



# ФУНДАМЕНТАЛЬНІ КОНЦЕПЦІЇ ООП. ВЗАЄМОДІЯ КЛАСІВ

Лекція 04  
Об'єктно-орієнтоване програмування

# Питання лекції

---

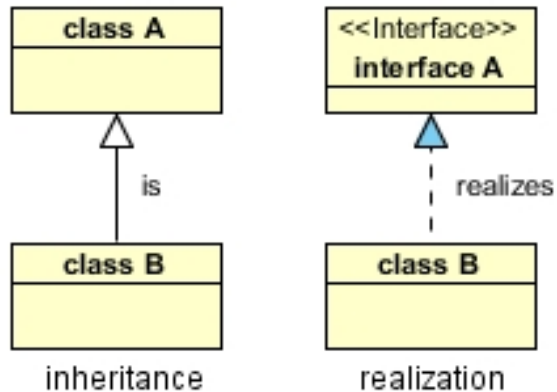
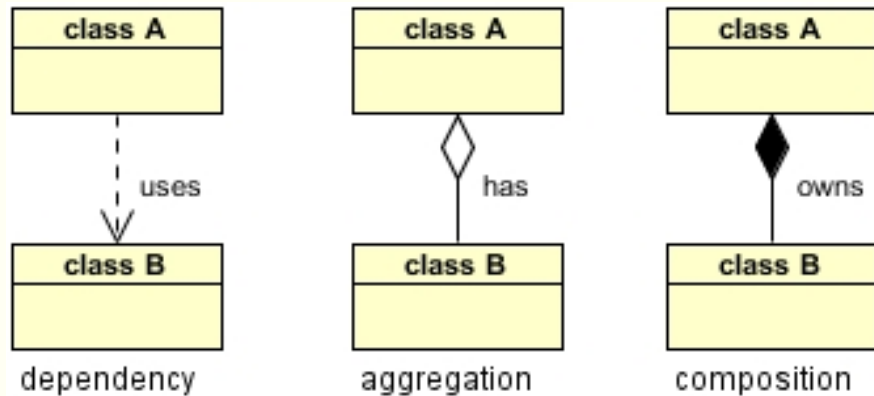
- Види взаємодії класів.
- Наслідування на прикладі структурованої обробки винятків.
- Спеціалізований (ad-hoc) поліморфізм.
- Поліморфізм підтипів.
- Параметричний поліморфізм. Узагальнені типи даних.



# ВИДИ ВЗАЄМОДІЇ КЛАСІВ

Питання 4.1.

# Види відношень між класами в UML. Залежність



- **Залежність (Dependency – посилається, використовує)**
  - Передбачає відсутність концептуального зв'язку між 2 об'єктами.
  - Наприклад, об'єкт EnrollmentService посилається на об'єкти Student та Course (через параметри методів чи вихідні типи)
  - ```
public class EnrollmentService {  
    public void enroll(Student s, Course c){ }  
}
```
- **Залежність** – направлений (directed) зв'язок, який позначає вимогу, потребу чи залежність одного іменованого елемента UML від інших іменованих елементів **для специфікації чи реалізації**.
  - Часто називають відношенням «постачальник-клієнт»
  - Зміни в постачальнику можуть вплинути на клієнта.

# Види відношень між класами в UML. Залежність

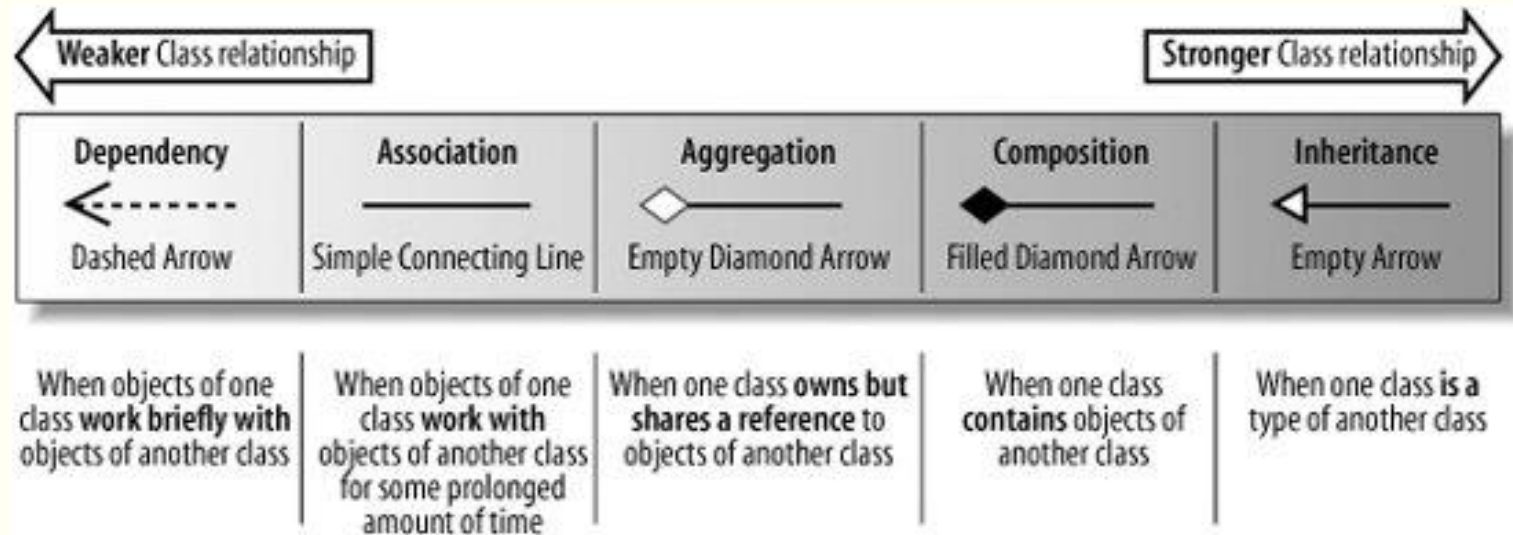
```
public class Customer
{
    public Guid CustomerId { get; set; }
    public String CustomerName { get; set; }

    // Other Customer related functions & properties
}

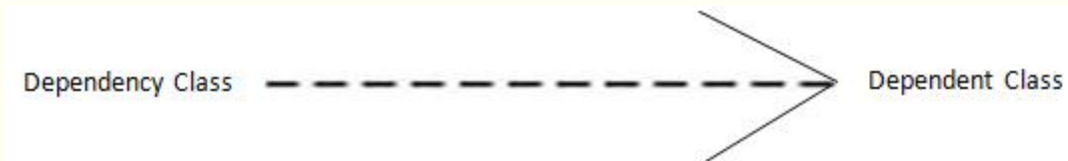
public class Order
{
    public Int32 OrderId { get; set; }
    public Guid OrderCustomerId { get; set; }
    public DateTime OrderDateTime { get; set; }

    // Other Order functions & properties

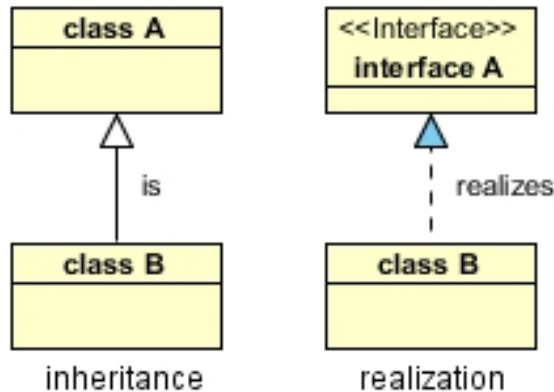
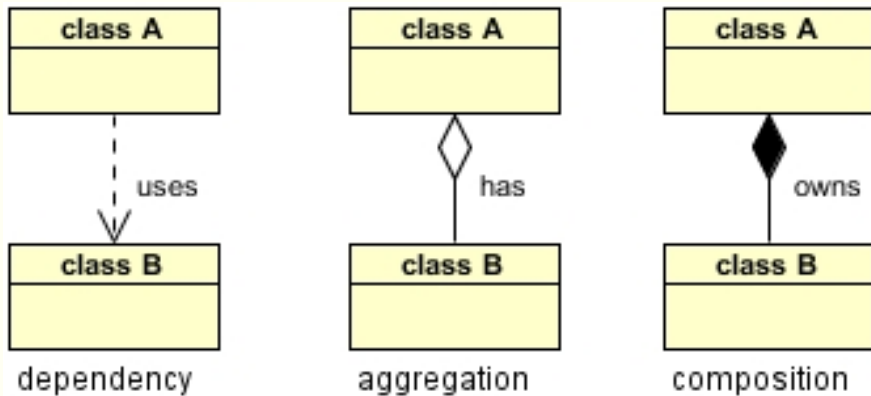
    public Order(Customer customer)
    {
        // Save order with CustomerId
        this.OrderCustomerId = customer.CustomerId;
    }
}
```



