматично знищуються, про що свідчить виклик деструкторів, визначених у протоколі оголошення класу. Використання локальних класів може застосовуватися у випадках, коли об’єкти мають обмежений час існування у програмі, або у тих випадках, коли вимагається гарантоване вивільнення пам’яті під великі масиви тимчасових об’єктів, які вже виконали покладену на них конкретну місію.

*Для самостійного вивчення*: Поглибити матеріал лекції за наданою літературою. Вивчення лекційного матеріалу та додаткових джерел. Розгляд запитань і виконання завдань для самостійної роботи, запропонованих на лекції.

*Контрольні запитання для самоперевірки*.

1. Дайте визначення ієрархії.
2. Які типи відношень можливі між класами?
3. Дайте визначення агрегації та композиції. В чому між ними різниця?
4. В чому полягають переваги композиції?
5. Чи може агрегація міститися в композиції та навпаки?

*Контрольні запитання для надання письмових відповідей*.

1. Як Ви вважаєте який тип відношень реалізується у вкладених класах?
2. Як Ви вважаєте з чим пов’язані обмеження щодо локальних класів?

*Література*

1. Джейс Либерти Освой самостоятельно С++ за 21 день: 3-е изд. пер. с англ.: Уч. пос. – М.: Издательский дом „Вильямс”, 2001. – 816 с.: ил..
2. Павловская Т.А. С/С++. Программирование на языке высокого уровня. СПб.: Питер, 2003. – 461 с. URL:  <http://www.ph4s.ru/bookprogramir_1.html>
3. Липпман С. Б., Лажойе Ж. Язык программирования С++: Вводный курс. — М.: ДМК, 2001. URL: <http://www.insycom.ru/html/metodmat/inf/Lipman.pdf>
4. Дейтел Х., Дейтел П. Основы программирования на С++. – М.: Бином, 1999. – 1024 с. URL:  <http://ijevanlib.ysu.am/wp-content/uploads/2018/03/deytel.pdf>
5. Бадд Т. Объектно-ориентированное программирование в действии. [2-е изд.] – СПб.: Изд-во "Питер". 1997.  URL: <http://khizha.dp.ua/library/Timothy_Budd_-_Introduction_to_OOP_(ru).pdf>.
6. *Герб Саттер,Андрей Александреску* "Стандарты программирования на С++ ", [*Вильямс, 2005*](http://www.williamspublishing.com/)*; 304 с.*
7. *Скотт Мейерс.* Эффективное использование C++. 50 рекомендаций по улучшению ваших программ и проектов. *"ДМК", 2000; 240 с.*
8. *Скотт Мейерс.* Наиболее эффективное использование C++. 35 новых рекомендаций по улучшению ваших программ и проектов.*"ДМК",2000;304 с.*
9. Г.Буч. Об‘єктно-орієнтоване проектування з прикладами застосування. – К.: Видавничий центр "Академія". 2002. ­ 499 с.
10. Страуструп Б. Язык программирования С++· Киев: "ДиаСофт", 1993. - 256 с.

**Додаткова інформація.**

##### Приклад. Клас A використовує ім’я класу B

using System;

using static System.Console;

namespace ConsoleApp1

{

// Взаємодія між класами типу uses

// Клас A використовує ім'я класу B

// Клас B - базовий клас

class B

{

// Прихована внутрішня змінна

private int b = 25;

// Віртуальний метод

public virtual void Print(B refB)

{

WriteLine("Method B.Print():");

WriteLine("b = {0}", refB.b);

}

}

// Клас A - похідний від класу B

class A : B

{

// Прихована внутрішня змінна

private int a = 15;

// У параметрі методу класу A використовується

// ім'я класу B, як базового для реалізації поліморфізму

public override void Print(B refB)

{

WriteLine("Method A.Print():");

base.Print(refB);

WriteLine("a = {0}", a);

}

}

class Program

{

static void Main(string[] args)

{

// 1. Створити екземпляр класу A

A objA = new A();

B objB = new B();

// 2. Оголосити посилання на базовий клас B

B refB;

// 3. Направити посилання refB на екземпляр класу A

refB = objA;

// 4. Викликати віртуальний метод Print() через посилання refB

refB.Print(refB); // викликається objA.Print(refB)

// 5. Перенаправити посилання на базовий клас B

refB = objB;

// 6. Викликати віртуальний метод Print() через посилання refB

refB.Print(refB); // викликається objB.Print(refB)

}

}

}

Результат виконання програми

Method A.Print():

Method B.Print():

b = 25

a = 15

Method B.Print():

b = 25

##### Приклад. Клас A використовує клас B

У прикладі демонструється декілька форм взаємодії між класами, що відноситься до типу uses.

