ЖИТОМИРСЬКА ПОЛІТЕХНІКА
ДЕРЖАВНИЙ УНІВЕРСИТЕТ

C#. класи

Лекція 07

План

1. Відмінності між класами та структурами в C#
2. Що таке клас?
3. Синтаксис оголошення класу
4. Основні принципи ООП
5. Поля класу
6. Методи, конструктори, властивости
7. Створення об'єктів
8. Приклад

Відмінності між класами та структурами в C#

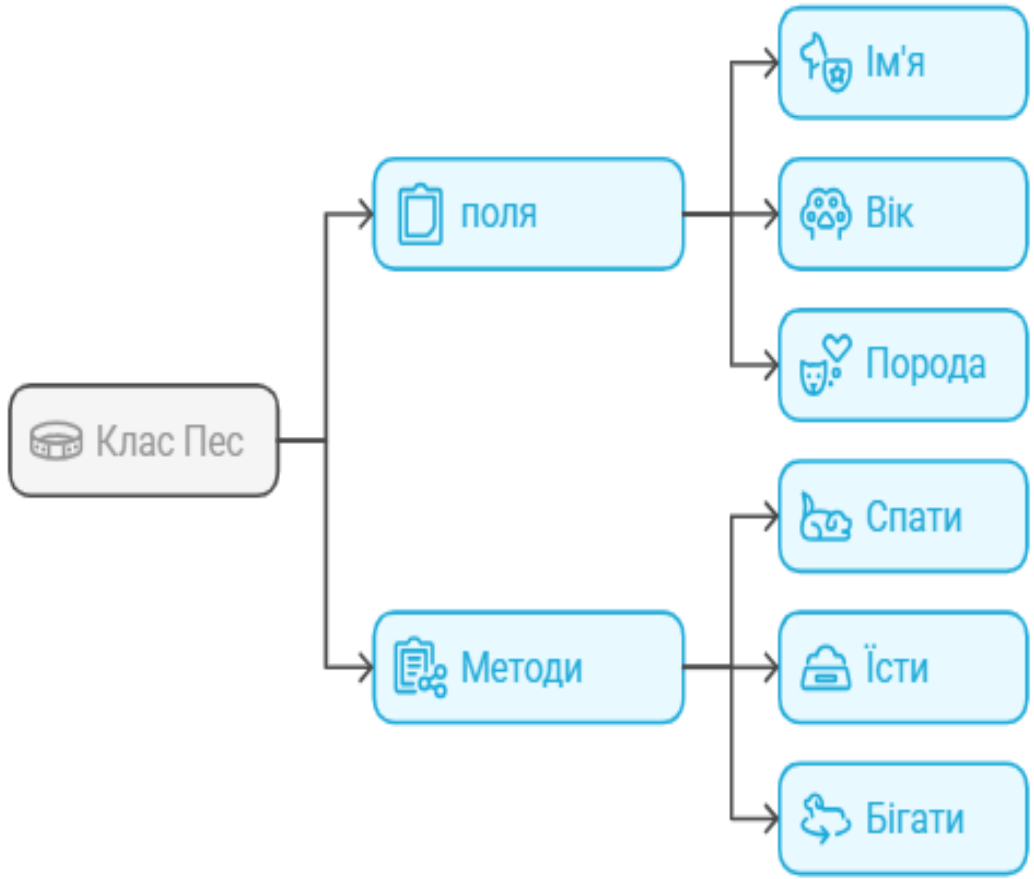
Характеристика	Класи (class)	Структури (struct)
Розташування в пам'яті	Купа	Стек
Успадкування	Підтримує успадкування	Не підтримує успадкування (але може реалізовувати інтерфейси)
Значення за замовчуванням	null (для посилальних типів)	Значення за замовчуванням для кожного поля (0, false, null тощо)
Передача параметрів	Передаються за посиланням (якщо не вказано ref або out	Передаються за значенням (копіюються)

Що таке клас?