- У прикладі маємо CustomerId в якості Guid.
  - Клас Order використовує екземпляр customer.
  - Якщо змінити тип даних CustomerId з Guid на int, це вплине на клас Order.



# Види відношень між класами в UML



## ■ Асоціація (Association – має, has-a)

- Майже завжди передбачає зв'язок між об'єктами. Наприклад, об'єкт Order має об'єкт Customer
- ```
public class Order {  
    private Customer customer;  
}
```

## ■ Агрегація (Aggregation – ціле має частину, has-a + whole-part)

- Спеціальний вид асоціації, в якому присутній зв'язок «ціле-частина» між 2 об'єктами, проте вони можуть існувати окремо.
- ```
public class Playlist {  
    private List<Song> songs;  
}
```
- Відрізнити від асоціації часто є проблемою.

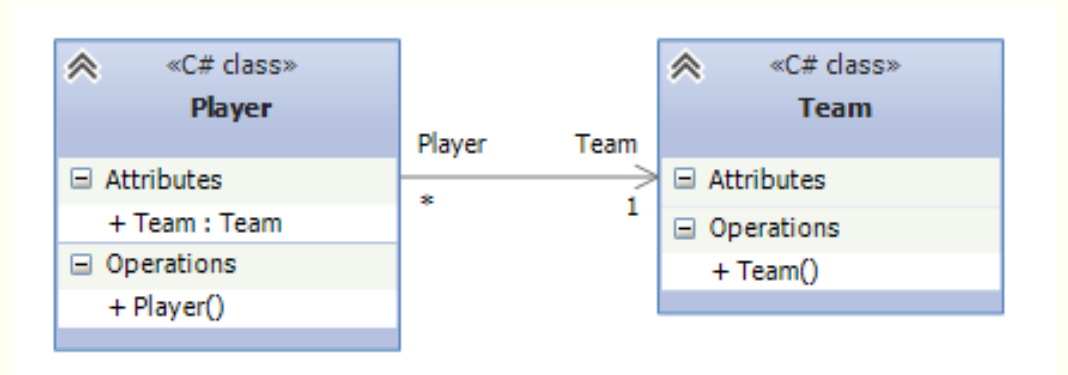
## ■ Композиція (Composition – ціле володіє частиною, has-a + whole-part + ownership)

- Спеціальний вид агрегації. Наприклад, квартира складається з окремих кімнат, проте кімнати не існують без квартири. При видаленні квартири всі пов'язані кімнати теж видаляються.
- ```
public class Apartment{  
    private Room bedroom;  
    public Apartment() {  
        bedroom = new Room();  
    }  
}
```