using System;

using static System.Console;

namespace ConsoleApp1

{

// Взаємодія між класами типу uses

// Клас A використовує клас B

class B

{

public const double Pi2 = 6.28;

public double Pi = 3.14;

static public void MethodB()

{

WriteLine("B.MethodB()");

}

}

class A

{

public void MethodA()

{

// 1. Виклик функції-члена класу B з класу A

B.MethodB();

// 2. Звертання до поля даних класу B з класу A

WriteLine(B.Pi2);

// 3. Запис поля даних класу B

B objB = new B(); // створити екземпляр класу B

objB.Pi = 3.141592; // записати поле даних

}

}

class Program

{

static void Main(string[] args)

{

// 1. Створити екземпляр класу A

A objA = new A();

// 2. Викликати метод MethodA() класу A

objA.MethodA();

}

}

}

Результат роботи програми

B.MethodB()

6.28

##### Приклад. Клас A динамічно створює об’єкт класу B

Даний приклад більш актуальний для мови C++, оскільки у цій мові об’єкт класу в методі можна створити двома способами:

// Мова C++. Спосіб 1

B objB;

// Мова C++. Спосіб 2 - з допомогою оператора new

B\* pB = new B();

У мові C# усі об’єкти (екземпляри) класу створюються динамічно з допомогою оператора new. Це пов’язано з тим, що класи відносяться до типів-посилання а не до типів-значення. У випадку з класами типи-посилання містять посилання на екземпляр класу. Саме посилання розміщується в стеку, а екземпляр розміщується в кучі (heap). Тому, для розміщення екземпляру потрібно виділити пам’ять динамічно.

Текст демонстраційної програми наступний.

using System;

using static System.Console;

namespace ConsoleApp1

{

// Взаємодія між класами типу uses

// Клас A динамічно створює об'єкт класу B

class B

{

// Внутрішнє поле

public int d;

// Метод Show()

public void Show()

{

WriteLine("d = {0}", d);

}

}

class A

{

// Метод, що динамічно створює об'єкт класу B

public void Demo()

{

WriteLine("Method A.Demo()");

// 1. Створити екземпляр класу B динамічно

WriteLine("Creating instance objB.");

B objB = new B();

WriteLine("OK!");

// 2. Заповнити значенням об'єкт класу B

WriteLine("Fill with values objB.d.");

objB.d = 330;

WriteLine("OK!");

}

}

class Program

{

static void Main(string[] args)

{

// 1. Створити екземпляр класу A

A objA = new A();

// 2. Викликати метод Demo() класу A

objA.Demo();

}

}

}

Результат роботи програми

Method A.Demo()

Creating instance objB.

OK!

Fill with values objB.d.

OK!

##### Приклад. Клас A отримує розміри класу B

У прикладі демонструється визначення розмірів екземпляру класу з допомогою механізму серіалізації. (У найбільш загальному випадку, серіалізація це властивість об’єкту зберігати свій стан у файловому потоці та (за вимогою) відновлювати його).

using System;

using static System.Console;

// Необхідно для використання можливостей класу Stream

using System.IO;

// Необхідно для використання класу BinaryFormatter

using System.Runtime.Serialization.Formatters.Binary;

namespace ConsoleApp1

{

// Вихідний клас B, який містить одне поле та один метод

// Перед класом вказується, що клас буде зберігатись в пам'яті

// (клас, який серіалізується)

[Serializable]

class B

{

public double x;

void ShowX()

{

WriteLine("x = {0}", x);

}

}

// Клас A, який містить метод,

// що визначає розмір екземпляру класу B

class A

{

// У методі створюється екземпляр класу B,

// та визначається його розмір

public void DemoSizeObjB()

{

// 1. Оголосити внутрішні змінні

long size = 0; // розмір екземпляру класу B

B objB = new B(); // створити екземпляр

// 2. Використання потоку в пам'яті у двійковому вигляді

// 2.1. Створити потік в пам'яті

using (Stream s = new MemoryStream())

{

// 2.2. Створити екземпляр бінарного (двійкового)

//      формату серіалізації

BinaryFormatter bf = new BinaryFormatter();

// 2.3. Записати екземпляр objB у двійковому форматі

//      в пам'ять

bf.Serialize(s, objB);

// 2.4. Визначити розмір пам'яті,

//       в який був записаний екземпляр

size = s.Length;

// 2.5. Вивести розмір на екран

WriteLine("size = {0}", size);

}

}

}

class Program

{

static void Main(string[] args)

{

// 1. Створити екземпляр класу A

A objA = new A();

// 2. Викликати метод DemoSizeObjB()

objA.DemoSizeObjB();

}

}

}

Результат роботи програми

size = 129