Клас у C# - це фундаментальний будівельний блок об'єктно-орієнтованого програмування. Він служить шаблоном для створення об'єктів, визначаючи їхні властивості (дані) та методи (поведінку).

У C# клас — це визначений користувачем тип даних, який описує стан і поведінку об’єкта. Стан представлено властивостями, а поведінка відноситься до дій або методів, які може виконувати об’єкт.

Що таке клас?


Клас Пес
поля
Ім'я
Вік
Порода
Методи
Спати
Їсти
Бігати


AKITA INU
PITBULL
RIESENSCHNAUZER
SAINT BERNARD
POMERANIAN
DALMATIAN

Основні принципи ООП

- Інкапсуляція. Клас об'єднує дані та методи, які працюють з цими даними, в єдину сутність, захищаючи дані від несанкціонованого доступу.
- Успадкування. Клас може успадковувати властивості та методи іншого класу, що дозволяє створювати ієрархії класів і повторно використовувати код.
- Поліморфізм. Об'єкти різних класів можуть мати однакові методи, але реалізовувати їх по-різному, що дозволяє створювати гнучкі та розширювані програми.
- Абстракція. Клас дозволяє абстрагувати складні сутності реального світу, представляючи їх у вигляді простих і зрозумілих об'єктів.

Оголошення класу

```
[модифікатори]  class Ім'яКласу
{
  // оголошення полів
  [модифікатори]   тип  Ім'яПоля ;

// оголошення методів
  [модифікатор]  тип результату  Ім'яМетода   ( параметри )
 { … }
}
```

Оголошення класу

```
[модифікатори] class Ім'яКласу [: базовий_клас] [: інтерфейси]
{
        поля,
        методи,
        конструктори,
        події,
        деструктори,
        константи,
        тощо
}
```

Оголошення класу

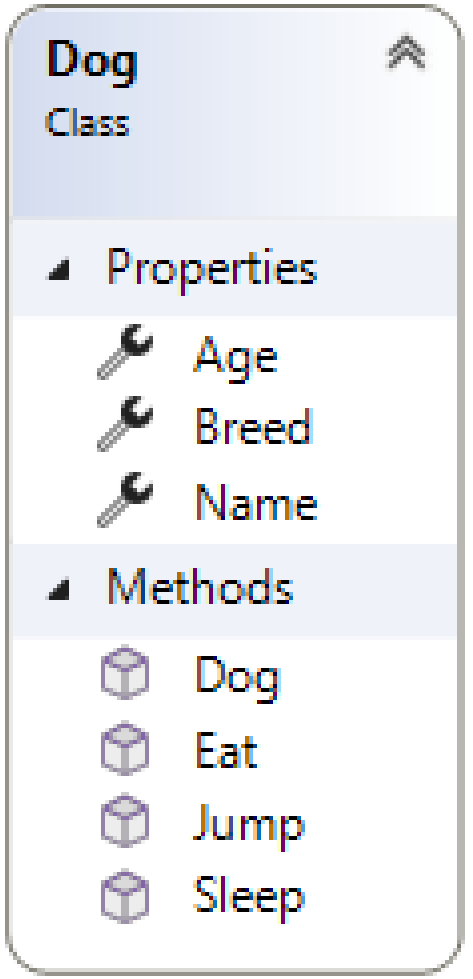
```
public class Dog
{
    public string name;
    public int age;
    public string breed;

    public Dog(string name, int age, string breed){
        this.name = name;
        this.age = age;
        this.breed = breed;
    }

    public void Sleep(){ Console.WriteLine($"{name} спить."); }

    public void Eat(){ Console.WriteLine($"{name} їсть."); }

    public void Jump() {Console.WriteLine($"{name} стрибає.");
    }
}
```