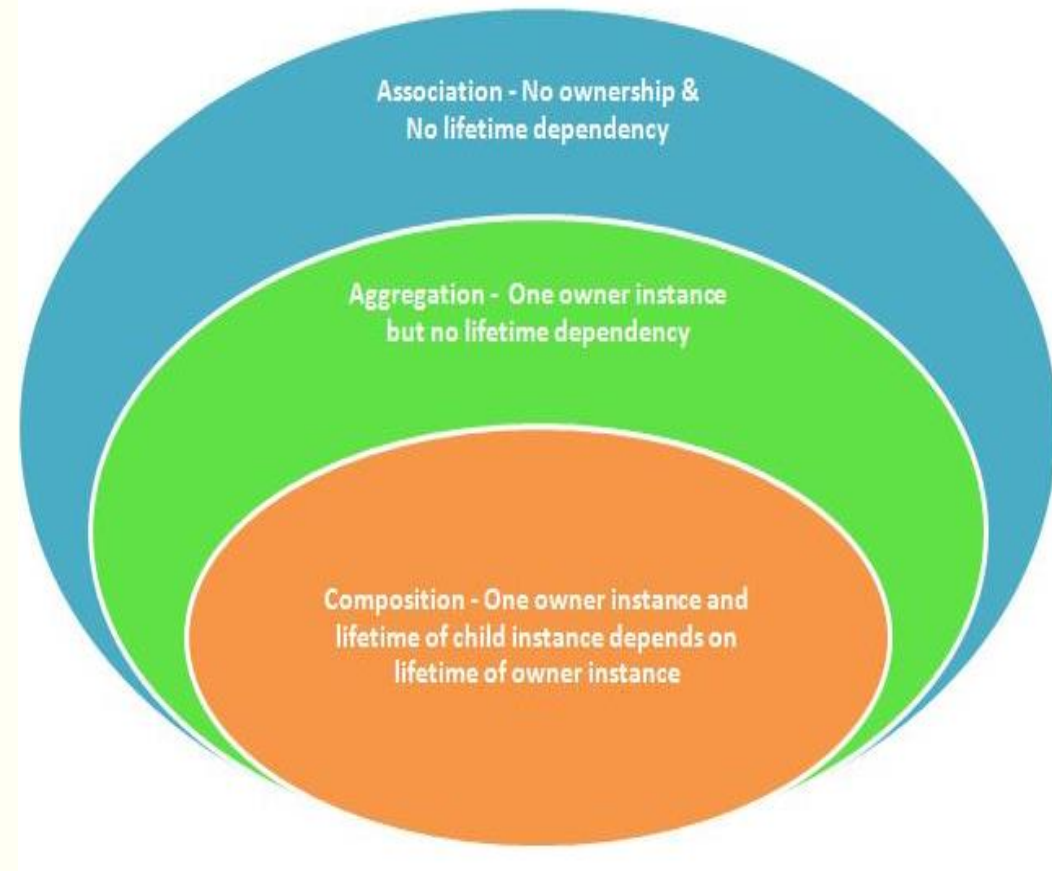
# Залежність vs асоціація

- Загалом асоціація представляє зв'язок, на зразок поля класу.
  - Зв'язок завжди тут, наприклад, можна встановити замовлення (order) за його замовником (customer).
  - При моделюванні з використанням інтерфейсів це не обов'язково буде поле: може бути метод, який повертатиме замовника замовлення.
- Асоціації також передбачають залежності: якщо існує асоціація між 2 класами, також існує і залежність.
  - Узагальнення теж.
- Ще один приклад асоціації: гравець грає в певній команді.:

```
class Team
{
}
class Player
{
    public Team Team { get; set; }
}
```



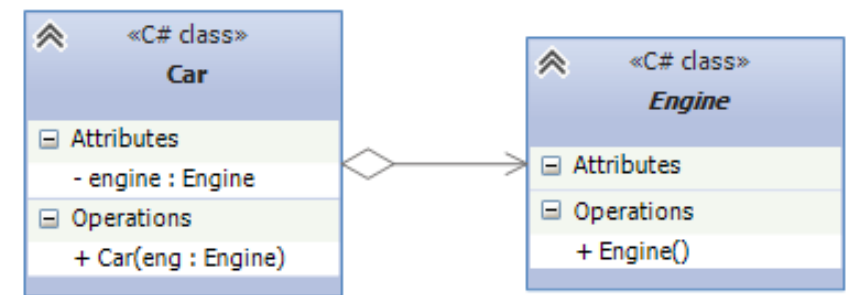
# Асоціація, агрегація та композиція



- При агрегації реалізується слабкий зв'язок.
  - Тут об'єкти Car і Engine будуть рівноправними.
  - У конструктор Car передається посилання на вже наявний об'єкт Engine.
  - Як правило, визначається посилання не на конкретний клас, а на абстрактний клас або інтерфейс, що збільшує гнучкість програми.

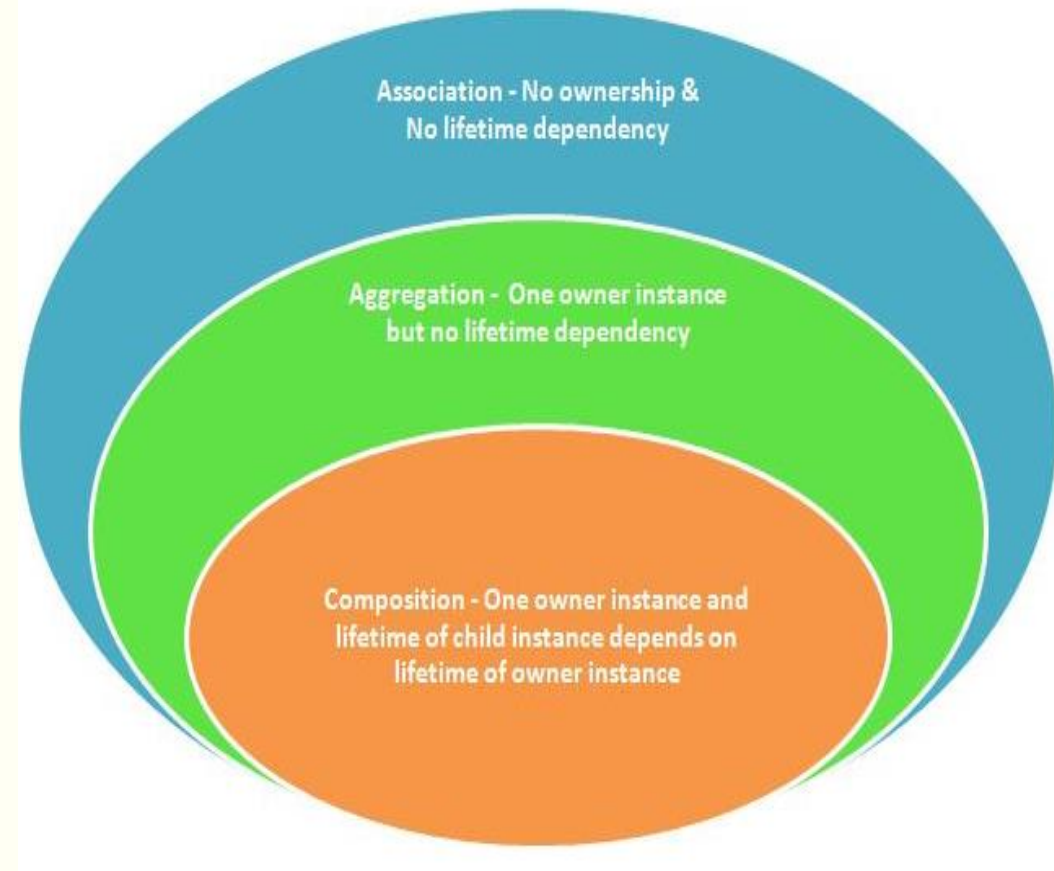
```
public abstract class Engine  
{ }
```

```
public class Car  
{  
    Engine engine;  
    public Car(Engine eng)  
    {  
        engine = eng;  
    }  
}
```