Основні модифікатори доступу до класів

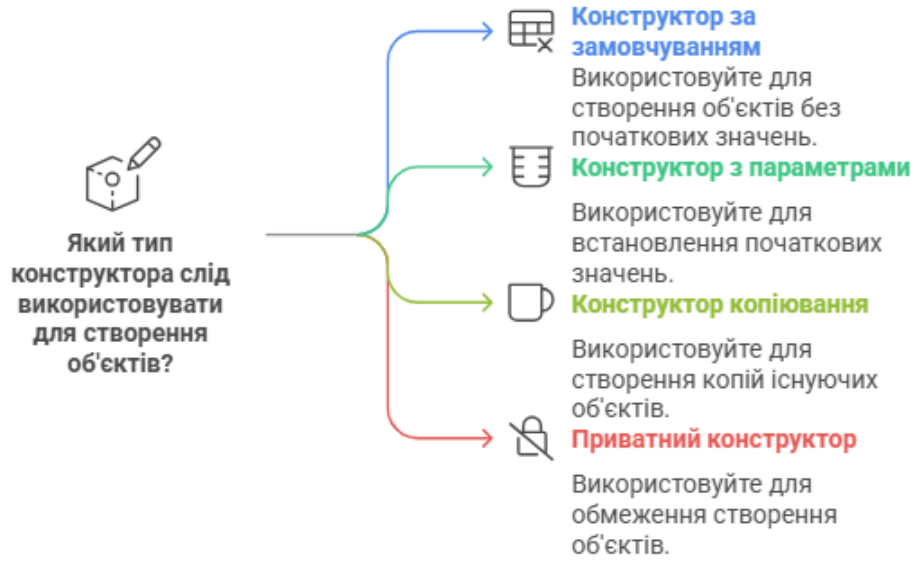
Модифікатор	Доступ з того ж класу	Доступ з похідного класу	Доступ з іншого класу в тій же збірці	Доступ з іншого класу в іншій збірці
public	✅	✅	✅	✅
private	✅	❌	❌	❌
protected	✅	✅	❌	❌
internal	✅	✅	✅	❌
protected internal	✅	✅	✅	✅

Конструктори

Конструктор – це спеціальний метод, який викликається при створенні нового екземпляру класу, він виділяє пам'ять необхідну для зберігання об’єкта, та як правило, виконує ініціалізацію полів та властивостей. Ім’я конструктора повинно бути ідентичним імені класу. Якщо в класі немає конструктора, то компілятор генерує конструктор за замовчуванням без параметрів.

Види конструкторів:

1. Конструктор за замовчуванням (Default Constructor).
2. Конструктор з параметрами (Parameterized Constructor).
3. Конструктор копіювання (Copy Constructor).
4. Приватний конструктор (Private Constructor).



Приклад

```
public class Dog{
    public string Name;
    public int Age;
    public string Breed;

    public Dog(string name, int age, string breed){Name = name;  Age = age;  Breed = breed; }

    public Dog(Dog copyDog){ Name = copyDog.Name; Age = copyDog.Age; Breed = copyDog.Breed; }

    private Dog() { }
    public static Dog PrivatDog (){
    Dog dog = new Dog();
    dog.Name = "Сінтія";
    dog.Age = 9;
    dog.Breed = "Golden Retriever";
    return dog;
    }
}
```

```
Dog dog1 = Dog.PrivatDog();
Dog dog2 = new Dog("Бадді", 3, "Beagle");
Dog dog3 = new Dog(dog2);
```

```
Ім'я: Сінтія, Вік: 9, Порода: Golden Retriever
Ім'я: Бадді, Вік: 3, Порода: Beagle
Ім'я: Бадді, Вік: 3, Порода: Beagle
```

Приклад

```
public class Dog
{
    public string Name;
    public int Age;
    public string Breed;
    public Dog(string name) => Name = name;
    public Dog(string name, int age) => SetParameters(name, age);

    private void SetParameters(string name, int age)
    {
        Name = name;
        Age = age;
    }

}
```

Конструктор приймає один параметр - name.
=> Name = name; - це скорочений синтаксис для присвоєння значення полю Name. Це означає, що коли створюється об'єкт Dog з ім'ям, це ім'я присвоюється полю Name.