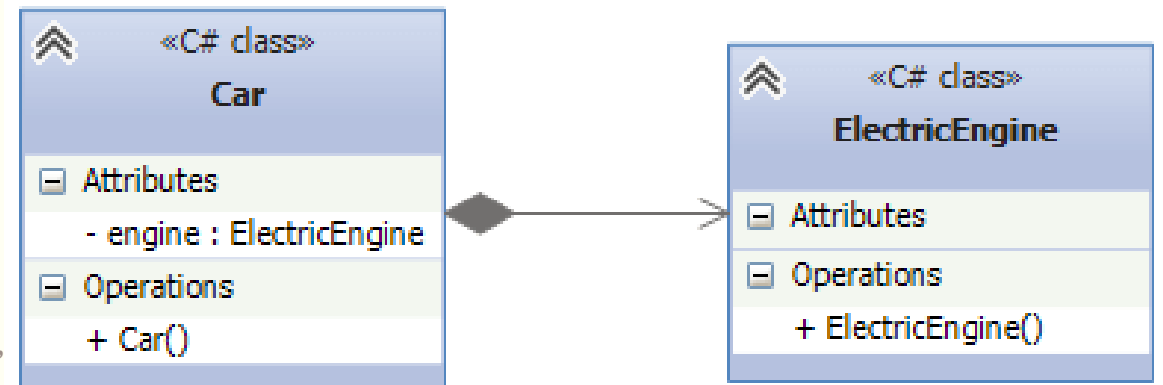
# Асоціація, агрегація та композиція



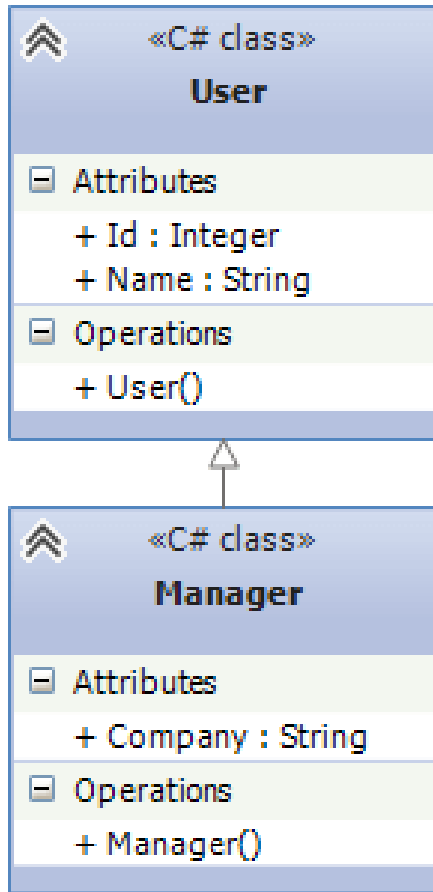
- Композиція визначає відношення HAS A.
  - Наприклад, клас автомобіля містить об'єкт класу електричного двигуна:

```
public class ElectricEngine
{ }

public class Car
{
    ElectricEngine engine;
    public Car()
    {
        engine = new ElectricEngine();
    }
}
```



# Види відношень між класами в UML. Узагальнення



- Наслідування є базовим принципом ООП та дозволяє одному класу (нащадку) успадковувати функціональність іншого класу (батьківського).
  - Наслідування визначає відношення IS-A, тобо "являється".

```
class User
{
    public int Id { get; set; }
    public string Name { get; set; }
}

class Manager : User
{
    public string Company { get; set; }
}
```

# Базовый механизм наследования

---

```
// Простой базовый класс.
class Car
{
    public readonly int maxSpeed;
    private int currSpeed;
    public Car(int max)
    {
        maxSpeed = max;
    }
    public Car()
    {
        maxSpeed = 55;
    }
    public int Speed
    {
        get { return currSpeed; }
        set
        {
            currSpeed = value;
            if (currSpeed > maxSpeed)
            {
                currSpeed = maxSpeed;
            }
        }
    }
}
```

- Наслідування — це аспект ООП, який полегшує повторне використання коду.
  - повторне використання коду існує в 2 видах: наслідування (відношення “являється”) та модель включення/делегатії (відношення “має”).
- Почнемо з класичної моделі наслідування.

```
static void Main(string[] args)
{
    Console.WriteLine("***** Basic Inheritance *****\n");
    // Создать экземпляр типа Car и установить максимальную скорость.
    Car myCar = new Car(80);
    // Установить текущую скорость и вывести ее на консоль.
    myCar.Speed = 50;
    Console.WriteLine("My car is going {0} MPH", myCar.Speed);
    Console.ReadLine();
}
```

# Вказування батьківського класу для існуючого класу

---

- Нехай планується побудова нового класу MiniVan.
  - Подібно до базового класу Car, необхідно, щоб MiniVan підтримував максимальну швидкість, поточну швидкість та властивість Speed, яка дозволяє користувачу змінювати стан об'єкта.
  - Існуючий клас Car, який слугуватиме основою для нового класу, називається **базовим** або **батьківським класом** або **суперкласом**.
  - Призначення базового класу полягає в означенні всіх спільних даних і членів для класів, які його розширяють.
  - Розширяючі класи формально називають **породженими** або **дочірніми класами** або **підкласами**.
- Об'єкти-MiniVan тепер мають доступ до всіх відкритих членів, визначених у базовому класі.
  - хоча до класу MiniVan не додані жодні члени, існує прямий доступ до відкритої властивості Speed батьківського класу і, таким чином, його код використовується повторно.
  - Це набагато краще, ніж створювати клас MiniVan, який матиме точно такі ж члени, що й Car.

```
// MiniVan "является" Car.  
class MiniVan : Car  
{  
}
```

# Наслідування зберігає інкапсуляцію

---

- Хоч конструктори зазвичай визначаються відкритими, породжений клас ніколи не успадковує конструктори свого батьківського класу.
  - Конструктори застосовуються тільки для створення екземпляру класу, всередині якого вони визначені.
  - В случае дублирования кода в этих двух классах придется сопровождать два фрагмента одинакового кода, что очевидно является непроизводительным расходом времени.
- Наслідування зберігає інкапсуляцію, тому наступний код викличе помилку компіляції, оскільки закриті члени ніколи не можуть бути доступними через посилання на об'єкт:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Basic Inheritance *****\n");
    ...
    // Создать объект MiniVan.
    MiniVan myVan = new MiniVan();
    myVan.Speed = 10;
    Console.WriteLine("My van is going {0} MPH",
        myVan.Speed);

    // Ошибка! Доступ к закрытым членам невозможен!
    myVan.currSpeed = 55;
    Console.ReadLine();
}
```

# Зауваження відносно багатьох базових класів

---

- С# вимагає, щоб будь-який конкретний клас мав тільки один безпосередній базовий клас.