Конструктор приймає два параметри – name і age.
=> SetParameters(name, age); - цей конструктор викликає метод SetParameters, передаючи йому параметри name та age.

Метод класу Dog. private означає, що цей метод доступний лише всередині класу Dog. Він приймає два параметри: name і age.
Цей метод присвоює значення параметрів name і age відповідним полям Name та Age об'єкта.

!!! Починаючи з 7.0 версії C# конструктор з одним виразом можна записати в скороченій формі

Приклад

```
Console.OutputEncoding = Encoding.Unicode;
Console.InputEncoding = Encoding.Unicode;

Dog dog1 = new Dog("Рекс");    // Використовуємо конструктор з одним параметром
dog1.Age = 3;                  // Встановлюємо вік окремо
dog1.Breed = "Німецька вівчарка"; // Встановлюємо породу окремо

Dog dog2 = new Dog("Бобік", 5);//Використовуємо конструктор з двома параметрами
dog2.Breed = "Дворняга";       // Встановлюємо породу окремо

Dog dog3 = new Dog("Лайка");
dog3.Age = 2;
dog3.Breed = "Сибірська лайка";

Console.WriteLine($"Ім'я: {dog1.Name}, Вік: {dog1.Age}, Порода: {dog1.Breed}");
Console.WriteLine($"Ім'я: {dog2.Name}, Вік: {dog2.Age}, Порода: {dog2.Breed}");
Console.WriteLine($"Ім'я: {dog3.Name}, Вік: {dog3.Age}, Порода: {dog3.Breed}");
```

Процес створення об'єктів в C#

У C# при створенні об'єктів класів використовується динамічна пам'ять, відома як купа (heap). Схема роботи :

1. **Запит пам'яті.** Коли створюється об'єкт за допомогою ключового слова new, система запитує у купи блок пам'яті, достатній для зберігання даних об'єкта.

2. Виділення пам'яті. Якщо в купі є достатньо вільної пам'яті, система виділяє необхідний блок і повертає адресу цього блоку.

3. Виклик конструктора. Після виділення пам'яті викликається конструктор класу, який ініціалізує поля об'єкта.

4. Повернення посилання. Оператор new повертає посилання на створений об'єкт, тобто адресу виділеного блоку в купі.

5. Використання об'єкта. Можна використовувати об'єкт, звертаючись до його полів і методів через отримане посилання.

6. Збирання сміття. Коли об'єкт більше не потрібен, збирач сміття (garbage collector) автоматично звільняє виділену пам'ять.

Усі дані, які зберігає клас, мають бути захищеними від прямого зовнішнього доступу. До них має бути дозволено доступ лише тільки за допомогою методів класу.

Не варто використовувати відкриті поля, це поганий стиль.
Для звернення до полів варто застосовувати методи!!!!

Приклад

```
public class Dog
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Breed { get; set; }

    public Dog(string name, int age, string breed){
        Name = name;
        Age = age;
        Breed = breed;
    }

    public void Sleep(){
        Console.WriteLine($"{Name} спить.");
    }
    public void Eat(){
        Console.WriteLine($"{Name} їсть.");
    }
    public void Jump(){
        Console.WriteLine($"{Name} стрибає.");
    }

}
```

```
static void Main(string[] args)
{
    Console.OutputEncoding = Encoding.Unicode;
    Console.InputEncoding = Encoding.Unicode;

    Dog myDog = new Dog("Рекс", 3, "Німецька вівчарка");

    myDog.Sleep();
    myDog.Eat();
    myDog.Jump();

    Console.WriteLine($"\nІм'я: {myDog.Name} \nВік:
{myDog.Age}\nПорода: {myDog.Breed}");}
}
```

```
Рекс спить.
Рекс їсть.
Рекс стрибає.

Ім'я: Рекс
Вік: 3
Порода: Німецька вівчарка
```

Властивості

У C# конструкція public string Name { get; set; } визначає властивість поля Name, яке має тип string.

Ключові слова get і set надають механізмами для доступу до приватного поля, що зберігає значення властивості, без необхідності безпосередньо маніпулювати цим полем.

get (аксессор) дозволяє отримати значення поля.
Коли іде звернення до поля Name для читання значення (наприклад, string myName = obj.Name;), викликається код, визначений у блоці get. Якщо блок get не містить явного коду, компілятор автоматично згенерує код для повернення значення приватного поля.

set (мутатор) дозволяє встановити значення поля.
Коли присвоюється значення Name (наприклад, obj.Name = "John";), викликається код, визначений у блоці set. Ключове слово value у блоці set представляє значення, яке потрібно присвоїти властивості. Якщо блок set не містить явного коду, компілятор автоматично згенерує код для присвоєння значення приватному полю.

Властивості

C# надає скорочений синтаксис для властивостей, які не потребують додаткової логіки в аксесорах (get, set)

```
public тип_даних Назва_Поля { get; set; }
```

Компілятор автоматично створює приватне поле для зберігання значення властивості.

Можна зробити поле доступним лише для читання (тільки get) або для запису (тільки set).

```
public тип_даних Назва_Поля { get;}
```

Зазвичай модифікатори властивостей public , оскільки вони входять у інтерфейс об'єкта.

Властивості

У C# властивості (properties) надають гнучкий механізм для доступу до полів класу. Вони дозволяють контролювати, як дані зберігаються та отримуються, і можуть включати додаткову логіку, таку як валідація даних або обчислення.

Властивість складається з двох спеціальних методів get, використовується для читання значення поля і set, використовується для запису значення в поле.

В аксесорах можна додати додаткову логіку, наприклад:

- перевірку коректності вхідних даних;
- обчислення значень на основі інших полів;
- виклик подій при зміні значення.

Властивості

Синтаксис:

```
public тип_даних [Назва_Поля/Назва_Властивості]
{
   get{ // Логіка для отримання значення
      return поле;
   }
   set{ // Логіка для встановлення значення
      поле = value;
   }
}
```

Основні характеристики властивостей:

Властивості забезпечують інкапсуляцію, приховуючи внутрішню реалізацію полів класу.
Дозволяє контролювати доступ, тобто, як дані можуть бути прочитані або змінені.
Можна додати логіку для валідації даних перед їх збереженням у поле.
Властивості можуть повертати обчислені значення, які не зберігаються безпосередньо в полі.

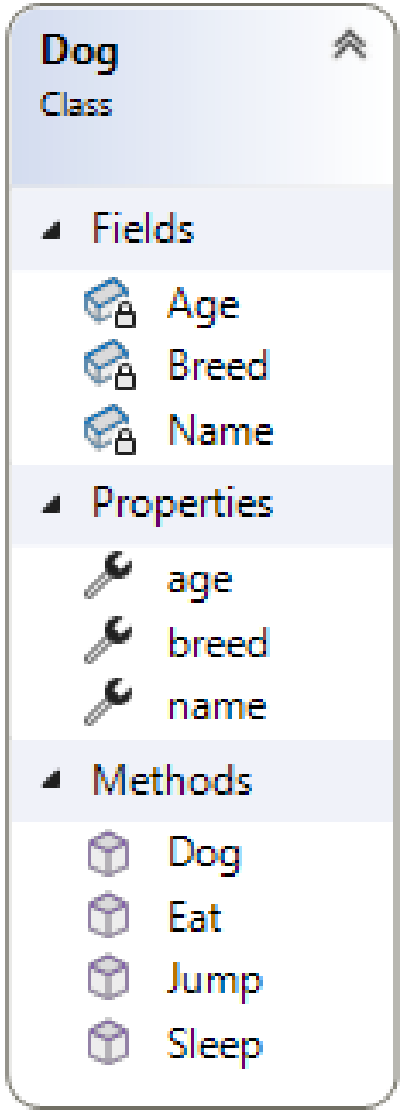
Приклад

```
public class Dog {
    private string Name;
    private  int Age;
    private string Breed;

    public string name {
       get { return Name; }
       set { Name = value; }
    }
    public int age {
       get { return Age; }
       set { Age = value; }
    }

    public string breed {
       get { return Breed; }
       set { Breed = value; }
    }
...
}
```

У класі Dog додаємо властивості таким чином реалізуємо інкапсуляцію.