```
// Не разрешено! Язык С# не допускает  
// множественного наследования классов!  
class WontWork  
    : BaseClassOne, BaseClassTwo  
{ }
```

- .NET дозволяє конкретному класу або структурі реалізовувати будь-яку кількість дискретних інтерфейсів.
- Завдяки цьому тип С# може надавати набір поведінок, уникаючи складнощів, характерних для множинного наслідування.
- У той час, як клас може мати тільки один безпосередній базовий клас, інтерфейс дозволено наслідувати від множини інших інтерфейсів.

# Ключеве слово sealed

---

- В C# підтримується ключеве слово — `sealed`, яке запобігає успадкуванню.
  - Если класс помечен как `sealed` (запечатанный), компилятор не позволяет наследовать от него.

```
// Класс Minivan не может быть расширен!  
sealed class Minivan : Car  
{  
}
```

- Зазвичай мають смисл при проектуванні обслуговуючого класу.
- У просторі імен `System` визначено багато запечатаних класів.
- Структури C# завжди неявно запечатані, тому успадкувати одну структуру від іншої чи клас від структури, чи структуру від класу неможливо.
- Структури можуть використовуватись тільки для моделювання окремих атомарних користувацьких типів.
- Для реалізації відношення “являється” повинні застосовуватись класи.

# Підрубиці щодо наслідування

---

```
class Employee
{
    ...
    // Новое поле и свойство.
    private int empAge;
    public int Age
    {
        get { return empAge; }
        set { empAge = value; }
    }

    // Обновленные конструкторы.
    public Employee() {}
    public Employee(string name, int id, float pay)
        :this(name, 0, id, pay){}
    public Employee(string name, int age, int id, float pay)
    {
        empName = name;
        empID = id;
        empAge = age;
        currPay = pay;
    }

    // Обновленный метод DisplayStats() теперь учитывает возраст.
    public void DisplayStats()
    {
        Console.WriteLine("Name: {0}", empName);
        Console.WriteLine("ID: {0}", empID);
        Console.WriteLine("Age: {0}", empAge);
        Console.WriteLine("Pay: {0}", currPay);
    }
}
```

- Загальна мета – створення сімейства класів, що моделюють різні типи працівників компанії.
- Нехай потрібно скористатись функціональністю класу Employee при створенні 2 нових класів (SalesPerson і Manager).
  - Новий клас SalesPerson “являється” Employee (як і Manager).
  - У моделі класичного наслідування базові класи (на зразок Employee) використовуються для визначення характеристик, спільних для всіх нащадків.
  - Підкласи (такі, як SalesPerson і Manager) розширяють спільну функціональність, додаючи додаткову специфічну функціональність.



## Підрообиці щодо наслідування

---

- Для нашого прикладу припустимо, що клас Manager розширяє Employee, зберігаючи кількість опціонів на акції, а клас SalesPerson підтримує кількість продажів.

```
// Менеджерам нужно знать количество их опционов на акции.  
class Manager : Employee  
{  
    public int StockOptions { get; set; }  
}
```

- Потім додайте новий файл класу (SalesPerson.cs), у якому визначено клас SalesPerson з відповідною автоматичною властивістю:

```
// Продавцям нужно знать количество продаж.  
class SalesPerson : Employee  
{  
    public int SalesNumber { get; set; }  
}
```

## Подробиці щодо наслідування

---

- Тепер після встановлення відношення “являється”, SalesPerson та Manager автоматично успадковують всі відкриті члени базового класу Employee.

```
// Создание объекта подкласса и доступ к функциональности базового класса.  
static void Main(string[] args)  
{  
    Console.WriteLine("***** The Employee Class Hierarchy *****\n");  
    SalesPerson fred = new SalesPerson();  
    fred.Age = 31;  
    fred.Name = "Fred";  
    fred.SalesNumber = 50;  
    Console.ReadLine();  
}
```

# Управління створенням базового класу за допомогою ключового слова `base`

---

- Зараз об'єкти `SalesPerson` і `Manager` можуть конструюватись лише з використанням стандартного конструктора

- Нехай до типу `Manager` додано новий конструктор, який приймає 6 аргументів

```
static void Main(string[] args)
{
    ...
    // Предположим, что у Manager есть конструктор со следующей сигнатурой:
    // (string fullName, int age, int empID,
    // float currPay, string ssn, int numbofOpts)
    Manager chucky = new Manager("Chucky", 50, 92, 100000, "333-23-2322", 9000);
    Console.ReadLine();
}
```

- Очевидно, що більшість з параметрів повинні зберігатись у змінних-членах, визначених у базовому класі `Employee`.

# Реалізуємо спеціальний конструктор

---

```
public Manager(string fullName, int age, int empID,
               float currPay, string ssn, int numbofOpts)
{
    // Это свойство определено в классе Manager.
    StockOptions = numbofOpts;

    // Присвоить входные параметры, используя
    // унаследованные свойства родительского класса.
    ID = empID;
    Age = age;
    Name = fullName;
    Pay = currPay;
    // Здесь возникнет ошибка компиляции, поскольку
    // свойство SSN доступно только для чтения!
    SocialSecurityNumber = ssn;
}
```

- Перша проблема: якщо визначити певну властивість як readonly (наприклад, SocialSecurityNumber), то присвоїти значення вхідного параметра string відповідному полю не вдасться.
- Друга проблема: було неявно створено досить неефективний конструктор, враховуючи, що в C#, якщо не вказати іншого, стандартний конструктор базового класу викликається автоматично перед виконанням логіки породженого конструктора.
  - Після цього поточна реалізація має доступ до багатьох відкритих властивостей базового класу Employee для встановлення його стану.

- 
- Для оптимізації створення породженого класу потрібно добре реалізувати конструктори підкласу, щоб вони явно викликали спеціальний конструктор базового класу замість стандартного конструктора.

- Модифікуємо спеціальний конструктор класу Manager, застосувавши ключове слово base:

```
public Manager(string fullName, int age, int empID,  
               float currPay, string ssn, int numbOfOpts)  
    : base(fullName, age, empID, currPay, ssn)  
    {  
        // Это свойство определено в классе Manager.  
        StockOptions = numbOfOpts;  
    }
```

- Тут явно викликається конструктор з 5 параметрами, визначений в Employee, що усуває зайві виклики під час створення екземпляру базового класу.

# Спеціальний конструктор SalesPerson

---

```
// В качестве общего правила, все подклассы должны явно вызывать
// соответствующий конструктор базового класса.
public SalesPerson(string fullName, int age, int empID,
                   float currPay, string ssn, int numbOfSales)
    : base(fullName, age, empID, currPay, ssn)
{
    // Это касается нас!
    SalesNumber = numbOfSales;
}
```

- Ключевое слово `base` можна використовувати повсюди, де підклас бажає звернутись до відкритого чи захищеного члена, визначеного в батьківському класі.
  - Застосування цього ключового слова не обмежується логікою конструктора.

# Збереження секретів сімейства: ключеве слово `protected`

---

- Коли базовий клас визначає захищені дані або захищені члени, він установлює набір елементів, які можуть бути доступними безпосередньо будь-якому нащадку.
  - Наприклад, щоб дозволити дочірнім класам `SalesPerson` і `Manager` безпосередньо звертатись до розділу даних, визначеному в `Employee`, можете змінити початковий клас `Employee`:

```
// Защищенные данные состояния.
partial class Employee
{
    // Теперь производные классы могут напрямую обращаться к этой информации.
    protected string empName;
    protected int empID;
    protected float currPay;
    protected int empAge;
    protected string empSSN;
    ...
}
```

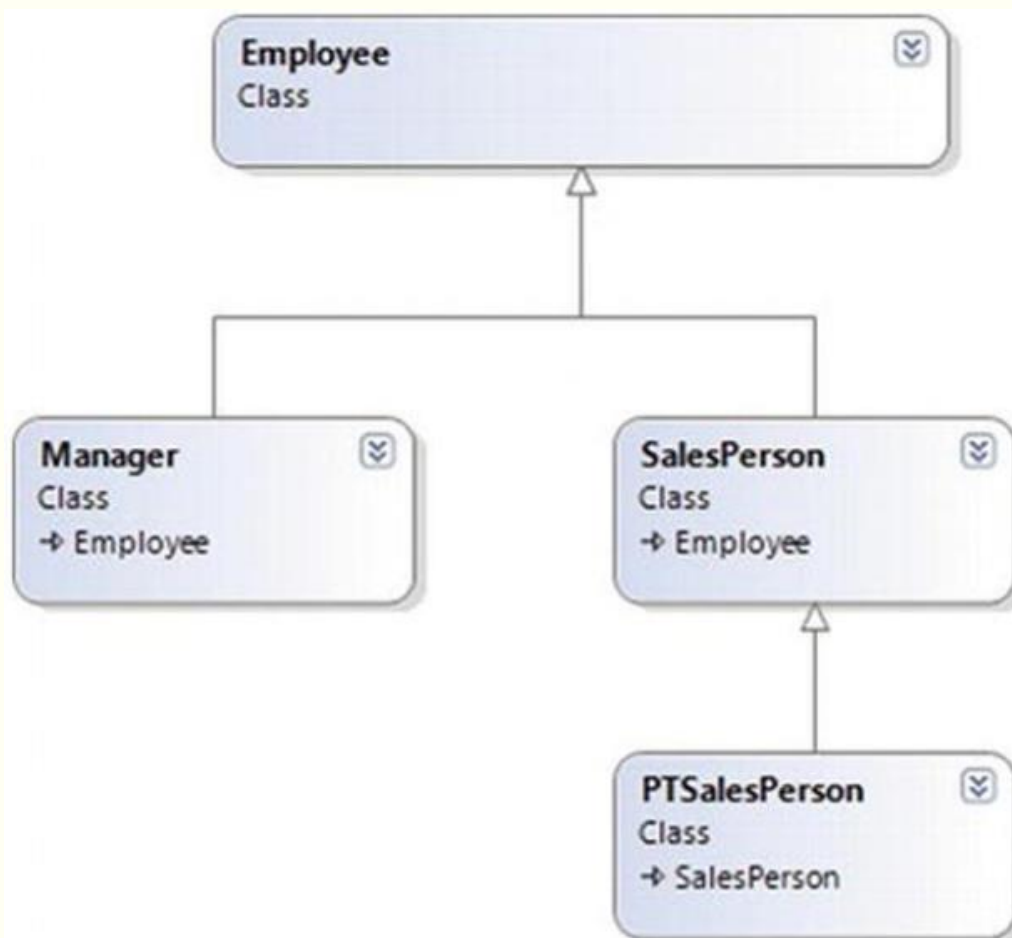
# Переваги і недоліки визначення захищених членів у базовому класі

---

- Перевага: породженим типам не потрібно опосередковано звертатись до даних, використовуючи відкриті методи та властивості.
- Можливий недолік: коли породжений тип має прямий доступ до внутрішніх даних свого батька, виникає ймовірність ненавмисного порушення існуючих бізнес-правил, які реалізовані в відкритих властивостях.
- З точки зору користувача об'єкта захищені дані трактуються як закриті.
  - Хоч захищені поля даних можуть порушити інкапсуляцію, оголошувати захищені методи досить безпечно (і корисно).
  - При побудові ієрархій класів дуже часто доводиться визначати набір методів, які використовуються тільки породженими типами й не призначені для застосування зовнішнім світом.



# Додавання запечатаного класу



- При побудові ієрархій класів можна з'ясувати, що деяка вітка в ланцюжку потребує «відсічення», оскільки подальше її розширення не має сенсу.

- Нехай у додаток додано ще один клас (PTSalesPerson), який розширяє існуючий тип SalesPerson.
- Клас PTSalesPerson представляє продавця, який працює на умовах часткової зайнятості.