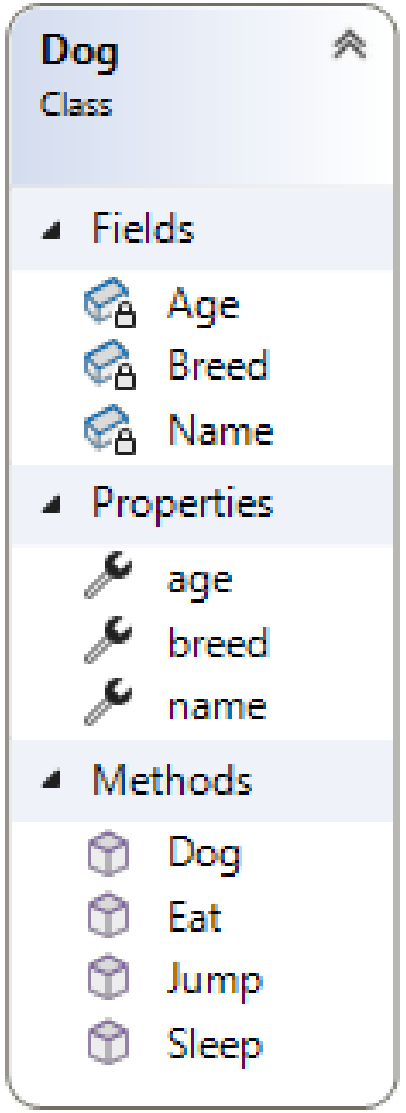


Приклад

```
public class Dog
{
    private string Name;
    private int Age;
    private string Breed;

    public string name {
        get => Name;
        set => Name = value;
    }
    public int age {
        get => Age;
        set => Age = value;
    }
    public string breed {
        get => Breed;
        set => Breed = value;
    } …
}
```

У версіях C#, починаючи з 7.0, синтаксис лямбда-виразів можна застосовувати всередині опису властивостей, якщо відповідний програмний блок можна реалізувати як один вираз.



Інкапсуляція

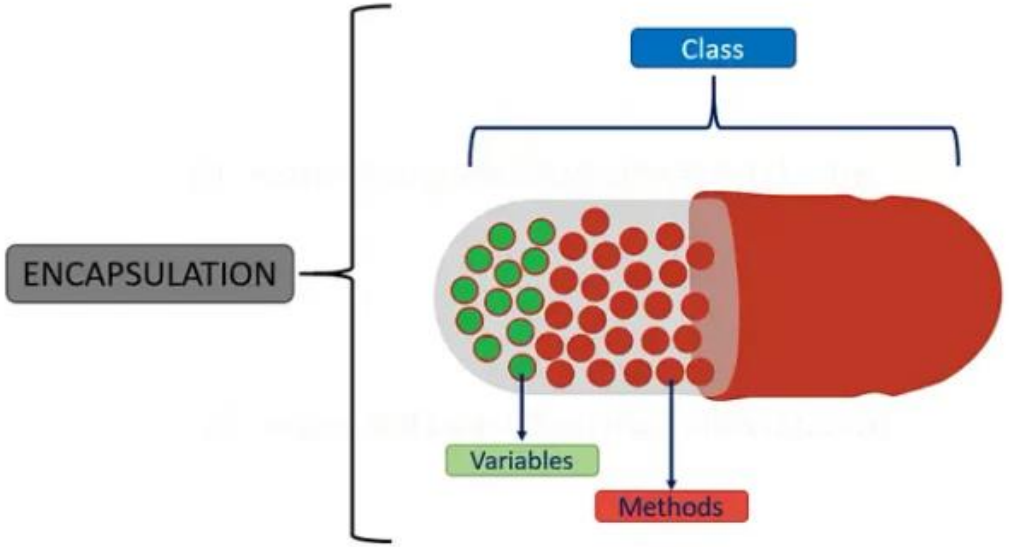
Інкапсуляція - це принцип ООП, який полягає в об'єднанні даних (полів) та методів, що працюють з цими даними, в одному класі. Це дозволяє приховати внутрішню реалізацію класу від зовнішнього світу і забезпечити контроль над доступом до даних.