```
sealed class PTSalesPerson : SalesPerson
{
    public PTSalesPerson(string fullName, int age, int empID,
                        float currPay, string ssn, int numbofSales)
        :base (fullName, age, empID, currPay, ssn, numbofSales)
    {
    }
    // Assume other members here...
}
```

# Реалізація моделі включення/делегації

---

```
// This new type will function as a contained class.
class BenefitPackage
{
    // Assume we have other members that represent
    // dental/health benefits, and so on.
    public double ComputePayDeduction()
    {
        return 125.0;
    }
}
```

■ Нехай створено новий клас, який моделює пакет пільг для працівників.

- Знадобиться виразити ідею про те, що кожний працівник «має» `BenefitPackage`.
- Для цього можна модифікувати визначення класу `Employee`
- Таким чином, один об'єкт успішно містить у собі інший об'єкт.

```
// Сотрудники имеют льготы.
partial class Employee
{
    // Содержит объект BenefitPackage.
    protected BenefitPackage empBenefits = new BenefitPackage();
    ...
}
```

# Реалізація моделі включення/делегації

---

```
public partial class Employee
{
    // Содержит объект BenefitPackage.
    protected BenefitPackage empBenefits = new BenefitPackage();

    // Открывает некоторое поведение, связанное с включенным объектом.
    public double GetBenefitCost()
    { return empBenefits.ComputePayDeduction(); }

    // Открывает объект через специальное свойство.
    public BenefitPackage Benefits
    {
        get { return empBenefits; }
        set { empBenefits = value; }
    }
    ...
}
```

```
static void Main(string[] args)
{
    Console.WriteLine("***** The Employee Class Hierarchy *****\n");
    ...
    Manager chucky = new Manager("Chucky", 50, 92, 100000, "333-23-2322", 9000);
    double cost = chucky.GetBenefitCost();
    Console.ReadLine();
}
```

- Проте для представлення функціональності включеного об'єкта зовнішньому світу, потрібна делегація.
  - *Делегація* — це акт додавання відкритих членів до включаючого (зовнішнього) класу, які використовують функціональність внутрішнього об'єкта.
  - Наприклад, можна було б оновити клас Employee, щоб він відкривав включений об'єкт empBenefits за допомогою спеціальної властивості, а також користуватись його функціональністю зсередини, через новий метод GetBenefitCost().
- BenefitsPackage визначений у типі Employee

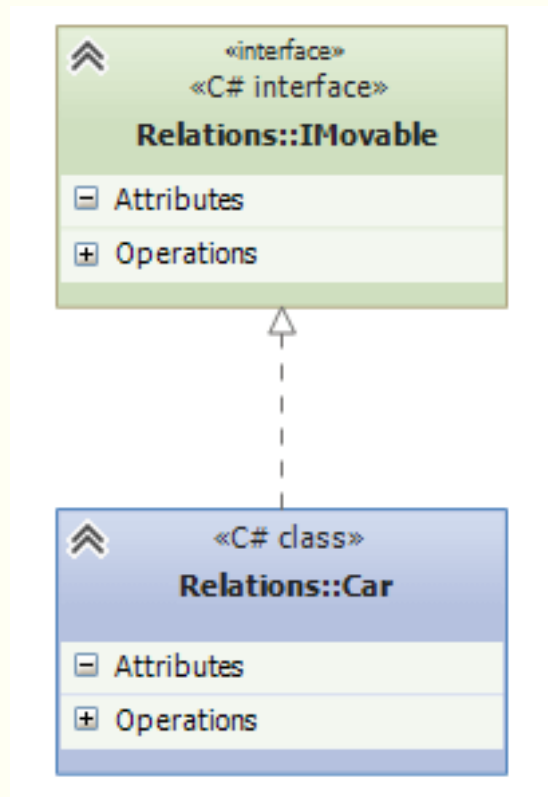
# Загальні рекомендації

---

- ***Замість наслідування краще обрати композицію.***
  - При успадкуванні вся функціональність класу-нащадка жорстко визначена на етапі компіляції.
  - Під час виконання програми ми не можемо її динамічно переозначити.
  - А клас-нащадок не завжди може переозначити, який визначено в батьківському класі.
  - Композиція, в свою чергу, дозволяє динамічно визначати поведінку об'єкта під час виконання, тому є більш гнучкою.
- ***Замість композиції слід віддавати перевагу агрегації,*** як більш гнучкому способу зв'язку компонентів.
  - Проте агрегація не завжди доречна.
  - Наприклад, класи Людина (головний компонент) та НервоваСистема (залежний компонент).
  - В реальності нервова система невіддільна від людини.
  - Створення та життєвий цикл компонентів буде спільний, тому тут краще композиція.

# Реалізація

---



- Реалізація передбачає визначення інтерфейсу та його реалізацію в класах.
  - Наприклад, маємо інтерфейс `IMovable` з методом `Move()`, який реалізується в класі `Car`:

```
public interface IMovable
{
    void Move();
}

public class Car : IMovable
{
    public void Move()
    {
        Console.WriteLine("Машина едет");
    }
}
```

# Вкладені типи

---

- У C# (та інших .NET-мовах) допускається визначати тип (перелічення, клас, інтерфейс, структуру або делегат) безпосередньо всередині контексту класу або структури.
  - При цьому вкладений (або “внутрішній”) тип вважається членом охоплюючого ( “зовнішнього”) класу, тому ним можна маніпулювати як будь-яким іншим членом.


```
public class OuterClass
{
    // Открытый вложенный тип может использоваться повсюду.
    public class PublicInnerClass {}

    // Закрытый вложенный тип может использоваться только членами включающего класса.
    private class PrivateInnerClass {}
}
```

- Вкладені типи дозволяють отримати повний контроль над рівнем доступу внутрішнього типу, оскільки вони можуть бути оголошені як закриті.
  - Невкладені класи не можуть оголошуватись з модифікатором `private`.
  - Оскільки вкладений тип є членом зовнішнього класу, він може мати доступ до його закритих членів.

```
private class MyClass { }
```

```
class Program
```

 class CitiesSurface.MyClass

CS1527: Элементы, определенные в пространстве имен, нельзя объявлять в явном виде как частные, защищенные, защищенные внутренние или частные защищенные.

- 
- Коли тип включає інший тип, він може створювати змінні-члени цього типу, як і будь-який інший елемент даних.
    - Проте якщо вкладений тип потрібно застосувати поза зовнішнім типом, його знадобиться кваліфікувати назвою зовнішнього типу.

```
static void Main(string[] args)
{
    // Создать и использовать открытый вложенный класс. Правильно!
    OuterClass.PublicInnerClass inner;
    inner = new OuterClass.PublicInnerClass();

    // Ошибка компиляции! Доступ к закрытому классу невозможен!
    OuterClass.PrivateInnerClass inner2;
    inner2 = new OuterClass.PrivateInnerClass();
}
```

# Вкладені типи та модифікатори доступу

---

```
public class Container
{
    class Nested
    {
        Nested() { }
    }
}
```

- Тип, визначений всередині класу, структури або інтерфейсу, називають **вкладеним (nested type)**.
  - За умовчанням вкладені типи закриті, тобто доступні тільки із зовнішнього типу.
  - Вкладеному класу можна встановити будь-який модифікатор доступу, проте якщо зовнішній клас запечатаний (sealed – не підтримує наслідування), модифікатори protected, protected internal або private protected для вкладеного класу згенерують попередження [CS0628](#): "new protected member declared in sealed class."
  - Вкладені структури можуть мати модифікатори доступу public, internal або private.
- Виклик конструктора вкладеного класу:
  - `Container.Nested nest = new Container.Nested();`
- Область видимості вкладеного класу обмежена областю видимості його зовнішнього класу.
  - Вкладеність типів підвищує інкапсульованість коду та надає користувачу можливість логічно групувати класи, які використовуються тільки в межах одного місця в коді.
  - Демонстрація роботи вкладених типів



---

---

```
// В Employee вложен класс BenefitPackage.  
public partial class Employee  
{  
    // В BenefitPackage вложено перечисление BenefitPackageLevel.  
    public class BenefitPackage  
    {  
        public enum BenefitPackageLevel  
        {  
            Standard, Gold, Platinum  
        }  
        public double ComputePayDeduction()  
        {  
            return 125.0;  
        }  
    }  
    ...  
}
```

```
static void Main(string[] args)  
{  
    ...  
    // Определить уровень льгот.  
    Employee.BenefitPackage.BenefitPackageLevel myBenefitLevel =  
        Employee.BenefitPackage.BenefitPackageLevel.Platinum;  
    Console.ReadLine()  
}
```

- Розглянемо концепцію на прикладі з працівниками.
  - Нехай визначення BenefitPackage вкладено безпосередньо в клас Employee (довільна глибина).
- Нехай потрібно створити перелічення BenefitPackageLevel, яке документує різні рівні пільг, що можуть надаватись працівнику.
  - Для програмного встановлення тісного зв'язку між Employee, BenefitPackage та BenefitPackageLevel, можна вкласти перелічення.

# Ініціалізація вкладених типів

---

```
class Rectangle
{
    private Point topLeft = new Point();
    private Point bottomRight = new Point();
    public Point TopLeft
    {
        get { return topLeft; }
        set { topLeft = value; }
    }
    public Point BottomRight
    {
        get { return bottomRight; }
        set { bottomRight = value; }
    }
    public void DisplayStats()
    {
        Console.WriteLine("[TopLeft: {0}, {1}, {2} BottomRight: {3}, {4}, {5}]",
            topLeft.X, topLeft.Y, topLeft.Color,
            bottomRight.X, bottomRight.Y, bottomRight.Color);
    }
}
```

- Відношення “має” дозволяє зіставляти нові класи, визначаючи змінні-члени існуючих класів.
  - Нехай існує клас Rectangle, який використовує тип Point для представлення координат верхнього лівого та нижнього правого кутів.
  - Оскільки автоматичні властивості задають усім внутрішнім змінним класів значення null, новий клас буде реалізовано з використанням “традиційного” синтаксису властивостей.

# Анонімні типи в C#

---

- **Анонімний тип** – це тип без назви, який містить тільки readonly-властивості.
  - Він не може містити інші члени: поля, методи, події тощо.
  - Створюється за допомогою оператора new разом з ініціалізатором об'єкта.
  - Ключове слово var використовується для тримання посилань на анонімні типи:

```
var student = new { Id = 1, FirstName = "James", LastName = "Bond" };
```

- Властивості анонімних типів не можуть ініціалізуватись null-значеннями, анонімною функцією чи вказівником.
  - Для доступу використовується стандартний оператор «.»

```
Console.WriteLine(student.Id); //виведе: 1
Console.WriteLine(student.FirstName); //виведе: James
Console.WriteLine(student.LastName); //виведе: Bond
student.Id = 2; //Error: cannot chage value
student.FirstName = "Steve"; //Error: cannot chage value
```

# Анонімні типи в C#

---

- Доступне створення анонімних масивів:

```
var students = new[] {  
    new { Id = 1, FirstName = "James", LastName = "Bond" },  
    new { Id = 2, FirstName = "Steve", LastName = "Jobs" },  
    new { Id = 3, FirstName = "Bill", LastName = "Gates" }  
};
```

- Анонімний тип завжди буде локальний для методу, де він був визначений.
  - Також анонімний тип не можна повертати з методу, проте можна передавати в метод як object (не рекомендується, краще обгорнути в клас або структуру).
- Всередині всі анонімні типи напряму породжені від класу `System.Object`.
    - Компілятор генерує клас із автозгенерованою назвою та застосовує доречний тип для кожної властивості залежно від значення виразу (value expression).
    - Проте з коду отримати його можна лише за допомогою методу `GetType()`:

```
static void Main(string[] args)  
{  
    var student = new { Id = 1, FirstName = "James", LastName = "Bond" };  
    Console.WriteLine(student.GetType().ToString());  
}
```

# Анонімні типи в C#

---

```
class Program {
    static void Main(string[] args) {
        IList<Student> studentList = new List<Student>() {
            new Student() { StudentID = 1, StudentName = "John", age = 18 },
            new Student() { StudentID = 2, StudentName = "Steve", age = 21 },
            new Student() { StudentID = 3, StudentName = "Bill", age = 18 },
            new Student() { StudentID = 4, StudentName = "Ram", age = 20 },
            new Student() { StudentID = 5, StudentName = "Ron", age = 21 }
        };
        var students = from s in studentList
                       select new { Id = s.StudentID, Name = s.StudentName };

        foreach(var stud in students)
            Console.WriteLine(stud.Id + "-" + stud.Name);
    }
}

public class Student {
    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public int age { get; set; }
}
```

- Переважно створюються в LINQ-запитах у результаті роботи оператора select.

- Вивід:  
1-John  
2-Steve  
3-Bill  
4-Ram  
5-Ron

- У коді оператор select з LINQ-запиту вибирає лише властивості StudentID та StudentName, а потім перейменовує їх в Id та Name відповідно.

# Самостійне вивчення

---

- Low Coupling і High Cohesion



# **ДЯКУЮ ЗА УВАГУ!**

**Наступне питання: Наслідування на прикладі структурованої обробки винятків**