Доступ до даних (полів) класу повинен здійснюватися тільки через методи

Інкапсуляція

Інкапсуляція дає можливість приховати внутрішню реалізацію, яка є важливою лише на етапі розробки класу.

На етапі використання об’єкта нам не важливо, яким чином він побудований в середині, нам важливі можливості, які він надає та як ними користуватися.

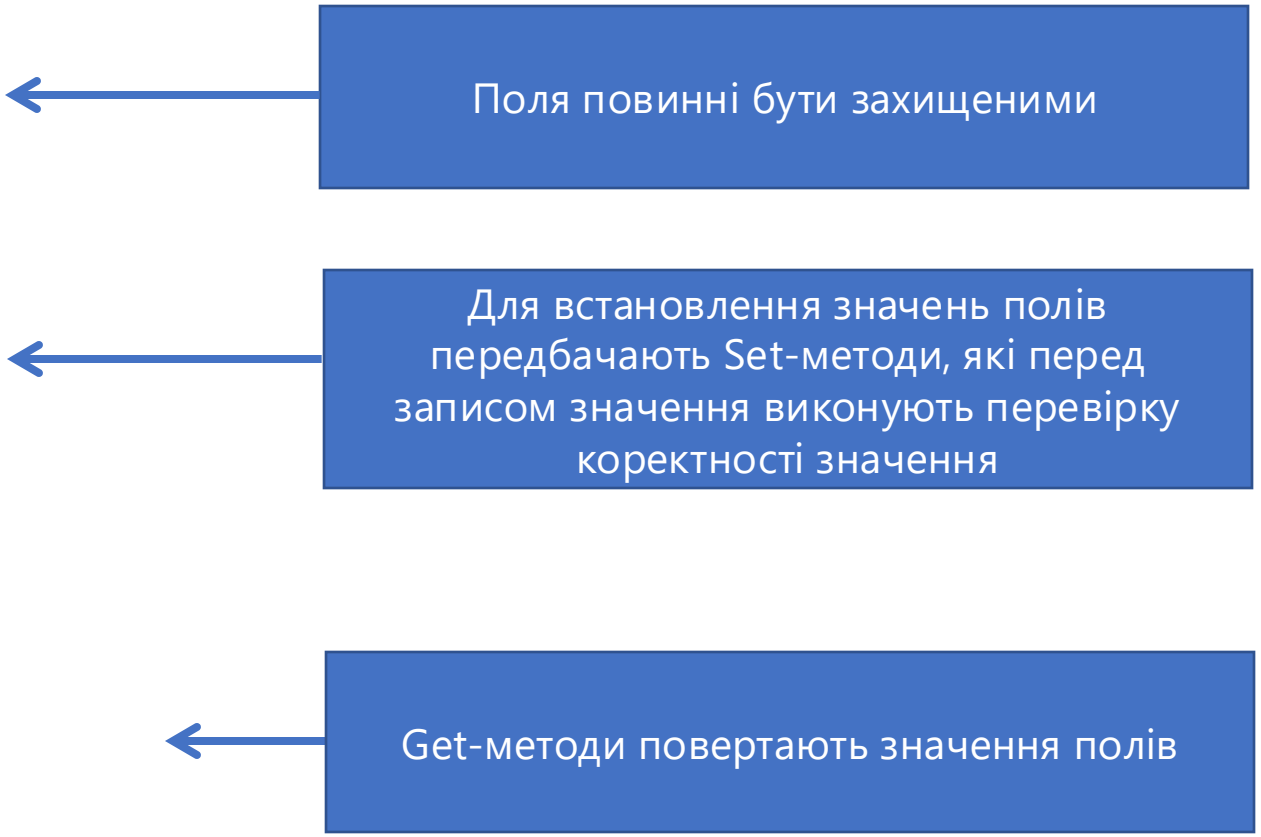


Приклад

```
public class Dog
{
    protected string Name;
    protected int Age;
    protected string Breed;

    public void SetName(string name){
        if (name.Length > 0)
            Name = name;
    }
    public void SetAge(int age){
        if (age > 1 && age < 18)
            Age = age;
    }
    public int GetAge(){ return Age;}
    public string GetName(){return Name;}

}
```



Інкапсуляція

Завдяки інкапсуляції клас «ззовні» виглядатиме так:

```
public class Dog
{
   public void   SetName(string name)
   public void   SetAge(int age)
   public int    GetAge()
   public string GetName()
}
```

Тобто будуть доступні лише відкриті елементи класу.

З назв методів зрозуміло, які дії вони виконують. При цьому клас «бере на себе» перевірку коректності даних, які будуть зберігатися у ньому. Перевірка здійснюється в Set-методах.

Інкапсуляція

При проектуванні класу, для деяких полів, не обов’язково надавати доступ для читання.

Наприклад,

пароль, який зберігається у класі повинен бути недоступний для читання. Але можна перевіряти правильність пароля. Пароль також можна змінювати правильно ввівши старий пароль.

Номер банківського рахунку. Ця інформація є конфіденційною і не повинна бути доступною для читання ззовні. Проте, можуть існувати методи для перевірки балансу або здійснення транзакцій.

Номер соціального страхування.

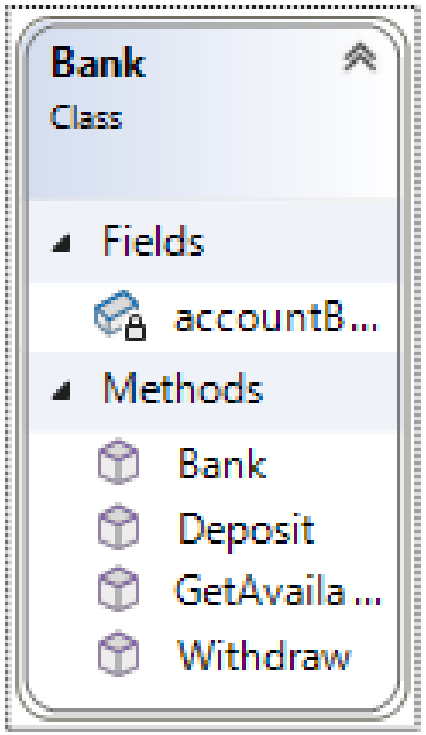
Дані медичних записів.

Приклад

```
public class Bank{
    private decimal accountBalance; // Поле доступне лише всередині класу

    public Bank(decimal initialBalance){
        accountBalance = initialBalance;
    }
    public void Deposit(decimal amount){
        if (amount > 0)  accountBalance += amount;
    }

    public void Withdraw(decimal amount){
        if (amount > 0 && amount <= accountBalance)
          accountBalance -= amount;
    }
    // Метод для отримання доступного балансу (для читання ззовні)
    public decimal GetAvailableBalance(){
        return accountBalance;
    }
}
```



Приклад

```
static void Main(string[] args)
{
    Console.OutputEncoding = Encoding.Unicode;
    Console.InputEncoding = Encoding.Unicode;

    Bank myBank = new Bank(1000);

    myBank.Deposit(500);

    myBank.Withdraw(200);

    decimal balance = myBank.GetAvailableBalance();

    System.Console.WriteLine($"Доступний баланс: {balance}");

}
```